



conference
proceedings

16th USENIX Symposium on Networked Systems Design and Implementation

Boston, MA, USA
February 26–28, 2019

Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation

Boston, MA, USA

February 26–28, 2019

ISBN 978-1-931971-49-2

Sponsored by
useenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

In cooperation with ACM SIGCOMM
and ACM SIGOPS

NSDI '19 Sponsors

Gold Sponsors

facebook



amazon

intel

Silver Sponsors

ByteDance

Google

Microsoft

NetApp®

Bronze Sponsors

**Futurewei
Technologies, Inc.**

ORACLE

TWO SIGMA

vmware®

General Sponsor

**CUBRO
NETWORK VISIBILITY**

Open Access Sponsor

NetApp®

Media Sponsor

No Starch Press

© 2019 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-931971-49-2

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google
Microsoft • NetApp

USENIX Benefactors

Amazon • Oracle • Two Sigma • VMware

USENIX Partners

BestVPN.com • Cisco Meraki
Teradactyl • TheBestVPN.com

Open Access Publishing Partner

PeerJ

USENIX Association

**Proceedings of the
16th USENIX Symposium
on Networked Systems Design
and Implementation**

**February 26–28, 2019
Boston, MA, USA**

Conference Organizers

Program Co-Chairs

Jay Lorch, *Microsoft Research*
Minlan Yu, *Harvard University*

Program Committee

Fadel Adib, *Massachusetts Institute of Technology*
Aditya Akella, *University of Wisconsin–Madison*
Katerina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*
Aruna Balasubramanian, *Stony Brook University*
Sujata Banerjee, *VMware Research*
Paul Barham, *Google*
Ranjita Bhagwan, *Microsoft Research*
Kai Chen, *Hong Kong University of Science and Technology*
Vijay Chidambaram, *The University of Texas at Austin and VMware Research*
Mosharaf Chowdhury, *University of Michigan*
Asaf Cidon, *Barracuda Networks*
Anja Feldmann, *Technische Universität Berlin*
Bryan Ford, *École Polytechnique Fédérale de Lausanne (EPFL)*
Roxana Geambasu, *Columbia University*
Manya Ghobadi, *Microsoft Research*
Jana Giceva, *Imperial College London*
Ronghui Gu, *Columbia University*
Haryadi Gunawi, *University of Chicago*
Andreas Haeberlen, *University of Pennsylvania*
Haitham Hassanieh, *University of Illinois at Urbana-Champaign*
Jon Howell, *VMware Research*
Rebecca Isaacs, *Twitter*
Xin Jin, *Johns Hopkins University*
Srikanth Kandula, *Microsoft Research*
Manos Kapritsos, *University of Michigan*
Dejan Kostić, *KTH Royal Institute of Technology*
Ramakrishna Kotla, *Amazon Web Services*
Arvind Krishnamurthy, *University of Washington*
Hongqiang Liu, *Alibaba*
Harsha V. Madhyastha, *University of Michigan and Google*
Dahlia Malkhi, *VMware Research*
Allison Mankin, *Salesforce*
Derek Murray, *Google*
Aurojit Panda, *New York University*
KyoungSoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*
Amar Phanishayee, *Microsoft Research*
Raluca Ada Popa, *University of California, Berkeley*

George Porter, *University of California, San Diego*

Lili Qiu, *The University of Texas at Austin*

K. K. Ramakrishnan, *University of California, Riverside*

Michael Schapira, *Hebrew University of Jerusalem*

Cole Schlesinger, *Amazon Web Services*

Vyas Sekar, *Carnegie Mellon University*

Ankit Singla, *ETH Zurich*

Anirudh Sivaraman, *New York University*

Alex C. Snoeren, *University of California, San Diego*

Michael Stumm, *University of Toronto*

Ryan Stutsman, *University of Utah*

Geoff Voelker, *University of California, San Diego*

Hakim Weatherspoon, *Cornell University*

John Wilkes, *Google*

Keith Winstein, *Stanford University*

Jie Xiong, *University of Massachusetts Amherst*

James Hongyi Zeng, *Facebook*

Irene Zhang, *Microsoft Research*

Xinyu Zhang, *University of California, San Diego*

Lin Zhong, *Rice University*

Poster Session Co-Chairs

Xin Jin, *Johns Hopkins University*

Ryan Stutsman, *University of Utah*

Test of Time Awards Committee

Tom Anderson, *University of Washington*

Jennifer Rexford, *Princeton University*

Emin Gun Sirer, *Cornell University*

Preview Sessions Chair

Aurojit Panda, *New York University*

Steering Committee

Aditya Akella, *University of Wisconsin–Madison*

Katerina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*

Sujata Banerjee, *VMware Research*

Paul Barham, *Google*

Nick Feamster, *Princeton University*

Casey Henderson, *USENIX Association*

Jon Howell, *VMware Research*

Arvind Krishnamurthy, *University of Washington*

Jeff Mogul, *Google*

Brian Noble, *University of Michigan*

Timothy Roscoe, *ETH Zurich*

Srinivasan Seshan, *Carnegie Mellon University*

External Reviewers

David Andersen
Tom Barbetter
Nathan Beckmann
Ignacio Cano
Philip Carns
Alex Dimakis
Alireza Farshin
Michael J. Freedman

Cheng Huang
Ryan Huang
Georgios Katsikas
Swarun Kumar
Steffen Maass
Stephanos Matsumoto
Max Mellette
Jayashree Mohan

Ravi Netravali
Soujanya Ponnappalli
Waleed Reda
Amir Rozbeh
Edo Roth
Brian Sandler
Steffen Smolka
Jon Stewart

Ion Stoica
Paul Wankadia
Tian Yang
Xinhao Yuan
Ennan Zhai
Shizhen Zhao
Haitao Zheng

Message from the NSDI '19 Program Co-Chairs

Welcome to NSDI '19!

NSDI is traditionally the top venue for papers on networked and distributed systems, and this year we continue that tradition with an excellent program featuring a record number of papers. The research and experiences described in this year's program include a wide range of topics including analytics, data center network architecture, distributed systems, host networking, machine learning, modern network hardware, monitoring and diagnosis, operating systems, privacy, security, and wireless networking.

This year, we made two major changes to the review process: we offered two submission deadlines (Spring and Fall) and we provided the possibility of getting one-shot-revision decisions in lieu of rejection. We received 332 submissions (92 in Spring and 240 in Fall), of which we accepted 49 papers. 45 of the 49 accepted papers were accepted outright; the remaining four were submitted in Spring, given a one-shot-revision decision, and accepted on resubmission in Fall. Both the overall submission count and the overall acceptance count were higher than any past instance of NSDI, indicating a dramatic increase in popularity this year. Having anticipated the high number of submissions, we also had a record-high program-committee size: 59 experts, spanning research and industry.

The review process was double-blind, befitting its role in the scientific process. We had two rounds of reviews, providing papers that advanced to the second round at least five reviews. In total, the program committee and external reviewers generated 1,191 reviews (338 in Spring, 853 in Fall). We strove to include valuable feedback in all these reviews, so we hope it benefited all authors who submitted their work. After writing reviews, we held online discussions to select papers to be discussed further at PC meetings or to be accepted without further need for discussion. We discussed 30 papers during the Spring PC meeting (three hours online on each of two consecutive days) and 57 papers during the 1.5-day Fall PC meeting on the Microsoft campus in Redmond, WA.

We'd like to thank the many, many people whose work was necessary to arrange this conference. Foremost, we thank all the authors who chose to send their strong work to NSDI. We also thank the program committee whose diligence, professionalism, expertise, excitement, and courtesy made the review process go smoothly and successfully. Special thanks to those of them who took on extra responsibilities beyond the considerable ones PC members already have: Ryan Stutsman and Xin Jin for serving as poster chairs; Aurojit Panda for serving in the new-this-year role of preview-sessions chair; Jon Howell and Lin Zhong for administering the review process for papers both chairs were conflicted with; and Geoff Voelker, KyoungSoo Park, Lili Qiu, and Rebecca Isaacs for selecting the Best Paper and Community Awards. Thanks also to the members of the Test of Time Awards Committee: Emin Gün Sirer, Jennifer Rexford, and Tom Anderson. We also want to thank the many people who helped us create and refine the proposal for multiple deadlines and one-shot revision, including Aditya Akella, Alex C. Snoeren, Apu Kapadia, Brad Karp, Bryan Parno, Casey Henderson, Dave Lomet, Dina Papagiannaki, Divesh Srivastava, H. V. Jagadish, James Mickens, Jason Flinn, Jeff Mogul, Jon Howell, Justin Levandoski, Nick Feamster, Phil Bernstein, Rachit Agarwal, Renata Teixeira, Timothy Roscoe, Zhi-Li Zhang, and the NSDI and SIGCOMM steering committees. The NSDI steering committee was also helpful to us in many other ways, providing guidance when we needed it. The chairs of recent NSDI conferences were also reliable sources of useful advice: Aditya Akella, Jon Howell, Katerina Argyraki, Rebecca Isaacs, Srinivasan Seshan, and Sujata Banerjee. We also wish to thank Arvind Krishnamurthy for his help arranging conferencing for the Spring online PC meetings, and Marissa Storm and Shuk Kam for their help facilitating the use of a Microsoft room for the Fall PC meeting. We're also very grateful to the USENIX staff, including Casey Henderson, Ginny Staubach, Jasmine Murcia, Jessica Kim, Michele Nelson, and Sarah TerHune, for the extraordinary levels of support they provided.

Jay Lorch, *Microsoft Research*
Minlan Yu, *Harvard University*
NSDI '19 Program Co-Chairs

NSDI '19: 16th USENIX Symposium on Networked Systems Design and Implementation

February 26–28, 2019

Boston, MA, USA

Host Networking

Datacenter RPCs can be General and Fast	1
Anuj Kalia, <i>Carnegie Mellon University</i> ; Michael Kaminsky, <i>Intel Labs</i> ; David Andersen, <i>Carnegie Mellon University</i>	
Eiffel: Efficient and Flexible Software Packet Scheduling	17
Ahmed Saeed and Yimeng Zhao, <i>Georgia Institute of Technology</i> ; Nandita Dukkipati, <i>Google</i> ; Ellen Zegura and Mostafa Ammar, <i>Georgia Institute of Technology</i> ; Khaled Harras, <i>Carnegie Mellon University</i> ; Amin Vahdat, <i>Google</i>	
Loom: Flexible and Efficient NIC Packet Scheduling.....	33
Brent Stephens, <i>UIC</i> ; Aditya Akella and Michael Swift, <i>UW-Madison</i>	

Distributed Systems

Exploiting Commutativity For Practical Fast Replication	47
Seo Jin Park and John Ousterhout, <i>Stanford University</i>	
Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification	65
Assaf Eisenman, <i>Stanford University</i> ; Asaf Cidon, <i>Stanford University and Barracuda Networks</i> ; Evgenya Pergament and Or Haimovich, <i>Stanford University</i> ; Ryan Stutsman, <i>University of Utah</i> ; Mohammad Alizadeh, <i>MIT CSAIL</i> ; Sachin Katti, <i>Stanford University</i>	
Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores	79
Diego Didona, <i>EPFL</i> ; Willy Zwaenepoel, <i>EPFL and University of Sydney</i>	
Monoxide: Scale out Blockchains with Asynchronous Consensus Zones	95
Jiaping Wang, <i>ICT/CAS, Sinovation AI Institute</i> ; Hao Wang, <i>Ohio State University</i>	

Modern Network Hardware

FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds.....	113
Daehyeok Kim and Tianlong Yu, <i>Carnegie Mellon University</i> ; Hongqiang Harry Liu, <i>Alibaba</i> ; Yibo Zhu, <i>Microsoft and Bytedance</i> ; Jitu Padhye and Shachar Raindel, <i>Microsoft</i> ; Chuanxiong Guo, <i>Bytedance</i> ; Vyas Sekar and Srinivasan Seshan, <i>Carnegie Mellon University</i>	

Direct Universal Access: Making Data Center Resources Available to FPGA.....	127
Ran Shu and Peng Cheng, <i>Microsoft Research</i> ; Guo Chen, <i>Microsoft Research & Hunan University</i> ; Zhiyuan Guo, <i>Microsoft Research & Beihang University</i> ; Lei Qu and Yongqiang Xiong, <i>Microsoft Research</i> ; Derek Chiou and Thomas Moscibroda, <i>Microsoft Azure</i>	

Stardust: Divide and Conquer in the Data Center Network	141
Noa Zilberman, <i>University of Cambridge</i> ; Gabi Bracha and Golan Schzukin, <i>Broadcom</i>	

Blink: Fast Connectivity Recovery Entirely in the Data Plane	161
Thomas Holterbach, Edgar Costa Molero, and Maria Apostolaki, <i>ETH Zurich</i> ; Alberto Dainotti, <i>CAIDA/UC San Diego</i> ; Stefano Vissicchio, <i>UC London</i> ; Laurent Vanbever, <i>ETH Zurich</i>	

Analytics

Hydra: a federated resource manager for data-center scale analytics	177
Carlo Curino, Subru Krishnan, and Konstantinos Karanasos, <i>Microsoft</i> ; Sriram Rao, <i>Facebook</i> ; Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan, <i>Microsoft</i>	

Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure	193
Qifan Pu, <i>UC Berkeley</i> ; Shivaram Venkataraman, <i>University of Wisconsin, Madison</i> ; Ion Stoica, <i>UC Berkeley</i>	

dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces	207
Da Yu, <i>Brown University</i> ; Yibo Zhu, <i>Microsoft and Bytedance</i> ; Behnaz Arzani, <i>Microsoft</i> ; Rodrigo Fonseca, <i>Brown University</i> ; Tianrong Zhang, Karl Deng, and Lihua Yuan, <i>Microsoft</i>	

Data Center Network Architecture

Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks	221
Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat, <i>Google, Inc.</i>	
Understanding Lifecycle Management Complexity of Datacenter Topologies	235
Mingyang Zhang, <i>University of Southern California</i> ; Radhika Niranjan Mysore, <i>VMware Research</i> ; Sucha Supittayapornpong and Ramesh Govindan, <i>University of Southern California</i>	
Shoal: A Network Architecture for Disaggregated Racks	255
Vishal Shrivastav, <i>Cornell University</i> ; Asaf Valadarsky, <i>Hebrew University of Jerusalem</i> ; Hitesh Ballani and Paolo Costa, <i>Microsoft Research</i> ; Ki Suh Lee, <i>Waltz Networks</i> ; Han Wang, <i>Barefoot Networks</i> ; Rachit Agarwal and Hakim Weatherspoon, <i>Cornell University</i>	

Wireless Technologies

NetScatter: Enabling Large-Scale Backscatter Networks	271
Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota, <i>University of Washington</i>	
Towards Programming the Radio Environment with Large Arrays of Inexpensive Antennas	285
Zhuqi Li, Yaxiong Xie, and Longfei Shangguan, <i>Princeton University</i> ; Rotman Ivan Zelaya, <i>Yale University</i> ; Jeremy Gummesson, <i>UMass Amherst</i> ; Wenjun Hu, <i>Yale University</i> ; Kyle Jamieson, <i>Princeton University</i>	
Pushing the Range Limits of Commercial Passive RFIDs	301
Jingxian Wang, <i>Carnegie Mellon University</i> ; Junbo Zhang, <i>Tsinghua University</i> ; Rajarshi Saha, <i>IIT Kharagpur</i> ; Haojian Jin and Swarun Kumar, <i>Carnegie Mellon University</i>	
SweepSense: Sensing 5 GHz in 5 Milliseconds with Low-cost Radios	317
Yeswanth Gudde, <i>UC San Diego</i> ; Raghav Subbaraman, <i>IIT Madras</i> ; Moein Khazraee, Aaron Schulman, and Dinesh Bharadia, <i>UC San Diego</i>	

Operating Systems

Slim: OS Kernel Support for a Low-Overhead Container Overlay Network	331
Danyang Zhuo and Kaiyuan Zhang, <i>University of Washington</i> ; Yibo Zhu, <i>Microsoft and Bytedance</i> ; Hongqiang Harry Liu, <i>Alibaba</i> ; Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson, <i>University of Washington</i>	
Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency	345
Kostis Kaffles, Timothy Chong, and Jack Tigar Humphries, <i>Stanford University</i> ; Adam Belay, <i>Massachusetts Institute of Technology</i> ; David Mazières and Christos Kozyrakis, <i>Stanford University</i>	
Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads	361
Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan, <i>MIT CSAIL</i>	

Monitoring and Diagnosis

End-to-end I/O Monitoring on a Leading Supercomputer	379
Bin Yang, <i>Shandong University, National Supercomputing Center in Wuxi</i> ; Xu Ji, <i>Tsinghua University, National Supercomputing Center in Wuxi</i> ; Xiaosong Ma, <i>Qatar Computing Research institute, HBKU</i> ; Xiyang Wang, <i>National Supercomputing Center in Wuxi</i> ; Tianyu Zhang and Xiupeng Zhu, <i>Shandong University, National Supercomputing Center in Wuxi</i> ; Nosayba El-Sayed, <i>Emory University</i> ; Haidong Lan and Yibo Yang, <i>Shandong University</i> ; Jidong Zhai, <i>Tsinghua University</i> ; Weiguo Liu, <i>Shandong University, National Supercomputing Center in Wuxi</i> ; Wei Xue, <i>Tsinghua University, National Supercomputing Center in Wuxi</i>	
Zeno: Diagnosing Performance Problems with Temporal Provenance	395
Yang Wu, <i>Facebook</i> ; Ang Chen, <i>Rice University</i> ; Linh Thi Xuan Phan, <i>University of Pennsylvania</i>	
Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks	421
Anurag Khandelwal, <i>UC Berkeley</i> ; Rachit Agarwal, <i>Cornell University</i> ; Ion Stoica, <i>UC Berkeley</i>	

(continued on next page)

DETER: Deterministic TCP Replay for Performance Diagnosis	437
Yuliang Li, <i>Harvard University</i> ; Rui Miao, <i>Alibaba Group</i> ; Mohammad Alizadeh, <i>Massachusetts Institute of Technology</i> ;	
Minlan Yu, <i>Harvard University</i>	

Improving Machine Learning

JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs	453
Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun, <i>Seoul National University</i>	
BLAS-on-flash: An Efficient Alternative for Large Scale ML Training and Inference?	469
Suhas Jayaram Subramanya and Harsha Vardhan Simhadri, <i>Microsoft Research India</i> ; Srajan Garg, <i>IIT Bombay</i> ;	
Anil Kag and Venkatesh Balasubramanian, <i>Microsoft Research India</i>	

Tiresias: A GPU Cluster Manager for Distributed Deep Learning.	485
Juncheng Gu, Mosharaf Chowdhury, and Kang G. Shin, <i>University of Michigan, Ann Arbor</i> ; Yibo Zhu, <i>Microsoft and Bytedance</i> ; Myeongjae Jeon, <i>Microsoft and UNIST</i> ; Junjie Qian, <i>Microsoft</i> ; Hongqiang Liu, <i>Alibaba</i> ;	
Chuanxiong Guo, <i>Bytedance</i>	

Network Functions

Correctness and Performance for Stateful Chained Network Functions	501
Junaid Khalid and Aditya Akella, <i>University of Wisconsin - Madison</i>	

Performance Contracts for Software Network Functions	517
Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Canea, <i>EPFL</i>	

FlowBlaze: Stateful Packet Processing in Hardware	531
Salvatore Pontarelli, <i>Axbryd/CNIT</i> ; Roberto Bifulco, <i>NEC Laboratories Europe</i> ; Marco Bonola, <i>Axbryd/CNIT</i> ;	
Carmelo Cascone, <i>Open Networking Foundation</i> ; Marco Spaziani and Valerio Bruschi, <i>CNIT/University of Rome Tor Vergata</i> ;	
Davide Sanvitto, <i>Politecnico di Milano</i> ; Giuseppe Siracusano, <i>NEC Laboratories Europe</i> ; Antonio Capone, <i>Politecnico di Milano</i> ;	
Michio Honda and Felipe Huici, <i>NEC Laboratories Europe</i> ; Giuseppe Bianchi, <i>CNIT/University of Rome Tor Vergata</i>	

Network Characterization

SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks	549
Yilong Geng, Shiyu Liu, and Zi Yin, <i>Stanford University</i> ; Ashish Naik, <i>Google Inc.</i> ; Balaji Prabhakar and Mendel Rosenblum, <i>Stanford University</i> ; Amin Vahdat, <i>Google Inc.</i>	

Is advance knowledge of flow sizes a plausible assumption?	565
Vojislav Đukić, <i>ETH Zurich</i> ; Sangeetha Abdu Jyothi, <i>University of Illinois at Urbana-Champaign</i> ; Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla, <i>ETH Zurich</i>	

Stable and Practical AS Relationship Inference with ProbLink	581
Yuchen Jin, <i>University of Washington</i> ; Colin Scott, <i>UC Berkeley</i> ; Amogh Dhamdhere, <i>CAIDA</i> ; Vasileios Giotas, <i>Lancaster University</i> ; Arvind Krishnamurthy, <i>University of Washington</i> ; Scott Shenker, <i>UC Berkeley, ICSI</i>	

NetBouncer: Active Device and Link Failure Localization in Data Center Networks	599
Cheng Tan, <i>NYU</i> ; Ze Jin, <i>Cornell University</i> ; Chuanxiong Guo, <i>Bytedance</i> ; Tianrong Zhang, <i>Microsoft</i> ; Haitao Wu, <i>Google</i> ; Karl Deng, Dongming Bi, and Dong Xiang, <i>Microsoft</i>	

Privacy and Security

Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services	615
Frank Wang, <i>MIT CSAIL</i> ; Ronny Ko and James Mickens, <i>Harvard University</i>	

Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs	631
Xiang Wang, Yang Hong, and Harry Chang, <i>Intel</i> ; KyoungSoo Park, <i>KAIST</i> ; Geoff Langdale, <i>branchfree.org</i> ; Jiayu Hu and Heqing Zhu, <i>Intel</i>	

Deniable Upload and Download via Passive Participation	649
David Sommer, Aritra Dhar, Luka Malisa, and Esfandiar Mohammadi, <i>ETH Zurich</i> ; Daniel Ronzani, <i>Ronzani Schlauri Attorneys</i> ; Srdjan Capkun, <i>ETH Zurich</i>	

CAUDIT: Continuous Auditing of SSH Servers To Mitigate Brute-Force Attacks	667
Phuong M. Cao, Yuming Wu, and Subho S. Banerjee, <i>UIUC</i> ; Justin Azoff and Alex Withers, <i>NCSA</i> ; Zbigniew T. Kalbarczyk and Ravishankar K. Iyer, <i>UIUC</i>	

Network Modeling

Dataplane equivalence and its applications	683
Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu, <i>University Politehnica of Bucharest</i>	
Alembic: Automated Model Inference for Stateful Network Functions	699
Soo-Jin Moon, <i>Carnegie Mellon University</i> ; Jeffrey Helt, <i>Princeton University</i> ; Yifei Yuan, <i>Intentionet</i> ; Yves Bieri, <i>ETH Zurich</i> ; Sujata Banerjee, <i>VMware Research</i> ; Vyas Sekar, <i>Carnegie Mellon University</i> ; Wenfei Wu, <i>Tsinghua University</i> ; Mihalis Yannakakis, <i>Columbia University</i> ; Ying Zhang, <i>Facebook, Inc.</i>	

Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations	719
Wei Sun and Lisong Xu, <i>University of Nebraska-Lincoln</i> ; Sebastian Elbaum, <i>University of Virginia</i> ; Di Zhao, <i>University of Nebraska-Lincoln</i>	

Wireless Applications

Scaling Community Cellular Networks with CommunityCellularManager	735
Shaddi Hasan, <i>UC Berkeley</i> ; Mary Claire Barela, <i>University of the Philippines, Diliman</i> ; Matthew Johnson, <i>University of Washington</i> ; Eric Brewer, <i>UC Berkeley</i> ; Kurtis Heimerl, <i>University of Washington</i>	
TrackIO: Tracking First Responders Inside-Out	751
Ashutosh Dhekne, <i>University of Illinois at Urbana-Champaign</i> ; Ayon Chakraborty, Karthikeyan Sundaresan, and Sampath Rangarajan, <i>NEC Labs America, Inc.</i>	
3D Backscatter Localization for Fine-Grained Robotics	765
Zhihong Luo, Qiping Zhang, Yunfei Ma, Manish Singh, and Fadel Adib, <i>MIT Media Lab</i>	
Many-to-Many Beam Alignment in Millimeter Wave Networks	783
Suraj Jog, Jiaming Wang, Junfeng Guan, Thomas Moon, Haitham Hassanieh, and Romit Roy Choudhury, <i>UIUC</i>	

Datacenter RPCs can be General and Fast

Anuj Kalia Michael Kaminsky[†] David G. Andersen
Carnegie Mellon University [†]*Intel Labs*

Abstract

It is commonly believed that datacenter networking software must sacrifice generality to attain high performance. The popularity of specialized distributed systems designed specifically for niche technologies such as RDMA, lossless networks, FPGAs, and programmable switches testifies to this belief. In this paper, we show that such specialization is not necessary. eRPC is a new general-purpose remote procedure call (RPC) library that offers performance comparable to specialized systems, while running on commodity CPUs in traditional datacenter networks based on either lossy Ethernet or lossless fabrics. eRPC performs well in three key metrics: message rate for small messages; bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, congestion, and background request execution. In microbenchmarks, one CPU core can handle up to 10 million small RPCs per second, or send large messages at 75 Gbps. We port a production-grade implementation of Raft state machine replication to eRPC without modifying the core Raft source code. We achieve 5.5 μ s of replication latency on lossy Ethernet, which is faster than or comparable to specialized replication systems that use programmable switches, FPGAs, or RDMA.

1 Introduction

“Using performance to justify placing functions in a low-level subsystem must be done carefully. Sometimes, by examining the problem thoroughly, the same or better performance can be achieved at the high level.”

— End-to-end Arguments in System Design

Squeezing the best performance out of modern, high-speed datacenter networks has meant painstaking specialization that breaks down the abstraction barriers between software and hardware layers. The result has been an explosion of co-designed distributed systems that depend on niche network technologies, including RDMA [18, 25, 26, 38, 50, 51, 58, 64, 66, 69], lossless networks [39, 47], FPGAs [33, 34], and programmable switches [37]. Add to that new distributed protocols with incomplete specifications, the inability to reuse existing software, hacks to enable consistent views of remote memory—and the typical developer is likely to give up and just use kernel-based TCP.

These specialized technologies were deployed with the belief that placing their functionality in the network will yield a

large performance gain. In this paper, we show that a general-purpose RPC library called eRPC can provide state-of-the-art performance on commodity datacenter networks without additional network support. This helps inform the debate about the utility of additional in-network functionality vs purely end-to-end solutions for datacenter applications.

eRPC provides three key performance features: high message rate for small messages; high bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, node failures, congestion control, and long-running background requests. eRPC is *not* an RDMA-based system: it works well with only UDP packets over lossy Ethernet without Priority Flow Control (PFC), although it also supports InfiniBand. Our goal is to allow developers to use eRPC in unmodified systems. We use as test-cases two existing systems: a production-grade implementation of Raft [14, 54] that is used in Intel’s distributed object store [11], and Masstree [49]. We successfully integrate eRPC support with both without sacrificing performance.

The need for eRPC arises because the communication software options available for datacenter networks leave much to be desired. The existing options offer an undesirable trade-off between performance and generality. Low-level interfaces such as DPDK [24] are fast, but lack features required by general applications (e.g., DPDK provides only unreliable packet I/O.) On the other hand, full-fledged networking stacks such as mTCP [35] leave significant performance on the table. Absent networking options that provide both high performance and generality, recent systems often choose to design and implement their own communication layer using low-level interfaces [18, 25, 26, 38, 39, 55, 58, 66].

The goal of our work is to answer the question: Can a general-purpose RPC library provide performance comparable to specialized systems? Our solution is based on two key insights. First, we optimize for the common case, i.e., when messages are small [16, 56], the network is congestion-free, and RPC handlers are short. Handling large messages, congestion, and long-running RPC handlers requires expensive code paths, which eRPC avoids whenever possible. Several eRPC components, including its API, message format, and wire protocol are optimized for the common case. Second, restricting each flow to at most one bandwidth-delay product (BDP) of outstanding data effectively prevents packet loss caused by switch buffer overflow for common traffic patterns. This is because datacenter switch buffers are much larger than the network’s BDP. For example, in our two

layer testbed that resembles real deployments, each switch has 12 MB of dynamic buffer, while the BDP is only 19 kB.

eRPC (*efficient* RPC) is available at <https://github.com/efficient/eRPC>. Our research contributions are:

1. We describe the design and implementation of a high-performance RPC library for datacenter networks. This includes (1) common-case optimizations that improve eRPC’s performance for our target workloads by up to 66%; (2) techniques that enable zero-copy transmission in the presence of retransmissions, node failures, and rate limiting; and (3) a scalable implementation whose NIC memory footprint is independent of the number of nodes in the cluster.
2. We are the first to show experimentally that state-of-the-art networking performance can be achieved without lossless fabrics. We show that eRPC performs well in a 100-node cluster with lossy Ethernet without PFC. Our microbenchmarks on two lossy Ethernet clusters show that eRPC can: (1) provide 2.3 μ s median RPC latency; (2) handle up to 10 million RPCs per second with one core; (3) transfer large messages at 75 Gbps with one core; (4) maintain low switch queueing during incast; and (5) maintain peak performance with 20000 connections per node (two million connections cluster-wide).
3. We show that eRPC can be used as a high-performance drop-in networking library for existing software. Notably, we implement a replicated in-memory key-value store with a production-grade version of Raft [14, 54] without modifying the Raft source code. Our three-way replication latency on lossy Ethernet is 5.5 μ s, which is competitive with existing specialized systems that use programmable switches (NetChain [37]), FPGAs [33], and RDMA (DARE [58]).

2 Background and motivation

We first discuss aspects of modern datacenter networks relevant to eRPC. Next, we discuss limitations of existing networking software that underlie the need for eRPC.

2.1 High-speed datacenter networking

Modern datacenter networks provide tens of Gbps per-port bandwidth and a few microseconds round-trip latency [73, §2.1]. They support polling-based network I/O from userspace, eliminating interrupts and system call overhead from the datapath [28, 29]. eRPC uses userspace networking with polling, as in most prior high-performance networked systems [25, 37, 39, 56].

eRPC works well in commodity, lossy datacenter networks. We found that restricting each flow to one BDP of outstanding data prevents most packet drops even on lossy networks. We discuss these aspects below.

Lossless fabrics. Lossless packet delivery is a link-level feature that prevents congestion-based packet drops. For ex-

ample, PFC for Ethernet prevents a link’s sender from overflowing the receiver’s buffer by using pause frames. Some datacenter operators, including Microsoft, have deployed PFC at scale. This was done primarily to support RDMA, since existing RDMA NICs perform poorly in the presence of packet loss [73, §1]. Lossless fabrics are useful even without RDMA: Some systems that do not use remote CPU bypass leverage losslessness to avoid the complexity and overhead of handling packet loss in software [38, 39, 47].

Unfortunately, PFC comes with a host of problems, including head-of-line blocking, deadlocks due to cyclic buffer dependencies, and complex switch configuration; Mittal et al. [53] discuss these problems in detail. In our experience, datacenter operators are often unwilling to deploy PFC due to these problems. Using simulations, Mittal et al. show that a new RDMA NIC architecture called IRN with improved packet loss handling can work well in lossy networks. Our BDP flow control is inspired by their work; the differences between eRPC’s and IRN’s transport are discussed in Section 5.2.3. Note that, unlike IRN, eRPC is a real system, and it does not require RDMA NIC support.

Switch buffer \gg BDP. The increase in datacenter bandwidth has been accompanied by a corresponding decrease in round-trip time (RTT), resulting in a small BDP. Switch buffers have grown in size, to the point where “shallow-buffered” switches that use SRAM for buffering now provide tens of megabytes of shared buffer. Much of this buffer is dynamic, i.e., it can be dedicated to an incast’s target port, preventing packet drops from buffer overflow. For example, in our two-layer 25 GbE testbed that resembles real datacenters (Table 1), the RTT between two nodes connected to different top-of-rack (ToR) switches is 6 μ s, so the BDP is 19 kB. This is unsurprising: for example, the BDP of the two-tier 10 GbE datacenter used in pFabric is 18 kB [15].

In contrast to the small BDP, the Mellanox Spectrum switches in our cluster have 12 MB in their dynamic buffer pool [13]. Therefore, the switch can ideally tolerate a 640-way incast. The popular Broadcom Trident-II chip used in datacenters at Microsoft and Facebook has a 9 MB dynamic buffer [9, 73]. Zhang et al. [70] have made a similar observation (i.e., buffer \gg BDP) for gigabit Ethernet.

In practice, we wish to support approximately 50-way incasts: congestion control protocols deployed in real datacenters are tested against comparable incast degrees. For example, DCQCN and Timely use up to 20- and 40-way incasts, respectively [52, 73]. This is much smaller than 640, allowing substantial tolerance to technology variations, i.e., we expect the switch buffer to be large enough to prevent most packet drops in datacenters with different BDPs and switch buffer sizes. Nevertheless, it is unlikely that the BDP-to-buffer ratio will grow substantially in the near future: newer 100 GbE switches have even larger buffers (42 MB in Mellanox’s Spectrum-2 and 32 MB in Broadcom’s Trident-III), and NIC-added latency is continuously decreasing. For ex-

ample, we measured InfiniBand’s RTT between nodes under different ToR’s to be only 3.1 μ s, and Ethernet has historically caught up with InfiniBand’s performance.

2.2 Limitations of existing options

Two reasons underlie our choice to design a new general-purpose RPC system for datacenter networks: First, existing datacenter networking software options sacrifice performance or generality, preventing unmodified applications from using the network efficiently. Second, co-designing storage software with the network is increasingly popular, and is largely seen as necessary to achieve maximum performance. However, such specialization has well-known drawbacks, which can be avoided with a general-purpose communication layer that also provides high performance. We describe a representative set of currently available options and their limitations below, roughly in order of increasing performance and decreasing generality.

Fully-general networking stacks such as mTCP [35] and IX [17] allow legacy sockets-based applications to run unmodified. Unfortunately, they leave substantial performance on the table, especially for small messages. For example, one server core can handle around 1.5 million and 10 million 64 B RPC requests per second with IX [17] and eRPC, respectively.

Some recent RPC systems can perform better, but are designed for specific use cases. For example, RAMCloud RPCs [56] are designed for low latency, but not high throughput. In RAMCloud, a single dispatch thread handles all network I/O, and request processing is done by other worker threads. This requires inter-thread communication for every request, and limits the system’s network throughput to one core. FaRM RPCs [25] use RDMA writes over connection-based hardware transports, which limits scalability and prevents use in non-RDMA environments.

Like eRPC, our prior work on FaSST RPCs [39] uses only datagram packet I/O, but requires a lossless fabric. FaSST RPCs do not handle packet loss, large messages, congestion, long-running request handlers, or node failure; researchers have believed that supporting these features in software (instead of NIC hardware) would substantially degrade performance [27]. We show that with careful design, we can support all these features and still match FaSST’s performance, while running on a lossy network. This upends conventional wisdom that losslessness or NIC support is necessary for high performance.

2.3 Drawbacks of specialization

Co-designing distributed systems with network hardware is a well-known technique to improve performance. Co-design with RDMA is popular, with numerous examples from key-value stores [25, 38, 50, 65, 66], state machine replication [58], and transaction processing systems [21, 26, 41, 66]. Programmable switches allow in-network optimizations such as reducing network round trips for distributed pro-

tocols [37, 43, 44], and in-network caching [36]. Co-design with FPGAs is an emerging technique [33].

While there are advantages of co-design, such specialized systems are unfortunately very difficult to design, implement, and deploy. Specialization breaks abstraction boundaries between components, which prevents reuse of components and increases software complexity. Building distributed storage systems requires tremendous programmer effort, and co-design typically mandates starting from scratch, with new data structures, consensus protocols, or transaction protocols. Co-designed systems often cannot reuse existing codebases or protocols, tests, formal specifications, programmer hours, and feature sets. Co-design also imposes deployment challenges beyond needing custom hardware: for example, using programmable switches requires user control over shared network switches, which may not be allowed by datacenter operators; and, RDMA-based systems are unusable with current NICs in datacenters that do not support PFC.

In several cases, specialization does not provide even a performance advantage. Our prior work shows that RPCs outperform RDMA-based designs for applications like key-value stores and distributed transactions, with the same amount of CPU [38, 39]. This is primarily because operations in these systems often require multiple remote memory accesses that can be done with one RPC, but require multiple RDMA. In this paper (§ 7.1), we show that RPCs perform comparably with switch- and FPGA-based systems for replication, too.

3 eRPC overview

We provide an overview of eRPC’s API and threading model below. In these aspects, eRPC is similar to existing high-performance RPC systems like Mellanox’s Accelio [4] and FaRM. eRPC’s threading model differs in how we sometimes run long-running RPC handlers in “worker” threads (§ 3.2).

eRPC implements RPCs on top of a transport layer that provides basic unreliable packet I/O, such as UDP or InfiniBand’s Unreliable Datagram transport. A userspace NIC driver is required for good performance. Our primary contribution is the design and implementation of end-host mechanisms and a network transport (e.g., wire protocol and congestion control) for the commonly-used RPC API.

3.1 RPC API

RPCs execute at most once, and are asynchronous to avoid stalling on network round trips; intra-thread concurrency is provided using an event loop. RPC servers register request handler functions with unique request types; clients use these request types when issuing RPCs, and get continuation callbacks on RPC completion. Users store RPC messages in opaque, DMA-capable buffers provided by eRPC, called msgbufs; a library that provides marshalling and unmarshalling can be used as a layer on top of eRPC.

Each user thread that sends or receives RPCs creates an exclusive Rpc endpoint (a C++ object). Each Rpc endpoint contains an RX and TX queue for packet I/O, an event loop, and several *sessions*. A session is a one-to-one connection between two Rpc endpoints, i.e., two user threads. The client endpoint of a session is used to send requests to the user thread at the other end. A user thread may participate in multiple sessions, possibly playing different roles (i.e., client or server) in different sessions.

User threads act as “dispatch” threads: they must periodically run their Rpc endpoint’s event loop to make progress. The event loop performs the bulk of eRPC’s work, including packet I/O, congestion control, and management functions. It invokes request handlers and continuations, and dispatches long-running request handlers to worker threads (§ 3.2).

Client control flow: `rpc->enqueue_request()` queues a request msgbuf on a session, which is transmitted when the user runs `rpc`’s event loop. On receiving the response, the event loop copies it to the client’s response msgbuf and invokes the continuation callback.

Server control flow: The event loop of the Rpc that owns the server session invokes (or dispatches) a request handler on receiving a request. We allow *nested* RPCs, i.e., the handler need not enqueue a response before returning. It may issue its own RPCs and call `enqueue_response()` for the first request later when all dependencies complete.

3.2 Worker threads

A key design decision for an RPC system is which thread runs an RPC handler. Some RPC systems such as RAMCloud use dispatch threads for only network I/O. RAMCloud’s dispatch threads communicate with *worker* threads that run request handlers. At datacenter network speeds, however, inter-thread communication is expensive: it reduces throughput and adds up to 400 ns to request latency [56]. Other RPC systems such as Accelio and FARM avoid this overhead by running all request handlers directly in dispatch threads [25, 38]. This latter approach suffers from two drawbacks when executing long request handlers: First, such handlers block other dispatch processing, increasing tail latency. Second, they prevent rapid server-to-client congestion feedback, since the server might not send packets while running user code.

Striking a balance, eRPC allows running request handlers in both dispatch threads and worker threads: When registering a request handler, the programmer specifies whether the handler should run in a dispatch thread. This is the only additional user input required in eRPC. In typical use cases, handlers that require up to a few hundred nanoseconds use dispatch threads, and longer handlers use worker threads.

3.3 Evaluation clusters

Table 1 shows the clusters used in this paper. They include two types of networks (lossy Ethernet, and lossless Infini-

Band), and three generations of NICs released between 2011 (CX3) and 2017 (CX5). eRPC works well on all three clusters, showing that our design is robust to NIC and network technology changes. We use traditional UDP on the Ethernet clusters (i.e., we do not use RoCE), and InfiniBand’s Unreliable Datagram transport on the InfiniBand cluster.

Currently, eRPC is primarily optimized for Mellanox NICs. eRPC also works with DPDK-capable NICs that support flow steering. For Mellanox Ethernet NICs, we generate UDP packets directly with `libibverbs` instead of going through DPDK, which internally uses `libibverbs` for these NICs.

Our evaluation primarily uses the large CX4 cluster, which resembles real-world datacenters. The ConnectX-4 NICs used in CX4 are widely deployed in datacenters at Microsoft and Facebook [3, 73], and its Mellanox Spectrum switches perform similarly to Broadcom’s Trident switches used in these datacenters (i.e., both switches provide dynamic buffering, cut-through switching, and less than 500 ns port-to-port latency.) We use 100 nodes out of the 200 nodes in the shared CloudLab cluster. The six switches in the CX4 cluster are organized as five ToRs with 40 25 GbE downlinks and five 100 GbE uplinks, for a 2:1 oversubscription.

4 eRPC design

Achieving eRPC’s performance goals requires careful design and implementation. We discuss three aspects of eRPC’s design in this section: scalability of our networking primitives, the challenges involved in supporting zero-copy, and the design of sessions. The next section discusses eRPC’s wire protocol and congestion control. A recurring theme in eRPC’s design is that we optimize for the common case, i.e., when request handlers run in dispatch threads, RPCs are small, and the network is congestion-free.

4.1 Scalability considerations

We chose plain packet I/O instead of RDMA writes [25, 66, 69] to send messages in eRPC. This decision is based on prior insights from our design of FaSST: First, packet I/O provides completion queues that can scalably detect received packets. Second, RDMA caches connection state in NICs, which does not scale to large clusters. We next discuss *new* observations about NIC hardware trends that support this design.

4.1.1 Packet I/O scales well

RPC systems that use RDMA writes have a *fundamental* scalability limitation. In these systems, clients write requests directly to per-client circular buffers in the server’s memory; the server must poll these buffers to detect new requests. The number of circular buffers grows with the number of clients, limiting scalability.

With traditional userspace packet I/O, the NIC writes an incoming packet’s payload to a buffer specified by a descriptor pre-posted to the NIC’s RX queue (RQ) by the receiver host; the packet is dropped if the RQ is empty. Then, the

Name	Nodes	Network type	Mellanox NIC	Switches	Intel Xeon E5 CPU code
CX3	11	InfiniBand	56 Gbps ConnectX-3	One SX6036	2650 (8 cores)
CX4	100	Lossy Ethernet	25 Gbps ConnectX-4 Lx	5x SN2410, 1x SN2100	2640 v4 (10 cores)
CX5	8	Lossy Ethernet	Dual-port 40 Gbps ConnectX-5	One SX1036	2697 v3 (14 c) or 2683 v4 (16 c)

Table 1: Measurement clusters. CX4 and CX3 are CloudLab [59] and Emulab [68] clusters, respectively.

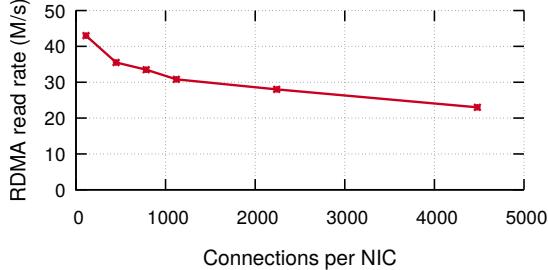


Figure 1: Connection scalability of ConnectX-5 NICs

NIC writes an entry to the host’s RX completion queue. The receiver host can then check for received packets in constant time by examining the head of the completion queue.

To avoid dropping packets due to an empty RQ with no descriptors, RQs must be sized proportionally to the number of independent connected RPC endpoints (§ 4.3.1). Older NICs experience cache thrashing with large RQs, thus limiting scalability, but we find that newer NICs fare better: While a Connect-IB NIC could support only 14 2K-entry RQs before thrashing [39], we find that ConnectX-5 NICs do not thrash even with 28 64K-entry RQs. This improvement is due to more intelligent prefetching and caching of RQ descriptors, instead of a massive 64x increase in NIC cache.

We use features of current NICs (e.g., multi-packet RQ descriptors that identify several contiguous packet buffers) in novel ways to guarantee a *constant* NIC memory footprint per CPU core, i.e., it does not depend on the number of nodes in the cluster. This result can simplify the design of future NICs (e.g., RQ descriptor caching is unneeded), but its current value is limited to performance improvements because current NICs support very large RQs, and are perhaps overly complex as a result. We discuss this in detail in Appendix A.

4.1.2 Scalability limits of RDMA

RDMA requires NIC-managed connection state. This limits scalability because NICs have limited SRAM to cache connection state. The number of in-NIC connections may be reduced by sharing them among CPU cores, but doing so reduces performance by up to 80% [39].

Some researchers have hypothesized that improvements in NIC hardware will allow using connected transports at large scale [27, 69]. To show that this is unlikely, we measure the connection scalability of state-of-the-art ConnectX-5 NICs, released in 2017. We repeat the connection scalability experiment from FaSST, which was used to evaluate the older Connect-IB NICs from 2012. We enable PFC on CX5 for this

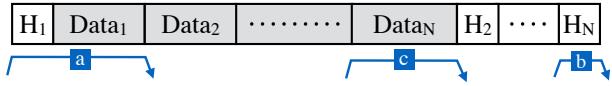


Figure 2: Layout of packet headers and data for an N -packet mbuf. Blue arrows show NIC DMAs; the letters show the order in which the DMAs are performed for packets 1 and N .

experiment since it uses RDMA; PFC is disabled in all experiments that use eRPC. In the experiment, each node creates a tunable number of connections to other nodes and issues 16-byte RDMA reads on randomly-chosen connections. Figure 1 shows that as the number of connections increases, RDMA throughput decreases, losing ≈50% throughput with 5000 connections. This happens because NICs can cache only a few connections, and cache misses require expensive DMA reads [25]. In contrast, eRPC maintains its peak throughput with 20000 connections (§ 6.3).

ConnectX-5’s connection scalability is, surprisingly, not substantially better than Connect-IB despite the five-year advancement. A simple calculation shows why this is hard to improve: In Mellanox’s implementation, each connection requires ≈375 B of in-NIC connection state, and the NICs have ≈2 MB of SRAM to store connection state as well as other data structures and buffers [1]. 5000 connections require 1.8 MB, so cache misses are unavoidable.

NIC vendors have been trying to improve RDMA’s scalability for a decade [22, 42]. Unfortunately, these techniques do not map well to RPC workloads [39]. Vendors have not put more memory in NICs, probably because of cost and power overheads, and market factors. The scalability issue of RDMA is exacerbated by the popularity of *multihost* NICs, which allow sharing a powerful NIC among 2–4 CPUs [3, 7].

eRPC replaces NIC-managed connection state with CPU-managed connection state. This is an explicit design choice, based upon fundamental differences between the CPU and NIC architectures. NICs and CPUs will both cache recently-used connection state. CPU cache misses are served from DRAM, whereas NIC cache misses are served from the CPU’s memory subsystem over the slow PCIe bus. The CPU’s miss penalty is therefore much lower. Second, CPUs have substantially larger caches than the ~2 MB available on a modern NIC, so the cache miss frequency is also lower.

4.2 Challenges in zero-copy transmission

eRPC uses zero-copy packet I/O to provide performance comparable to low-level interfaces such as DPDK and RDMA. This section describes the challenges involved in doing so.

4.2.1 Message buffer layout

eRPC provides DMA-capable message buffers to applications for zero-copy transfers. A msgbuf holds one, possibly multi-packet message. It consists of per-packet headers and data, arranged in a fashion optimized for small single-packet messages (Figure 2). Each eRPC packet has a header that contains the transport header, and eRPC metadata such as the request handler type and sequence numbers. We designed a msgbuf layout that satisfies two requirements.

1. The data region is contiguous to allow its use in applications as an opaque buffer.
2. The first packet’s data and header are contiguous. This allows the NIC to fetch small messages with one DMA read; using multiple DMAs for small messages would substantially increase NIC processing and PCIe use, reducing message rate by up to 20% [40].

For multi-packet messages, headers for subsequent packets are at the end of the message: placing header 2 immediately after the first data packet would violate our first requirement. Non-first packets require two DMAs (header and data); this is reasonable because the overhead for DMA-reading small headers is amortized over the large data DMA.

4.2.2 Message buffer ownership

Since eRPC transfers packets directly from application-owned msgbufs, msgbuf references must never be used by eRPC after msgbuf ownership is returned to the application. In this paper, we discuss msgbuf ownership issues for only clients; the process is similar but simpler for the server, since eRPC’s servers are passive (§ 5). At clients, we must ensure the following invariant: *no eRPC transmission queue contains a reference to the request msgbuf when the response is processed*. Processing the response includes invoking the continuation, which permits the application to reuse the request msgbuf. In eRPC, a request reference may be queued in the NIC’s hardware DMA queue, or in our software rate limiter (§ 5.2).

This invariant is maintained trivially when there are no retransmissions or node failures, since the request must exit all transmission queues before the response is received. The following **example** demonstrates the problem with retransmissions. Consider a client that falsely suspects packet loss and retransmits its request. The server, however, received the first copy of the request, and its response reaches the client before the retransmitted request is dequeued. Before processing the response and invoking the continuation, we must ensure that there are no queued references to the request msgbuf. We discuss our solution for the NIC DMA queue next, and for the rate limiter in Appendix C.

The conventional approach to ensure DMA completion is to use “signaled” packet transmission, in which the NIC writes completion entries to the TX completion queue. Unfortunately, doing so reduces message rates by up to 25% by us-

ing more NIC and PCIe resources [38], so we use unsignaled packet transmission in eRPC.

Our method of ensuring DMA completion with unsignaled transmission is in line with our design philosophy: we choose to make the common case (no retransmission) fast, at the expense of invoking a more-expensive mechanism to handle the rare cases. We flush the TX DMA queue after queueing a retransmitted packet, which blocks until all queued packets are DMA-ed. This ensures the required invariant: when a response is processed, there are no references to the request in the DMA queue. This flush is moderately expensive ($\approx 2\ \mu s$), but it is called during rare retransmission or node failure events, and it allows eRPC to retain the 25% throughput increase from unsignaled transmission.

During server node failures, eRPC invokes continuations with error codes, which also yield request msgbuf ownership. It is possible, although extremely unlikely, that server failure is suspected while a request (not necessarily a retransmission) is in the DMA queue or the rate limiter. Handling node failures requires similar care as discussed above, and is discussed in detail in Appendix B.

4.2.3 Zero-copy request processing

Zero-copy reception is harder than transmission: To provide a contiguous request msgbuf to the request handler at the server, we must strip headers from received packets, and copy only application data to the target msgbuf. However, we were able to provide zero-copy reception for our common-case workload consisting of single-packet requests and dispatch-mode request handlers as follows. eRPC owns the packet buffers DMA-ed by the NIC until it re-adds the descriptors for these packets back to the receive queue (i.e., the NIC cannot modify the packet buffers for this period.) This ownership guarantee allows running dispatch-mode handlers without copying the DMA-ed request packet to a dynamically-allocated msgbuf. Doing so improves eRPC’s message rate by up to 16% (§ 6.2).

4.3 Sessions

Each session maintains multiple outstanding requests to keep the network pipe full. Concurrently requests on a session can complete *out-of-order* with respect to each other. This avoids blocking dispatch-mode RPCs behind a long-running worker-mode RPC. We support a constant number of concurrent requests (default = 8) per session; additional requests are transparently queued by eRPC. This is inspired by how RDMA connections allow a constant number of operations [10]. A session uses an array of *slots* to track RPC metadata for outstanding requests.

Slots in server-mode sessions have an MTU-size preallocated msgbuf for use by request handlers that issue short responses. Using the preallocated msgbuf does not require user input: eRPC chooses it automatically at run time by examining the handler’s desired response size. This opti-

mization avoids the overhead of dynamic memory allocation, and improves eRPC’s message rate by up to 13% (§ 6.2).

4.3.1 Session credits

eRPC limits the number of unacknowledged packets on a session for two reasons. First, to avoid dropping packets due to an empty RQ with no descriptors, the number of packets that may be sent to an Rpc must not exceed the size of its RQ ($|RQ|$). Because each session sends packets independently of others, we first limit the number of sessions that an Rpc can participate in. Each session then uses *session credits* to implement packet-level flow control: we limit the number of packets that a client may send on a session before receiving a reply, allowing the server Rpc to replenish used RQ descriptors before sending more packets.

Second, session credits automatically implement end-to-end flow control, which reduces switch queueing (§ 5.2). Allowing BDP/MTU credits per session ensures that each session can achieve line rate. Mittal et al. [53] have proposed similar flow control for RDMA NICs (§ 5.2.3).

A client session starts with a quota of C packets. Sending a packet to the server consumes a credit, and receiving a packet replenishes a credit. An Rpc can therefore participate in up to $\lfloor RQ \rfloor / C$ sessions, counting both server-mode and client-mode sessions; session creation fails after this limit is reached. We plan to explore statistical multiplexing in the future.

4.3.2 Session scalability

eRPC’s scalability depends on the user’s desired value of C , and the number and size of RQs that the NIC and host can effectively support. Lowering C increases scalability, but reduces session throughput by restricting the session’s packet window. Small values of C (e.g., $C = 1$) should be used in applications that (a) require only low latency and small messages, or (b) whose threads participate in many sessions. Large values (e.g., BDP/MTU) should be used by applications whose sessions individually require high throughput.

Modern NICs can support several very large RQs, so NIC RQ capacity limits scalability only on older NICs. In our evaluation, we show that eRPC can handle 20000 sessions with 32 credits per session on the widely-used ConnectX-4 NICs. However, since each RQ entry requires allocating a packet buffer in host memory, needlessly large RQs waste host memory and should be avoided.

5 Wire protocol

We designed a wire protocol for eRPC that is optimized for small RPCs and accounts for per-session credit limits. For simplicity, we chose a simple *client-driven* protocol, meaning that each packet sent by the server is in response to a client packet. A client-driven protocol has fewer “moving parts” than a protocol in which both the server and client can independently send packets. Only the client maintains

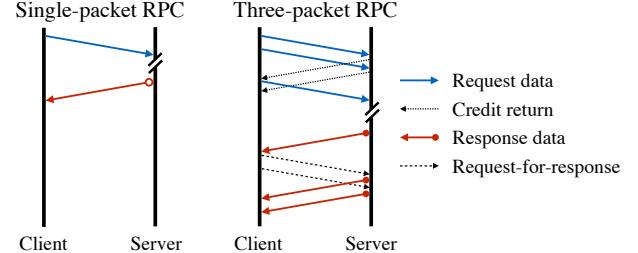


Figure 3: Examples of eRPC’s wire protocol, with 2 credits/session.

wire protocol state that is rolled back during retransmission. This removes the need for client-server coordination before rollback, reducing complexity. A client-driven protocol also shifts the overhead of rate limiting entirely to clients, freeing server CPU that is often more valuable.

5.1 Protocol messages

Figure 3 shows the packets sent with $C = 2$ for a small single-packet RPC, and for an RPC whose request and response require three packets each. Single-packet RPCs use the fewest packets possible. The client begins by sending a window of up to C request data packets. For each request packet except the last, the server sends back an explicit *credit return* (CR) packet; the credit used by the last request packet is implicitly returned by the first response packet.

Since the protocol is client-driven, the server cannot immediately send response packets after the first. Subsequent response packets are triggered by *request-for-response* (RFR) packets that the client sends after receiving the first response packet. This increases the latency of multi-packet responses by up to one RTT. This is a fundamental drawback of client-driven protocols; in practice, we found that the added latency is less than 20% for responses with four or more packets.

CRs and RFRs are tiny 16 B packets, and are sent only for large multi-packet RPCs. The additional overhead of sending these tiny packets is small with userspace networking that our protocol is designed for, so we do not attempt complex optimizations such as cumulative CRs or RFRs. These optimizations may be worthwhile for kernel-based networking stacks, where sending a 16 B packet and an MTU-sized packet often have comparable CPU cost.

5.2 Congestion control

Congestion control for datacenter networks aims to reduce switch queueing, thereby preventing packet drops and reducing RTT. Prior high-performance RPC implementations such as FaSST do not implement congestion control, and some researchers have hypothesized that doing so will substantially reduce performance [27]. Can effective congestion control be implemented efficiently in software? We show that optimizing for uncongested networks, and recent advances in software rate limiting allow congestion control with only 9% overhead (§ 6.2).

5.2.1 Available options

Congestion control for high-speed datacenter networks is an evolving area of research, with two major approaches for commodity hardware: RTT-based approaches such as Timely [52], and ECN-based approaches such as DCQCN [73]. Timely and DCQCN have been deployed at Google and Microsoft, respectively. We wish to use these protocols since they have been shown to work at scale.

Both Timely and DCQCN are rate-based: client use the congestion signals to adjust per-session sending rates. We implement Carousel’s rate limiter [61], which is designed to efficiently handle a large number of sessions. Carousel’s design works well for us as-is, so we omit the details.

eRPC includes the hooks and mechanisms to easily implement either Timely or DCQCN. Unfortunately, we are unable to implement DCQCN because none of our clusters performs ECN marking¹. Timely can be implemented entirely in software, which made it our favored approach. eRPC runs all three Timely components—per-packet RTT measurement, rate computation using the RTT measurements, and rate limiting—at client session endpoints. For Rpc’s that host only server-mode endpoints, there is no overhead due to congestion control.

5.2.2 Common-case optimizations

We use three optimizations for our common-case workloads. Our evaluation shows that these optimizations reduce the overhead of congestion control from 20% to 9%, and that they do not reduce the effectiveness of congestion control. The first two are based on the observation that datacenter networks are typically uncongested. Recent studies of Facebook’s datacenters support this claim: Roy et al. [60] report that 99% of all datacenter links are less than 10% utilized at one-minute timescales. Zhang et al. [71, Fig. 6] report that for Web and Cache traffic, 90% of top-of-rack switch links, which are the most congested switches, are less than 10% utilized at 25 μ s timescales.

When a session is uncongested, RTTs are low and Timely’s computed rate for the session stays at the link’s maximum rate; we refer to such sessions as *uncongested*.

1. **Timely bypass.** If the RTT of a packet received on an uncongested session is smaller than Timely’s low threshold, below which it performs additive increase, we do not perform a rate update. We use the recommended value of 50 μ s for the low threshold [52, 74].
2. **Rate limiter bypass.** For uncongested sessions, we transmit packets directly instead of placing them in the rate limiter.
3. **Batched timestamps for RTT measurement.** Calling `rdtsc()` costs 8 ns on our hardware, which is sub-

¹The Ethernet switch in our private CX5 cluster does not support ECN marking [5, p. 839]; we do not have admin access to the shared CloudLab switches in the public CX4 cluster; and InfiniBand NICs in the CX3 cluster do not relay ECN marks to software.

stantial when processing millions of small packets per second. We reduce timer overhead by sampling it once per RX or TX batch instead of once per packet.

5.2.3 Comparison with IRN

IRN [53] is a new RDMA NIC architecture designed for lossy networks, with two key improvements. First, it uses BDP flow control to limit the outstanding data per RDMA connection to one BDP. Second, it uses efficient selective acks instead of simple go-back-N for packet loss recovery.

IRN was evaluated with simulated switches that have small (60–480 kB) static, per-port buffers. In this buffer-deficient setting, they found SACKs necessary for good performance. However, dynamic-buffer switches are the de-facto standard in current datacenters. As a result, packet losses are very rare with only BDP flow control, so we currently do not implement SACKs, primarily due to engineering complexity. eRPC’s dependence on dynamic switch buffers can be reduced by implementing SACK.

With small per-port switch buffers, IRN’s maximum RTT is a few hundred microseconds, allowing a ~300 μ s retransmission timeout (RTO). However, the 12 MB dynamic buffer in our main CX4 cluster (25 Gbps) can add up to 3.8 ms of queueing delay. Therefore, we use a conservative 5 ms RTO.

5.3 Handling packet loss

For simplicity, eRPC treats reordered packets as losses by dropping them. This is not a major deficiency because datacenter networks typically use ECMP for load balancing, which preserves intra-flow ordering [30, 71, 72] except during rare route churn events. Note that current RDMA NICs also drop reordered packets [53].

On suspecting a lost packet, the client rolls back the request’s wire protocol state using a simple go-back-N mechanism. It then reclaims credits used for the rolled-back transmissions, and retransmits from the updated state. The server never runs the request handler for a request twice, guaranteeing at-most-once RPC semantics.

In case of a false positive, a client may violate the credit agreement by having more packets outstanding to the server than its credit limit. In the extremely rare case that such an erroneous loss detection occurs *and* the server’s RQ is out of descriptors, eRPC will have “induced” a real packet loss. We allow this possibility and handle the induced loss like a real packet loss.

6 Microbenchmarks

eRPC is implemented in 6200 SLOC of C++, excluding tests and benchmarks. We use static polymorphism to create an `Rpc` class that works with multiple transport types without the overhead of virtual function calls. In this section, we evaluate eRPC’s latency, message rate, scalability, and bandwidth using microbenchmarks. To understand eRPC’s performance in commodity datacenters, we primarily use the large CX4

Cluster	CX3 (InfiniBand)	CX4 (Eth)	CX5 (Eth)
RDMA read	1.7 μ s	2.9 μ s	2.0 μ s
eRPC	2.1 μ s	3.7 μ s	2.3 μ s

Table 2: Comparison of median latency with eRPC and RDMA

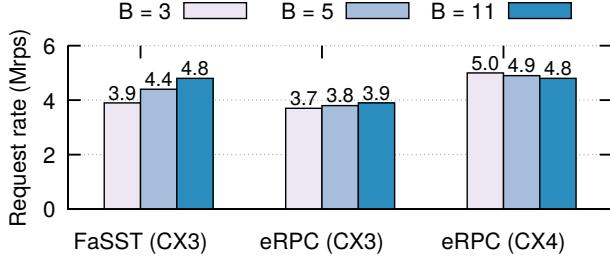


Figure 4: Single-core small-RPC rate with B requests per batch

cluster. We use CX5 and CX3 for their more powerful NICs and low-latency InfiniBand, respectively. eRPC’s congestion control is enabled by default.

6.1 Small RPC latency

How much latency does eRPC add? Table 2 compares the median latency of 32 B RPCs and RDMA reads between two nodes connected to the same ToR switch. Across all clusters, eRPC is at most 800 ns slower than RDMA reads.

eRPC’s median latency on CX5 is only 2.3 μ s, showing that latency with commodity Ethernet NICs and software networking is much lower than the widely-believed value of 10–100 μ s [37, 57]. CX5’s switch adds 300 ns to every layer-3 packet [12], meaning that end-host networking adds only \approx 850 ns each at the client and server. This is comparable to switch-added latency. We discuss this further in § 7.1.

6.2 Small RPC rate

What is the CPU cost of providing generality in an RPC system? We compare eRPC’s small message performance against FaSST RPCs, which outperform other RPC systems such as FaRM [39]. FaSST RPCs are *specialized* for single-packet RPCs in a lossless network, and they do not handle congestion.

We mimic FaSST’s experiment setting: one thread per node in an 11-node cluster, each of which acts both as RPC server and client. Each thread issues batches of B requests, keeping multiple request batches in flight to hide network latency. Each request in a batch is sent to a randomly-chosen remote thread. Such batching is common in key-value stores and distributed online transaction processing. Each thread keeps up to 60 requests in flight, spread across all sessions. RPCs are 32 B in size. We compare eRPC’s performance on CX3 (InfiniBand) against FaSST’s reported numbers on the same cluster. We also present eRPC’s performance on the CX4 Ethernet cluster. We omit CX5 since it has only 8 nodes.

Figure 4 shows that eRPC’s per-thread request issue rate is at most 18% lower than FaSST across all batch sizes, and

Action	RPC rate	% loss
Baseline (with congestion control)	4.96 M/s	–
Disable batched RTT timestamps (§5.2)	4.84 M/s	2.4%
Disable Timely bypass (§5.2)	4.52 M/s	6.6%
Disable rate limiter bypass (§5.2)	4.30 M/s	4.8%
Disable multi-packet RQ (§4.1.1)	4.06 M/s	5.6%
Disable preallocated responses (§4.3)	3.55 M/s	12.6%
Disable 0-copy request processing (§4.2.3)	3.05 M/s	14.0%

Table 3: Impact of disabling optimizations on small RPC rate (CX4)

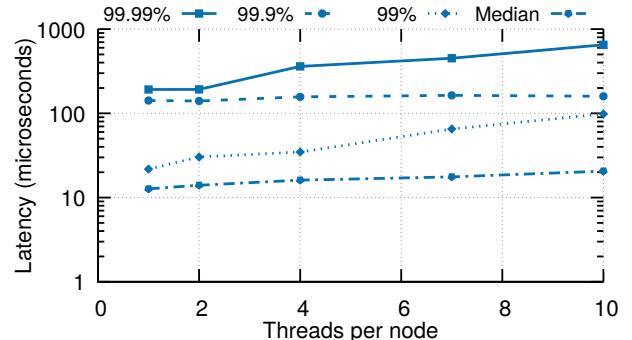


Figure 5: Latency with increasing threads on 100 CX4 nodes

only 5% lower for $B = 3$. This performance drop is acceptable since eRPC is a full-fledged RPC system, whereas FaSST is highly specialized. On CX4, each thread issues 5 million requests per second (Mrps) for $B = 3$; due to the experiment’s symmetry, it simultaneously also handles incoming requests from remote threads at 5 Mrps. Therefore, each thread processes 10 million RPCs per second.

Disabling congestion control increases eRPC’s request rate on CX4 ($B = 3$) from 4.96 Mrps to 5.44 Mrps. This shows that the overhead of our optimized congestion control is only 9%.

Factor analysis. How important are eRPC’s common-case optimizations? Table 3 shows the performance impact of *disabling* some of eRPC’s common-case optimizations on CX4; other optimizations such as our single-DMA msgbuf format and unsignaled transmissions cannot be disabled easily. For our baseline, we use $B = 3$ and enable congestion control. Disabling all three congestion control optimizations (§ 5.2.2) reduces throughput to 4.3 Mrps, increasing the overhead of congestion control from 9% to 20%. Further disabling preallocated responses and zero-copy request processing reduces throughput to 3 Mrps, which is 40% lower than eRPC’s peak throughput. *We therefore conclude that optimizing for the common case is both necessary and sufficient for high-performance RPCs.*

6.3 Session scalability

We evaluate eRPC’s scalability on CX4 by increasing the number of nodes in the previous experiment ($B = 3$) to 100. The five ToR switches in CX4 were assigned between 14 and

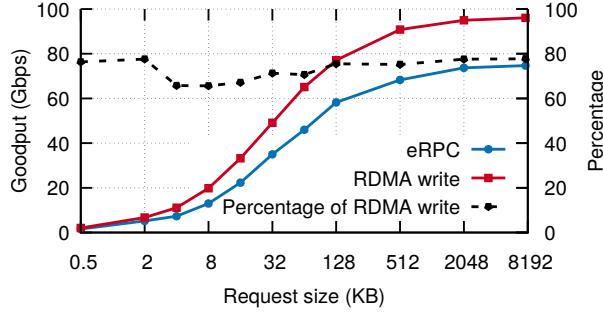


Figure 6: Throughput of large transfers over 100 Gbps InfiniBand

27 nodes each by CloudLab. Next, we increase the number of threads per node: With T threads per node, there are $100T$ threads in the cluster; each thread creates a client-mode session to $100T - 1$ threads. Therefore, each node hosts $T * (100T - 1)$ client-mode sessions, and an equal number of server-mode sessions. Since CX4 nodes have 10 cores, each node handles up to 19980 sessions. This is a challenging traffic pattern that resembles distributed online transaction processing (OLTP) workloads, which operate on small data items [26, 39, 66, 69].

With 10 threads/node, each node achieves 12.3 Mrps on average. At 12.3 Mrps, each node sends and receives 24.6 million packets per second (packet size = 92 B), corresponding to 18.1 Gbps. This is close to the link’s achievable bandwidth (23 Gbps out of 25 Gbps), but is somewhat smaller because of oversubscription. We observe retransmissions with more than two threads per node, but the retransmission rate stays below 1700 packets per second per node.

Figure 5 shows the RPC latency statistics. The median latency with one thread per node is 12.7 μ s. This is higher than the 3.7 μ s for CX4 in Table 2 because most RPCs now go across multiple switches, and each thread keeps 60 RPCs in flight, which adds processing delay. Even with 10 threads per node, eRPC’s 99.99th percentile latency stays below 700 μ s.

These results show that eRPC can achieve high message rate, bandwidth, and scalability, and low latency in a large cluster with lossy Ethernet. Distributed OLTP has been a key application for lossless RDMA fabrics; our results show that it can also perform well on lossy Ethernet.

6.4 Large RPC bandwidth

We evaluate eRPC’s bandwidth using a client thread that sends large messages to a remote server thread. The client sends R -byte requests and keeps one request outstanding; the server replies with a small 32 B response. We use up to 8 MB requests, which is the largest message size supported by eRPC. We use 32 credits per session. To understand how eRPC performs relative to hardware limits, we compare against R -byte RDMA writes, measured using perf-test.

On the clusters in Table 1, eRPC gets bottlenecked by network bandwidth in this experiment setup. To understand eRPC’s performance limits, we connect two nodes in the

Loss rate	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}
Bandwidth (Gbps)	73	71	57	18	2.5

Table 4: eRPC’s 8 MB request throughput with packet loss

Incast degree	Total bw	50% RTT	99% RTT
20	21.8 Gbps	39 μ s	67 μ s
20 (no cc)	23.1 Gbps	202 μ s	204 μ s
50	18.4 Gbps	34 μ s	174 μ s
50 (no cc)	23.0 Gbps	524 μ s	524 μ s
100	22.8 Gbps	349 μ s	969 μ s
100 (no cc)	23.0 Gbps	1056 μ s	1060 μ s

Table 5: Effectiveness of congestion control (cc) during incast

CX5 cluster to a 100 Gbps switch via ConnectX-5 InfiniBand NICs. (CX5 is used as a 40 GbE cluster in the rest of this paper.) Figure 6 shows that eRPC achieves up to 75 Gbps with one core. eRPC’s throughput is at least 70% of RDMA write throughput for 32 kB or larger requests.

In the future, eRPC’s bandwidth can be improved by freeing-up CPU cycles. First, on-die memory copy accelerators can speed up copying data from RX ring buffers to request or response msgbufs [2, 28]. Commenting out the memory copies at the server increases eRPC’s bandwidth to 92 Gbps, showing that copying has substantial overhead. Second, cumulative credit return and request-for-response (§ 5.1) can reduce packet processing overhead.

Table 4 shows the throughput with $R = 8$ MB (the largest size supported by eRPC), and varying, artificially-injected packet loss rates. With the current 5 ms RTO, eRPC is usable while the loss probability is up to .01%, beyond which throughput degrades rapidly. We believe that this is sufficient to handle packet corruptions. RDMA NICs can handle a somewhat higher loss rate (.1%) [73].

6.5 Effectiveness of congestion control

We evaluate if our congestion control is successful at reducing switch queueing. We create an incast traffic pattern by increasing the number of client nodes in the previous setup ($R = 8$ MB). The one server node acts as the incast victim. During an incast, queuing primarily happens at the victim’s ToR switch. We use per-packet RTTs measured at the clients as a proxy for switch queue length [52].

Table 5 shows the total bandwidth achieved by all flows and per-packet RTT statistics on CX4, for 20, 50, and 100-way incasts (one flow per client node). We use two configurations: first with eRPC’s optimized congestion control, and second with no congestion control. Disabling our common-case congestion control optimizations does not substantially affect the RTT statistics, indicating that these optimizations do not reduce the quality of congestion control.

Congestion control successfully handles our target workloads of up to 50-way incasts, reducing median and 99th percentile queuing by over 5x and 3x, respectively. For 100-way incasts, our implementation reduces median queueing by 3x, but fails to substantially reduce 99th percentile queueing. This is in line with Zhu et al. [74, § 4.3]’s analysis, which shows that Timely-like protocols work well with up to approximately 40 incast flows.

The combined incast throughput with congestion control is within 20% of the achievable 23 Gbps. We believe that this small gap can be further reduced with better tuning of Timely’s many parameters. Note that we can also support ECN-based congestion control in eRPC, which may be a better congestion indicator than RTT [74].

Incast with background traffic. Next, we augment the setup above to mimic an experiment from Timely [52, Fig 22]: we create one additional thread at each node that is not the incast victim. These threads exchange latency-sensitive RPCs (64 kB request and response), keeping one RPC outstanding. During a 100-way incast, the 99th percentile latency of these RPCs is 274 μ s. This is similar to Timely’s latency (\approx 200-300 μ s) with a 40-way incast over a 20 GbE lossless RDMA fabric. Although the two results cannot be directly compared, this experiment shows that the latency achievable with software-only networking in commodity, lossy datacenters is comparable to lossless RDMA fabrics, even with challenging traffic patterns.

7 Full-system benchmarks

In this section, we evaluate whether eRPC can be used in real applications with unmodified existing storage software: We build a state machine replication system using an open-source implementation of Raft [54], and a networked ordered key-value store using Masstree [49].

7.1 Raft over eRPC

State machine replication (SMR) is used to build fault-tolerant services. An SMR service consists of a group of server nodes that receive commands from clients. SMR protocols ensure that each server executes the same sequence of commands, and that the service remains available if servers fail. Raft [54] is such a protocol that takes a *leader-based* approach: Absent failures, the Raft replicas have a stable leader to which clients send commands; if the leader fails, the remaining Raft servers elect a new one. The leader appends the command to replicas’ logs, and it replies to the client after receiving acks from a majority of replicas.

SMR is difficult to design and implement correctly [31]: the protocol must have a specification and a proof (e.g., in TLA+), and the implementation must adhere to the specification. We avoid this difficulty by using an existing implementation of Raft [14]. (It had no distinct name, so we term it LibRaft.) We did not write LibRaft ourselves; we found it on GitHub

Measurement	System	Median	99%
Measured at client	NetChain	9.7 μ s	N/A
	eRPC	5.5 μ s	6.3 μ s
Measured at leader	ZabFPGA	3.0 μ s	3.0 μ s
	eRPC	3.1 μ s	3.4 μ s

Table 6: Latency comparison for replicated PUTs

and used it as-is. LibRaft is well-tested with fuzzing over a network simulator and 150+ unit tests. Its only requirement is that the user provide callbacks for sending and handling RPCs—which we implement using eRPC. Porting to eRPC required no changes to LibRaft’s code.

We compare against recent consistent replication systems that are built from scratch for two specialized hardware types. First, NetChain [37] implements chain replication over programmable switches. Other replication protocols such as conventional primary-backup and Raft are too complex to implement over programmable switches [37]. Therefore, despite the protocol-level differences between LibRaft-over-eRPC and NetChain, our comparison helps understand the relative performance of end-to-end CPU-based designs and switch-based designs for in-memory replication. Second, Consensus in a Box [33] (called ZabFPGA here), implements ZooKeeper’s atomic broadcast protocol [32] on FPGAs. eRPC also outperforms DARE [58], which implements SMR over RDMA; we omit the results for brevity.

Workloads. We mimic NetChain and ZabFPGA’s experiment setups for latency measurement: we implement a 3-way replicated in-memory key-value store, and use one client to issue PUT requests. The replicas’ command logs and key-value store are stored in DRAM. NetChain and ZabFPGA use 16 B keys, and 16–64 B values; we use 16 B keys and 64 B values. The client chooses PUT keys uniformly at random from one million keys. While NetChain and ZabFPGA also implement their key-value stores from scratch, we reuse existing code from MICA [45]. We compare eRPC’s performance on CX5 against their published numbers because we do not have the hardware to run NetChain or ZabFPGA. Table 6 compares the latencies of the three systems.

7.1.1 Comparison with NetChain

NetChain’s key assumption is that software networking adds 1–2 orders of magnitude more latency than switches [37]. However, we have shown that eRPC adds 850 ns, which is only around 2x higher than latency added by current programmable switches (400 ns [8]).

Raft’s latency over eRPC is 5.5 μ s, which is substantially lower than NetChain’s 9.7 μ s. This result must be taken with a grain of salt: On the one hand, NetChain uses NICs that have higher latency than CX5’s NICs. On the other hand, it has numerous limitations, including key-value size and capacity constraints, serial chain replication whose latency increases linearly with the number of replicas, absence of

congestion control, and reliance on a complex and external failure detector. The main takeaway is that microsecond-scale consistent replication is achievable in commodity Ethernet datacenters with a general-purpose networking library.

7.1.2 Comparison with ZabFPGA

Although ZabFPGA’s SMR servers are FPGAs, the clients are commodity workstations that communicate with the FPGAs over slow kernel-based TCP. For a challenging comparison, we compare against ZabFPGA’s commit latency measured at the leader, which involves only FPGAs. In addition, we consider its “direct connect” mode, where FPGAs communicate over point-to-point links (i.e., without a switch) via a custom protocol. Even so, eRPC’s median leader commit latency is only 3% worse.

An advantage of specialized, dedicated hardware is low jitter. This is highlighted by ZabFPGA’s negligible leader latency variance. This advantage does not carry over directly to end-to-end latency [33] because storage systems built with specialized hardware are eventually accessed by clients running on commodity workstations.

7.2 Masstree over eRPC

Masstree [49] is an ordered in-memory key-value store. We use it to implement a single-node database index that supports low-latency point queries in the presence of less performance-critical longer-running scans. This requires running scans in worker threads. We use CX3 for this experiment to show that eRPC works well on InfiniBand.

We populate a Masstree server on CX3 with one million random 8B keys mapped to 8B values. The server has 16 Hyper-Threads, which we divide between 14 dispatch threads and 2 worker threads. We run 64 client threads spread over 8 client nodes to generate the workload. The workload consists of 99% GET(key) requests that fetch a key-value item, and 1% SCAN(key) requests that sum up the values of 128 keys succeeding the key. Keys are chosen uniformly at random from the inserted keys. Two outstanding requests per client was sufficient to saturate our server.

We achieve 14.3 million GETs/s on CX3, with 12 μ s 99th percentile GET latency. If the server is configured to run only dispatch threads, the 99th percentile GET latency rises to 26 μ s. eRPC’s median GET latency under low load is 2.7 μ s. This is around 10x faster than Cell’s single-node B-Tree that uses multiple RDMA reads [51]. Despite Cell’s larger key-/value sizes (64 B/256 B), the latency differences are mostly from RTTs: At 40 Gbps, an additional 248 B takes only 50 ns more time to transmit.

8 Related work

RPCs. There is a vast amount of literature on RPCs. The practice of optimizing an RPC wire protocol for small RPCs originates with Birrell and Nelson [19], who introduce the idea of an implicit-ACK. Similar to eRPC, the Sprite RPC

system [67] directly uses raw datagrams and performs re-transmissions only at clients. The Direct Access File System [23] was one of the first to use RDMA in RPCs. It uses SEND/RECV messaging over a connected transport to initiate an RPC, and RDMA reads or writes to transfer the bulk of large RPC messages. This design is widely used in other systems such as NFS’s RPCs [20] and some MPI implementations [48]. In eRPC, we chose to transfer all data over datagram messaging to avoid the scalability limits of RDMA. Other RPC systems that use RDMA include Mellanox’s Accelio [4] and RFP [63]. These systems perform comparably to FARM’s RPCs, which are slower than eRPC at scale by an order of magnitude.

Co-design. There is a rapidly-growing list of projects that co-design distributed systems with the network. This includes key-value stores [38, 46, 50, 65], distributed databases and transaction processing systems [21, 25, 66, 69], state machine replication [33, 58], and graph-processing systems [62]. We believe the availability of eRPC will motivate researchers to investigate how much performance these systems can achieve without sacrificing the networking abstraction. On the other hand, there is a smaller set of recent projects that also prefer RPCs over co-design, including RAMCloud, FaSST, and the distributed data shuffler by Liu et al. [47]. However, their RPCs lack either performance (RAMCloud) or generality (FaSST), whereas eRPC provides both.

9 Conclusion

eRPC is a fast, general-purpose RPC system that provides an attractive alternative to putting more functions in network hardware, and specialized system designs that depend on these functions. eRPC’s speed comes from prioritizing common-case performance, carefully combining a wide range of old and new optimizations, and the observation that switch buffer capacity far exceeds datacenter BDP. eRPC delivers performance that was until now believed possible only with lossless RDMA fabrics or specialized network hardware. It allows unmodified applications to perform close to the hardware limits. Our ported versions of LibRaft and Masstree are, to our knowledge, the fastest replicated key-value store and networked database index in the academic literature, while operating end-to-end without additional network support.

Acknowledgments We received valuable insights from our SIGCOMM and NSDI reviewers, Miguel Castro, John Ousterhout, and Yibo Zhu. Sol Boucher, Jack Kosaian, and Hyeontaek Lim helped improve the writing. CloudLab [59] and Emulab [68] resources were used in our experiments. This work was supported by funding from the National Science Foundation under awards CCF-1535821 and CNS-1700521, and by Intel via the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS). Anuj Kalia is supported by the Facebook Fellowship.

Appendix A. eRPC’s NIC memory footprint

Primarily, four on-NIC structures contribute to eRPC’s NIC memory footprint: the TX and RX queues, and their corresponding completion queues. The TX queue must allow sufficient pipelining to hide PCIe latency; we found that 64 entries are sufficient in all cases. eRPC’s TX queue and TX completion queue have 64 entries by default, so their footprint does not depend on cluster size. The footprint of on-NIC page table entries required for eRPC is negligible because we use 2 MB hugepages [25].

As discussed in Section 4.3.1, eRPC’s RQs must have sufficient descriptors for all connected sessions. If traditional RQs are used, their footprint grows with the number of connected sessions supported. Modern NICs (e.g., ConnectX-4 and newer NICs from Mellanox) support *multi-packet* RQ descriptors that specify multiple contiguous packet buffers using base address, buffer size, and number of buffers. With eRPC’s default configuration of 512-way RQ descriptors, RQ size is reduced by 512x, making it negligible. This optimization has the added advantage of almost eliminating RX descriptor DMA, which is now needed only once every 512 packets. While multi-packet RQs were originally designed for large receive offload of one message [6], we use this feature to receive packets of independent messages.

What about the RX completion queue (CQ)? By default, NICs expect the RX CQ to have sufficient space for each received packet, so using multi-packet RQ descriptors does not reduce CQ size. However, eRPC does not need the information that the NIC DMA-writes to the RX CQ entries. It needs only the number of new packets received. Therefore, we shrink the CQ by allowing it to *overrun*, i.e., we allow the NIC to overwrite existing entries in the CQ in a round-robin fashion. We poll the overrunning CQ to check for received packets. It is possible to use a RX CQ with only one entry, but we found that doing so causes cache line contention between eRPC’s threads and the CPU’s on-die PCIe controller. We solve this issue by using 8-entry CQs, which makes the contention negligible.

Appendix B. Handling node failures

eRPC launches a session management thread that handles sockets-based management messaging for creating and destroying sessions, and detects failure of remote nodes with timeouts. When the management thread suspects a remote node failure, each dispatch thread with sessions to the remote node acts as follows. First, it flushes the TX DMA queue to release msgbuf references held by the NIC. For client sessions, it waits for the rate limiter to transmit any queued packets for the session, and then invokes continuations for pending requests with an error code. For server-mode sessions, it frees session resources after waiting (non-blocking) for request handlers that have not enqueued a response.

Appendix C. Rate limiting with zero-copy

Recall the request retransmission example discussed in § 4.2.2: On receiving the response for the first copy of a retransmitted request, we wish to ensure that the rate limiter does not contain a reference to the retransmitted copy. Unlike eRPC’s NIC DMA queue that holds only a few tens of packets, the rate limiter tracks up to milliseconds worth of transmissions during congestion. As a result, flushing it like the DMA queue is too slow. Deleting references from the rate limiter turned out to be too complex: Carousel requires a bounded difference between the current time and a packet’s scheduled transmission time for correctness, so deletions require rolling back Timely’s internal rate computation state. Each Timely instance is shared by all slots in a session (§ 4.3), which complicates rollback.

We solve this problem by dropping response packets received while a retransmitted request is in the rate limiter. Each such response indicates a false positive in our retransmission mechanism, so they are rare. This solution does not work for the NIC DMA queue: since we use unsigned transmission, it is generally impossible for software to know whether a request is in the DMA queue without flushing it.

References

- [1] Private communication with Mellanox.
- [2] Fast memcpy with SPDK and Intel I/OAT DMA Engine. <https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine>.
- [3] A peek inside Facebook’s server fleet upgrade. <https://www.nextplatform.com/2017/03/13/peek-inside-facebooks-server-fleet-upgrade/>, 2017.
- [4] Mellanox Accelio. <http://www.accelio.org>, 2017.
- [5] Mellanox MLNX-OS user manual for Ethernet. http://www.mellanox.com/related-docs/prod_management_software/MLNX-OS_ETH_v3_6_3508_UM.pdf, 2017.
- [6] Mellanox OFED for Linux release notes. http://www.mellanox.com/related-docs/prod-software/Mellanox_OFED_Linux_Release_Notes_3_2-1_0_1_1.pdf, 2017.
- [7] Oak Ridge leadership computing facility - Summit. <https://www.olcf.ornl.gov/summit/>, 2017.
- [8] Aurora 710 based on Barefoot Tofino switching silicon. <https://netbergtw.com/products/aurora-710/>, 2018.
- [9] Facebook open switching system FBOSS and Wedge in the open. <https://code.facebook.com/posts/843620439027582/facebook-open-switching-system-fboss-and-wedge-in-the-open/>, 2018.

- [10] RDMAmojo - blog on RDMA technology and programming by Dotan Barak. http://www.rdmamojo.com/2013/01/12/ibv_modify_qp/, 2018.
- [11] Distributed asynchronous object storage stack. <https://github.com/daos-stack>, 2018.
- [12] Tolly report: Mellanox SX1016 and SX1036 10/40GbE switches. http://www.mellanox.com/related-docs/prod_eth_switches/Tolly212113MellanoxSwitchSXPerformance.pdf, 2018.
- [13] Jim Warner's switch buffer page. <https://people.ucsc.edu/~warner/buffer.html>, 2018.
- [14] C implementation of the Raft consensus protocol. <https://github.com/willemt/raft>, 2018.
- [15] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, Hong Kong, China, Aug. 2013.
- [16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.
- [17] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected data-plane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.
- [18] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. In *Proc. VLDB*, New Delhi, India, Aug. 2016.
- [19] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 1984.
- [20] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, 2003.
- [21] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.
- [22] D. Crupnicoff, M. Kagan, A. Shahar, N. Bloch, and H. Chapman. Dynamically-connected transport service, May 19 2011. URL <https://www.google.com/patents/US20110116512>. US Patent App. 12/621,523.
- [23] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The direct access file system. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, 2003.
- [24] DPDK. Data Plane Development Kit (DPDK). <http://dpdk.org/>, 2017.
- [25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.
- [26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.
- [27] A. Dragojevic, D. Narayanan, and M. Castro. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 2017.
- [28] M. D. et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.
- [29] D. Firestone et al. Azure accelerated networking: Smart-NICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, Apr. 2018.
- [30] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.
- [31] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.
- [32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.
- [33] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proc. 13th USENIX NSDI*, Santa Clara, CA, May 2016.
- [34] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. Aug. 2017.
- [35] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.
- [36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.
- [37] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.
- [38] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM*

- SIGCOMM*, Chicago, IL, Aug. 2014.
- [39] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.
- [40] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high-performance RDMA systems. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.
- [41] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. HyperLoop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.
- [42] M. J. Koop, J. K. Sridhar, and D. K. Panda. Scalable MPI design over InfiniBand using eXtended Reliable Connection. In *2008 IEEE International Conference on Cluster Computing*, 2008.
- [43] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say no to Paxos overhead: Replacing consensus with network ordering. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.
- [44] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.
- [45] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ISCA*, 2015.
- [46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.
- [47] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proc. 12th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2017.
- [48] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.
- [49] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.
- [50] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference*, San Jose, CA, June 2013.
- [51] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the Cell distributed B-Tree store. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.
- [52] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zets. TIMELY: RTT-based congestion control for the datacenter. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.
- [53] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting network support for RDMA. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.
- [54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, Philadelphia, PA, June 2014.
- [55] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.
- [56] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM TOCS*, 2015.
- [57] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.
- [58] M. Poke and T. Hoefler. DARE: High-performance state machine replication on RDMA networks. In *HPDC*, 2015.
- [59] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login;*, 2014.
- [60] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.
- [61] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proc. ACM SIGCOMM*, Los Angeles, CA, Aug. 2017.
- [62] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.
- [63] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proc. 12th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2017.
- [64] Y. Wang, X. Meng, L. Zhang, and J. Tan. C-hint: An effective and reliable cache management for RDMA-accelerated key-value stores. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Seattle, WA, Nov.

2014.

- [65] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: A resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.
- [66] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.
- [67] B. B. Welch. The Sprite remote procedure call system. Technical report, Berkeley, CA, USA, 1986.
- [68] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.
- [69] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. In *Proc. VLDB*, Munich, Germany, Aug. 2017.
- [70] J. Zhang, F. Ren, X. Yue, R. Shu, and C. Lin. Sharing bandwidth by allocating switch buffer in data center networks. *IEEE Journal on Selected Areas in Communications*, 2014.
- [71] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC ’17, 2017.
- [72] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proc. 9th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.
- [73] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.
- [74] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye. ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY. In *Proc. CoNEXT*, Dec. 2016.

Eiffel: Efficient and Flexible Software Packet Scheduling

Ahmed Saeed[†], Yimeng Zhao[†], Nandita Dukkipati*, Mostafa Ammar[†], Ellen Zegura[†],

Khaled Harras[‡], Amin Vahdat^{*}

[†]*Georgia Institute of Technology*, ^{*}*Google*, [‡]*Carnegie Mellon University*

Abstract

Packet scheduling determines the ordering of packets in a queuing data structure with respect to some ranking function that is mandated by a scheduling policy. It is the core component in many recent innovations to optimize network performance and utilization. Our focus in this paper is on the design and deployment of packet scheduling in software. Software schedulers have several advantages over hardware including shorter development cycle and flexibility in functionality and deployment location. We substantially improve current software packet scheduling performance, while maintaining flexibility, by exploiting underlying features of packet ranking; namely, packet ranks are integers and, at any point in time, fall within a limited range of values. We introduce Eiffel, a novel programmable packet scheduling system. At the core of Eiffel is an integer priority queue based on the Find First Set (FFS) instruction and designed to support a wide range of policies and ranking functions efficiently. As an even more efficient alternative, we also propose a new approximate priority queue that can outperform FFS-based queues for some scenarios. To support flexibility, Eiffel introduces novel programming abstractions to express scheduling policies that cannot be captured by current, state-of-the-art scheduler programming models. We evaluate Eiffel in a variety of settings and in both kernel and userspace deployments. We show that it outperforms state of the art systems by 3-40x in terms of either number of cores utilized for network processing or number of flows given fixed processing capacity.

1 Introduction

Packet scheduling is the core component in many recent innovations to optimize network performance and utilization. Typically, packet scheduling targets network-wide objectives (e.g., meeting strict deadlines of flows [34], reducing flow completion time [14]), or provides isolation and differentiation of service (e.g., through bandwidth allocation [40, 35] or Type of Service levels [44, 15, 32]). It is also used for resource allocation within the packet processing system (e.g., fair CPU utilization in middleboxes [56, 30] and software switches [33]).

Packet scheduling determines the ordering of packets in a *queuing data structure* with respect to some *ranking function* that is mandated by a *scheduling policy*. In particular, as

packets arrive at the scheduler they are *enqueued*, a process that involves ranking based on the scheduling policy and ordering the packets according to the rank. Then, periodically, packets are *dequeued* according to the packet ordering. In general, the dequeuing of a packet might, for some scheduling policies, prompt recalculation of ranks and a reordering of the remaining packets in the queue. A packet scheduler should be *efficient* by performing a minimal number of operations on packet enqueue and dequeue thus enabling the handling of packets at high rates. It should also be *flexible* by providing the necessary abstractions to implement as many scheduling policies as possible.

In modern networks, hardware and software both play an important role [23]. While hardware implementation of network functionality will always be faster than its corresponding software implementation, software schedulers have several advantages. First, the short development cycle and flexibility of software makes it an attractive replacement or precursor for hardware schedulers. Second, the number of rate limiters and queues deployed in hardware implementations typically lags behind network needs. For instance, three years ago, network needs were estimated to be in the tens of thousands of rate limiters [46] while hardware network cards offered 10-128 queues [4]. Third, software packet schedulers can be deployed in multiple platforms and locations, including middleboxes as Virtual Network Functions and end hosts (e.g., implementation based on BESS [33], or OpenVSwitch [45]). Hence, we assert that software solutions will always be needed to replace or augment hardware schedulers [19, 36, 47, 22, 39]. However, as will be discussed in Section 2, current software schedulers do not meet our efficiency and flexibility objectives.

Our focus in this paper is on the design and implementation of efficient and flexible packet scheduling in software. The need for programmable schedulers is rising as more sophisticated policies are required of networks [27, 50] with schedulers deployed at multiple points on a packet’s path. It has proven difficult to achieve scheduler efficiency in software schedulers, especially handling packets at high line rates, without limiting the supported scheduling policies [47, 50, 36, 47, 19, 22]. Furthermore, CPU-efficient implementation of even the simplest scheduling policies is still an open problem for most platforms. For instance, kernel packet pacing can cost CPU utilization of up to 10% [47] and up to 12% for hierarchical weighted fair queuing scheduling in

NetIOC of VMware’s hypervisor [37]. This overhead will only grow as more programmability is added to the scheduler, assuming basic building blocks remain the same (e.g., OpenQueue [39]). The inefficiency of these systems stems from relying on $O(\log n)$ comparison-based priority queues.

At a fundamental level, a scheduling policy that has m ranking functions associated with a packet (e.g., pacing rate, policy-based rate limit, weight-based share, and deadline-based ordering) typically requires m priority queues in which this packet needs to be enqueued and dequeued [49], which translates roughly to $O(m \log n)$ operations per packet for a scheduler with n packets enqueued. We show how to reduce this overhead to $O(m)$ for any scheduling policy (i.e., constant overhead per ranking function).

Our approach to providing both flexibility and efficiency in software packet schedulers is two fold. First, we observe (§2) that packet ranks can be represented as integers that at any point in time fall within a limited window of values. We exploit this property (§3.1.1) to employ integer priority queues that have $O(1)$ overhead for packet insertion and extraction. We achieve this by proposing a modification to priority queues based on the Find First Set (FFS) instruction, found in most CPUs, to support a wide range of policies and ranking functions efficiently. We also propose a new approximate priority queue that can outperform FFS-based queues for some scenarios (§3.1.2). Second, we observe (§3.2) that packet scheduling programming models (i.e., PIFO [50] and OpenQueue [39]) do not support per-flow packet scheduling nor do they support reordering of packets on a dequeue operation. We augment the PIFO scheduler programming model to capture these two abstractions.

We introduce Eiffel, an efficient and flexible software scheduler that instantiates our proposed approach. Eiffel is a software packet scheduler that can be deployed on end-hosts and software switches to implement any scheduling algorithm. To demonstrate this we implement Eiffel (§4) in: 1) the kernel as a Queuing Discipline (qdisc) and compare it to Carousel [47] and FQ/Pacing [26] and 2) the Berkeley Extensible Software Switch (BESS) [8, 33] using Eiffel-based implementations of pFabric [14] and hClock [19]. We evaluate Eiffel in both settings (§5). Eiffel outperforms Carousel by 3x and FQ/Pacing by 14x in terms of CPU overhead when deployed on Amazon EC2 machines with line rate of 20 Gbps. We also find that an Eiffel-based implementation of pFabric and hClock outperforms an implementation using comparison-based priority queues by 5x and 40x respectively in terms of maximum number of flows given fixed processing capacity and target rate.

2 Background and Objectives

In modern networks, packet scheduling can easily become the system bottleneck. This is because schedulers are burdened with the overhead of maintaining a large number of buffered packets sorted according to scheduling policies.

Despite the growing capacity of modern CPUs, packet processing overhead remains a concern. Dedicating CPU power to networking takes from CPU capacity that can be dedicated to VM customers especially in cloud settings [28]. One approach to address this overhead is to optimize the scheduler for a specific scheduling policy [26, 25, 19, 47, 22]. However, with specialization two problems linger. First, in most cases inefficiencies remain because of the typical reliance on generic default priority queues in modern libraries (e.g., RB-trees in kernel and Binary Heaps in C++). Second, even if efficiency is achieved, through the use of highly efficient specialized data structures (e.g., Carousel [47] and QFQ [22]) or hybrid hardware/software systems (e.g. SENIC [46]), this efficiency is achieved at the expense of programmability. The Eiffel system we develop in this paper is designed to be both efficient and programmable. In this section we examine these two objectives, show how existing solutions fall short of achieving them and highlight our approach to successfully combine efficiency with flexibility.

Efficient Priority Queueing: Priority queuing is fundamental to computer science with a long history of theoretical results. Packet priority queues are typically developed as comparison-based priority queues [26, 19]. A well known result for such queues is that they require $O(\log n)$ steps for either insertion or extraction for a priority queue holding n elements [52]. This applies to data structures that are widely used in software packet schedulers such as RB-trees, used in kernel Queuing Disciplines, and Binary Heaps, the standard priority queue implementation in C++.

Packet queues, however, have the following characteristics that can be exploited to significantly lower the overhead of packet insertion and extraction:

- *Integer packet ranks:* Whether it is deadlines, transmission time, slack time, or priority, the calculated rank of a packet can always be represented as an integer.
- *Packet ranks have specific ranges:* At any point in time, the ranks of packets in a queue will typically fall within a limited range of values (i.e., with well known maximum and minimum values). This range is policy and load dependent and can be determined in advance by operators (e.g., transmission time where packets can be scheduled a maximum of a few seconds ahead, flow size, or known ranges of strict priority values). Ranges of priority values are diverse ranging from just eight levels [1], to 50k for a queue implementing per flow weighted fairness which requires a number of priorities corresponding to the number of flows (i.e., 50k flows on a video server [47]), and up to 1 million priorities for a time indexed priority queue [47].

- *Large numbers of packets share the same rank:* Modern line rates are in the range of 10s to 100s of Gbps. Hence, multiple packets are bound to be transmitted with nanosecond time gaps. This means that packets with small differences in their ranks can be grouped and said to have

System	Efficiency	HW/SW	Flexibility				Notes
			Unit of Scheduling	Work-Conserving	Supports Shaping	Programmable	
FQ/Pacing qdisc [26] hClock [19] Carousel [47] OpenQueue [39] PIFO [50] <i>Eiffel</i>	$O(\log n)$	SW	Flows	No	Yes	No	Only non-work conserving FQ
	$O(\log n)$	SW	Flows	Yes	Yes	No	Only HWPQ Sched.
	$O(1)$	SW	Packets	No	Yes	No	Only non-work conserving sched.
	$O(\log n)$	SW	Packets & Flows	Yes	No	On enq/deq	Inefficient building blocks
	$O(1)$	HW	Packets	Yes	Yes	On enq	Max. # flows 2048
	$O(1)$	SW	Packets & Flows	Yes	Yes	On enq/deq	-

Table 1: Proposed work in the context of the state of the art in scheduling

the same rank with minimal or no effect on the accurate implementation of the scheduling policy. For instance, consider a busy-polling-based packet pacer that can dequeue packets at fixed intervals (e.g., order of 10s of nanoseconds). In that scenario, packets with gaps smaller than 10 nanoseconds can be considered to have the same rank.

These characteristics make the design of a packet priority queue effectively the design of bucketed integer priority queues over a finite range of rank values $[0, C]$ with number of buckets N , each covering C/N interval of the range. The number of buckets, and consequently the range covered by each bucket, depend on the required ranking granularity which is a characteristic of the scheduling policy. The number of buckets is typically in the range of a few thousands to hundreds of thousands. Elements falling within a range of a bucket are ordered in FIFO fashion. Theoretical complexity results for such bucketed integer priority queues are reported in [53, 29, 52].

Integer priority queues do not come for free. Efficient implementation of integer priority queues requires pre-allocation of buckets and meta data to access those buckets. In a packet scheduling setting the number of buckets is fixed, making the overhead per packet a constant whose value is logarithmic in the number of buckets, because searching is performed on the bucket list not the list of elements. Hence, bucketed integer priority queues achieve CPU efficiency at the expense of maintaining elements unsorted within a single bucket and pre-allocation of memory for all buckets. Note that the maintaining elements unsorted within a bucket is inconsequential because packets within a single bucket effectively have equivalent rank. Moreover, the memory required for buckets, in most cases, is minimal (e.g., tens to hundreds of kilobytes), which is consistent with earlier work on bucketed queues [47]. Another advantage of bucketed integer priority queues is that elements can be (re)moved with $O(1)$ overhead. This operation is used heavily in several scheduling algorithms (e.g., hClock [19] and pFabric [14]).

Recently, there has been some attempts to employ data structures specifically developed or re-purposed for efficiently implementing specific packet scheduling algorithms. For instance, Carousel [47], a system developed for rate limiting at scale, relies on Timing Wheel [54], a data structure that can support time-based operations in $O(1)$ and requires comparable memory to our proposed approach. How-

ever, Timing Wheel supports only non-work conserving time-based schedules in $O(1)$. Timing Wheel is efficient as buckets are indexed based on time and elements are accessed when their deadline arrives. However, Timing Wheel does not support operations needed for non-work conserving schedules (i.e., ExtractMin or ExtractMax). Another example is efficient approximation of popular scheduling policies (e.g., Start-Time Fair Queueing [31] as an approximation of Weighted Fair Queuing [24], or the more recent Quick Fair Queue (QFQ) [22]). This approach of developing a new system or a new data structure per scheduling policy does not provide a path to the efficient implementation of more complex policies. Furthermore, it does not allow for a truly programmable network. These limitations lead us to our first objective for Eiffel:

Objective 1: Develop data structures that can be employed for any scheduling algorithm providing $O(1)$ processing overhead per packet leveraging integer priority queues (§3.1).

Flexibility of Programmable Packet Schedulers: There has been recent interest in developing flexible, programmable, packet schedulers [50, 39]. This line of work is motivated by the support for programmability in all aspects of modern networks. Work on programmable schedulers focuses on providing the infrastructure for network operators to define their own scheduling policies. This approach improves on the current standard approach of providing a small fixed set of scheduling policies as currently provided in modern switches. A programmable scheduler provides building blocks for customizing packet ranking and transmission timing. Proposed programmable schedulers differ based on the flexibility of their building blocks. A flexible scheduler allows a network operator to specify policies according to the following specifications:

- *Unit of Scheduling:* Scheduling policies operate either on per packet basis (e.g., pacing) or on per flow basis (e.g., fair queuing). This requires a model that provides abstractions for both.
- *Work Conservation:* Scheduling policies can be work-conserving or non-work-conserving.
- *Ranking Trigger:* Efficient implementation of policies can require ranking packets on their enqueue, dequeue, or both.

Recent programmable packet schedulers export primitives that enable the specification of a scheduling policy and its

parameters, often within limits. The PIFO scheduler programming model is the most prominent example [50]. It is implemented in hardware relying on Push-In-First-Out (PIFO) building blocks where packets are ranked only on enqueue. The scheduler is programmed by arranging the blocks to implement different scheduling policies. Due to its hardware implementation, the PIFO model employs compact constructs with considerable flexibility. However, PIFO remains very limited in its capacity (i.e., PIFO can handle a maximum of 2048 flows at line rate), and expressiveness (i.e., PIFO can't express per flow scheduling). OpenQueue is an example of a flexible programmable packet scheduler in software [39]. However, the flexibility of OpenQueue comes at the expense of having three of its building blocks as priority queues, namely queues, buffers, and ports. This overhead, even in the presence of efficient priority queues, will form a memory and processing overhead. Furthermore, OpenQueue does not support non-work-conserving schedules.

The design of a flexible and efficient packet scheduler remains an open research challenge. It is important to note here that the efficiency of programmable schedulers is different from the efficiency of policies that they implement. An efficient programmable platform aims to reduce the overhead of its building blocks (i.e., Objective 1) which makes the overhead primarily a function of the complexity of the policy itself. Thus, the efficiency of a scheduling policy becomes a function of only the number of building blocks required to implement it. Furthermore, an efficient programmable platform should allow the operator to choose policies based on their requirements and available resources by allowing the platform to capture a wide variety of policies. To address this challenge, we choose to extend the PIFO model due to its existing efficient building blocks. In particular, we introduce flows as a unit of scheduling in the PIFO model. We also allow modifications to packet ranking and relative ordering both on enqueue and dequeue.

Objective 2: Provide a fully expressive scheduler programming abstraction by extending the PIFO model (§3.2).

Eiffel's place in Scheduling Research Landscape: This section reviewed scheduling support in software¹. Table 1 summarizes the discussed related work. Eiffel fills the gap in earlier work by being the first efficient $O(1)$ and programmable software scheduler. It can support both per flow policies (e.g., hClock and pFabric) and per packet scheduling policies (e.g., Carousel). It can also support both work-conserving and non-work-conserving schedules.

¹Scheduling is widely supported in hardware switches using a short list of scheduling policies, including shaping, strict priority, and Weighted Round Robin [7, 6, 9, 50]. An approach to efficient hardware packet scheduling relies on pipelined-heaps [18, 38, 55] to help position Eiffel. Pipelined-heaps are composed of pipelined-stages for enqueueing and dequeuing elements in a priority queue. However, such approaches are not immediately applicable to software.

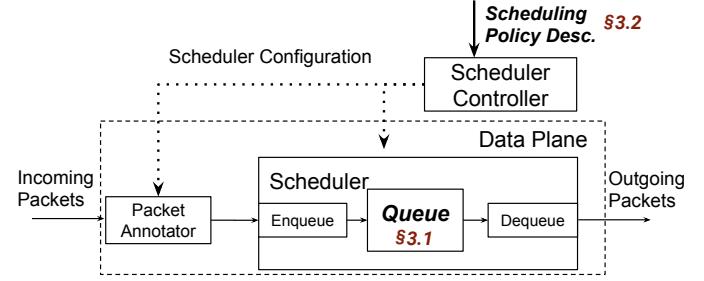


Figure 1: Eiffel programmable scheduler architecture highlighting Eiffel's extensions.

3 Eiffel Design

Figure 1 shows the architecture of Eiffel with four main components: 1) a packet annotator to set the input to the enqueue component (e.g., packet priority), 2) an enqueue component that calculates a rank for incoming packets, 3) a queue that holds packets sorted based on their rank, and 4) a dequeue component which is triggered to re-rank elements in the queue, for some scheduling algorithms. Eiffel leverages and extends the PIFO scheduler programming model to describe scheduling policies [50, 2]. The functions of the packet annotator, the enqueue module, and the dequeue module are derived in a straightforward manner from the scheduling policy. The only complexity in the Scheduler Controller, namely converting scheduling policy description to code, has been addressed in earlier work on the PIFO model [2]. The two complicated components in this architecture, therefore, correspond with the two objectives discussed in the previous section: the Queue (*Objective 1*) and The Scheduling Policy Description (*Objective 2*). For the rest of this section, we explain our efficient queue data structures along with our extensions to the programming model used to configure the scheduler.

3.1 Priority Queueing in Eiffel

A priority queue maintains a list of elements, each tagged with a priority value. A priority queue supports one of two operations efficiently: ExtractMin or ExtractMax to get the element with minimum or maximum priority respectively. Our goal, as stated in Objective 1 in the previous section, is to enable these operations with $O(1)$ overhead. To this end we first develop a circular extension of efficient priority queues that rely on the FindFirstSet (FFS) operation, found in all modern CPUs [3, 10]. Our extensions allow FFS-based queues to operate over large moving ranges while maintaining CPU efficiency. We then improve on the FFS-based priority queue by introducing the approximate gradient queue, which can perform priority queuing in $O(1)$ under some conditions. The approximate priority queue can outperform the FFS-based queue by up to 9% for scenarios of a highly occupied bucketed priority queue (§5.2). Note

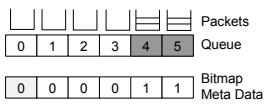


Figure 2: FFS-based queue where FFS of a bit-map of six bits can be processed in $O(1)$.

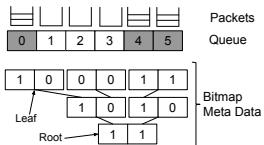


Figure 3: Hierarchical FFS-based queue where FFS of bit-map of two bits can be processed in $O(1)$ using a 3-level hierarchy

that for all Integer Priority Queues discussed in this section, enqueue operation is trivial as buckets are identified by the priority value of their elements. This makes the enqueue operation a simple bucket lookup based on the priority value of the enqueued element.

3.1.1 Circular FFS-based Queue (cFFS)

FFS-based queues are bucketed priority queues with a bitmap representation of queue occupancy. Zero represents an empty bucket, and one represents a non-empty bucket. FFS produces the index of the leftmost set bit in a machine word in constant time. All modern CPUs support a version of Find First Set at a very low overhead (e.g., Bit-Scan-Forward (BSR) takes three cycles to complete [3]). Hence, a priority queue, with a number of buckets equal to or smaller than the width of the word supported by the FFS operation can obtain the smallest set bit, and hence the element with the smallest priority, in $O(1)$ (e.g., Figure 2). In the case that a queue has more buckets than the width of the word supported by a single FFS operation, a set of words can be processed sequentially to represent the queue, with every bit representing a bucket. This results in an $O(M)$ algorithm that is very efficient for very small M , where M is the number of words. For instance, realtime process scheduling in the linux kernel has a hundred priority levels. An FFS-based priority queue is used where FFS is applied sequentially on two words, in case of 64-bit words, or four words in case of 32-bit words [11]. This algorithm is not efficient for large values of M as it requires scanning all words, in the worst case, to find the index of the highest priority element. FFS instruction is also used in QFQ to sort groups of flows based on the eligibility for transmission where the number of groups is limited to a number smaller than 64 [22]. QFQ is an efficient implementation of fair queuing which uses FFS efficiently over a small number of elements. However, QFQ does not provide any clear direction towards implementing other policies efficiently.

To handle an even larger numbers of priority levels, hierarchical bitmaps may be used. One example is Priority Index Queue (PIQ) [55], a hardware implementation of FFS-based queues, which introduces a hierarchical structure where each node represents the occupancy of its children, and the chil-

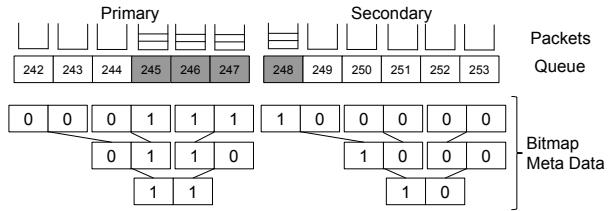


Figure 4: Circular Hierarchical FFS-based queue is composed of two Hierarchical FFS-based queues, one acting as the main queue and the other as a buffer.

dren of leaf nodes are buckets. The minimum element can be found by recursively navigating the tree using FFS operation (e.g., Figure 3 for a word width of two). Hierarchical FFS-based queues have an overhead of $O(\log_w N)$ where w is the width of the word that FFS can process in $O(1)$ and N is the number of buckets. It is important to realize that, for a given scheduling policy, the value of N is a given fixed value that doesn't change once the scheduling policy is configured. Hence, a specific instance of a Hierarchical FFS-based queue has a constant overhead independent of the number of enqueued elements. In other words, once an implementation is created N does not change.

Hierarchical FFS-based queues only work for a fixed range of priority values. However, as discussed earlier, typical priority values for packets span a moving range. PIQ avoids this problem by assuming support for the universe of possible values of priorities. This is an inefficient approach because it requires generating and maintaining a large number of buckets, with relatively few of them in use at any given time.

Typical approaches to operating over a large moving range while maintaining a small memory footprint rely on *circular queues*. Such queues rely on the *mod* operation to map the moving range to a smaller range. However, the typical approach to circular queuing does not work in this case as it results in an incorrect bitmap. For example, if we add a packet with priority value six to the queue in Figure 2 selecting the bucket with a *mod* operation, the packet will be added in slot zero and consequently mark the bit map at slot zero. Hence, once the range of an FFS-based queue is set, all elements enqueued in that range have to be dequeued before the queue can be assigned a new range so as to avoid unnecessary resetting of elements. In that scenario, enqueued elements that are out of range are enqueue at the last bucket, and thus losing their proper ordering. Otherwise, the bitmap meta data will have to be reset in case any changes are made to the range of the queue.

A natural solution to this problem is to introduce an overflow queue where packets with priority values outside the current range are stored. Once all packets in the current range are dequeued, packets from that “secondary” queue are inserted using the new range. However, this introduces a significant overhead as we have to go through all pack-

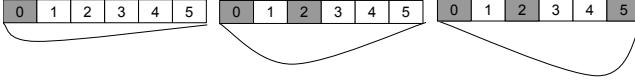


Figure 5: A sketch of a curvature function for three states of a **maximum** priority queue. As the maximum index of nonempty buckets increases, the critical point shifts closer to that index.

ets in the buffer every time the range advances. We solve this problem by making the secondary queue an FFS-based queue, covering the range that is immediately after the range of the queue (Figure 4). Elements outside the range of the secondary queue are enqueued at the last bucket in the secondary queue and their values are not sorted properly. However, we find that to not be a problem as ranges for the queues are typically easy to figure out given a specific scheduling policy.

A *Circular Hierarchical FFS-based queue*, referred to hereafter simply as a cFFS, maintains the minimum priority value supported by the primary queue (`h_index`), the number of buckets (`q_size`) per queue, two pointers to the two sets of buckets, and two pointers to the two sets of bitmaps. Hence, the queue “circulates” by switching the pointers of the two queues from the buffer range to the primary range and back based on the location of the minimum element along with their corresponding bitmaps.

Note that work on efficient priority queues has a very long history in computer science with examples including van Emde Boas tree [53] and Fusion trees [29]. However, such theoretical data structures are complicated to implement and require complex operations. cFFS is highly efficient both in terms of complexity and the required bit operations. Moreover, it is relatively easy to implement.

3.1.2 Approximate Priority Queueing

cFFS queues still require more than one step to find the minimum element. We explore a tradeoff between accuracy and efficiency by developing a gradient queue, a data structure that can find a *near* minimum element in one step.

Basic Idea The Gradient Queue (GQ) relies on an algebraic approach to calculating FFS. In other words, it attempts to find the index of the most significant bit using algebraic calculations. *This makes it amenable to approximation.* The intuition behind GQ is that the contribution of the most significant set bit to the value of a word is larger than the sum of the contributions of the rest of the set bits. We consider the weight of a non-empty bucket to be proportional to its index. Hence, Gradient Queue occupancy is represented by its curvature function. The curvature function of the queue is the sum of the weight functions of all nonempty buckets in the queue. More specifically, a specific curvature shape corresponds to a specific occupancy pattern. A *proper weight function* ensures the uniqueness of the cur-

vature function per occupancy pattern. It also makes finding the non-empty bucket with the maximum index equivalent to finding the critical point of the queue’s curvature (i.e., the point where the derivative of the curvature function of the queue is zero). A sample sketch of a curvature function is illustrated in Figure 5.

Exact Gradient Queue On a bucket becoming nonempty, we add its weight function to the queue’s curvature function, and we subtract its function when it becomes empty. We define a desirable weight function as one that is: 1) easy to differentiate to find the critical point, and 2) easy to maintain when bucket state changes between empty and non-empty. We use weight function, $2^i(x - i)^2$ where i is the index of the bucket and x is the variable in the space of the curvature function.

This weight function results in queue curvature of the form of $ax^2 - bx + c$, where the critical point is located at $x = b/2a$. Hence, we only care about a and b where $a = \sum_i 2^i$ and $b = \sum_i i2^i$ for all non-empty buckets i . The maintenance of the curvature function of the queue becomes as simple as incrementing and decrementing a and b when a bucket becomes non-empty or empty respectively. Theorem 1, in Appendix A, shows that determining the highest priority non-empty queue can be calculated using $\text{ceil}(b/a)$.

A Gradient Queue with a single curvature function is limited by the the range of values a and b can take, which is analogous to the limitation of FFS-based queues by the size of words for which FFS can be calculated in $O(1)$. A natural solution is to develop a hierarchical Gradient Queue. This makes Gradient Queue an equivalent of FFS-based queue with more expensive operations (i.e., division is more expensive than bit operations). However, due to its algebraic nature, Gradient Queue allows for approximation that is not feasible using bit operations.

Approximate Gradient Queue Like FFS-based queues, gradient queue has a complexity of $O(\log_w N)$ where w is the width of the representation of a and b and N is the number of buckets. Our goal is reduce the number of steps even further for each lookup. We are particularly interested in having lookups that can be made in one operation, which can be achieved through approximation. The advantage of the curvature representation of the Gradient Queue compared to FSS-based approaches is that it lends itself naturally to approximation.

A simple approximation is to make the value of a and b corresponding to a certain queue curvature smaller which will allow them to represent a larger number of priority values. In particular, we change the weight function to $2^{f(i)}(x - i)^2$ which results in $a = \sum_i 2^{f(i)}$ and $b = \sum_i i2^{f(i)}$ where $f(i) = i/\alpha$ and α is a positive integer. This approach leads to two natural results: 1) the biggest gain of the approximation is that a and b can now represent a much larger range of values for i which eliminates the need for hierarchical Gradient Queue and allows for finding the minimum ele-

ment with one step, and 2) the employed weight function is no longer proper. While BSR instruction is 8-32x faster than DIV [3], the performance gained from the reduced memory lookups required per BSR operation.

This approximation stems from using an “improper” weight function. This leads to breaking the two guarantees of a proper weight function, namely: 1) the curvature shape is no longer unique per queue occupancy pattern, and 2) the index of the maximum non-empty bucket no longer corresponds to the critical point of the curvature *in all cases*. In other words, the index of the maximum non-empty bucket, M , is no longer $\text{ceil}(b/a)$ due the fact that the weight of the maximum element no longer dominates the curvature function as the growth is sub-exponential. However, this ambiguity does not exist for all curvatures (i.e., queue occupancy patterns).

We characterize the conditions under which ambiguity occurs causing error in identifying the highest priority non-empty bucket. Hence, we identify scenarios where using the approximate queue is acceptable. The effect of $f(i) = i/\alpha$ can be described as introducing ambiguity to the value of $\text{ceil}(b/a)$. This is because exponential growth in a and b occurs not between consecutive indices but every α indices. In particular, we find solving the geometric and arithmetic-geometric sums of a and b that $\frac{b}{a} = \frac{M}{1-g(\alpha,M)} + u(\alpha)$ where $g(\alpha,M) = (2^{1/\alpha})^{-M-1}$ is a logarithmically decaying function of M and α . $u(\alpha) = 1/(1 - 2^{1/\alpha})$ is non-linear but slowly growing function of α . Hence, an approximate GQ can operate as a bucketed-queue where indices start from I_0 where $g(\alpha,M_0) \approx 0$ and end at I_{max} where $2^{f(I_{max})}$ can be precisely represented in the CPU word used to represent a and b . In this case, there is a constant shift in the value $\text{ceil}(b/a)$ that is calculated by $u(\alpha)$. For instance, consider an approximate queue with an α of 16. The function $g(\alpha,M)$ decays to near zero at $M = 124$ making the shift $u(\alpha) = 22$. Hence, $I_0 = 124$ and $I_{max} = 647$ which allows for the creation of an approximate queue that can handle 523 buckets. Note that this configuration results in an exact queue only when all buckets between I_0 and I_{max} are nonempty. However, error is introduced when some elements are missing. In Section 5.2, we show the effect of this error through extensive experiments; more examples are shown in Appendix B.

Typical scheduling policies (e.g., timestamp-based shaping, Least Slack Time First, and Earliest Deadline First) will generate priority values for packets that are uniformly distributed over priority levels. For such scenarios, the approximate gradient queue will have zero error and extract the minimum element in one step. This is clearly not true for *all* scheduling policies (e.g., strict priority will probably have more traffic for medium and low level priorities compared to high priority). For cases where the index suggested by the function is of an empty bucket, we perform linear search until we find a nonempty bucket. Moreover, for a cases of a moving range, a circular approximate queue can be imple-

mented as with cFFS.

Approximate queues have been used before for different use cases. For instance, Soft-heap [21] is an approximate priority queue with a bounded error that is inversely proportional to the overhead of insertion. In particular, after n insertions in a soft-heap with an error bound $0 < \epsilon \leq 1/2$, the overhead of insertion is $O(\log(1/\epsilon))$. Hence, ExtractMin operation which can have a large error under Soft-heap. Another example is the RIPQ which was developed for caching [51]. RIPQ relies on a bucket-sort-like approach. However, the RIPQ implementation is suited for static caching, where elements are not moved once inserted, which makes it not very suitable for the dynamic nature of packet scheduling.

3.2 Flexibility in Eiffel

Our second objective is to deploy flexible schedulers that have full expressive power to implement a wide range of scheduling policies. Our goal is to provide the network operator with a compiler that takes as input policy description and produces an initial implementation of the scheduler using the building blocks provided in the previous section. Our starting point is the work in PIFO which develops a model for programmable packet scheduling [50]. PIFO, however, suffers from several drawbacks, namely: 1) it doesn’t support reordering packets already enqueued based on changes in their flow ranking, 2) it does not support ranking of elements on packet dequeue, and 3) it does not support shaping the output of the scheduling policy. In this section, we show our augmentation of the PIFO model to enable a completely flexible programming model in Eiffel. We address the first two issues by adding programming abstractions to the PIFO model, and we address the third problem by enabling arbitrary shaping with Eiffel by changing how shaping is handled within the PIFO model. We discuss the implementation of an initial version of the compiler in Section 4.

3.2.1 PIFO Model Extensions

Before we present our new abstractions, we review briefly the PIFO programming model [50]. The model relies on the Push-In-First-Out (PIFO) conceptual queue as its main building block. In programming the scheduler, the PIFO blocks are arranged to implement different scheduling algorithms.

The PIFO programming model has three abstractions: 1) scheduling transactions, 2) scheduling trees, and 3) shaping transactions. A scheduling transaction represents a single ranking function with a single priority queue. Scheduling trees are formed by connecting scheduling transactions, where each node’s priority queue contains an ordering of its children. The tree structure allows incoming packets to change the relative ordering of packets belonging to different policies. Finally, a shaping transaction can be attached to any non-root node in the tree to enforce a rate limit on it.

There are several examples of the PIFO programming model in action presented in the original paper [50]. The primitives presented in the original PIFO model capture scheduling policies that have one of the following features: 1) distinct packet rank enumerations, over a small range of values (e.g., strict priority), 2) per-packet ranking over a large range of priority values (e.g., Earliest Deadline First [41]), and 3) hierarchical policy-based scheduling (e.g., Hierarchical Packet Fair Queuing [17]).

Eiffel augments the PIFO model by adding two additional scheduler primitives. The first primitive is *per-flow ranking and scheduling* where the rank of all packets of a flow depend on a ranking that is a function of the ranks of all packets enqueued for that specific flow. We assume that a sequence of packets that belong to a single flow should not be reordered by the scheduler. Existing PIFO primitives keep per-flow state but use them to rank each packet individually where an incoming packet for a certain flow does not change the ranking of packets already enqueued that belong to the same flow. The per-flow ranking extension keeps track of that information along with a queue per flow for all packets belonging to that flow. A single PIFO block orders flows, rather than packets, based on their rank. The second primitive is *on-dequeue scheduling* where incoming and outgoing packets belonging to a certain flow can change the rank of all packets belonging to that flow on enqueue and dequeue.

The two primitives can be integrated in the PIFO model. All flows belonging to a *per-flow* transaction are treated as a single flow by scheduling transactions higher in the hierarchical policy. Also note that every individual flow in the flow-rank policy can be composed of multiple flows that are scheduled according to per packet scheduling transactions. We realize that this specification requires tedious work to describe a complex policy that handles thousands of different flows or priorities. However, this specification provides a direct mapping to the underlying priority queues. We believe that defining higher level programming languages describing packet schedulers as well as formal description of the expressiveness of the language to be topics for future research.

3.2.2 Arbitrary Shaping

A flexible packet scheduler should support any scheme of bandwidth division between incoming flows. Earlier work on flexible schedulers either didn't support shaping at all (e.g., OpenQueue) or supported it with severe limitations (e.g., PIFO). We allow for arbitrary shaping by decoupling work conserving scheduling from shaping. A natural approach to this decoupling is to allow any flow or group of flows to have a shaper associated with them. This can be achieved by assigning a separate queue to the shaped aggregate whose output is then enqueued into its proper location in the scheduling hierarchy. However, this approach is extremely inefficient as it requires a queue per rate limit, which can lead to increased CPU and memory overhead. We im-

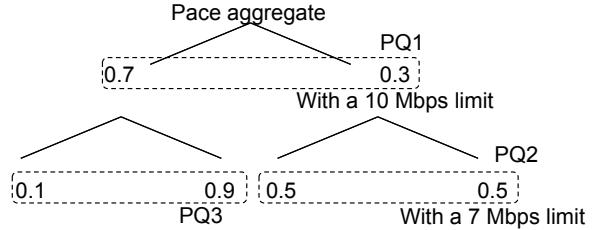


Figure 6: Example of a policy that imposes two limits on packets the belong to the rightmost leaf.

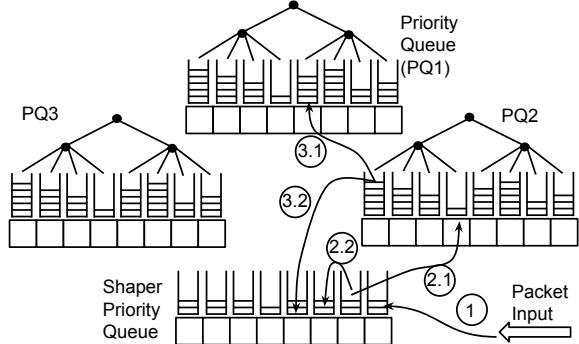


Figure 7: A diagram of the implementation of the example in Figure 6.

prove the efficiency of this approach by leveraging recent results that show that any rate limit can be translated to a timestamp per packet, which yields even better adherence to the set rate than token buckets [47]. Hence, we use only one shaper for the whole hierarchy which is implemented using a single priority queue.

As an example, consider the hierarchical policy in Figure 6. Each node represents a policy-defined flow with the root representing the aggregate traffic. Each node has a share of its parent's bandwidth, defined by the fraction in the figure. Each node can also have a policy-defined rate limit. In this example, we have a rate limit at a non-leaf node and a leaf node. Furthermore, we require the aggregate traffic to be paced. We map the hierarchical policy in Figure 6 to its priority-queue-based realization in Figure 7. Per the PIFO model, each non-leaf node is represented by a priority queue. Per our proposal, a single shaper is added to rate limit all packets according to all policy-defined rate limits.

To illustrate how this single shaper works, consider packets belonging to the rightmost leaf policy. We explore the journey of packets belonging to that leaf policy through the different queues, shown in Figure 7. These packets will be enqueued to the shaper with timestamps set based on a 7 Mbps rate to enforce the rate on their node (step 1). Once dequeued from the shaper, each packet will be enqueued to PQ2 (step 2.1) and the shaper according to the 10 Mbps rate limit (step 2.2). After the transmission time of a packet belonging to PQ2 is reached, which is defined by the shaper, the packet is inserted in both the root's (PQ1) priority queue

(3.1) and the shaper according to the pacing rate (3.2). When the transmission time, calculated based on the pacing rate, is reached the packet is transmitted. To achieve this functionality, each packet holds a pointer to the priority queue they should be enqueued to. This pointer avoids searching for the queue a packet should be enqueued to. Note that having the separate shaper allows for specifying rate limits on any node in the hierarchical policy (e.g., the root and leaves) which was not possible in the PIFO model, where shaping transactions are tightly coupled with scheduling transactions.

4 Eiffel Implementation

Packet scheduling is implemented in two places in the network: 1) hardware or software switches, and 2) end-host kernel. We focus on the software placements (kernel and userspace switches) and show that Eiffel can outperform the state of the art in both settings. We find that userspace and kernel implementations of packet scheduling face significantly different challenges as the kernel operates in an event-based setting while userspace operates in a busy polling setting. We explain here the differences between both implementations and our approach to each. We start with our approach to policy creation.

Policy Creation: We extend the existing PIFO open source model to configure the scheduling algorithm [50, 2]. The existing implementation represents the policy as a graph using the DOT description language and translates the graph into C++ code. We rely on the cFFS for our implementation, unless otherwise stated. This provides an initial implementation which we tune according to whether the code is going to be used in kernel or userspace. We believe automating this process can be further refined, but the goal of this work is to evaluate the performance of Eiffel algorithms and data structures.

Kernel Implementation We implement Eiffel as a qdisc [36] kernel module that implements enqueue and dequeue functions and keeps track of the number of enqueued packets. The module can also set a timer to trigger dequeue. Access to qdiscs is serialized through a global qdisc lock. In our design, we focus on two sources of overhead in a qdisc: 1) the overhead of the queuing data structure, and 2) the overhead of properly setting the timer. Eiffel reduces the first overhead by utilizing one of the proposed data structures to reduce the cost of both enqueue and dequeue operations. The second overhead can be mitigated by improving the efficiency of finding the smallest deadline of an enqueued packet. This operation of `SoonestDeadline()` is required to efficiently set the timer to wake up at the deadline of the next packet. Either of our supported data structures can support this operation efficiently as well.

Userspace Implementation We implement Eiffel in the Berkeley Extensible Software Switch (BESS, formerly SoftNIC [33]). BESS represents network processing elements as a pipeline of modules. BESS is busy polling-based where

a set of connected modules form a unit of execution called a task. A scheduler tracks all tasks and runs them according to assigned policies. Tasks are scheduled based on the amount of resources (CPU cycles or bits) they consume. Our implementation of Eiffel in BESS is done in self-contained modules.

We find that two main parameters determine the efficiency of Eiffel in BESS: 1) batch size and 2) queue size. Batching is already well supported in BESS as each module receives packets in batches and passes packets to its subsequent module in a batch. However, we find that batching per flow has an intricate impact on the performance of Eiffel. For instance, with small packet sizes, if no batching is performed per flow, then every incoming batch of packets will activate a large number of queues without any of the packets being actually queued (due to small packet size) which increases the overhead per packet (i.e., queue lookup of multiple queues rather than one). This is not the case for large packet sizes where the lookup cost is amortized over the larger size of the packet improving performance compared to batching of large packets. Batching large packets results in large queues for flows (i.e., large number of flows with large number of enqueued packets). We find that batching should be applied based on expected traffic pattern. For that purpose, we setup Buffer modules per traffic class before Eiffel’s module in the pipeline when needed. We also perform output batching per flow in units of 10KB worth of payload which was suggested as a good threshold that does not affect fairness at a macroscale between flows [19]. We also find that limiting the number of packets enqueued in Eiffel can significantly affect the performance of Eiffel in BESS. We limit the number of packets per flow to 32 packets which we find, empirically, to maintain performance.

5 Evaluation

5.1 Eiffel Use Cases

Methodology: We evaluate our kernel and userspace implementation through a set of use cases each with its corresponding baseline. We implement two common use cases, one in kernel and another in userspace². In each use case, we evaluated Eiffel’s scheduling behavior as well as its CPU performance as compared to the baseline. The comparison of scheduling behavior was done by comparing aggregate rates achieved as well as order of released packets. However, we only report CPU efficiency results as we find that Eiffel matches the scheduling behavior of the baselines.

A key aspect of our evaluation is determining the metrics of comparisons in kernel and userspace settings. The main difference is that a kernel module can support line rate by using more CPU. This requires us to fix the packet rate we are evaluating at and look at the CPU utilization of different

²A third use case the implements hClock in userspace can be found in the extended version of this paper [48]

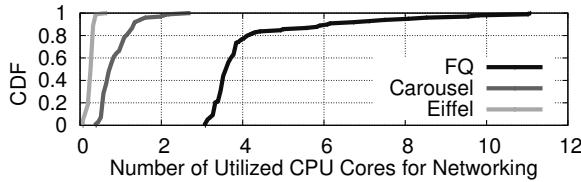


Figure 8: A comparison between the CPU overhead of the networking stack using FQ/pacing, Carousel, and Eiffel.

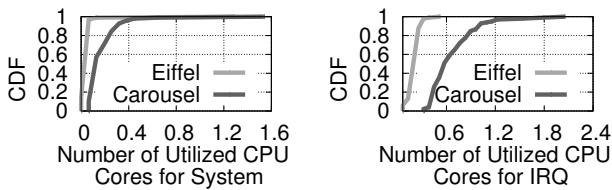


Figure 9: A Comparison between detailed CPU utilization of Carousel and Eiffel in terms of system processes (left) and soft interrupt servicing (right).

scheduler implementations. On the other hand, a userspace implementation relies on busy polling on one or more CPU cores to support different packet rates. Hence, in the case of userspace, we fix the number of cores used, to one core unless otherwise is stated, and compare the different scheduler implementations based on the maximum achievable rate.

5.1.1 Use Case 1: Shaping in Kernel

Traffic shaping (i.e., rate limiting and pacing) is an essential operation for efficient utilization [16] and correct operation of modern protocols (e.g., both TIMELY [43] and BBR [20] require per flow pacing). Recently, it has been shown that the canonical kernel shapers (i.e., FQ/pacing [26] and HTB qdiscs [25]) are inefficient due to reliance on inefficient data structures; there are outperformed by the userspace-based implementation in Carousel [47]. To offer a fair comparison we implement all systems in the kernel. We implement a rate limiting qdisc whose functionality matches the rate limiting features of the existing FQ/pacing qdisc [26].

We implemented Eiffel as a qdisc. The queue is configured with 20k buckets with a maximum horizon of 2 seconds and only the shaper is used. We implemented the qdisc in kernel v4.10. We modified only `sock.h` to keep the state of each socket allowing us to avoid having to keep track of each flow in the qdisc. We conduct experiments for egress traffic shaping between two servers within the same cluster in Amazon EC2. We use two `m4.16xlarge` instances equipped with 64 cores and capable of sustaining 25 Gbps. We use `neper` [5] to generate traffic with a large number of TCP flows. In particular, we generate traffic from 20k flows and use `SO_MAX_PACING_RATE` to rate limit individual flows to achieve a maximum aggregate rate of 24 Gbps. This configuration constitutes a worst case in terms of load for all evaluated qdiscs as it requires the maximum amount of cal-

```
#On enqueue of packet p of flow f:
f.rank = min(p.rank, f.rank)
#On dequeue of packet p of flow f:
f.rank = min(p.rank, f.front().rank)
```

Figure 10: Implementation of pFabric in Eiffel.

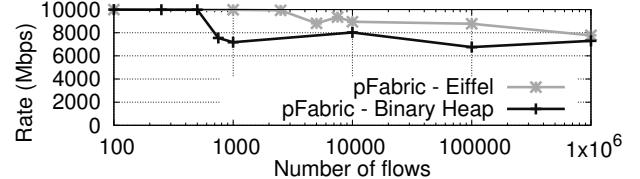


Figure 11: Performance of pFabric implementation using cFFS and a binary heap showing Eiffel sustaining line rate at 5x number of flows.

culations. We measure overhead in terms of the number of cores used for network processing which we calculate based on the observed fraction of CPU utilization. Without `neper` operating, CPU utilization is zero, hence, we attribute any CPU utilization during our experiments to the networking stack, except for the CPU portion attributed to userspace processes. We track CPU utilization using `dstat`. We run our experiments for 100 seconds and record the CPU utilization every second. This continuous behavior emulates the behavior handled by content servers which were used to evaluate Carousel [47].

Figure 8 shows the overhead of all three systems. It is clear that Eiffel is superior, outperforming FQ by a median 14x and Carousel by 3x. We find the overhead of FQ to be consistent with earlier results [47]. This is due to its complicated data structure which keeps track internally of active and inactive flows and requires continuous garbage collection to remove old inactive flows. Furthermore, it relies on RB-trees which increases the overhead of reordering flows on every enqueue and dequeue. To better understand the comparison with Carousel, we look at the breakdown of the main components of CPU overhead, namely overhead spent on *system processes* and *servicing software interrupts*. Figure 9 details the comparison. We find that the main difference is in the overhead introduced by Carousel in firing timers at constant intervals while Eiffel can trigger timers exactly when needed (Figure 9 right). The overhead of the data structures in both cases introduces minimal overhead in system processes (Figure 9 left).

5.1.2 Use Case 2: Least/Largest X First in Userspace

One of the most widely used patterns for packet scheduling is ordering packets such that the flow or packet with the least or most of some feature exits the queue first. Many examples of such policies have been promoted including Least Slack Time First (LSTF) [42], Largest Queue First (LQF), and Shortest/Least Remaining Time First (SRTF). We refer to this class of algorithms as L(X)F. This class of algorithms

is interesting as some of them were shown to provide theoretically proven desirable behavior. For instance, LSTF was shown to be a universal packet scheduler that can emulate the behavior of any scheduling algorithm [42]. Furthermore, SRTF was shown to schedule flows close to optimally within the pFabric architecture [14]. We show that Eiffel can improve the performance of this class of scheduling algorithms.

We implement pFabric as an instance of such class of algorithms where flows are ranked based on their remaining number of packets. Every incoming and outgoing packet changes the rank of all other packets belonging to the same flow, requiring on dequeue ranking. Figure 10 shows the representation of pFabric using the PIFO model with per-flow ranking and on dequeue ranking provided by Eiffel. We also implemented pFabric using $O(\log n)$ priority queue based on a Binary Heap to provide a baseline. Both designs were implemented as queue modules in BESS. We used packets of size 1500B. Operations are all done on a single core with a simple flow generator. All results are the average of ten experiments each lasting for 20 seconds. Figure 11 shows the impact of increasing the number of flows on the performance of both designs. It is clear that Eiffel has better performance. The overhead of pFabric stems from the need to continuously move flows between buckets which has $O(1)$ using bucketed queues while it has an overhead of $O(n)$ as it requires re-heapifying the heap every time. The figure also shows that as the number of flows increases the value of Eiffel starts to decrease as Eiffel reaches its capacity.

5.2 Eiffel Microbenchmark

Our goal in this section is evaluate the impact of different parameters on the performance of different data structures. We also evaluate the effect of approximation in switches on network-wide objectives. Finally, we provide guidance on how one should choose among the different queuing data structures within Eiffel, given specific scheduler user-case characteristics. To inform this decision we run a number of microbenchmark experiments. We start by evaluating the performance of the proposed data structures compared to a basic bucketed priority queue implementation. Then, we explore the impact of approximation using the gradient queue both on a single queue and at a large network scale through ns2-simulation. Finally, we present our guide for choosing a priority queue implementation.

Experiment setup: We perform benchmarks using Google’s benchmark tool [12]. We develop a baseline for bucketed priority queues by keeping track of non-empty buckets in a binary heap, we refer to this as BH. We ignore comparison-based priority queues (e.g., Binary Heaps and RB-trees) as we find that bucketed priority queues performs 6x better in most cases. We compare cFFS, approximate gradient queue (Approx), and BH. In all our experiments, the queue is initially filled with elements according to queue occupancy rate or average number of packet per bucket param-

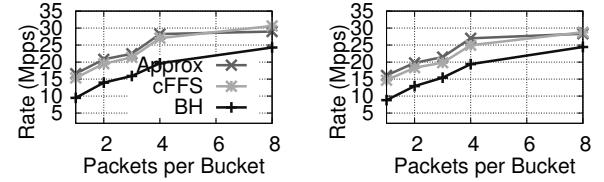


Figure 12: Effect of number of packets per bucket on queue performance for 5k (left) and 10k (right) buckets.

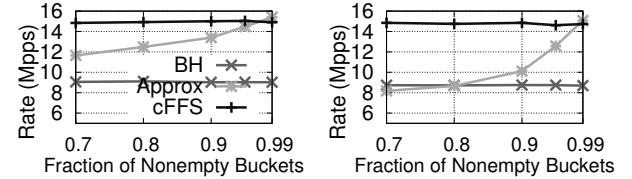


Figure 13: Effect of queue occupancy on performance of Approximate Queue for 5k (left) and 10k (right) buckets.

eters. Then, packets are dequeued from the queue. Reported results (i.e., y-axis of figures 12 and 13) are in terms of million packets per seconds.

Effect of number of packet per bucket: The number of buckets configured in a queue is the main determining factor for the overhead of a bucketed queue. Note that this parameter controls queue granularity which is the priority interval covered by a bucket. High granularity (i.e., large number of buckets) implies a smaller number of packets per bucket for the same workload. Hence, the number of packets per bucket is a good proxy to the configured number of buckets. For instance, if we choose a large number of buckets with high granularity, the chance of empty buckets increases. On the other hand, if we choose a small number of buckets with coarser granularity, we get higher number of elements per bucket. This proxy is important because in the case of the approximate queue, the main factor affecting its performance is the number of empty buckets.

Figure 12 shows the effect of increasing the average number of packets per bucket for all three queues for 5k and 10k buckets. For a small number of packets per bucket, which also reflects choosing a fine grain granularity, the approximate queue introduces up to 9% improvement in performance in the case of 10k buckets. In such cases, the approximate queue function has zero error which makes it significantly better. As the number of the packets per bucket increases, the overhead of finding the smallest indexed bucket is amortized over the total number of elements in the bucket which makes FFS-based and approximate queues similar in performance.

We also explore the effect of having empty buckets on the performance of the approximate queue. Empty buckets cause errors in the curvature function of the approximate queue which in turn trigger linear search for non-empty buckets. Figure 12 shows throughput of the queue for different ratios

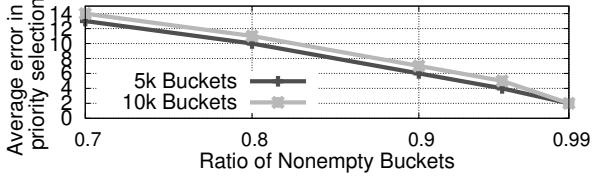


Figure 14: Effect of having empty buckets on the error of fetching the minimum element for the approximate queue.

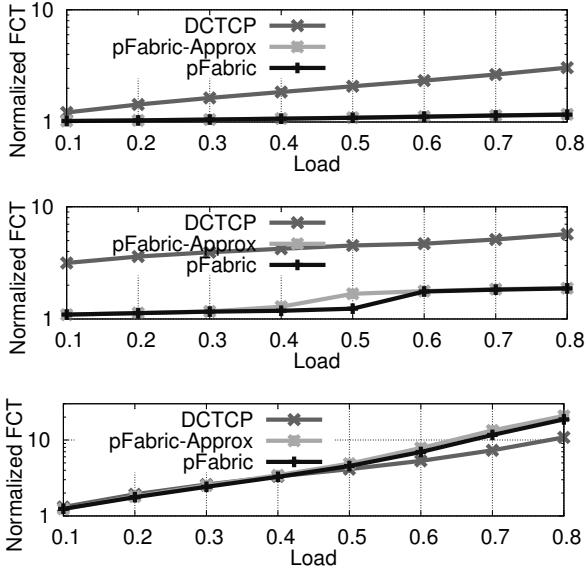


Figure 15: Effect of using an Approximate Queue on the performance of pFabric in terms of normalized flow completion times under different load characteristics: Average FCT for (0, 100kB] flow sizes, 99th percentile FCT for (0, 100kB] for sizes, and Average FCT for (10MB, inf) flow sizes.

of non-empty buckets. As expected, as the ratio increases the overhead decreases which improves the throughput of the approximate queue. Figure 14 shows the error in the approximate queue’s fetching of elements. As the number of empty buckets increases the error in the approximate queue is larger and the overhead of linear search grows. We suggest that cases where the queue is more than 30% empty should trigger changes in the queue’s granularity based on the queue’s CPU performance and to avoid allocating memory to buckets that are not used.

The granularity of the queue determines the representation capacity of the queue. It is clear for our results that picking low granularity (i.e., high number of packets per bucket) yields better performance in terms of packets per second. On the other hand, from a networking perspective, high granularity yields exact ordering of packets. For instance, a queue with a granularity of 100 microseconds cannot insert gaps between packets that are smaller than 100 microseconds. Hence, we recommend configuring the queue’s granularity such that each bucket has at least one packet. This

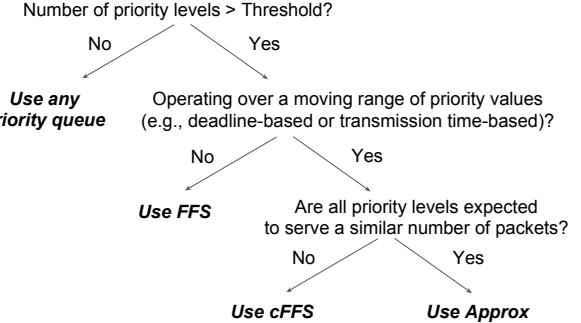


Figure 16: Decision tree for selecting a priority queue based on the characteristics of the scheduling algorithm.

can be determined by observing the long term behavior of the queue. We also note that this problem can be solved by having non-uniform bucket granularity which is dynamically set to achieve the result of at least one packet per bucket. We leave this problem for future work.

Impact of Approximation on Network-wide Objectives: A natural question is: how does approximate prioritization, at *every* switch in a network, affect network-wide objectives? To answer that question, we perform simulations of pFabric, which requires prioritization at every switch. Our simulation are based on ns2 simulations provided by the authors of pFabric [14] and the plotting tools provided by the authors of QJump [32]. We change only the priority queuing implementation from a linear search-based priority queue to our Approximate priority queue and increase queue size to handle 1000k elements. We use DCTCP [13] as a baseline to put the result in context. Figure 15 shows a snapshot of results of the simulations of a 144 node leaf-spine topology. Due to space limitations, We show results for only web-search workload simulations which are based on clusters in Microsoft datacenters [13]. The load is varied between 10% to 80% of the load observed. We note that the setting of the simulations is not relevant for the scope of this paper, however, what is relevant is comparing the performance of pFabric using its original implementation to pFabric using our approximate queue. We find that approximation has minimal effect on overall network behavior which makes performance on a mircorscale the only concern in selecting a queue for a specific scheduler.

A Guide for Choosing a Priority Queue for Packet Scheduling Figure 16 summarizes our takeaways from working with the proposed queues. For a small number of priority levels, we find that the choice of priority queue has little impact and for most scenarios a bucket-based queue might be overkill due to its memory overhead. However, when the number of priority levels or buckets is larger than a threshold the choice of queues makes a significant difference. We found in our experiments that this threshold is 1k and that the difference in performance is not significant around the threshold. We find that if the priority levels are

over a fixed range (e.g., job remaining time [14]) then an FFS-based priority queue is sufficient. When the priority levels are over a moving range, where the number of levels are not all equally likely (e.g., rate limiting with a wide range of limits [47]), it is better to use cFFS priority queue. However, for priority levels over a moving range with highly occupied priority levels (e.g., Least Slack Time-based [42] or hierarchical-based schedules [19]) approximate queue can be beneficial.

Another important aspect is choosing the number of buckets to assign to a queue. This parameter should be chosen based on both the desired granularity and efficiency which form a clear trade-off. Proposed queues have minimal CPU overhead (e.g., a queue with a billion buckets will require six bit operations to find the minimum non-empty bucket using a cFFS). Hence, the main source of efficiency overhead is the memory overhead which has two components: 1) memory footprint, and 2) cache freshness. However, we find that most scheduling policies require thousands to tens of thousands of elements which require small memory allocation for our proposed queues.

6 Conclusion

Efficient packet scheduling is a crucial mechanism for the correct operation of networks. Flexible packet scheduling is a necessary component of the current ecosystem of programmable networks. In this paper, we showed how Eiffel can introduce both efficiency and flexibility for packet scheduling in software relying on integer priority queuing concepts and novel packet scheduling programming abstractions. We showed that Eiffel can achieve orders of magnitude improvements in performance compared to the state of the art while enabling packet scheduling at scale in terms of both number of flows or rules and line rate. We believe that our work should enable network operators to have more freedom in implementing complex policies that correspond to current networks needs where isolation and strict sharing policies are needed.

We believe that the biggest impact Eiffel will have is making the case for a reconsideration of the basic building blocks of the packet schedulers in hardware. Current proposals for packet scheduling in hardware (e.g., PIFO model [50] and SmartNICs [28]), rely on parallel comparisons of elements in a single queue. This approach limits the size of the queue. Earlier proposals that rely on pipelined-heaps [18, 38, 55] required a priority queue that can capture the whole universe of possible packet rank values, which requires significant hardware overhead. We see Eiffel as a step on the road of improving hardware packet schedulers by reducing the number of parallel comparisons through an FFS-based queue meta data or through an approximate queue metadata. For instance, Eiffel can be employed in a hierarchical structure with parallel comparisons to increase the capacity of individual queues in a PIFO-like setting. Future programmable schedulers can

implement a hardware version of cFFS or the approximate queue and provide an interface that allows for connecting them according to programmable policies. While the implementation is definitely not straight forward, we believe this to be the natural next step in the development of scalable packet schedulers.

Acknowledgments

The authors would like to thank the NSDI Shepherd, K. K. Ramakrishnan, and the anonymous reviewers for providing excellent feedback. This work is funded in part by NSF grants NETS 1816331.

References

- [1] IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-2005 (Incorporates IEEE Std 802.1Q1998, IEEE Std 802.1u-2001, IEEE Std 802.1v-2001, and IEEE Std 802.1s-2002)* (May 2006), 1–300.
- [2] C++ reference implementation for Push-In First-Out Queue, 2016. <https://github.com/programmable-scheduling/pifo-machine>.
- [3] Intel 64 and ia-32 architectures optimization reference manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>, 2016.
- [4] Intel 82599 10gbe controller. <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf>, 2016.
- [5] neper: a Linux networking performance tool, 2016. <https://github.com/google/neper>.
- [6] Arista 7010T Gigabit Ethernet Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7010T-48_Datasheet.pdf, 2017.
- [7] Arista 7500 series data center switch. https://www.arista.com/assets/data/pdf/Datasheets/7500_Datasheet.pdf, 2017.
- [8] Bess: Berkeley extensible software switch. <https://github.com/NetSys/bess/wiki>, 2017.
- [9] Cisco: Understanding Quality of Service on the Catalyst 6500 Switch. https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11_538840.html, 2017.
- [10] MD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions. <https://support.amd.com/TechDocs/24594.pdf>, 2017.
- [11] Real-Time Scheduling Class (mapped to the SCHED_FIFO and SCHED_RR policies). <https://elixir.bootlin.com/linux/latest/source/kernel/sched/rt.c#L1494>, 2017.
- [12] Benchmark Tools, 2018. <https://github.com/google/benchmark>.
- [13] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2010), ACM, pp. 63–74.
- [14] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., McKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2013), ACM, pp. 435–446.

- [15] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 455–468.
- [16] BEHESHTI, N., GANJALI, Y., GHOBADI, M., MCKEOWN, N., AND SALMON, G. Experimental study of router buffer sizing. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2008), IMC ’08, ACM, pp. 197–210.
- [17] BENNETT, J. C. R., AND ZHANG, H. Hierarchical packet fair queuing algorithms. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 1996), SIGCOMM ’96, ACM, pp. 143–156.
- [18] BHAGWAN, R., AND LIN, B. Fast and scalable priority queue architecture for high-speed network switches. In *INFOCOM ’00* (2000), pp. 538–547.
- [19] BILLAUD, J.-P., AND GULATI, A. hclock: Hierarchical qos for packet scheduling in a hypervisor. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys ’13, ACM, pp. 309–322.
- [20] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September–October (2016), 20 – 53.
- [21] CHAZELLE, B. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)* 47, 6 (2000), 1012–1027.
- [22] CHECCONI, F., RIZZO, L., AND VALENTE, P. QFQ: Efficient packet scheduling with tight guarantees. *IEEE/ACM Transactions on Networking (TON)* 21, 3 (2013), 802–816.
- [23] DALTON, M., SCHULTZ, D., ADRIAENS, J., AREFIN, A., GUPTA, A., FAHS, B., RUBINSTEIN, D., ZERMENO, E. C., RUBOW, E., DOCAUER, J. A., ALPERT, J., AI, J., OLSON, J., DECABOOTER, K., DE KRUIJF, M., HUA, N., LEWIS, N., KASINADHUNI, N., CREPALDI, R., KRISHNAN, S., VENKATA, S., RICHTER, Y., NAIK, U., AND VAHDAT, A. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 373–387.
- [24] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queuing algorithm. In *Symposium Proceedings on Communications Architectures & Protocols* (New York, NY, USA, 1989), SIGCOMM ’89, ACM, pp. 1–12.
- [25] DEVERA, M. Linux Hierarchical Token Bucket, 2003. <http://luxik.cdi.cz/~devik/qos/htb/>.
- [26] DUMAZET, E., AND CORBET, J. Tso sizing and the fq scheduler. <https://lwn.net/Articles/564978/>, 2013.
- [27] FEAMSTER, N., AND REXFORD, J. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583* (2017).
- [28] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 51–66.
- [29] FREDMAN, M. L., AND WILLARD, D. E. Blasting through the information theoretic barrier with fusion trees. In *STOC’ 90* (1990), pp. 1–7.
- [30] GHODSI, A., SEKAR, V., ZAHARIA, M., AND STOICA, I. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM ’12, ACM, pp. 1–12.
- [31] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 1996), SIGCOMM ’96, ACM, pp. 157–168.
- [32] GROSVENOR, M. P., SCHWARZKOPF, M., GOG, I., WATSON, R. N. M., MOORE, A. W., HAND, S., AND CROWCROFT, J. Queues don’t matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 1–14.
- [33] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.
- [34] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review* 42, 4 (Aug. 2012), 127–138.
- [35] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2013), ACM, pp. 15–26.
- [36] HUBERT, B., GRAF, T., MAXWELL, G., VAN MOOK, R., VAN OOSTERHOUT, M., SCHROEDER, P., SPAANS, J., AND LARROY, P. Linux advanced routing & traffic control. In *Ottawa Linux Symposium* (2002), p. 213.
- [37] INC., V. Performance Evaluation of Network I/O Control in VMware vSphere 6. <https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/network-ioc-vsphere6-performance-evaluation-white-paper.pdf>, 2015.
- [38] IOANNOU, A., AND KATEVENIS, M. G. H. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking* 15, 2 (April 2007), 450–461.
- [39] KOGAN, K., MENIKKUMBURA, D., PETRI, G., NOH, Y., NIKOLENKO, S., SIROTKIN, A. V., AND EUGSTER, P. A programmable buffer management platform. In *ICNP ’17* (2017).
- [40] KUMAR, A., JAIN, S., NAIK, U., RAGHURAMAN, A., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2015), ACM, pp. 1–14.
- [41] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.
- [42] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal Packet Scheduling. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI ’16)* (Mar. 2016), pp. 501–521.
- [43] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM Conference* (2015), pp. 537–550.
- [44] MUNIR, A., BAIG, G., IRTEZA, S. M., QAZI, I. A., LIU, A. X., AND DOGAR, F. R. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2014), ACM, pp. 491–502.

- [45] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)* (2015).
- [46] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: scalable NIC for end-host rate limiting. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)* (2014).
- [47] SAEED, A., DUKKIPATI, N., VALANCIUS, V., LAM, T., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable Traffic Shaping at End-Hosts. In *Proceedings of the ACM SIGCOMM Conference* (2017).
- [48] SAEED, A., ZHAO, Y., DUKKIPATI, N., AMMAR, M., ZEGURA, E., HARRAS, K., AND VAHDAT, A. Eiffel: Efficient and Flexible Software Packet Scheduling. *arXiv preprint arXiv:1810.03060* (2018).
- [49] SIVARAMAN, A., SUBRAMANIAN, S., AGRAWAL, A., CHOLE, S., CHUANG, S.-T., EDSALL, T., ALIZADEH, M., KATTI, S., MCKEOWN, N., AND BALAKRISHNAN, H. Towards programmable packet scheduling. In *HotNets-XIV* (2015).
- [50] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *Proceedings of the ACM SIGCOMM Conference* (2016), pp. 44–57.
- [51] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. Ripq: Advanced photo caching on flash for facebook. In *FAST* (2015), pp. 373–386.
- [52] THORUP, M. Equivalence between priority queues and sorting. *J. ACM* 54, 6 (2007).
- [53] VAN EMDE BOAS, P. Preserving order in a forest in less than logarithmic time. In *FOCS' 75* (1975), pp. 75–84.
- [54] VARGHESE, G., AND LAUCK, T. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1987), SOSP '87, ACM, pp. 25–38.
- [55] WANG, H., AND LIN, B. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems* 24, 7 (2013), 1380–1389.
- [56] WANG, W., FENG, C., LI, B., AND LIANG, B. On the fairness-efficiency tradeoff for packet processing with multiple resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, ACM, pp. 235–248.

A Gradient Queue Correctness

Theorem 1. *The index of the maximum non-empty bucket, N , is $\text{ceil}(b/a)$.*

Proof. We encode the occupancy of buckets by a bit string of length N where zeros represent empty buckets and ones represent nonempty buckets. The value of the bit string is the value of the critical point $x = \frac{b}{a}$ for queue represented by the bit of strings. We prove the theorem by showing an ordering between all bit strings, where the maximum value is N and the minimum value is larger than $N - 1$. The minimum value is when all buckets are nonempty (i.e., all

ones). In that case, $a = \sum_{i=1}^N 2^i$ and $b = \sum_{i=1}^N i2^i$. Note that b is an Arithmetic-Geometric Progression that can be simplified to $N2^{N+1} - (2^{N+1} - 2)$ and a is a Geometric Progression that can be simplified to $2^{N+1} - 2$. Hence, the critical point $x = \frac{N2^{N+1}}{2^{N+1}-2} - 1 = \frac{N}{1-2^{-N}} - 1$ where $\frac{N}{1-2^{-N}} < N + 1$ and $\text{ceil}(x) = N$. The maximum value occurs when only bucket N is nonempty (i.e., all zeros). It is straightforward to show that the critical point is exactly $x = N$. Now, consider any N -bit string, where the N th bit is 1, if we flip one bit from 1 to zero, the value of the critical point increases. It is straight forward to show that $\frac{b-j2^j}{a-2^j} - \frac{b}{a} > 0$, where j is the index of the flipped bit. \square

B Examples of Errors in Approximate Gradient Queue

To better understand the effect of missing elements on the accuracy of the approximate queue, consider the following cases of elements distribution for a maximum priority queue with N buckets:

- Elements are evenly distributed over the queue with frequency $1/\alpha$, which is equivalent to an Exact Gradient Queue with N/α elements,
- $N/2$ elements are present in buckets from 0 to $N/2$ and then a single element is present in bucket indexed $3N/4$, where the concentration of the elements at the beginning of the queue will create an error on the estimation of the index of the maximum element $\varepsilon = \text{ceil}(b/a) + u(\alpha) - 3N/4$. We note that in this case $\varepsilon < 0$ because the estimation of $\text{ceil}(b/a)$ will be closer to the concentration of elements that is pulling the curvature away from $3N/4$. The error in such cases grows proportional to size of the concentration and inversely proportional to the distance between the low concentration and the high concentration.
- All elements are present, which allows the value $\varepsilon = \text{ceil}(b/a) + u(\alpha)$ to be exactly where the maximum element is.

Loom: Flexible and Efficient NIC Packet Scheduling

Brent Stephens
University of Illinois at Chicago

Aditya Akella
University of Wisconsin-Madison

Michael M. Swift
University of Wisconsin-Madison

Abstract

In multi-tenant cloud data centers, operators need to ensure that competing tenants and applications are isolated from each other and fairly share limited network resources. With current NICs, operators must either 1) use a single NIC queue and enforce network policy in software, which incurs high CPU overheads and struggles to drive increasing line-rates (100Gbps), or 2) use multiple NIC queues and accept imperfect isolation and policy enforcement. These problems arise due to inflexible and static NIC packet schedulers and an inefficient OS/NIC interface.

To overcome these limitations, we present Loom, a new NIC design that moves all per-flow scheduling decisions out of the OS and into the NIC. The key aspects of Loom’s design are 1) a new network policy abstraction: restricted directed acyclic graphs (DAGs), 2) a programmable hierarchical packet scheduler, and 3) a new expressive and efficient OS/NIC interface that enables the OS to precisely control how the NIC performs packet scheduling while still ensuring low CPU utilization. Loom is the only multiqueue NIC design that is able to efficiently enforce network policy. We find empirically that Loom lowers latency, increases throughput, and improves fairness for collocated applications and tenants.

1 Introduction

Many large organizations today operate data centers (DCs) with tens to hundreds of thousands of multi-core servers [54, 47, 24]. In virtualized DCs, there are many competing tenants, and operators need to ensure that these tenants are isolated from each other and share resources according to what they are allocated. With VMs and containers, it is currently possible to ensure that tenants fairly share CPU and memory. However, providing network isolation for competing tenants on a server continues to remain a problem [50, 30, 10, 32, 29, 41, 57]. Further, each tenant may run a variety of applications with different performance needs, ranging from latency-sensitive applications such as web services, search, and key-value stores, to throughput-sensitive applications such as Web indexing and batch analytics. It is similarly difficult to ensure that tenants’ applications do not harm each other’s network performance objectives [34].

Network isolation is hard because more functionality is moving to the network interface card (NIC), including packet scheduling. Data center operators are upgrading server NICs from 10Gbps to 100Gbps and beyond. To drive these high line-rates, NICs provide function offloading to reduce CPU load and multiple queues to enable parallel processing of packets [51, 52, 57, 43, 29]. Without these optimizations, applications struggle to drive line-rates. However, with them,

it is not always possible to ensure suitable isolation among competing applications/tenants/flows. System software multiplexes applications and tenants into a small number of queues, and the NIC schedules packets from queues with coarse grain policies. As a result, the on-NIC packet scheduler, and not the OS, is now ultimately responsible for deciding which packets to send and when to send them.

The main goal of our work is to enable rich hierarchies of application-, tenant-, and DC operator-level policies to be realized on NICs while driving high line rates. This helps to simultaneously ensure that applications’ network SLOs can be met, that tenants can be isolated from each other on the data center network, and that operators’ network performance objectives are satisfied.

To solve this problem, we created *Loom*, a new NIC design that moves all per-flow scheduling decisions out of the OS and into the NIC. Loom provides a customizable, hierarchical on-NIC packet scheduler and an efficient OS/NIC interface with a queue per flow. This enables Loom to implement a variety of scheduling algorithms while also enabling the OS to drive line-rates (100Gbps). Loom takes inspiration from recent advances in switch design such as PIFOs and others [12, 55, 15, 56]. These switches utilize a programmable match+action pipeline and generic scheduling queues to support a variety of hierarchical scheduling algorithms. However, NICs are a fundamentally different environment than switches, and these existing approaches are not immediately applicable. Loom addresses the problems that arise when implementing programmable scheduling on NICs.

There are three key components to Loom. First, Loom introduces a new network policy abstraction: restricted directed acyclic graphs (DAGs). Existing abstractions, such as flat traffic classes or strict hierarchies cannot express a common type of end-host network policy. Specifically, hosts may want to rate limit aggregated traffic by destination, such as over intra-DC or WAN links (e.g., BwE [32] or EyeQ [30]). Loom’s new DAG policy abstraction allows for per-destination rate-limits to be expressed independently of the traffic classes used to determine how tenants and applications share bandwidth.

Second, Loom introduces a new programmable packet scheduling hierarchy designed for NICs. In switches, packet headers are available to make scheduling decisions. However, NICs have limited on-NIC SRAM, so they must make scheduling decisions prior to reading packet headers from main memory via DMA. In Loom, the OS enqueues scheduling metadata along with the descriptors used to notify the NIC of new packets, and the NIC only fetches packet content when it is scheduled to be sent.

Third, Loom contributes a new expressive and efficient

OS/NIC interface that utilizes batching and metadata in-lining to reduce the CPU overheads of communicating the network policy to the NIC. Specifically, Loom uses a doorbell queue per core to efficiently aggregate both multiple packets and policy changes into a single PCIe write.

We build a software Loom prototype based on BESS [1], and conduct experiments at both 10Gbps and 40Gbps. We find that Loom is able to enforce complex hierarchical network policies. Also, we show that Loom is able to enforce policies that are not expressible in existing policy abstractions. In contrast, we find that it is not possible to enforce even simple policies with existing multiqueue NICs. Further, we demonstrate that improving network isolation translates into reductions in latency, increases in throughput, and improvements in fairness for competing tenants and applications that are collocated on the same servers. Through an analysis of worst-case behavior, we argue that Loom can still operate at 100Gbps line-rate, even with minimally sized packets. Finally, we evaluate the overheads of our new OS/NIC interface and find that Loom can reduce the number of generated PCIe writes by up to 43x when compared with existing approaches [43, 35].

2 Motivation

Different DC applications and tenants have different performance requirements and service level objectives (SLOs). Ideally, DC operators would be able to ensure that competing applications and tenants are isolated according to some high-level policy, and that application- and tenant-specific SLOs are met [30, 10, 41, 11, 34]. Unfortunately, today, this is not always possible. A key part of the reason is static and inflexible packet scheduling on server NICs today. We elaborate on this issue in the rest of this section.

2.1 High-Level Network Policies

Multi-tenant DCs run different classes of applications, each with their own performance objectives. Different applications can benefit from customized scheduling algorithms [16, 18, 19, 17, 42, 59]. At the same time, cloud providers need to ensure that tenants on the same server fairly share resources, and operators want high infrastructure utilization [34]. Meeting these performance goals amounts to specifying and enforcing a policy that determines how packets from all flows, applications, and tenants on a server are scheduled.

Figure 1 shows a possible high-level network policy for a server. This policy should be enforced no matter how applications access the NIC, such as with SR-IOV that bypasses the hypervisor and/or kernel. Our example is motivated by recent work which demonstrated that there is significant potential for increasing infrastructure utilization by collocating big data applications with latency sensitive applications like key-value stores (KVS) [34]. In this policy graph, the “leaves” shown at the top are packet sources (*e.g.*, different flows). Nodes in this figure determine how packets from different flows are

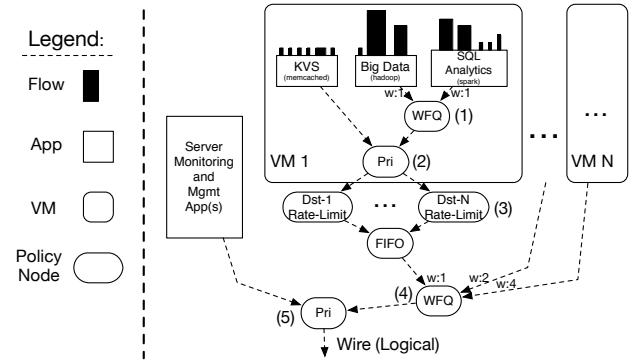


Figure 1: A scheduling hierarchy for a server. Different parts of this hierarchy are specified by different entities (Section 3). The OS is responsible for dynamically enforcing this policy. scheduled. Policies at different levels of the graph come from different entities (*e.g.*, operator, tenant, application).

In VM 1 in Figure 1, first, all the flows from an application are aggregated. Each application can specify the scheduling algorithm used for its own flows. Next, the tenant that owns VM 1 specifies how traffic from its applications is scheduled. Node (1) specifies that the competing flows from Hadoop and Spark should use weighted fair queuing (WFQ) to fairly share bandwidth. Node (2) then specifies that traffic from the key-value store (KVS) should have strict priority over both Hadoop and Spark traffic. Together, these nodes specify a work-conserving policy for how competing traffic from VM 1 should share limited NIC bandwidth.

Additionally, some network operators may want to specify per-destination rate limits that are enforced at hosts. This is useful for ensuring network isolation across intra-DC and wide-area links (*e.g.*, BWE [32] and EyeQ [30]). In the example, an operator specifies separate per-destination rate-limits to all the traffic created by VM 1 with the Dst-1 through Dst-N nodes (3), which may be replicated for other VMs.

Finally, an operator specifies how traffic from competing tenants is scheduled. Node (4) specifies that different tenants should use weighted fair queuing to share bandwidth in proportion to the resources they are allocated ($w:1, w:2, w:4$). Node (5) then specifies that management and hypervisor traffic have strict priority over tenant traffic.

2.2 Issues with Multiqueue NICs

As the above illustrates, it is desirable to schedule traffic leaving a server according to a high-level policy. However, the principle challenge in doing this is in ensuring that applications with competing network objectives (*e.g.*, bandwidth-hungry vs. latency sensitive) do not impact each other’s network performance. This is currently not always possible because OSes need to use many independent NIC transmit queues to drive line-rate¹. As a result, the NIC and not the OS

¹From our own experiments, we have found that, even with batching [7, 2] and TSO, single core throughput in Linux is limited to around 36-40 Gbps.

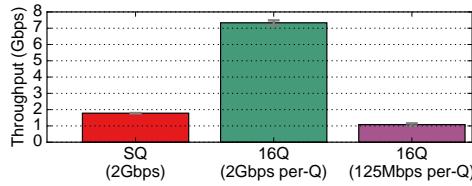


Figure 2: Achieved rate for memcached with 16 threads when trying to enforce a 2 Gbps rate-limit.

is now ultimately responsible for deciding which packets to send and when to send them. Unfortunately, current on-NIC packet schedulers are static, inflexible, and only support a limited number of traffic classes and scheduling algorithms [57].

2.2.1 Background

Modern NICs provide multiple transmit/receive queues and interrupt lines. This reduces overhead by allowing multiple cores to simultaneously send and receive packets. Current best practices are to configure a separate transmit queue for each core. By default, most NICs service queues using simple deficit round robin scheduling [53].

Many NICs support a few additional features for controlling the packet scheduler. NICs with DCB support [21] provide priority scheduling and partition the queues on a NIC into a different pool of traffic for each of 8 DCB priorities. Some advanced NICs allow the OS to set per-queue or per-priority rate-limits [35, 43]. A few NICs support a one-queue-per-flow (QPF) model [43, 35]. These NICs only provide rate limits, deficit round robin [53], and a small number of priorities.

While these scheduling features provide a limited ability to implement fair scheduling, today’s NICs are unable to enforce many other useful, rich network policies.

2.2.2 Inflexible NIC Packet Schedulers

To illustrate the problems with enforcing network policy on multiqueue NICs, we performed a few experiments with Linux and an Intel 10 Gbps NIC [27]. For each transmit queue, we configure Linux *Qdisc* to classify and schedule packets according to a network policy. We find that this can enforce policy when only a single transmit queue (SQ) is used, but that network policy is violated when multiple NIC queues (MQ) or a queue-per-flow (QPF) are used. Although we use XPS [9] to pin transmit queues to cores in these MQ experiments, we see similar results without XPS.

First, Figure 2 illustrates the difficulty in enforcing a rate-limit for all traffic from a single multi-threaded application. In this experiment, the policy is that the sum of all traffic from a 16-threaded memcached application should be rate-limited to 2 Gbps. Because network traffic is not uniformly spread across application threads, it is not possible to configure a per-queue rate limiter. Setting a rate-limit of 2 Gbps per-queue leads to over-utilization, while setting a fair rate-limit of 125 Mbps (2 Gbps/16) leads to under-utilization.

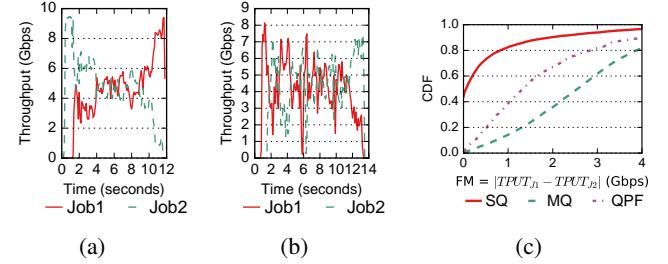


Figure 3: Unfairness when two Spark jobs are allocated equal bandwidth shares. (a) Time series of the achieved throughput for two competing Spark jobs with the MQ configuration. (b) Throughput with the QPF configuration. (c) CDF of the difference in aggregate throughput for two competing jobs.

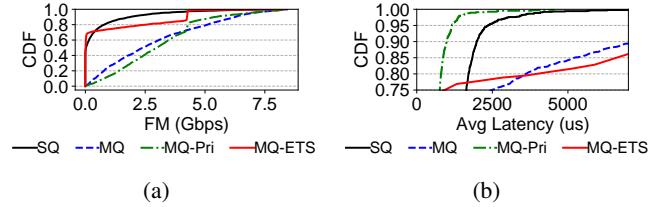


Figure 4: Hybrid approaches that combine DCB and software to enforce a policy. (a) Difference in Spark throughput for two competing tenants. (b) Memcached latency for 32KB values.

Next, Figure 3 shows an experiment where the network policy is that two Spark jobs (each with multiple tasks) should share network bandwidth equally, and each task is allocated its own CPU core. Figure 3a and 3b plot a time series showing unequal throughput achieved by each job for the MQ and QPS configurations, respectively, and Figure 3c plots a CDF of a fairness metric $FM = |TPUT_{J1} - TPUT_{J2}|$, i.e., the difference in achieved throughput for the two jobs when they are both active. Even though each job has the same number of queues, it is still not possible to enforce network policy in MQ because the active flows are not uniformly spread across the cores. Similarly, because the NIC performs per-queue fair scheduling and each job does not have the same number of active flows, QPF does not fairly schedule traffic (Figure 3b). In contrast, the OS can ensure a fair bandwidth distribution when using a single queue (SQ). This is shown in Figure 3c.

With DCB, it is reasonable to try to enforce part of a network policy in hardware, such as prioritizing latency sensitive traffic, and then enforce the rest in software. Figure 4 shows the results from two hybrid approaches. In this experiment, there are two tenants who should fairly share the link bandwidth. The first only runs Spark. The second runs both memcached and Spark. The second tenant’s local policy is that memcached traffic should have strict priority. First, the MQ-Pri approach assigns memcached traffic to a DCB traffic class given strict priority over other traffic, while the traffic from both Spark applications share a traffic class and the cross-tenant fairness policy is enforced by software. Second, the MQ-Fair approach does the opposite and assigns traffic from each tenant to different DCB traffic classes, with each

class being given an equal share of bandwidth, and enforces the priority policy for memcached traffic in software.

Figure 4a shows that MQ-Pri is unable to fairly share bandwidth between the competing tenants but achieves low memcached latency in Figure 4b. Conversely, MQ-Fair has good fairness, but very high latencies. This demonstrates that it is not possible to have an OS or hypervisor both use multiple queues and enforce only part of the network policy in software with the NIC enforcing the rest in hardware.

2.2.3 Inefficient OS/NIC Interfaces

It is possible to use a dynamic approach to enforcing network policy by collapsing it into appropriate per-queue weights, rate limits, and priorities. However, here, scheduling metadata for each queue needs to be updated as flows start and stop.

We find two performance problems to this approach. First, with current NIC interfaces drivers must write a per-queue PCIe doorbell register after adding data to a queue; for small flows across many queues, this can lead to many PCIe writes. Each update requires a separate PCIe write, taking up to 900ns [23]. During this time, the CPU is otherwise unavailable. Furthermore, past research shows that when two cores send data on different queues and write a doorbell for each packet they are unable to achieve a 40 Gbps line rate [46].

Second, updating policy also requires expensive PCIe writes. These overheads are prohibitive, especially if many different flows/queues must be updated simultaneously, and they directly undermine the benefits of offloading packet scheduling to NICs. In this case, it may not be possible to both drive line-rate and update the NIC’s packet scheduler.

To illustrate this problem, we measured the overheads of configuring per-queue rate-limiters on a Mellanox ConnectX-4 NIC [35]. Installing a new limit takes a median of 2.07ms. Setting a queue to use an existing limit takes a median of 64 μ s to complete. We also modified the driver to apply an existing rate asynchronously, and not wait for completion events (and errors). Even so, the median update takes 950ns!

3 DAG Policy Abstraction

Loom is a NIC design with a programmable packet scheduler that offloads the enforcement of high-level network policy. This section describes the design of the Loom policy DAG.

The key aspects of the policy DAG in Loom are: (1) what scheduling algorithms can be expressed at each node in the DAG, (2) how nodes in the DAG can be connected, (3) how different sub-policies from individual applications and tenants are composed, and (4) how traffic is classified into different leaf nodes in the hierarchy.

Node Types. There are two types of non-leaf nodes in the policy: work conserving *scheduling* nodes that determine the relative ordering of different packets, and rate-limiting *shaping* nodes that determine the timing of packets. Every node in the policy is annotated with the specific scheduling or shaping algorithm that should be used. All scheduling algorithms in

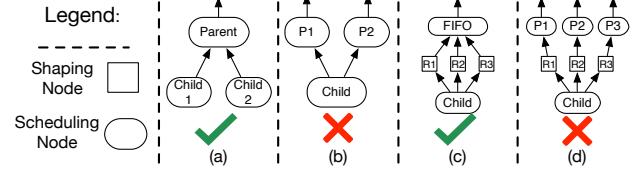


Figure 5: An illustration of the relationships allowed between scheduling and shaping nodes in Loom. A check indicates that the relationship is allowed ((a) and (c)), while an ‘x’ indicates the relationship is forbidden ((b) and (d)).

Loom are expressed by enqueue and dequeue functions that compute a priority (rank). For convenience, Loom provides default implementations of common algorithms including strict priority scheduling (Pri), rate-limits (RL), weighted fair queuing (WFQ), and least slack time first (LSTF) scheduling [38]. However, Loom also allows for installing custom scheduling algorithms at a node.

Restricted DAG Hierarchy. Network policies in Loom are expressed as a restricted DAG like what is shown in Figure 1. The restriction is that the policy graph forms a tree if all shaping nodes are removed. Each shaping node may optionally be a nested set of parallel shaping nodes. Once traffic is aggregated for scheduling, it may only be separated again for shaping. This prevents scheduling nodes from reordering packets that were already ordered by a child while still allowing for separate per-destination and per-path rate-limits.

Ideally, an operator would be able to compose a network policy that specifies both a work-conserving policy for sharing the bandwidth of the local NIC port and separate rate-limit classes to manage bandwidth in the network core and over WAN links. Unfortunately, such policies are not expressible as a tree because once flows for different destinations have been aggregated, they cannot be disaggregated and have separate rate-limits applied. *The Loom policy DAG, however, allows for policies where the traffic classes used to order competing requests (i.e., work-conserving scheduling classes) may be different than the rate-limit classes used to shape traffic.* In contrast, such policies are not expressible via Qdisc.

In more detail, the DAG is restricted in the following way: if a node N ’s parent is a scheduling node, then N may have at most one parent (and outgoing edge). Loom imposes the restriction that a node may have multiple parents (and outgoing edges) *only* if the parents are all shaping nodes. Further, all sibling shaping nodes must share a single FIFO parent. This ensures that scheduling nodes closer to the root do not reorder packets that were already ordered by a child node.

These DAG restrictions are illustrated in Figure 5. Figure 5(a) and Figure 5(b) show that each node may have only one parent if the parent is a scheduling node. Figure 5(c) and Figure 5(d) show that all parallel shaping nodes must share the same parent and child.

Traffic Classification. When packets are sent to the NIC, the OS tags them with the appropriate metadata, and the NIC then uses lookup tables to map the packet to the appropriate

leaf nodes in the policy DAG. This happens before Loom enqueues the packet. Example metadata includes per-socket priorities, socket IDs, process IDs, users, cgroup names, and virtual interface IDs. Policies may also be expressed in terms of network addresses (*e.g.*, IP destination). For example, this is the case with per-destination rate-limits.

Composing Sub-Policies. As Figure 1 illustrates, different sub-graphs of the network policy need to come from different entities. For example, each application may have its own local scheduling algorithms. In Loom, each entity expresses its own local policy as a separate policy DAG. These sub graphs are composed at each level, such as the OS or VM, by attaching the root node of each sub-graph to a leaf node in the next level of the policy. Finally, the VMM passes the final graph to the Loom NIC. When policies are not specified (*e.g.*, when a legacy application does not specify how its flows should be scheduled), Loom uses FIFO packet scheduling.

Policy Limitations. Although our DAG policy abstraction addresses a key limitation with applying rate-limits in prior work [56], there are still some limitations to this abstraction. One limitation is that policies are local to a node. In a cloud data center, it may be desirable to express *network-wide* policies across a cluster of servers [50, 30, 10, 32, 29, 41, 44], *e.g.*, “All of the servers for Tenant A and Tenant B fairly share network bandwidth.” However, such a policy is not directly expressible in our abstraction. It would have to be implemented by (dynamically) mapping this high-level policy onto a collection of per-server and possibly per-switch policies. To enforce such policies, rate-limiters need to be updated whenever either VMs [29] or flows [44] start and stop.

Another limitation is that Loom does not guarantee that the algorithms expressed as part of a policy DAG are efficient or that a policy will map well onto the underlying hardware. Providing default implementations of common algorithms helps overcome this limitation. We also allow the NIC to reject policies when it cannot efficiently compute the enqueue and dequeue functions or when the DAG is too large. This avoids poor NIC performance from inefficient policies.

4 Loom Design

There are two key components to the Loom programmable NIC design: (1) a new scheduling hierarchy, and (2) a new OS/NIC interface that enables the OS to efficiently and precisely control how NICs perform packet scheduling while still ensuring low CPU utilization. This section first provides background on programmable scheduling for switches and then discusses these two components.

4.1 Programmable Scheduling Background

Loom’s design leverages recent advances in switch design for programmable and stateful match+action forwarding pipelines [12, 55, 15], and programmable hierarchical packet scheduling [56]. In these systems, lookup tables arranged in a pipeline map packet headers to logical queues and scheduling

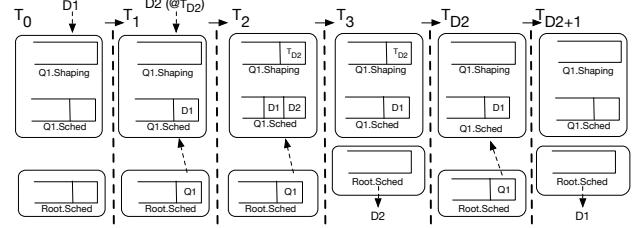


Figure 6: An illustration how a prior PIFO-based scheduler [56] operates that also shows how rate-limiters will be incorrectly applied when only some of the packets at a scheduling node should be rate-limited. Although the packet for D2 should be rate-limited, the packet for D1 is incorrectly rate-limited because D2 has a higher rank than D1.

metadata. For example, the NIC needs the leaf traffic class for the packet. Similarly, WFQ tracks virtual time and the number of transmitted bytes per class. Once scheduling metadata for a packet has been found, packets are enqueued into a logical scheduling hierarchy implemented by a tree of priority queues, also known as push-in first-out queues, or PIFOs.

Scheduling: Because all policies in both these systems and Loom are expressed by computing a priority (rank), many policies can be implemented with a common tree of PIFOs. Different scheduling algorithms are implemented by changing how the rank is computed (*e.g.*, priority, deadline, virtual time, slack, etc.). In general, within a single node, this model is sufficient to emulate any scheduling algorithm [38].

A scheduling algorithm is expressed through an enqueue function, a dequeue function, and the state maintained across function calls. The enqueue function runs when packet is enqueued at a node in the DAG. Using the local state and metadata associated with a packet, this function then computes and returns a rank (priority). On dequeue, the node returns the lowest rank packet and a dequeue function updates local state. For example, an implementation of fair queuing would update a global virtual time in the dequeue function. Shaping algorithms behave similarly, but they compute a transmit time as a rank. This enables Loom to implement a wide range of scheduling and shaping algorithms.

For example, strict priority scheduling is implemented by computing a priority on enqueue. WFQ is implemented by computing a virtual time for the packet on enqueue and updating a global (to the node) virtual time on dequeue. Both strict and token bucket rate-limits can be implemented by computing a wall clock transmit time on either enqueue or dequeue. Other scheduling algorithms can also be implemented (*e.g.*, for LSTF [38], we compute a slack time on enqueue).

Enqueuing and dequeuing from different nodes in the hierarchy operates as follows. First, a match+action pipeline finds all necessary metadata from the leaf traffic class. Then, enqueueing starts at the leaf node. The enqueue function for this node is used to compute a rank and enqueue a pointer to the packet. Traversing the hierarchy from the leaf node to

the root, the local enqueue function at each node computes a rank, and the NIC enqueues a pointer to the appropriate child node at each parent.

The leftmost picture in Figure 6 illustrates this behavior. When a new packet arrives at T_0 , it is first pushed into Q_1 , and then a pointer to Q_1 is pushed into the Root PIFO. Conversely, when the transmit port is ready to transmit a packet, it starts by dequeuing the element at the head of the root of the PIFO tree, which will be a pointer to a child PIFO. After running the dequeue function, dequeuing then continues to follow this pointer chain (e.g., Q_1 and Q_2 could have other pointers enqueued) until a leaf node is reached and the original packet is ultimately dequeued.

Rate Limiting: To implement rate-limiting, each node in the prior PIFO design uses both a shaping queue for rate-limiting and a scheduling queue for ordering packets. When the enqueue function at a node determines that a packet should be rate-limited, its transmission time is pushed into the shaping queue for the node and the packet is added to the scheduling queue. However, no pointers are enqueued at subsequent parent nodes. Then, only once the computed transmit time has expired, is the packet enqueued at the rest of the parent nodes and it will eventually be scheduled.

Unfortunately, this design cannot apply rate-limits to only some of the packets sharing a PIFO node in the hierarchy (e.g., those going to specific sets of destinations).

Figure 6 illustrates this behavior and the problem that it causes. In this example, there are two queues in the hierarchy, the Root queue, and Q_1 . At time T_1 , a packet for D1 that should not be rate-limited is enqueued in the tree. Then at T_2 a high-priority packet for D2 that should be rate-limited until time T_{D2} arrives. Because of its priority, it is ordered in Q_1 ahead of D1, but it is not enqueued higher up the tree (per the PIFO rate limiting description above). Then, because there is still a reference to Q_1 at the root, the next time a packet is dequeued from the root, the packet at the head of the scheduling queue at Q_1 will be returned. However, in this case, the packet for D2 is dequeued because its rank in the scheduling queue at Q_1 for D2 is higher than the rank for D1. At this point, there are no pointers to Q_1 at the root, so D1 will not be sent. Later, at time T_{D2} a pointer to Q_1 will be enqueued at the root and D1 will eventually be sent. In effect, the packet for D1 is rate-limited when it should not be, while the packet for D2 is not rate-limited when it should be!

This problem occurs because the existing design does not distinguish rate-limited from non-rate limited packets at a node. For the same reason, having different rate limits at a node can also lead to packets being sent at the wrong time.

4.2 Programmable Scheduling for NICs

NICs provide a different environment than switches, so existing approaches for programmable scheduling on switches are not immediately usable on NICs. Switches have buffer space to hold complete packet contents. In contrast, NICs perform

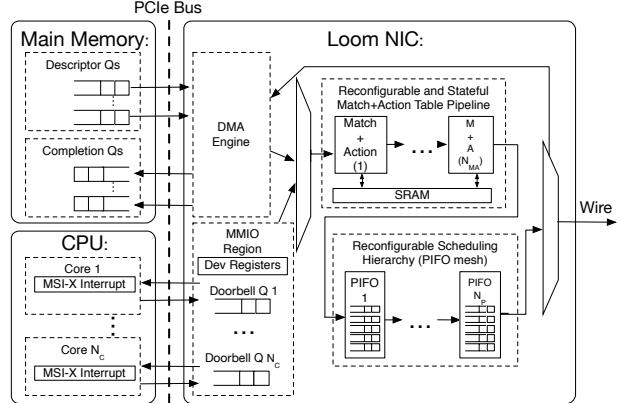


Figure 7: A breakdown of the different components of Loom. Note that descriptor queues are per-flow while the doorbell queues are per-core.

scheduling over packet descriptors, and defer reading packets from main memory as long as possible to keep memory requirements small. Reading just the headers is infeasible because it would require two DMA operations per packet. Instead, NICs must perform scheduling in advance of reading a packet from main memory.

4.2.1 Scheduling Operations

To overcome this challenge, Loom relies on the OS to communicate any necessary scheduling metadata to the NIC. This metadata may be explicitly set for each packet by including scheduling information in-line with transmit descriptors and doorbell writes. This metadata may also be implicitly determined by the queue the OS uses to send a packet.

Figure 7 illustrates the design of our Loom prototype. The OS assigns each flow its own descriptor queue. After writing descriptors, the OS then rings an on-NIC doorbell. Doorbells are then parsed and processed by an on-NIC stateful match+action pipeline. This pipeline is used to lookup scheduling metadata (Section 4.3.2) and per-queue DMA state (e.g., descriptor ring buffer address). Next, pointers to descriptor queues are enqueued in a PIFO tree that is used to schedule competing packet transmissions. At this step, all scheduling ranks are computed using information from the OS. When the request is dequeued, it is sent to the DMA engine. Once the DMA engine has read the necessary descriptor and packet data from main memory, packets are then parsed and processed by the match+action pipeline before being transmitted. For example, packet headers are updated as needed to support features like segmentation offload (TSO) and network virtualization in the match+action pipeline.

Regardless of whether metadata is explicitly in-lined with descriptors and doorbells or implicitly associated with queues, with this division of labor, it is possible to schedule the processing of different transmit requests from different transmit queues without already having access to the packet data. The

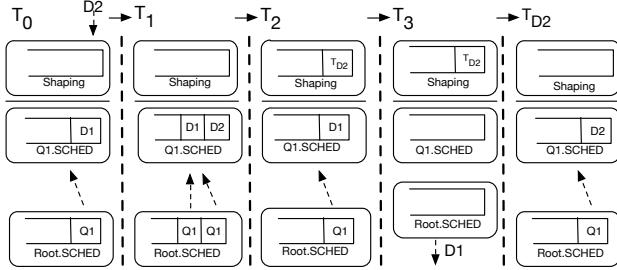


Figure 8: An illustration how Loom enforces different rate-limits for different packets sharing the same PIFO. When a packet is dequeued before its computed transmission time, it is instead pushed into a separate global shaping queue. After its transmission time, it is re-enqueued in the hierarchy.

trade-off between these two approaches is that including metadata in descriptors and doorbells increases the descriptor and doorbell size while using implicit per-queue metadata consumes additional on-NIC SRAM.

4.2.2 DAG Rate-Limiting

To implement the rate-limiter DAG abstraction in Loom, we created a new design for rate-limiting with PIFOs that allows multiple rate-limiting classes to be applied at the same PIFO node. Crucially, this design uses a *global* shaping queue to implement rate-limits instead of per-node shaping queues. This overcomes the limitations of previous designs that cannot support separate scheduling and shaping queues.

When no rate-limits are exceeded, packets are enqueued into the PIFO hierarchy as if no rate-limit were applied. Instead of proactively enforcing rate-limits, transmission times are computed and stored. Then, on dequeue, if the transmission time is in the past, dequeuing continues as normal.

However, if the transmission time is in the future on dequeue, the request is enqueued in a separate global shaping queue. Because all shaping is done with respect to wall-clock time, independent shaping queues are not needed for each node in the hierarchy. Next, after the computed (rate-limited) transmission time expires, the request is then re-enqueued according to its previous ranks in the scheduling queues, after which it will be transmitted according to the policy as normal. This ensures that subsequent high-priority traffic can bypass rate-limited traffic. In contrast with previous designs [56], Loom can correctly enforce rate-limits even when different transmission times are computed for different requests otherwise sharing the same scheduling queue in the hierarchy.

Figure 8 illustrates this design with an example. At time T0, a packet for D2 shows up that should be rate limited until time T_{D2} . However, at time T1, it is initially only enqueued in the scheduling queues. Then, when it is dequeued from the root PIFO at T2, it is re-enqueued in a global shaping queue until its reaches its transmission time of T_{D2} . At this point, it is then re-inserted into the hierarchy.

4.2.3 Line-rate (100Gbps) Operation

An apparent trade-off of this design is that it can increase the number of enqueue and dequeue operations performed per packet. In the worst case, a packet in Loom may be enqueued and dequeued two times from the PIFO hierarchy. However, this does not prevent Loom from operating at line-rate. Although our design, like prior work [56], uses PIFOs that only support a single enqueue and dequeue operation per cycle, a Loom NIC does not need to schedule a unique packet every cycle to still be able to forward at line-rate.

Consider the following analysis: Assume a 100Gbps NIC that operates at a frequency of 1GHz. The OS sends minimum-sized 64-byte packets, and each packet requires the worst-case 2 enq/deqs per packet. Even in this case, the NIC can still schedule a packet every 2ns. Achieving line-rate only needs to send a packet every 5.12ns. Further, we note that this worst-case analysis is far from the common case. For example, packets are often bigger, and only some rate limited packets need multiple enq/deqs.

We employ other optimizations to further reduce operations. One such optimization is that we include a flag in scheduling metadata that indicates whether a rate-limit class is currently being rate-limited. This allows for packets to be immediately enqueued in the shaping queue, reducing the average number of operations per packet. Second, to bound the number of subsequent dequeue operations that do not yield a transmittable packet, we limit the number of outstanding packets from each traffic class.

However, deep policy hierarchies can require multiple average cycles per dequeue, preventing 100Gbps operation. While different stages of the hierarchy can be pipelined, if the depth of the hierarchy exceeds the number of PIFO blocks, again it may not be possible to guarantee an average rate of 1 enq/deq per cycle. We expect policies in practice to be not arbitrarily deep. For example, the graph in Figure 1 is quite rich, yet it uses ≤ 5 PIFOs. A deeper policy graph is unlikely to need more than 10 PIFOs. Thus, we believe we can support 100Gbps for practical policies.

4.3 OS/NIC Interface

Loom introduces an efficient and expressive OS/NIC interface for notifying the NIC of new segments and communicating network policy updates. Loom minimizes the *total number of PCIe* writes needed by the OS. This is accomplished through two mechanisms: First, the OS uses batched doorbells to notify the NIC of new segments. Second, scheduling and metadata updates are passed in-line with packet descriptors to avoid generating additional PCIe writes.

4.3.1 Batched Doorbells

To efficiently support a large number of transmit queues, doorbells are separated from (per-flow) transmit descriptor queues in Loom (Figure 7). Instead of writing to a separate

doorbell for each queue, in Loom, doorbells are written indirectly through *doorbell queues*. When the driver needs to notify the NIC about new segments from different flows it writes a batch of doorbell descriptors (16-bit integers) to a per-core doorbell queue. These queues are stored on the NIC, and, with write-combining, 32 doorbells (1 cacheline) can be written in a single PCIe write. This design builds on top of the fact that modern OSes already send segments to the NIC in batches [7, 2]. Unlike existing NICs that generate a PCIe write per queue to ring per-queue doorbells, Loom only generates a single PCIe write per batch.

Figure 7 illustrates this design by showing the different types of queues that are used in Loom for OS/NIC communication and where they are located in memory. The control flow when the OS needs to send a batch of packets in Loom is as follows: first, descriptors for individual packets and configuration updates (described below) are written to per-flow descriptor queues. These are fast writes to main memory. Additionally, descriptors for all of the packets in the batch that belong to the same flow are created in a single write. Next, the OS writes a batch of doorbell descriptors to a doorbell queue. The NIC detects the write, finds the queues from the doorbell descriptors, and processes the segments in the batch.

Although Loom relies on batching to reduce the number of PCIe writes, it still provides low latency. If there are fewer packets than can fit in a batch, the OS and/or application need not wait before ringing a doorbell. If new packets are generated in the interim, they will be part of the next batch. Similarly, due to per-flow queues and on-NIC scheduling, high-priority packets in later batches are not blocked by packets in earlier batches.

4.3.2 Scheduling Metadata

In Loom, the OS provides the NIC with the necessary metadata to compute scheduling ranks before the NIC has read segment data. This is accomplished in one of two ways. First, when the metadata applies to all the traffic in a flow (i.e., transmit descriptor queue), scheduling metadata updates are sent through doorbell descriptors. Second, when the policy applies to individual segments, scheduling metadata is passed in segment descriptors. By passing metadata inline, there is no extra OS overhead to communicate new scheduling information when new flows arrive. Furthermore, Loom saves per-queue metadata in on-NIC lookup tables, which allows for future segments to be sent without any additional metadata. All of the information needed by the NIC (*e.g.*, the address of the queue in main memory) is already implicitly associated with the queue the segment is enqueued in.

Because the match+action pipeline processes packets sequentially, dynamically reconfiguring the scheduling hierarchy is straightforward. When processing a segment or doorbell, each stage in the pipeline saves provided configuration values (if present) to local SRAM as the stage processes the segment or doorbell. These values can include scheduling

metadata, shared memory regions, algorithms, and traffic classes. With Loom’s efficient doorbells this can be accomplished without additional PCIe writes.

4.4 Discussion

There are a wide spectrum of different NIC designs, ranging from NICs with specialized ASICs [3, 35], NICs with some FPGAs [36, 33, 22], NICs built from tiled or network processors [39, 13, 14, 58, 37], and purely virtual NICs [48, 1]. We believe that Loom’s design is applicable to all of these different NIC types. While the PIFO abstraction is well suited for efficient hardware implementation (ASIC) [56], recent work has proved that all local scheduling algorithms are expressible with PIFOs [38]. Thus, the Loom design can be applied even to NICs with more flexible architectures. Similarly, all NIC types benefit from an efficient OS/NIC interface. Even in a virtual NIC, the use of doorbell queues reduces the number of memory regions that need to be polled by the NIC’s backend.

Self-virtualizing NICs (*e.g.*, SR-IOV) can reduce the CPU overheads of virtual networking. Supporting SR-IOV with Loom is straightforward. Instead of providing per-core doorbells, Loom provides per-VCPU doorbells. VMs are then allocated their own doorbells and events. For security, queue creation/initialization is controlled by the hypervisor. However, common case operations like updating scheduling metadata are handled directly by the guest.

Similarly, Loom is also compatible with kernel bypass frameworks like RDMA, DPDK [28], and netmap [45]. Like netmap [45], we require such applications to call into the kernel to ring doorbells and configure queues. Even with a large number of kernel-bypass queues and applications, this still allows for efficient doorbell batching and scheduling of competing doorbells from different traffic classes.

5 Implementation

There are two major aspects to our implementation. The first is a compiler for the Loom policy DAG, and the second is a prototype Loom NIC. Loom is open source and available at <https://github.com/bestephe/loom>.

For the policy DAG compiler, we made modifications to an existing compiler for scheduling trees written in Domino [5]. Scheduling policies are expressed as a DAG, and each node contains both an enqueue and a dequeue function. The Loom network policy is expressed in a restricted subset of C++. The output of the compiler is C++ code that combines generic PIFOs with custom scheduling algorithms.

We created a software prototype Loom NIC using the Berkeley Extensible Software Switch (BESS [1], formerly SoftNIC [25]). To interface with the OS, BESS loads a kernel module that registers a new Ethernet adapter. The driver for this adapter communicates with a backend userspace application that emulates the functionality of the Loom NIC hardware. The driver and backend in BESS communicate through descriptor queues implemented with shared memory.

We modified the BESS kernel driver to implement Loom’s OS/NIC interface. We replaced BESS’s per-core descriptor queues with per-flow descriptor queues and per-core doorbell queues. Also, we modified the descriptor format and the driver so that OS and flow-level metadata is included along with transmit descriptors and doorbells. We also identified and fixed a problem in the driver that caused excessive packet loss when transmitting packets.

To implement the compiled Loom policy in the NIC backend, we extended a C++ implementation of a pipeline of PIFOs [6]. We modified the shaping queues used for rate-limiting to support our DAG abstraction (Section 4). We also modified the model to support functions called on dequeue.

6 Methodology

For a baseline in our experiments, we compare our Loom prototype against three different BESS configurations. The first uses a single transmit descriptor queue for all packets (SQ), the second uses a descriptor queue per-core (MQ), and the third uses one queue per-flow (QPF) with round-robin scheduling between competing queues. Although SQ is not able to always drive line-rate in our experiments, it provides a baseline where the OS is able to enforce network policy. In contrast, while MQ and QPF are able to drive line-rate, they provide a baseline that is not able to enforce network policy.

In both the SQ and MQ configurations, when possible we configure Qdisc to try to enforce the network policy in software (Loom policies that use DAG rate-limits are not expressible in the Qdisc tree hierarchy). Because Qdisc does not allow for packet scheduling to be configured based on container, process, or socket ids, we rely on IP addresses and ports to express the network policy when using Qdisc.

We perform two different types of experiments to evaluate Loom. First, we use network-bound applications to profile the network behavior of our implementation. Specifically, we use the iperf3 [4] program to saturate network throughput and the sockperf [8] application to measure end-to-end latency. In these experiments, applications from different tenants are isolated by placing them in their own containers (cgroups). Second, we run real data center applications. We try to capture the performance of both latency and bandwidth sensitive applications. We use memcached for a latency-sensitive application. As a bandwidth-sensitive application, we use Spark with the TeraSort benchmark to perform a 25 GB shuffle. We compute throughput over a 50ms window.

We evaluate both types of experiments by sending data between two servers on CloudLab [20]. In order to stress Loom’s packet processing, we use a 1500 byte MTU for all experiments, and do not use large segments, either for transmit segmentation offload (TSO) or large receive offload (LRO). The small packet size increases the number of packets per second that must be scheduled by a single core.

The software NIC in every experiment uses one core for packet transmission and one core for reception. This implies

that the CPU utilization of the software prototype is 200%.

We use two different experiment configurations. In the first configuration (HW1), we use two servers with Intel X520 10GbE NICs, two Intel E5-2683 v3 14-core CPUs (Haswell) and 256GB of memory. In this configuration, all servers use the Loom prototype NIC for both sending and receiving.

In the second configuration (HW2), we use two servers with 40 Gbps Mellanox ConnectX-3 NICs, two Intel E5-2650 8-core CPUs (Haswell) and 64GB of memory. These experiments are asymmetric: only the server *sending* uses the Loom prototype NIC. The BESS SoftNIC [25], upon which our prototype is based, is currently unable to *receive* at 40 Gbps with a 1500 byte MTU with a single receive core (although it can receive at 40 Gbps with 9000B jumbo frames)². To demonstrate that Loom can schedule and *transmit* packets at 40 Gbps with a 1500 MTU, we perform asymmetric experiments where the receiving host uses a vanilla driver.

7 Evaluation

First, we evaluate the ability of our Loom prototype to enforce network policies. Next, we evaluate the efficiency of our new OS/NIC interface. Finally, we evaluate the performance of two different DC applications when used with Loom.

7.1 Policy Enforcement

Our first experiments demonstrate that our Loom prototype can isolate tenants. We perform the following experiment: There are five active tenants (T0-T4). Fair queuing (FQ) with equal shares is used to share bandwidth between tenants. The first tenant runs a single latency-sensitive application (sockperf). Then, in successive two-second intervals, another tenant starts running a bandwidth-hungry application. Starting with Tenant T1, each tenant i starts 4^i flows (4-256). Two seconds after the final tenant starts, the applications for each tenant successively finish at two-second intervals.

Figure 9 shows the throughput achieved over time by different tenants in the single queue (SQ), multi-queue (MQ), queue-per-flow (QPF), and Loom configurations. Figure 9a shows that SQ is able to approximately enforce this policy of tenant bandwidth isolation. However, because only a single transmit queue is used, SQ is not able to fully drive the 10 Gbps line-rate. Next, Figure 9b shows that MQ is not able to enforce this policy. Instead, each successive tenant is able to use more than its fair share of bandwidth because it has more flows. As shown in Figure 9c QPF also leads to unfairness because the NIC’s per-queue deficit round robin scheduling favors the tenant with the most flows. In contrast, Figure 9d shows that Loom is able to more precisely enforce the network policy than SQ while also driving full line-rate.

Next, Figure 10 shows the 90th percentile latency observed in each 250ms interval by tenant T0 in the same experiment

²The limit is not intrinsic to Loom. It arises from a BESS design choice to copy packet data in the kernel and not in the backend, and we are working with the BESS authors to address the problem.

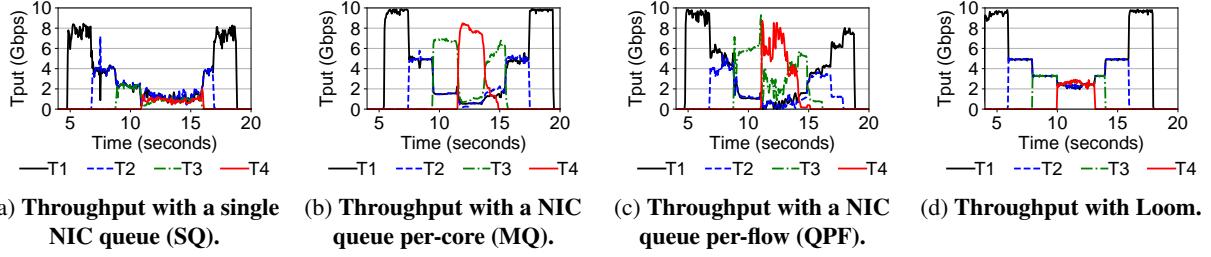


Figure 9: Aggregate throughput achieved by different tenants when each tenant should receive an equal share of bandwidth.

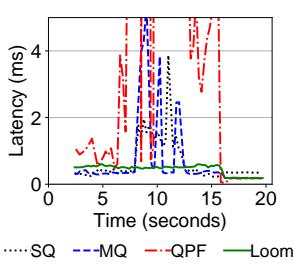


Figure 10: 90th Percentile latency over time for a latency-sensitive application (T0) that is configured to fairly share bandwidth with up to four other bandwidth hungry applications (T1-T4) for SQ, MQ, 10 Gbps and 40 Gbps network, and Loom.

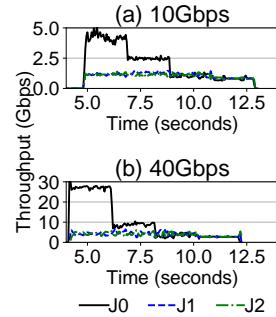


Figure 11: Throughput for the different jobs from tenant T1 when only jobs J1 and J2 are subject to a combined 2.5 Gbps and 10 Gbps rate-limit on a 10 Gbps and 40 Gbps network, respectively.

for the SQ, MQ, QPF, and Loom configurations. Because T0 uses less than its share of bandwidth, its packets are prioritized ahead of those from the other tenants. When the number of flows is small, Figure 10 shows that all of the approaches are able to provide similar latency. However, as the number of flows increases, only Loom is able to provide consistent low latency. QPF incurs the highest latency of any configuration because it uses more queues than SQ and MQ.

We compared the CPU utilization of Loom to existing designs: SQ, MQ, and QPF³. Loom performs similarly to MQ and QPF because there is little CPU scheduling work and no coordination across cores. In contrast, SQ has 25-300% higher CPU utilization due to complicated packet scheduling in software and contention for a single queue.

We study Loom’s ability to enforce shaping policies where the rate-limit classes are different than the traffic classes used for scheduling. This experiment uses the same tenants as the previous one, but replaces Tenant T1. It now runs three competing jobs: J0 for Dst-1, and J1 and J2 for Dst-2. The traffic for Dst-2 is rate-limited to 10 Gbps, and for Dst-1 is not rate-limited. As before, every two seconds, an additional tenant starts flows, but in this case all stop at the same time.

The results of this experiment are shown in Figure 11.

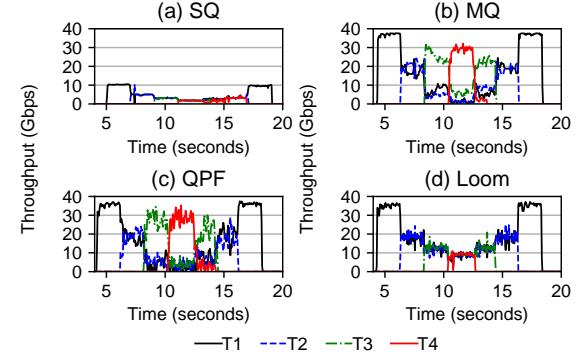


Figure 12: Aggregate throughput achieved on a 40 Gbps network when the network policy is that each tenant should receive an equal share of bandwidth.⁴

When the only active flows are from T1 (5-7 seconds), together, jobs J1 and J2 are appropriately rate-limited to 10 Gbps while J0 receives the remaining bandwidth. Further, according to the scheduling policy, jobs J1 and J2 fairly share the bandwidth available to their rate-limit class. However, after Tenant T2 and T3 have started flows at 9 seconds, tenant T1’s fair share is 3.3 Gbps and the fair share of jobs J1 and J2 are less than 10 Gbps, we see that Jobs J1, J2, and J3 fairly share tenant T1’s bandwidth.

From Figure 11, we see that Loom is able to enforce the policy DAG. In contrast, this policy is not expressible with the static Linux Qdisc or the scheduling tree policies used by prior work if the number of jobs is dynamic [56]. In a strict hierarchy, the rate limit must be applied before applying fair queuing across jobs, but the relative weight of the shaped and not shaped traffic changes with each new job.

Performance at 40 Gbps and Beyond: To demonstrate that our prototype scales with increasing line rates, we repeated some of the previous experiments at 40 Gbps line-rates with the small change that tenant T1 starts 16 flows instead of 4. These results are shown in Figure 12 and Figure 11(b). In both of these experiments, the trends stay the same. Figure 12 shows that SQ is only able to drive roughly ~10 Gbps due to contention for the queue. In contrast, while MQ, QPF, and Loom are able to drive ~37 Gbps, only Loom is able to

³Note that this CPU utilization is *in addition* to that used by our software prototype, which uses two cores.

⁴Our current prototype has a small throughput hit (1.3% median and 3.4% average) when compared with MQ that we are working on improving.

Number of Tenants	SENIC/QPF write/s	Loom write/s	% Reduction
32 Tenants	150K	128K	19.2%
64 Tenants	243K	205K	15.8%
96 Tenants	303K	263K	13.2%

Table 1: The median number of PCIe writes per second generated by SENIC/QPF and Loom and the percent reduction in number of PCIe writes with Loom’s batched doorbells.

enforce network policy. Similarly, Figure 11(b) shows that Loom can rate-limit traffic while also driving a 40 Gbps link.

Based on these results, we believe that it should be possible to continue to scale Loom with increasing line-rates, *e.g.*, 100 and 200 GbE. These results demonstrate that a single CPU core can schedule 1500B packets at 40 Gbps, and we expect that a hardware NIC implementation should be able to exceed this performance. NPU-based NICs can parallelize computing ranks in enqueue and dequeue functions. Also, prior work has already demonstrated that custom hardware can enqueue/dequeue a billion packets per-second [56].

7.2 OS/NIC Interface

To evaluate Loom’s batched doorbells, we estimate the difference in number of PCIe writes using Loom’s batched doorbells and a NIC that has a queue-per-core with unbatched doorbells, such as SENIC [43] and the ConnectX-4. We instrument Linux to count how many doorbells Loom would use, and how many extra doorbell rings are needed without batching. With Loom, there is one PCIe write to the doorbell queue for a batch of segments added to any queue by a single core; this is approximately the same Linux behavior with a queue-per-core, which adds a batch of segments from multiple flows to a single queue. We calculate the number of PCIe doorbell writes in Loom as the number of `skbs` for which the `xmit_more` flag is false, indicating the OS has no more data to enqueue. Without batching, additional PCIe doorbell writes are needed for each different flow. We calculate the number of additional writes as the number of times a segment in a batch comes from a different flow than the preceding packet, indicating it would be sent on a different queue.

We evaluate the interface efficiency with a variable number of tenants each with 16 active flows sending traffic with iperf3. We note that this experiment is a best case scenario because it uses long-lived flows that are able to benefit from segmentation offload, which reduces the total number of segments sent to the NIC. Table 1 shows total number of writes per second and the percent reduction in number of PCIe writes with Loom when compared with current approaches. In these experiments, the total number of writes generated by the one flow per-queue approach varies from 150K–303K writes per second. Even with this benign workload, batched doorbells can reduce the number of writes by up to 19.2%. As part of future work, we are working to improve batching in Linux, which we expect can lead to even further reductions.

Next, because the benefits of batched doorbells are work-

Line-rate	64KB Batch		256KB Batch	
	SENIC/QPF write/s	Loom write/s	SENIC/QPF write/s	Loom write/s
10 Gbps	833K	19K	833K	4.8K
40 Gbps	3.3M	76K	3.3M	19K
100 Gbps	8.3M	191K	8.3M	48K

Table 2: The number of PCIe writes per second generated by SENIC/QPF and Loom given a worst-case traffic pattern.

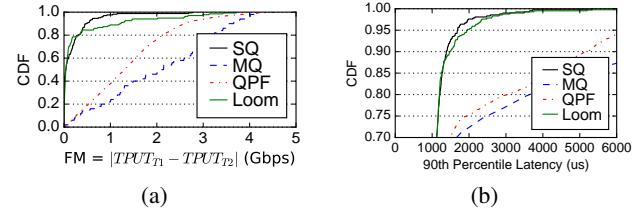


Figure 13: Hierarchical policy performance. (a) CDF of fairness for Tenants 1 and 2. (b) CDF of memcached latency

load dependent, we also performed a worst case analysis to estimate the benefits of Loom’s approach. We approximate an RPC-style workload with many clients and assume that each 1500B packet in a batch is sent from a different flow. In this scenario, existing approaches such as SENIC that use a queue per flow generate a write per packet, while Loom generates one write per *batch*. Table 2 shows computed write rates of SENIC/QPF and Loom for different batch sizes and line-rates. At 40 Gbps with 64KB batches, SENIC generates 3.3M writes/second, while Loom causes only 76,000, a 43x reduction! For context, prior work found that the Intel XL710 40 Gbps NIC cannot drive line-rate when the OS generates more than 3.3M writes per second (*i.e.*, a single doorbell is rung per 1500B packet) [46].

This analysis demonstrates that existing one flow per-queue approaches will have difficulty driving increasing line-rates for workloads with many short flows. Additionally, this analysis shows that Loom can reduce PCIe overheads by increasing the batch size, which is not possible with existing one flow per-queue approaches under some workloads.

Finally, we also note that current approaches would generate even more PCIe writes than we estimate as flows start and stop to update scheduling metadata and the network policy. In contrast, configuration updates in Loom do not generate any additional updates as they are inlined with data.

7.3 DC Applications

We also used applications that are not always network bound to evaluate Loom. We show that Loom addresses the problems associated with using multiqueue NICs demonstrated in Section 2. We omit most graphs due to lack of space.

Memcached rate limit. First, we found that despite using multiple queues, Loom can accurately rate-limit memcached traffic to 2 Gbps in the experiment from Figure 2.

Equal shares. When two Spark jobs are active (*cf.* Figure 3), Loom is able to fairly schedule traffic from the competing jobs, even as different flows start and stop. In contrast,

MQ and QPF lead to job-level unfairness. SQ provides the same fairness as Loom, but is not able to drive line rate.

Priority. We evaluated Loom’s ability to prioritize flows from one Spark job over those of another. We find that Loom can enforce this policy while MQ cannot. With Loom, the average and 90th percentile job completion times for the high-priority Job are 51.8s and 59.2s, respectively. In contrast, with MQ, the average and 90th percentile job completion times are 69.38s and 151.6s, respectively.

Multi-workload priority. We conducted an experiment where a single tenant is running two multi-threaded and multi-process applications with different performance requirements (memcached and Spark). The policy is that all of the traffic from memcached should have strict priority over that from Spark. We measure the impact that traffic from Spark has on memcached latency by examining the 90th percentile latency each 1 second interval. Both SQ and Loom isolate memcached from the Spark job, but MQ and QPF have 2.5X worse latencies in at least 25% of the intervals. Thus, even though each application thread has its own core in MQ and each flow has its own queue in QPF, neither is able to ensure that memcached response times are not impacted by a competing Spark job.

Hierarchical policy. Figure 13 shows that Loom can enforce hierarchical policies that cannot be enforced by DCB. In this experiment, tenants T1 and T2 are allocated equal shares of bandwidth. Tenant T1 runs memcached and Spark with the local policy that memcached has strict priority over Spark. Tenant T2 only runs Spark. T1’s memcached latency should not be impacted by T1’s Spark job, and each tenant should receive at most their fair share of throughput when they are both active. Figure 13a plots per-tenant unfairness, and Figure 13b plots the 90th percentile memcached request latency in each one second interval during the experiment. These figures show that Loom provides both isolation and fairness, while MQ and QPF cannot.

8 Related Work

SENIC [43] is related to Loom because it provides each flow with its own transmit descriptor queue and uses a doorbell FIFO to notify the NIC of new segments. However, SENIC only provides a single shared doorbell FIFO that requires synchronization across multiple cores. Further, SENIC writes individual 16B queue updates, and SENIC does not support programmable policies.

The ConnectX-4 [35] (CX-4) NIC is also similar to Loom. It supports many hundreds of thousands of descriptor queues and uses per-CPU registers for doorbell updates. However, individual doorbells and configuration updates need to be written to the NIC one at a time. The CX-4 also does not support hierarchical or programmable policies.

Titan [57] and MQFQ [26] can both improve fairness on multiqueue NICs. However, neither can enforce traffic priorities or hierarchical policies, and Titan incurs high scheduling

update overheads with dynamic workloads.

Carousel [48] and PSPAT [46] both use multiple dedicated CPU cores to onload packet scheduling. However, both approaches still need to use multiple independent cores to drive 100 Gbps line-rates. Although they both use as few cores as possible, this does not solve the problems associated with multiqueue NICs. Further, Carousel can only express rate-limits, and, while PSPAT can express the same policies as Qdisc, it is still not able to express some policies that use per-destination rate-limits.

Like PSPAT [46], Neugebauer *et. al* [40] also benchmark PCIe performance. They find that PCIe can significantly impact the performance of end host networking. Both of these projects motivate the need for an efficient OS/NIC interface.

Eiffel [49] improves the efficiency of software packet scheduling, and, like Loom, Eiffel also addresses some shortcomings of PIFOs. Eiffel and Loom are complementary. Eiffel would benefit from Loom if it is ever not able to drive line-rate with a single core, and Eiffel can be used to enforce Loom policy sub-trees in software.

FlexNIC [31] introduces new interface for *receiving* packets on a programmable NIC. Loom’s focus on packet transmission makes it complementary to FlexNIC.

Silo [29] also finds that multiqueue NICs prevent per-destination rate-limits from being enforced. However, Silo cannot ensure that competing applications share NIC bandwidth according to a high-level policy and has difficulty driving high line-rates because it only uses a single NIC queue.

9 Conclusions

DCs need to enforce a high-level hierarchical network policy. However, with today’s multiqueue NICs, this is not possible. To address this deficiency, we created *Loom*, a new NIC design that moves all per-flow scheduling decisions out of the OS and into the NIC. Loom contributes 1) a new policy DAG abstraction that can express per-destination rate-limits independent of scheduling traffic classes, 2) a new flexible programmable scheduling hierarchy designed for NICs, and 3) a new expressive and efficient OS/NIC interface. Together, these aspects enable Loom to completely offload all packet scheduling to the NIC with low CPU overhead while still driving increasing line-rates (100Gbps). For collocated tenants’ applications, these benefits translate into reductions in latency, increases in throughput, and improvements in fairness for competing tenants and applications.

Acknowledgments: We would like to thank the anonymous reviewers and our shepherd Alex C. Snoeren for their thoughtful feedback. Brent Stephens, Aditya Akella, and Michael Swift are supported in part by the NSF grant CNS-1717039.

References

- [1] BESS: Berkeley Extensible Software Switch. <https://github.com/NetSys/bess>.

- [2] Bulk network packet transmission. <https://lwn.net/Articles/615238/>.
- [3] Intel ethernet switch fm10000 datasheet. <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-multi-host-controller-fm10000-family-datasheet.pdf>.
- [4] iperf3: Documentation. <http://software.es.net/iperf/>.
- [5] pifo-compiler: Compiler for packet scheduling programs. <https://github.com/programmable-scheduling/pifo-compiler>.
- [6] pifo-machine: C++ reference implementation for push-in first-out queue. <https://github.com/programmable-scheduling/pifo-machine>.
- [7] qdisc: bulk dequeue support. <https://lwn.net/Articles/615240/>.
- [8] sockperf: Network benchmarking utility. <https://github.com/Mellanox/sockperf>.
- [9] xps: Transmit packet steering. <https://lwn.net/Articles/412062/>.
- [10] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *SIGCOMM* (2011).
- [11] BALLANI, H., JANG, K., KARAGIANNIS, T., KIM, C., GUNAWARDENA, D., AND O'SHEA, G. Chatty tenants and the cloud network sharing problem. In *NSDI* (2013).
- [12] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F. A., AND HOROWITZ, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *SIGCOMM* (2013).
- [13] CAVIUM CORPORATION. Cavium CN63XX-NIC10E. http://cavium.com/Intelligent_Network_Adapters_CN63XX_NIC10E.html.
- [14] CAVIUM CORPORATION. Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf.
- [15] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching. In *SIGCOMM* (2017).
- [16] CHOWDHURY, M., AND STOICA, I. Coflow: A networking abstraction for cluster applications. In *HotNets* (2012).
- [17] CHOWDHURY, M., AND STOICA, I. Efficient coflow scheduling without prior knowledge. In *SIGCOMM* (2015).
- [18] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra. In *SIGCOMM* (2011), ACM.
- [19] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with Varys. In *SIGCOMM* (2014).
- [20] Cloudlab. <http://cloudlab.us/>.
- [21] DATA CENTER BRIDGING TASK GROUP. <http://www.ieee802.org/1/pages/dcbridges.html>.
- [22] EXABLAZE. ExaNIC V5P. <https://exablaze.com/exanic-v5p>.
- [23] FLAJSLIK, M., AND ROSENBLUM, M. Network interface design for low latency request-response protocols. In *USENIX ATC* (2013).
- [24] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM* (2015).
- [25] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. SoftNIC: A software NIC to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>.
- [26] HEDAYATI, M., SCOTT, M. L., SHEN, K., , AND MARTY, M. Multi-queue fair queuing. Tech. Rep. UR CSD / 1005, Department of Computer Science, University of Rochester, October 2018. <http://hdl.handle.net/1802/34380>.
- [27] INTEL. Intel 82599 10 GbE controller datasheet. <http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf>.
- [28] Intel Data Plane Development Kit. <http://dpdk.org>.
- [29] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *SIGCOMM* (2015).
- [30] JEYAKUMAR, V., ALIZADEH, M., MAZIERES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. EyeQ: Practical network performance isolation at the edge. In *NSDI* (2013).
- [31] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with FlexNIC. In *ASPLOS* (2016).
- [32] KUMAR, A., JAIN, S., NAIK, U., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM* (2015).
- [33] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. K. Just say NO to paxos overhead: Replacing consensus with network ordering. In *OSDI* (2016).
- [34] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In *ISCA* (2015).
- [35] MELLANOX TECHNOLOGIES. ConnectX-4 VPI. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.
- [36] MELLANOX TECHNOLOGIES. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.
- [37] MELLANOX TECHNOLOGIES. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.
- [38] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *NSDI* (2016).
- [39] NETRONOME. NFP-6xxx flow processor. <https://netronome.com/product/nfp-6xxx/>.
- [40] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding PCIe performance for end host networking. In *SIGCOMM* (2018).
- [41] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the network in cloud computing. In *SIGCOMM* (2012).
- [42] QIU, Z., STEIN, C., AND ZHONG, Y. Minimizing the total weighted completion time of coflows in datacenter networks. In *SPAA* (2015).
- [43] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for end-host rate limiting. In *NSDI* (2014).
- [44] RAGHAVAN, B., VISHWANATH, K., RAMABHADRAN, S., YOCUM, K., AND SNOEREN, A. C. Cloud control with distributed rate limiting. In *SIGCOMM* (2007).

- [45] RIZZO, L. netmap: A novel framework for fast packet I/O. In *USENIX ATC* (2012).
- [46] RIZZO, L., VALENTE, P., LETTIERI, G., AND MAFFIONE, V. PSPAT: Software packet scheduling at hardware speed. <http://info.ipt.unipi.it/~luigi/pspat/>. Preprint; accessed May 31 2017.
- [47] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).
- [48] SAEED, A., DUKKIPATI, N., VALANCUS, V., LAM, T., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable traffic shaping at end-hosts. In *SIGCOMM* (2017).
- [49] SAEED, A., ZHAO, Y., DUKKIPATI, N., AMMAR, M., ZEGUR, E., A KHALED HARRAS, AND VAHDAT, A. Eiffel: Efficient and flexible software packet scheduling. In *NSDI* (2019).
- [50] SHIEH, A., KANDULA, S., GREENBERG, A., AND KIM, C. Sharing the data center network. In *NSDI* (2011).
- [51] SHINDE, P., KAUFMANN, A., KOURTIS, K., AND ROSCOE, T. Modeling NICs with Unicorn. In *PLOS* (2013).
- [52] SHINDE, P., KAUFMANN, A., ROSCOE, T., AND KAESTLE, S. We need to talk about NICs. In *HotOS* (2013).
- [53] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *SIGCOMM* (1995).
- [54] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *SIGCOMM* (2015).
- [55] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., McKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM* (2016).
- [56] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND McKEOWN, N. Programmable packet scheduling at line rate. In *SIGCOMM* (2016).
- [57] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue NICs. In *USENIX ATC* (2017).
- [58] TILER. Tile Processor Architecture Overview For the TILE-GX Series. <http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf>.
- [59] ZHAO, Y., CHEN, K., BAI, W., TIAN, C., GENG, Y., ZHANG, Y., LI, D., AND WANG, S. RAPIER: Integrating routing and scheduling for coflow-aware data center networks. In *INFOCOM* (2015).

Exploiting Commutativity For Practical Fast Replication

Seo Jin Park
Stanford University

John Ousterhout
Stanford University

Abstract

Traditional approaches to replication require client requests to be ordered before making them durable by copying them to replicas. As a result, clients must wait for two round-trip times (RTTs) before updates complete. In this paper, we show that this entanglement of ordering and durability is unnecessary for strong consistency. The Consistent Unordered Replication Protocol (CURP) allows clients to replicate requests that have not yet been ordered, as long as they are commutative. This strategy allows most operations to complete in 1 RTT (the same as an unreplicated system). We implemented CURP in the Redis and RAMCloud storage systems. In RAMCloud, CURP improved write latency by $\sim 2x$ ($14 \mu s \rightarrow 7.1 \mu s$) and write throughput by $4x$. Compared to unreplicated RAMCloud, CURP’s latency overhead for 3-way replication is just $1 \mu s$ ($6.1 \mu s$ vs $7.1 \mu s$). CURP transformed a non-durable Redis cache into a consistent and durable storage system with only a small performance overhead.

1 Introduction

Fault-tolerant systems rely on replication to mask individual failures. To ensure that an operation is durable, it cannot be considered complete until it has been properly replicated. Replication introduces a significant overhead because it requires round-trip communication to one or more additional servers. Within a datacenter, replication can easily double the latency for operations in comparison to an unreplicated system; in geo-replicated environments the cost of replication can be even greater.

In principle, the cost of replication could be reduced or eliminated if replication could be overlapped with the execution of the operation. In practice, however, this is difficult to do. Executing an operation typically establishes an ordering between that operation and other concurrent operations, and the order must survive crashes if the system is to provide consistent behavior. If replication happens in parallel with execution, different replicas may record different orders for the operations, which can result in inconsistent behavior after crashes. As a result, most systems perform ordering before replication: a client first sends an operation to a server that orders the operation (and usually executes it as well); then that server issues replication requests to other servers, ensuring a consistent ordering among replicas. As a result, the minimum latency for an operation is two round-trip times (RTTs). This problem affects all systems that provide consistency and replication, including both primary-backup approaches and consensus approaches.

Consistent Unordered Replication Protocol (CURP) reduces the overhead for replication by taking advantage of the fact that most operations are commutative, so their order of execution doesn’t matter. CURP supplements a system’s existing replication mechanism with a lightweight form of replication without ordering based on *witnesses*. A client replicates each operation to one or more witnesses in parallel with sending the request to the primary server; the primary can then execute the operation and return to the client without waiting for normal replication, which happens asynchronously. This allows operations to complete in 1 RTT, as long as all witnessed-but-not-yet-replicated operations are commutative. Non-commutative operations still require 2 RTTs. If the primary crashes, information from witnesses is combined with that from the normal replicas to re-create a consistent server state.

CURP can be easily applied to most existing systems using primary-backup replication. Changes required by CURP are not intrusive, and it works with any kind of backup mechanism (e.g. state machine replication [31], file writes to network replicated drives [1], or scattered replication [26]). This is important since most high-performance systems optimize their backup mechanisms, and we don’t want to lose those optimizations (e.g. CURP can be used with RAMCloud without sacrificing its fast crash recovery [26]).

To show its performance benefits and applicability, we implemented CURP in two NoSQL storage systems: Redis [30] and RAMCloud [27]. Redis is generally used as a non-durable cache due to its very expensive durability mechanism. By applying CURP to Redis, we were able to provide durability and consistency with similar performance to the non-durable Redis. For RAMCloud, CURP reduced write latency by half (only a $1 \mu s$ penalty relative to RAMCloud without replication) and increased throughput by $3.8x$ without compromising consistency.

Overall, CURP is the first replication protocol that completes linearizable deterministic update operations within 1 RTT without special networking. Instead of relying on special network devices or properties for fast replication [21, 28, 22, 12, 3], CURP exploits commutativity, and it can be used for any system where commutativity of client requests can be checked just from operation parameters (CURP cannot use state-dependent commutativity). Even when compared to Speculative Paxos or NOPaxos (which require a special network topology and special network switches), CURP is faster since client request packets do not need to detour to get ordered by a networking device (NOPaxos has an overhead of $16 \mu s$, but CURP only increased latency by $1 \mu s$).

2 Separating Durability from Ordering

Replication protocols supporting concurrent clients have combined the job of ordering client requests consistently among replicas and the job of ensuring the durability of operations. This entanglement causes update operations to take 2 RTTs.

Replication protocols must typically guarantee the following two properties:

- **Consistent Ordering:** if a replica completes operation a before b , no client in the system should see the effects of b without the effects of a .
- **Durability:** once its completion has been externalized to an application, an executed operation must survive crashes.

To achieve both consistent ordering and durability, current replication protocols need 2 RTTs. For example, in master-backup (a.k.a. primary-backup) replication, client requests are always routed to a master replica, which serializes requests from different clients. As part of executing an operation, the master replicates either the client request itself or the result of the execution to backup replicas; then the master responds back to clients. This entire process takes 2 RTTs total: 1 from clients to masters and another RTT for masters to replicate data to backups in parallel.

Consensus protocols with strong leaders (e.g. Multi-Paxos [17] or Raft [25]) also require 2 RTTs for update operations. Clients route their requests to the current leader replica, which serializes the requests into its operation log. To ensure durability and consistent ordering of the client requests, the leader replicates its operation log to a majority of replicas, and then it executes the operation and replies back to clients with the results. In consequence, consensus protocols with strong leaders also require 2 RTTs for updates: 1 RTT from clients to leaders and another RTT for leaders to replicate the operation log to other replicas.

Fast Paxos [19] and Generalized Paxos [18] reduced the latency of replicated updates from 2 RTTs to 1.5 RTT by allowing clients to optimistically replicate requests with presumed ordering. Although their leaders don't serialize client requests by themselves, leaders must still wait for a majority of replicas to durably agree on the ordering of the requests before executing them. This extra waiting adds 0.5 RTT overhead. (See §B.3 for a detailed explanation on why they cannot achieve 1 RTT.)

Network-Ordered Paxos [21] and Speculative Paxos [28] achieve near 1 RTT latency for updates by using special networking to ensure that all replicas receive requests in the same order. However, since they require special networking hardware, it is difficult to deploy them in practice. Also, they can't achieve the minimum possible latency since client requests detour to a common root-layer switch (or a middlebox).

The key idea of CURP is to separate durability and consistent ordering, so update operations can be done in 1 RTT in the normal case. Instead of replicating totally ordered

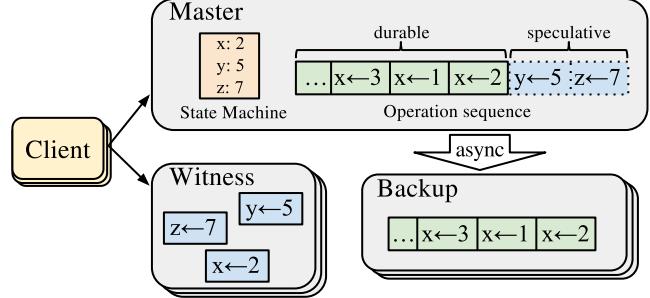


Figure 1: CURP clients directly replicate to witnesses. Witnesses only guarantee durability without ordering. Backups hold data that includes ordering information. Witnesses are temporary storage to ensure durability until operations are replicated to backups.

operations in 2 RTTs, CURP achieves durability without ordering and uses the commutativity of operations to defer agreement on operation order.

To achieve durability in 1 RTT, CURP clients directly record their requests in temporary storage, called a *witness*, without serializing them through masters. As shown in Figure 1, witnesses do not carry ordering information, so clients can directly record operations into witnesses in parallel with sending operations to masters so that all requests will finish in 1 RTT. In addition to the unordered replication to witnesses, masters still replicate ordered data to backups, but do so asynchronously after sending the execution results back to the clients. Since clients directly make their operations durable through witnesses, masters can reply to clients as soon as they execute the operations without waiting for permanent replication to backups. If a master crashes, the client requests recorded in witnesses are replayed to recover any operations that were not replicated to backups. A client can then complete an update operation and reveal the result returned from the master if it successfully recorded the request in witnesses (optimistic fast path: 1 RTT), or after waiting for the master to replicate to backups (slow path: 2 RTT).

CURP's approach introduces two threats to consistency: ordering and duplication. The first problem is that the order in which requests are replayed after a server crash may not match the order in which the master processed those requests. CURP uses commutativity to solve this problem: all of the *unsynced* requests (those that a client considers complete, but which have not been replicated to backups) must be commutative. Given this restriction, the order of replay will have no visible impact on system behavior. Specifically, a witness only accepts and saves an operation if it is commutative with every other operation currently stored by that witness (e.g., writes to different objects). In addition, a master will only execute client operations speculatively (by responding before replication is complete), if that operation is commutative with every other unsynced operation. If either a witness or master finds that a new operation is not commutative, the client must ask the master to sync with backups. This adds an extra RTT of latency, but it flushes all of the speculative operations.

The second problem introduced by CURP is duplication.

When a master crashes, it may have completed the replication of one or more operations that are recorded by witnesses. Any completed operations will be re-executed during replay from witnesses. Thus there must be a mechanism to detect and filter out these re-executions. The problem of re-executions is not unique to CURP, and it can happen in distributed systems for a variety of other reasons. There exist mechanisms to filter out duplicate executions, such as RIFL [20], and they can be applied to CURP as well.

We can apply the idea of separating ordering and durability to both consensus-based replicated state machines (RSM) and primary-backup, but this paper focuses on primary-backup since it is more critical for application performance. Fault-tolerant large-scale high-performance systems are mostly configured with a single cluster coordinator replicated by consensus and many data servers using primary-backup (e.g. Chubby [6], ZooKeeper [15], Raft [25] are used for cluster coordinators in GFS [13], HDFS [32], and RAMCloud [27]). The cluster coordinators are used to prevent split-brains for data servers, and operations to the cluster coordinators (e.g. change of master node during recovery) are infrequent and less latency sensitive. On the other hand, operations to data servers (e.g. insert, replace, etc) directly impact application performance, so the rest of this paper will focus on the CURP protocol for primary-backup, which is the main replication technique for data servers. In §B.2, we sketch how the same technique can be applied for consensus.

3 CURP Protocol

CURP is a new replication protocol that allows clients to complete linearizable updates within 1 RTT. Masters in CURP speculatively execute and respond to clients before the replication to backups has completed. To ensure the durability of the speculatively completed updates, clients multicast update operations to witnesses. To preserve linearizability, witnesses and masters enforce commutativity among operations that are not fully replicated to backups.

3.1 Architecture and Model

CURP provides the same guarantee as current primary-backup protocols: it provides linearizability to client requests in spite of failures. CURP assumes a fail-stop model and does not handle byzantine faults. As in typical primary-backup replications, it uses a total of $f + 1$ replicas composed of 1 master and f backups, where f is the number of replicas that can fail without loss of availability. In addition to that, it uses f witnesses to ensure durability of updates even before replications to backups are completed. As shown in Figure 2, witnesses may fail independently and may be co-hosted with backups. CURP remains available (i.e. immediately recoverable) despite up to f failures, but will still be strongly consistent even if all replicas fail.

Throughout the paper, we assume that witnesses are separate from backups. This allows CURP to be applied to a wide range of existing replicated systems without modi-

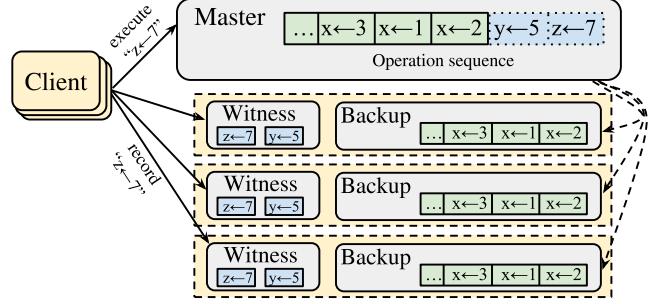


Figure 2: CURP architecture for $f = 3$ fault tolerance.

fying their specialized backup mechanisms. For example, CURP can be applied to a system which uses file writes to network replicated drives as a backup mechanism, where the use of witnesses will improve latency while retaining its special backup mechanism. However, when designing new systems, witnesses may be combined with backups for extra performance benefits. (See §B.1 for details.)

CURP makes no assumptions about the network. It operates correctly even with networks that are asynchronous (no bound on message delay) and unreliable (messages can be dropped). Thus, it can achieve 1 RTT updates on replicated systems in any environment, unlike other alternative solutions. (For example, Speculative Paxos [28] and Network-Ordered Paxos [21] require special networking hardware and cannot be used for geo-replication.)

3.2 Normal Operation

3.2.1 Client

Client interaction with masters is generally the same as it would be without CURP. Clients send update RPC requests to masters. If a client cannot receive a response, it retries the update RPC. If the master crashes, the client may retry the RPC with a different server.

For 1 RTT updates, masters return to clients before replication to backups. To ensure durability, clients directly record their requests to *witnesses* concurrently while waiting for responses from masters. Once all f witnesses have accepted the requests, clients are assured that the requests will survive master crashes, so clients complete the operations with the results returned from masters.

If a client cannot record in all f witnesses (due to failures or rejections by witnesses), the client cannot complete an update operation in 1 RTT. To ensure the durability of the operation, the client must wait for replication to backups by sending a **sync** RPC to the master. Upon receiving **sync** RPCs, the master ensures the operation is replicated to backups before returning to the client. This waiting for sync increases the operation latency to 2 RTTs in most cases and up to 3 RTT in the worst case where the master hasn't started syncing until it receives a **sync** RPC from a client. If there is no response to the **sync** RPC (indicating the master might have crashed), the client restarts the entire process; it resends the update RPC to a new master and tries to record the RPC request in witnesses of the new master.

3.2.2 Witness

Witnesses support 3 basic operations: they record operations in response to client requests, hold the operations until explicitly told to drop by masters, and provide the saved operations during recovery.

Once a witness accepts a **record** RPC for an operation, it guarantees the durability of the operation until told that the operation is safe to drop. To be safe from power failures, witnesses store their data in non-volatile memory (such as flash-backed DRAM). This is feasible since a witness needs only a small amount of space to temporarily hold recent client requests. Similar techniques are used in strongly-consistent low-latency storage systems, such as RAMCloud [27].

A witness accepts a new **record** RPC from a client only if the new operation is commutative with all operations that are currently saved in the witness. If the new request doesn't commute with one of the existing requests, the witness must reject the record RPC since the witness has no way to order the two noncommutative operations consistent with the execution order in masters. For example, if a witness already accepted " $x \leftarrow 1$ ", it cannot accept " $x \leftarrow 5$ ".

Witnesses must be able to determine whether operations are commutative or not just from the operation parameters. For example, in key-value stores, witnesses can exploit the fact that operations on different keys are commutative. In some cases, it is difficult to determine whether two operations commute each other. SQL UPDATE is an example; it is impossible to determine the commutativity of "UPDATE T SET rate = 40 WHERE level = 3" and "UPDATE T SET rate = rate + 10 WHERE dept = SDE" just from the requests themselves. To determine the commutativity of the two updates, we must run them with real data. Thus, witnesses cannot be used for operations whose commutativity depends on the system state. In addition to the case explained, determining commutativity can be more subtle for complex systems, such as DBMS with triggers and views.

Each of f witnesses operates independently; witnesses need not agree on either ordering or durability of operations. In an asynchronous network, record RPCs may arrive at witnesses in different order, which can cause witnesses to accept and reject different sets of operations. However, this does not endanger consistency. First, as mentioned in §3.2.1, a client can proceed without waiting for sync to backups only if all f witnesses accepted its record RPCs. Second, requests in each witness are required to be commutative independently, and only one witness is selected and used during recovery (described in §3.3).

3.2.3 Master

The role of masters in CURP is similar to their role in traditional primary-backup replications. Masters in CURP receive, serialize, and execute all update RPC requests from clients. If an executed operation updates the system state, the master synchronizes (*syncs*) its current state with backups by replicating the updated value or the log of ordered operations.

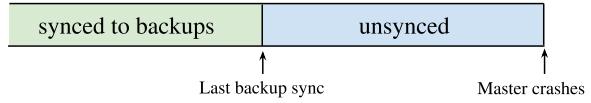


Figure 3: Sequence of executed operations in the crashed master.

Unlike traditional primary-backup replication, masters in CURP generally respond back to clients *before* syncing to backups, so that clients can receive the results of update RPCs within 1 RTT. We call this *speculative* execution since the execution may be lost if masters crash. Also, we call the operations that were speculatively executed but not yet replicated to backups *unsynced* operations. As shown in Figure 3, all unsynced operations are contiguous at the tail of the masters' execution history.

To prevent inconsistency, a master must sync before responding if the operation is not commutative with any existing unsynced operations. If a master responds for a non-commutative operation before syncing, the result returned to the client may become inconsistent if the master crashes. This is because the later operation might complete and its result could be externalized (because it was recorded to witnesses) while the earlier operation might not survive the crash (because, for example, its client crashed before recording it to witnesses). For example, if a master speculatively executes " $x \leftarrow 2$ " and "read x ", the returned read value, 2, will not be valid if the master crashes and loses " $x \leftarrow 2$ ". To prevent such unsafe dependencies, masters enforce commutativity among unsynced operations; this ensures that all results returned to clients will be valid as long as they are recorded in witnesses.

If an operation is synced because of a conflict, the master tags its result as "synced" in the response; so, even if the witnesses rejected the operation, the client doesn't need to send a **sync** RPC and can complete the operation in 2 RTTs.

3.3 Recovery

CURP recovers from a master's crash in two phases: (1) restoration from backups and (2) replay from witnesses. First, the new master restores data from one of the backups, using the same mechanism it would have used in the absence of CURP.

Once all data from backups have been restored, the new master replays the requests recorded in witnesses. The new master picks any available witness. If none of the f witnesses are reachable, the new master must wait. After picking the witness to recover from, the new master first asks it to stop accepting more operations; this prevents clients from erroneously completing update operations after recording them in a stale witness whose requests will not be retried anymore. After making the selected witness immutable, the new master retrieves the requests recorded in the witness. Since all requests in a single witness are guaranteed to be commutative, the new master can execute them in any order. After replaying all requests recorded in the selected witness, the new master finalizes the recovery by syncing to backups and resetting witnesses for the new master (or assigning a new set of witnesses). Then the new master can start accepting

client requests again.

Some of the requests in the selected witness may have been executed and replicated to backups before the master crashed, so the replay of such requests will result in re-execution of already executed operations. Duplicate executions of the requests can violate linearizability [20].

To avoid duplicate executions of the requests that are already replicated to backups, CURP relies on exactly-once semantics provided by RIFL [20], which detects already executed client requests and avoids their re-execution. Such mechanisms for exactly-once semantics are already necessary to achieve linearizability for distributed systems [20], so CURP does not introduce a new requirement. In RIFL, clients assign a unique ID to each RPC; servers save the IDs and results of completed requests and use them to detect and answer duplicate requests. The IDs and results are durably preserved with updated objects in an atomic fashion. (If a system replicates client requests to backups instead of just updated values, providing atomic durability becomes trivial since each request already contains its ID and its result can be obtained from its replay during recovery.)

This recovery protocol together with the normal operation protocol described in §3.2 guarantee linearizability of client operations even with server failures. An informal proof of correctness can be found in appendix §A.

3.4 Garbage Collection

To limit memory usage in witnesses and reduce possible rejections due to commutativity violations, witnesses must discard requests as soon as possible. Witnesses can drop the recorded client requests after masters make their outcomes durable in backups. In CURP, masters send garbage collection RPCs for the synced updates to their witnesses. The garbage collection RPCs are batched: each RPC lists several operations that are now durable (using RPC IDs provided by RIFL [20]).

3.5 Reconfigurations

This section discusses three cases of reconfiguration: recovery of a crashed backup, recovery of a crashed witness, and data migration for load balancing. First, CURP doesn't change the way to handle backup failures, so a system can just recover a failed backup as it would without CURP.

Second, if a witness crashes or becomes non-responsive, the system configuration manager (the owner of all cluster configurations) decommissions the crashed witness and assigns a new witness for the master; then it notifies the master of the new witness list. When the master receives the notification, it syncs to backups to ensure f -fault tolerance and responds back to the configuration manager that it is now safe to recover from the new witness. After this point, clients can use f witnesses again to record operations. However, CURP does not push the new list of witnesses to clients. Since clients cache the list of witnesses, clients may still use the decommissioned witness (if it was temporarily disconnected, the witness will continue to accept record RPCs from clients).

This endangers consistency since requests recorded in the old witnesses will not be replayed during recovery.

To prevent clients from completing an unsynced update operation with just recording to old witnesses, CURP maintains a monotonically increasing integer, *WitnessListVersion*, for each master. A master's *WitnessListVersion* is incremented every time the witness configuration for the master is updated, and the master is notified of the new version along with the new witness list. Clients obtain the *WitnessListVersion* when they fetch the witness list from the configuration manager. On all update requests, clients include the *WitnessListVersion*, so that masters can detect and return errors if the clients used wrong witnesses; if they receive errors, the clients fetch new witness lists and retry the updates. This ensures that clients' update operations can never complete without syncing to backups or recording to current witnesses.

Third, for load balancing, a master can split its data into two partitions and migrate a partition to a different master. Migrations usually happen in two steps: a prepare step of copying data while servicing requests and a final step which stops servicing (to ensure that all recent operations are copied) and changes configuration. To simplify the protocol changes from the base primary-backup protocol, CURP masters sync to backups and reset witnesses before the final step of migration, so witnesses are completely ruled out of migration protocols. After the migration is completed, some clients may send updates on the migrated partition to the old master and old witnesses; the old master will reject and tell the client to fetch the new master information (this is the same as without CURP); then the client will fetch the new master and its witness information and retry the update. Meanwhile, the requests on the migrated partition can be accidentally recorded in the old witness, but this does not cause safety issues; masters will ignore such requests during the replay phase of recovery by the filtering mechanism used to reject requests on not owned partitions during normal operations.

3.6 Read Operations

CURP handles read operations in a fashion similar to that of primary-backup replication. Since such operations don't modify system state, clients can directly read from masters, and neither clients nor masters replicate read-only operations to witnesses or backups.

However, even for read operations, a master must check whether a read operation commutes with all currently *unsynced* operations as discussed in §3.2.3. If the read operation conflicts with some unsynced update operations, the master must sync the unsynced updates to backups before responding for the read.

3.7 Consistent Reads from Backups

In primary-backup replication, clients normally issue all read operations to the master. However, some systems allow reading from backups because it reduces the load on masters and can provide better latency in a geo-replicated environment (clients can read from a backup in the same

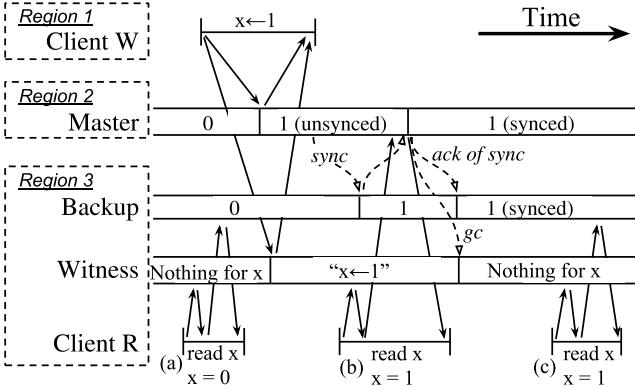


Figure 4: Three cases of reading the value of x from a backup replica while another client is changing the value of x from 0 to 1: (a) client R first confirms that a nearby witness has no request that is not commutative with “read x ,” so the client directly reads the value of x from a nearby backup. (b) Just after client W completes “ $x \leftarrow 1$ ”, client R starts another read. Client R finds that there is a non-commutative request saved in a nearby witness, so it must read from a remote master to guarantee consistency. (c) After syncing “ $x \leftarrow 1$ ” to the backup, the master garbage collected the update request from witnesses and acknowledged the full sync to backups. Now, client R sees no non-commutative requests in the witness and can complete read operation by reading from the nearby backup.

region to avoid wide-area RTTs). However, naively reading from backups can violate linearizability since updates in CURP can complete before syncing to backups.

To avoid reading stale values, clients in CURP use a nearby witness (possibly colocated with a backup) to check whether the value read from a nearby backup is up to date. To perform a consistent read, a client must first ask a witness whether the read operation commutes with the operations currently saved in the witness (as shown in Figure 4). If it commutes, the client is assured that the value read from a backup will be up to date. If it doesn’t commute (i.e. the witness retains a write request on the key being read), the value read from a backup might be stale. In this case, the client must read from the master.

In addition, we assume that the underlying primary-backup replication mechanism prevents backups from returning new values that are not yet fully synced to all backups. Such mechanism is necessary even before applying CURP since returning a new value prematurely can cause inconsistency; even if a value is replicated to some of backups, the value may get lost if the master crashes and a new master recovers from a backup that didn’t receive the new value. A simple solution for this problem is that backups don’t allow reading values that are not yet fully replicated to all backups. For backups to track which values are fully replicated and ok to be read, a master can piggyback the acknowledgements for successful previous syncs when it sends **sync** requests to backups. When a client tries to read a value that is not known to be yet fully replicated, the backup can wait for full replication or ask the client to retry.

Thanks to the safety mechanisms discussed above, CURP still guarantees linearizability. With a concurrent update, reading from backups could violate linearizability in two ways: (1) a read sees the old value after the completion of the update operation and (2) a read sees the old value

after another read returned the new value. The first issue is prevented by checking a witness before reading from a backup. Since clients can complete an update operation only if it is synced to *all* backups or recorded in *all* witnesses, a reader will either see a noncommutative update request in the witness being checked or find the new value from the backup; thus, it is impossible for a read after an update to return the old value. For the second issue, since both a master and backups delay reads of a new value until it is fully replicated to all backups, it is impossible to read an older value after another client reads the new value.

4 Implementation on NoSQL Storage

This section describes how to implement CURP on low-latency NoSQL storage systems that use primary-backup replications. With the emergence of large-scale Web services, NoSQL storage systems became very popular (e.g. Redis [30], RAMCloud [27], DynamoDB [33] and MongoDB [7]), and they range from simple key-value stores to more fully featured stores supporting secondary indexing and multi-object transactions; so, improving their performance using CURP is an important problem with a broad impact.

The most important piece missing from §3 to implement CURP is how to efficiently detect commutativity violations. Fortunately for NoSQL systems, CURP can use *primary keys* to efficiently check the commutativity of operations. NoSQL systems store data as a collection of objects, which are identified by *primary keys*. Most update operations in NoSQL specify the affected object with its primary key (or a list of primary keys), and update operations are commutative if they modify disjoint sets of objects. The rest of this section describes an implementation of CURP that exploits this efficient commutativity check.

4.1 Life of A Witness

Witnesses have two modes of operation: normal and recovery. In each mode, witnesses service a subset of operations listed in Figure 5. When it receives a **start** RPC, a witness starts its life for a master in normal mode, in which the witness is allowed to mutate its collection of saved requests. In normal mode, the witness services **record** RPCs for client requests targeted to the master for which the witness was configured by **start**; by accepting only requests for the correct master, CURP prevents clients from recording to incorrect witnesses. Also, witnesses drop their saved client requests as they receive **gc** RPCs from masters.

A witness irreversibly switches to a recovery mode once it receives a **getRecoveryData** RPC. In recovery mode, mutations on the saved requests are prohibited; witnesses reject all **record** RPCs and only service **getRecoveryData** or **end**. As a recovery is completed and the witness becomes useless, the cluster coordinator may send **end** to free up the resources, so that the witness server can start another life for a different master.

CLIENT TO WITNESS:
$\text{record}(\text{masterID}, \text{list of } \langle \text{keyHash}, \text{rpcId}, \text{request} \rangle) \rightarrow \{\text{ACCEPTED or REJECTED}\}$
Saves the client <i>request</i> (with <i>rpcId</i>) of an update on <i>keyHashes</i> . Returns whether the witness could accommodate and save the request.
MASTER TO WITNESS:
$\text{gc}(\text{list of } \langle \text{keyHash}, \text{rpcId} \rangle) \rightarrow \text{list of } \text{request}$
Drops the saved requests with the given <i>keyHashes</i> and <i>rpcIds</i> . Returns stale requests that haven't been garbage collected for a long time.
getRecoveryData() $\rightarrow \text{list of } \text{request}$
Returns all requests saved for a particular crashed master.
CLUSTER COORDINATOR TO WITNESS:
$\text{start}(\text{masterId}) \rightarrow \{\text{SUCCESS or FAIL}\}$
Start a witness instance for the given master, and return SUCCESS. If the server fails to create the instance, FAIL is returned.
end() $\rightarrow \text{NULL}$
This witness is decommissioned. Destruct itself.

Figure 5: The APIs of Witnesses.

4.2 Data Structure of Witnesses

Witnesses are designed to minimize the CPU cycles spent for handling **record** RPCs. For client requests mutating a single object, recording to a witness is similar to inserting in a set-associative cache; a record operation finds a set of slots using a hash of the object's primary key and writes the given request to an available slot in the set. To enforce commutativity, the witness searches the occupied slots in the set and rejects if there is another request with the same primary key (for performance, we compare 64-bit hashes of primary keys instead of full keys). If there is no slot available in the set for the key, the record operation is rejected as well.

For client requests mutating multiple objects, witnesses perform the commutativity and space check for every affected object; to accept an update affecting n objects, a witness must ensure that (1) no existing client request mutates any of the n objects and (2) there is an available slot in each set for all n objects. If the update is commutative and space is available, the witness writes the update request n times as if recording n different requests on each object.

4.3 Commutativity Checks in Masters

Every NoSQL update operation changes the values of one or more objects. To enforce commutativity, a master can check if the objects touched (either updated or just read) by an operation are *unsynced* at the time of its execution. If an operation touches any *unsynced* value, it is not commutative and the master must sync all unsynced operations to backups before responding back to the client.

If the object values are stored in a log, masters can determine if an object value is synced or not by comparing its position in the log against the last synced position.

If the object values are not stored in a log, masters can use monotonically increasing timestamps. Whenever a master updates the value of an object, it tags the new value with a current timestamp. Also, the master keeps the timestamp of when last backup sync started. By comparing the timestamp of an object against the timestamp of the last backup sync, a master can tell whether the value of the object has been synced to backups.

4.4 Improving Throughput of Masters

Masters in primary-backup replication are usually the bottlenecks of systems since they drive replication to backups. Since masters in CURP can respond to clients before syncing to backups, they can delay syncs until the next batch without impacting latency. This batching of syncs improves masters' throughput in two ways.

First, by batching replication RPCs, CURP reduces the number of RPCs a master must handle per client request. With 3-way primary-backup replication, a master must process 4 RPCs per client request (1 update RPC and 3 replication RPCs). If the master batches replication and syncs every 10 client requests, it handles 1.3 RPCs on average. On NoSQL storage systems, sending and receiving RPCs takes a significant portion of the total processing time since NoSQL operations are not compute-heavy.

Second, CURP eliminates wasted resources and other inefficiencies that arise when masters wait for syncs. For example, in the RAMCloud [27] storage system, request handlers use a polling loop to wait for completion of backup syncs. The syncs complete too quickly to context-switch to a different activity, but the polling still wastes more than half of the CPU cycles of the polling thread. With CURP, a master can complete a request without waiting for syncing and move on to the next request immediately, which results in higher throughput.

The batch size of syncs is limited in CURP to reduce witness rejections. Delaying syncs increases the chance of finding non-commutative operations in witnesses and masters, causing extra rejections in witnesses and more blocking syncs in masters. A simple way to limit the batching would be for masters to issue a sync immediately after responding to a client if there is no outstanding sync; this strategy gives a reasonable throughput improvement since at most one CPU core will be used for syncing, and it also reduces witness rejections by syncing aggressively. However, to find the optimal batch size, an experiment with a system and real workload is necessary since each workload has a different sensitivity to larger batch sizes. For example, workloads which randomly access large numbers of keys uniformly can use a very large batch size without increasing the chance of commutativity conflicts.

4.5 Garbage Collection

As discussed in §3.4, masters send garbage collection RPCs for synced updates to their witnesses. Right after syncing to backups, masters send **gc** RPCs (in Figure 5), so the witnesses can discard data for the operations that were just synced.

To identify client requests for removal, CURP uses 64-bit key hashes and RPC IDs assigned by RIFL [20]. Upon receiving a **gc** RPC, a witness locates the sets of slots using the *keyHashes* and resets the slots whose occupying requests have the matching RPC IDs. Witnesses ignore *keyHashes* and *rpcIds* that are not found since the record RPCs might have been rejected. For client requests that mutate multiple objects, **gc** RPCs include multiple $\langle \text{keyHash}, \text{rpcIds} \rangle$ pairs for all affected objects, so that witnesses can clear all slots

occupied by the request.

Although the described garbage collection can clean up most records, some slots may be left uncollected: if a client crashes before sending the update request to the master, or if the **record** RPC is delayed significantly and arrives after the master finished garbage collection for the update. Uncollected garbage will cause witnesses to indefinitely reject requests with the same keys.

Witnesses detect such uncollected records and ask masters to retry garbage collection for them. When it rejects a **record**, a witness recognizes the existing record as uncollected garbage if there have been many garbage collections since the record was written (three is a good number if a master performs only one **gc** RPC at a time). Witnesses notify masters of the requests that are suspected as uncollected garbage through the response messages of **gc** RPCs; then the masters retry the requests (most likely filtered by RIFL), sync to backups, and thus include them in the next **gc** requests.

4.6 Recovery Steps

To recover a crashed master, CURP first restores data from backups and then replays requests from a witness. To fetch the requests to replay, the new master sends a **getRecoveryData** RPC (in Figure 5), which has two effects: (1) it irreversibly sets the witness into recovery mode, so that the data in the witness will never change, (2) it provides the entire list of client requests saved in the witness.

With the provided requests, the new master replays all of them. Since operations already recovered from backups will be filtered out by RIFL [20], the replay step finishes very quickly. In total, CURP increases recovery time by the execution time for a few requests plus 2 RTT (1 RTT for **getRecoveryData** and another RTT for backup sync after replay).

4.7 Zombies

For a fault-tolerant system to be consistent, it must neutralize *zombies*. A zombie is a server that has been determined to have crashed, so some other server has taken over its functions, but the server has not actually crashed (e.g., it may have suffered temporary network connectivity problems). Clients may continue to communicate with zombies; reads or updates accepted by a zombie may be inconsistent with the state of the replacement server.

CURP assumes that the underlying system already has mechanisms to neutralize zombies (e.g., by asking backups to reject replication requests from a crashed master [27]). The witness mechanism provides additional safeguards. If a zombie responds to a client request without waiting for replication, then the client must communicate with all witnesses before completing the request. If it succeeds before the witness data has been replayed during recovery, then the update will be reflected in the new master. If the client contacts a witness after its data has been replayed, the witness will reject the request; the client will then discover that the old master has crashed and reissue its request to the new master. Thus, the witness mechanism does not create new

	RAMCloud cluster	Redis cluster
CPU	Xeon X3470 (4x2.93 GHz)	Xeon D-1548 (8x2.0 GHz)
RAM	24 GB DDR3 at 800 MHz	64 GB DDR4
Flash	2x Samsung 850 PRO SSDs	Toshiba NVMe flash
NIC	Mellanox ConnectX-2 InfiniBand HCA (PCIe 2.0)	Mellanox ConnectX-3 10 Gbps NIC (PCIe 3.0)
Switch	Mellanox SX6036 (2 level)	HPE 45XGc
OS	Linux 3.16.0-4-amd64	Linux 3.13.0-100-generic

Table 1: The server hardware configuration for benchmarks.

safety issues with respect to zombies.

4.8 Modifications to RIFL

In order to work with CURP, the garbage collection mechanism of RIFL described in [20] must be modified. See §C.1 for details.

5 Evaluation

We evaluated CURP by implementing it in the RAMCloud and Redis storage systems, which have very different backup mechanisms. First, using the RAMCloud implementation, we show that CURP improves the performance of consistently replicated systems. Second, with the Redis implementation, we demonstrate that CURP can make strong consistency affordable in a system where it had previously been too expensive for practical use.

5.1 RAMCloud Performance Improvements

RAMCloud [27] is a large-scale low latency distributed key-value store, which primarily focuses on reducing latency. Small read operations take 5 μ s, and small writes take 14 μ s. By default, RAMCloud replicates each new write to 3 backups, which asynchronously flush data into local drives. Although replicated data are stored in slow disk (for cost saving), RAMCloud features a technique to allow fast recovery from a master crash (it recovers within a few seconds) [26].

With the RAMCloud implementation of CURP, we answered the following questions:

- How does CURP improve RAMCloud’s latency and throughput?
- How many resources do witness servers consume?
- Will CURP be performant under highly-skewed workloads with hot keys?

Our evaluations using the RAMCloud implementation were conducted on a cluster of machines with the specifications shown in Table 1. All measurements used InfiniBand networking and RAMCloud’s fastest transport, which bypasses the kernel and communicates directly with InfiniBand NICs. Our CURP implementation kept RAMCloud’s fast crash recovery [26], which recovers from master crashes within a few seconds using data stored on backup disks. Servers were configured to replicate data to 1–3 different backups (and 1–3 witnesses for CURP results), indicated as a replication factor f . The log cleaner of RAMCloud did not run in any measurements; in a production system, the log cleaner can reduce the throughput.

For RAMCloud, CURP moved backup syncs out of the critical path of write operations. This decoupling not only improved latency but also improved the throughput of

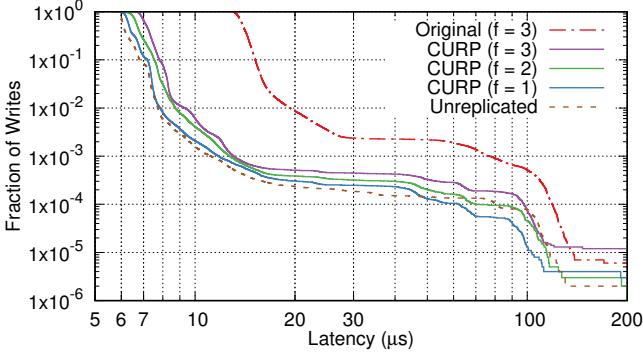


Figure 6: Complementary cumulative distribution of latency for 100B random RAMCloud writes with CURP. Writes were issued sequentially by a single client to a single server, which batches 50 writes between syncs. A point (x, y) indicates that y of the 1M measured writes took at least x μ s to complete. f refers to fault tolerance level (i.e. number of backups and witnesses). “Original” refers to the base RAMCloud system before adopting CURP. “Unreplicated” refers to RAMCloud without any replication. The median latency for synchronous, CURP ($f = 3$), and unreplicated writes were 14 μ s, 7.1 μ s, and 6.1 μ s respectively.

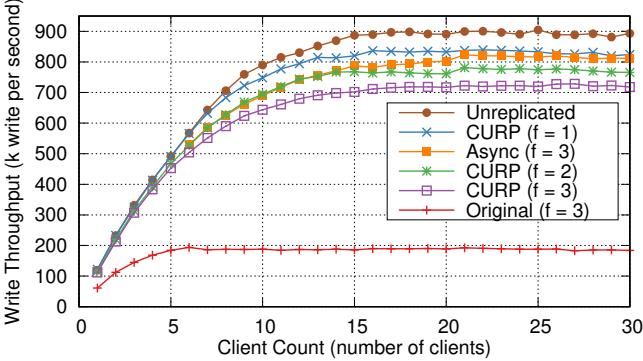


Figure 7: The aggregate throughput for one server serving 100B RAMCloud writes with CURP, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server, which batches 50 writes before syncs. Each experiment was run 15 times, and median values are displayed. “Original” refers to the base RAMCloud system before adding CURP. “Unreplicated” refers to RAMCloud without any replication. In “Async” RAMCloud, masters return to clients before backup syncs, and clients complete writes without replication to witnesses or backups.

RAMCloud writes.

Figure 6 shows the latency of RAMCloud write operations before and after applying CURP. CURP cuts the median write latencies in half. Even the tail latencies are improved overall. When compared to unreplicated RAMCloud, each additional replica with CURP adds 0.3 μ s to median latency.

Figure 7 shows the single server throughput of write operations with and without CURP by varying the number of clients. The server batches 50 writes before starting a sync. By batching backup syncs, CURP improves throughput by about 4x. When compared to unreplicated RAMCloud, adding an additional CURP replica drops throughput by \sim 6%.

To illustrate the overhead of CURP on throughput (e.g. sending gc RPCs to witnesses), we measured RAMCloud with asynchronous replication to 3 backups, which is identical to CURP ($f=3$) except that it does not record information on witnesses. Achieving strong consistency with CURP reduces

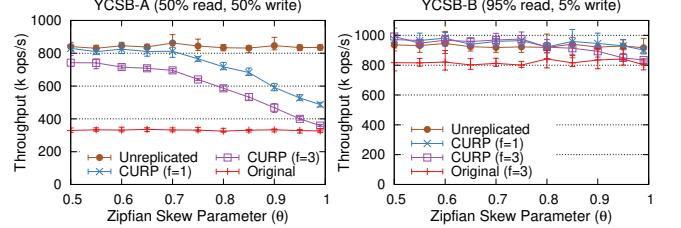


Figure 8: Throughput of a single RAMCloud server for YCSB-A and YCSB-B workloads with CURP at different Zipfian skewness levels. Each experiment was run 5 times, and median values are displayed with errorlines for min and max.

throughput by 10%. In all configurations except the original RAMCloud, masters are bottlenecked by a dispatch thread which handles network communications for both incoming and outgoing RPCs. Sending witness gc RPCs burdens the already bottlenecked dispatch thread and reduces throughput.

We also measured the latency and throughput of RAMCloud read operations before and after applying CURP, and there were no differences.

5.2 Resource Consumption by Witness Servers

Each witness server implemented in RAMCloud can handle 1270k record requests per second with occasional garbage collection requests (1 every 50 writes) from master servers. A witness server runs on a single thread and consumes 1 hyper-thread core at max throughput. Considering that each RAMCloud master server uses 8 hyper-thread cores to achieve 728k writes per second, adding 1 witness increases the total CPU resources consumed by RAMCloud by 7%. However, CURP reduces the number of distinct backup operations performed by masters, because it enables batching; this offsets most of the cost of the witness requests (both backup and witness operations are so simple that most of their cost is the fixed cost of handling an RPC; a batched replication request costs about the same as a simple one).

The second resource overhead is memory usage. Each witness server allocates 4096 request storage slots for each associated master, and each storage slot is 2KB. With additional metadata, the total memory overhead per master-witness pair is around 9MB.

The third issue is network traffic amplification. In CURP, each update request is replicated both to witnesses and backups. With 3-way replication, CURP increases network bandwidth use for update operations by 75% (in the original RAMCloud, a client request is transferred over the network to a master and 3 backups).

5.3 Impact of Highly-Skewed Workloads

CURP may lose its performance benefits when used with highly-skewed workloads with hot keys; in CURP, an unsynced update on a key causes conflicts on all following updates or reads on the same key until the sync completes. To measure the impact of hot keys, we measured RAMCloud’s performance with CURP using a highly-skewed Zipfian distribution [14] with 1M objects. Specifically, we used two different workloads similar to YCSB-A and YCSB-B [9];

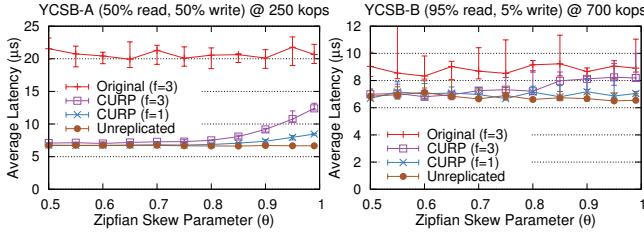


Figure 9: Average RAMCloud client request latency for YCSB-A and YCSB-B workloads with CURP at different Zipfian skewness levels. 10 clients issued requests to maintain a certain throughput level (250 kops for YCSB-A and 700 kops for YCSB-B). Each experiment was run 5 times, and median values are displayed with errorlines for min and max. Latency values are averaged over both read and write operations.

since RAMCloud is a key-value store and doesn't support 100B field writes in 1k objects, we modified the YCSB benchmark to read and write 100B objects with 30B keys.

Figure 8 shows the impact of workload skew (defined in [14]) on the throughput of a single server. For YCSB-A (write-heavy workload), the server throughput with CURP is similar to an unreplicated server when skew is low, but it drops as the workload gets more heavily skewed. For YCSB-B, since most operations are reads, the throughput is less affected by skew. CURP's throughput benefit degrades starting at a Zipfian parameter $\theta = 0.8$ (about 3% of accesses are on hot keys) and almost disappears at $\theta = 0.99$.

Figure 9 shows the impact of skew on CURP's latency; unlike the throughput benefits, CURP retains its latency benefits even with extremely skewed workloads. We measured latencies under load since an unloaded system will not experience conflicts even with extremely skewed workloads. For YCSB-A, the latency of CURP increases starting at $\theta = 0.85$, but CURP still reduces latency by 42% even at $\theta = 0.99$. For YCSB-B, only 5% of operations are writes, so the latency improvements are not as dramatic as YCSB-A.

Figure 10 shows the latency distributions of reads and writes separately at $\theta = 0.95$ under the same loaded conditions as Figure 9. For YCSB-A, CURP increases the tail latency for read operations slightly since reads occasionally conflict with unsynced writes on the same keys. CURP reduces write latency by 2–4x: write latency with CURP is almost as low as for unreplicated writes until the 50th percentile, where conflicts begin to cause blocking on syncs. Overall, the improvement of write latency by CURP more than compensates for the degradation of read latency.

For YCSB-B, operation conflicts are more rare since all reads (which compose 95% of all operations) are commutative with each other. In this workload, CURP actually improves the overall read latency; this is because, by batching replication, CURP makes CPU cores more readily available for incoming read requests (which is also why unreplicated reads have lower latency). For YCSB-A, CURP doesn't improve read latency much since frequent conflicts limit batching replication. In general, read-heavy workloads experience fewer conflicts and are less affected by hot keys.

5.4 Making Redis Consistent and Durable

Redis [30] is another low-latency in-memory key-value store, where values are data structures, such as lists, sets, etc. For Redis, the only way to achieve durability and consistency after crashes is to log client requests to an append-only file and invoke `fsync` before responding to clients. However, `fsyncs` can take several milliseconds, which is a 10–100x performance penalty. As a result, most Redis applications do not use synchronous mode; they use Redis as a cache with no durability guarantees. Redis also offers replication to multiple servers, but the replication mechanism is asynchronous, so updates can be lost after crashes; as a result, this feature is not widely used either.

For this experiment, we used CURP to hide the cost of Redis' logging mechanism: we modified Redis to record operations on witnesses, so that operations can return without waiting for log syncs. Log data is then written asynchronously in the background. The result is a system with durability and consistency, but with performance equivalent to a system lacking both of these properties. In this experiment the log data is not replicated, but the same mechanism could be used to replicate the log data as well.

With the Redis implementation of CURP, we answered the following questions:

- Can CURP transform a fast in-memory cache into a strongly-consistent durable storage system without degrading performance?
- How wide a range of operations can CURP support?

Measurements of the Redis implementation were conducted on a cluster of machines in CloudLab [29], whose specifications are in Table 1. All measurements were collected using 10 Gbps networking and NVMe SSDs for Redis backup files. Linux `fsync` on the NVMe SSDs takes around 50–100 μ s; systems with SATA3 SSDs will perform worse with the `fsync-always` option.

For the Redis implementation, we used Redis 3.2.8 for servers and “C++ Client” [34] for clients. We modified “C++ Client” to construct Redis requests more quickly.

Figure 11 shows the performance of Redis before and after adding CURP to its local logging mechanism; it graphs the cumulative distribution of latencies for Redis SET operations. After applying CURP (using 1 witness server), the median latency increased by 3 μ s (12%). The additional cost is caused primarily by the extra syscalls for send and recv on the TCP socket used to communicate with the witness; each syscall took around 2.5 μ s.

When a second witness server is added in Figure 11, latency increases significantly. This occurs because the Redis RPC system has relatively high tail latency. Even for the non-durable original Redis system, which makes only a single RPC request per operation, latency degrades rapidly above the 80th percentile. With two witnesses, CURP must wait for three RPCs to finish (the original to the server, plus two witness RPCs). At least one of these is likely

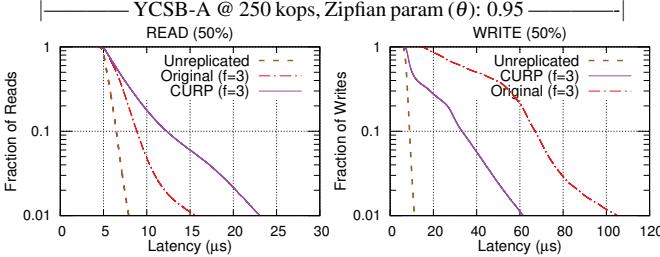


Figure 10: Complementary cumulative distribution of read and write latencies with CURP on a loaded server (250 kops for YCSB-A and 700 kops for YCSB-B). 10 clients issued read / write operations (using the read / write mix ratio of YCSB) for 1 min to a single server. The workloads used a Zipfian distribution with $\theta = 0.95$, which means 16% of operations are on keys that were accessed within the last 100 executed operations.

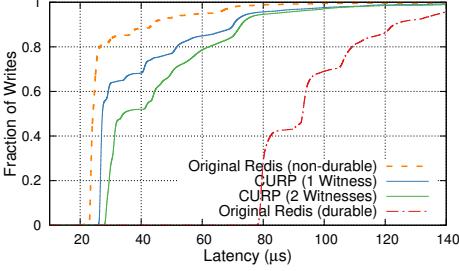


Figure 11: Cumulative distribution of latency for 100B random Redis SET requests with CURP. Writes were issued sequentially by a single client to a single Redis server. CURP used one or two additional Redis servers as witnesses. “Original Redis (durable)” refers to the base Redis without CURP, configured to invoke `fsync` on a backup file before replying to clients.

to experience high tail latency and slow down the overall completion. We didn’t see a similar effect in RAMCloud because its latency is consistent out to the 99th percentile: when issuing three concurrent RPCs, it is unlikely that any of them will experience high latency.

Figure 12 shows the throughput of Redis SET operations for a single Redis server with varying numbers of clients. Applying CURP reduced the throughput of Redis about 18%. With a large number of clients, the original synchronous form of Redis can offer throughput approaching non-durable Redis. The reason for this is that Redis batches `fsyncs` in synchronous mode: in each cycle through its event loop, it processes all of the requests waiting on its incoming sockets, issues a single `fsync`, then responds to all of those requests. The disadvantage of this approach is that it results in very high latency for clients.

5.5 Applicability of CURP

CURP can be applied to a variety of operations, not just write operations in key-value stores. Redis supports many data structures, such as strings, hashmaps, lists, counters, and so on. All of these update operations (including ones that are non-idempotent or return read values) can benefit from CURP. Since each data structure is assigned to a specific key, CURP can execute many update operations on different keys without blocking on syncs.

Figure 13 shows the median latency with and without CURP on three different Redis commands: `SET`, which writes ASCII data to a string data structure; `HMSET`, which

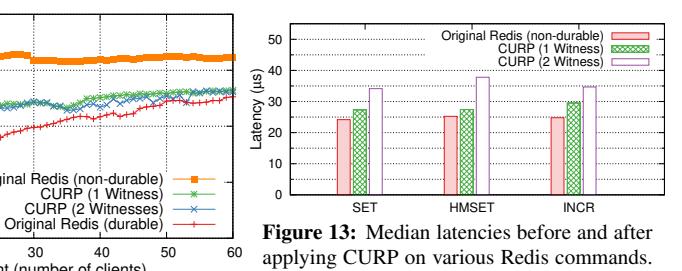
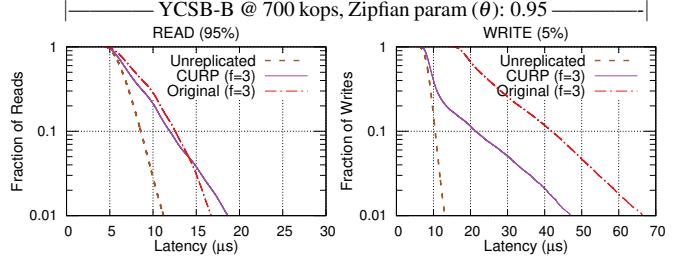


Figure 12: The aggregate throughput for one server serving 100B Redis SET operations with CURP, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server. “Original Redis (durable)” refers to the base Redis without CURP, but configured to invoke `fsync` before replying to clients.

writes data to a member of a hashmap; and `INCR`, which increments an integer counter and returns its current value. For all three operations, latency overheads were small for CURP with 1 witness. CURP with 2 witnesses increased latency about 10 μ s because of tail latency issues. We believe that the TCP transport library used by the C++ client is inefficient for waiting for multiple responses concurrently, and we will continue to investigate this.

6 Related work

Table 2 summarizes the performance of CURP and other fast replication protocols. The paragraphs below explain these numbers in detail. We present analytical performance instead of empirical results since empirical performance depends too much on implementation and underlying systems (e.g. CURP on RAMCloud and CURP on Redis have very different absolute performance).

Generalized Paxos [18] allows clients to complete operations (i.e. receive execution results) in 1.5 RTTs and supersedes *Fast Paxos* [19]. Both protocols allow clients to send requests directly to replicas and reduce latency from 2 RTTs to 1.5 RTT. Fast Paxos has a contention problem and performs well only at low throughput. Generalized Paxos resolves the contention problem by using commutativity; it groups commutative requests from concurrent clients into an unordered set, and it only orders between sets. Although Generalized Paxos allows a leader replica to learn that operations are committed in 1 RTT, clients need to wait another half RTT to receive the execution results from the leader; so

		CURP	Gen.Paxos	EPaxos	NOPaxos	
Latency	LAN	read write	1 RTT 1 RTT	1.5 RTTs 1.5 RTTs	2 RTTs 2 RTTs	1 RTT + α 1 RTT + α
	WAN	read write	~ 0 RTT 1 RTT	1.5 RTTs 1.5 RTTs	~ 1 RTT ~ 1 RTT	Not Avail. Not Avail.
load on leader	read write	<1 RPC 1 RPC	$\sim n$ RPCs $\sim n$ RPCs	~ 2 RPCs ~ 2 RPCs	1 RPC 1 RPC	

Table 2: Performance comparisons of replication protocols. “LAN” means intra-datacenter replications. “WAN” means geo-replication and assumes that all clients have a local replica; clients in a datacenter without local replicas must send requests to a remote replica and experience the WAN RTTs same as in “LAN”. NOPaxos’s RTT is longer than usual since network packets must detour through a sequencer. All latency numbers omitted the time to make data persistent, which is same for all protocols (1 persistence time per request) and insignificant with the use of modern fast storage technologies. “Load on leader” shows how many RPCs a leader (or master) processes per client request. “ n ” denotes the number of replicas.

its end-to-end latency becomes 1.5 RTTs, as opposed to 1 RTT for CURP. (See §B.3 for a detailed explanation why they cannot achieve 1 RTT.)

Egalitarian Paxos (EPaxos) [22] relies on commutativity to allow multiple leaders to propose and execute operations concurrently. This approach improves throughput. In geo-replicated environments, EPaxos allows clients to choose a nearby replica as leader, so operations can complete in 1 wide-area RTT. However, in LAN environments, EPaxos clients cannot hide the message delay to a leader, so operations take 2 RTT. Also, since EPaxos does not have a strong leader, read operations must run through full consensus and be written to replicated command logs; for read-heavy workloads, EPaxos will perform worse than traditional 2 RTT protocols with read leases, such as Raft [25]. On the other hand, CURP can directly execute read operations in masters or even in backups with the help of witnesses. Another limitation of EPaxos is that clients in a datacenter that doesn’t host a replica must use a remote leader, increasing its latency to 2 wide-area RTTs.

Speculative Paxos [28] and *Network-Ordered Paxos* (NOPaxos) [21] reduce latency almost to 1 RTT by serializing client requests within network. Both protocols use SDNs to detour requests from all clients through a single network device (a root layer switch or middlebox); so, they can be deployed only in specialized environments (e.g. a privately-owned datacenter). Also, due to detouring of packets, they actually add latency overhead over unreplicated systems; Speculative Paxos (~ 25 μ s) or NOPaxos(~ 16 μ s) have higher latency overhead compared to CURP (~ 1 μ s).

TAIR [37] and Janus [23] commit distributed transactions in 1 wide-area RTT; before them, transaction commits took 2 RTTs: 1 for transaction prepares and 1 for geo-replicating the data of prepare. They flattened out these serial steps by replicating data before the prepare is executed. They modified concurrency control protocols to fix inconsistencies in replications. They also require commutativity of workloads for 1 RTT commits.

To avoid the performance penalty of consistent replications, *eventual consistency* [36] has been widely adopted in

industry [10, 8, 5]. Systems using eventual consistency return from updates before replication is complete, and replications happen asynchronously; since nearby replicas are stale, clients must read from far-away masters for consistency. Pileus [35] and Tuba [2] allowed applications to declare their consistency and latency priorities, and they dynamically select replicas to read from.

Broadcast-broadcast (BB) protocols [4, 3, 12, 16] for total order broadcasts [11] have similarities to CURP. Senders in BB protocols broadcast a message to all destinations (replicated processes) plus a sequencer before ordering, followed by a second broadcast from the sequencer about the ordering information. Some variants of BB protocols [3, 12] exploit the fact that broadcasts are mostly delivered in-order in small LAN environments and let processes optimistically consume messages without waiting for the ordering information from the sequencer. If the suspected order turned out to be different from the order determined by the sequencer, the process must rollback to correct the inconsistency. On the other hand, in CURP, replicas wait for the ordered replication from a master instead of executing operations with a presumed ordering, so CURP doesn’t require rollbacks, which is expensive and difficult to implement. Furthermore, even if client requests arrive in a master and witnesses out of order, CURP still achieves 1 RTT as long as the reordered requests are commutative.

7 Conclusion

In this paper we have uncovered an opportunity for introducing concurrency into mechanisms for consistent replication. By exploiting the commutativity of operations, replication without ordering can be performed in parallel with sending requests to an execution server. This general approach can be applied to improve a variety of replication mechanisms, including primary-backup approaches and consensus protocols with strong leaders. We presented Consistent Unordered Replication Protocol (CURP), which supplements standard primary-backup replication mechanisms. CURP reduces the latency to complete operations from 2 RTTs to 1 RTT while retaining strong consistency. We implemented CURP in RAMCloud and Redis to demonstrate its benefits.

Acknowledgements

We thank our shepherd, Manos Kapritsos, and our anonymous NSDI and OSDI reviewers for their feedback. Thanks to Stephen Yang and Collin Lee for helping on improving the clarity of this paper. This work was supported by the industrial affiliates of the Stanford Platform Lab and by the Samsung Scholarship.

References

- [1] GlusterFS. <https://www.gluster.org>, 2017. Accessed: 2017-09-22.
- [2] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design*

- and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 367–381.
- [3] BALAKRISHNAN, M., BIRMAN, K., AND PHANISHAYEE, A. PLATO: Predictive latency-aware total ordering. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems* (Leeds, UK, 2006), SRDS ’06, IEEE Computer Society, pp. 175–188.
 - [4] BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* 9, 3 (Aug. 1991), 272–314.
 - [5] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook’s distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 49–60.
 - [6] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, 2006), OSDI ’06, USENIX Association, pp. 335–350.
 - [7] CHODOROW, K., AND DIROLF, M. *MongoDB: The Definitive Guide*, 1st ed. O’Reilly Media, Inc., 2010.
 - [8] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!’s hosted data serving platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288.
 - [9] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, IN, 2010), SoCC ’10, ACM, pp. 143–154.
 - [10] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon’s highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, WA, 2007), SOSP ’07, ACM, pp. 205–220.
 - [11] DÉFAGO, X., SCHIPER, A., AND URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.* 36, 4 (Dec. 2004), 372–421.
 - [12] FELBER, P., AND SCHIPER, A. Optimistic active replication. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (Phoenix, AZ, USA, 2001), ICDCS ’01, IEEE Computer Society, pp. 333–341.
 - [13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 29–43.
 - [14] GRAY, J., SUNDARESAN, P., ENGLERT, S., BACLAWSKI, K., AND WEINBERGER, P. J. Quickly generating billion-record synthetic databases. *SIGMOD Rec.* 23, 2 (May 1994), 243–252.
 - [15] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA, 2010), USENIXATC’10, USENIX Association, pp. 11–11.
 - [16] KAASHOEK, M. F., AND TANENBAUM, A. S. Group communication in the amoeba distributed operating system. In *[1991] Proceedings. 11th International Conference on Distributed Computing Systems* (May 1991), pp. 222–230.
 - [17] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169.
 - [18] LAMPORT, L. Generalized consensus and Paxos. Tech. rep., March 2005.
 - [19] LAMPORT, L. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103.
 - [20] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, CA, 2015), SOSP ’15, ACM, pp. 71–86.
 - [21] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. K. Just say no to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI’16, USENIX Association, pp. 467–483.
 - [22] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, PA, 2013), SOSP ’13, ACM, pp. 358–372.
 - [23] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX*

- Conference on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI'16, USENIX Association, pp. 517–532.
- [24] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada, 1988), PODC '88, ACM, pp. 8–17.
- [25] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, 2014), USENIX Association, pp. 305–319.
- [26] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 29–41.
- [27] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud storage system. *ACM Trans. Comput. Syst.* 33, 3 (Aug. 2015), 7:1–7:55.
- [28] PORTS, D. R. K., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015), NSDI'15, USENIX Association, pp. 43–57.
- [29] RICCI, R., EIDE, E., AND TEAM, C. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. ; *login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.
- [30] SANFILIPPO, S., ET AL. Redis. <https://redis.io/>, 2015. Accessed: 2017-04-18.
- [31] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319.
- [32] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2010), pp. 1–10.
- [33] SIVASUBRAMANIAN, S. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, AZ, 2012), SIGMOD '12, ACM, pp. 729–730.
- [34] SPRENKER, L., AND HAMMOND, B. Redis C++ Client. <https://github.com/mrpi/redis-cplusplus-client>, 2011. Accessed: 2017-04-20.
- [35] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABULIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, PA, 2013), SOSP '13, ACM, pp. 309–324.
- [36] VOGELS, W. Eventually consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44.
- [37] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, CA, 2015), SOSP '15, ACM, pp. 263–278.
- [38] ZHAO, W. Fast Paxos made easy: Theory and implementation. *International Journal of Distributed Systems and Technologies (IJDST)* 6, 1 (2015), 15–33.

A Informal Proof of Correctness

With the normal operation behaviors described in §3.2, the recovery protocol in §3.3 guarantees the following correctness properties.

- **Durability:** if a client completes an operation, it survives server crashes.
- **Consistency:** if a client completes an operation, its result returned to an application remains consistent after server crash recoveries.
- **Linearizability:** an operation appears to be executed exactly once between start and completion.

Before presenting proofs, we reiterate some key behaviors of the CURP protocol.

(*Rule 1*) from §3.2.1, a client only completes an update operation if (1) it is recorded in all f witnesses or (2) it is replicated to f backups.

(*Rule 2*) a completed unsynced operation must be individually commutative with all preceding operations that are not synced yet. This is the behavior described in §3.2.3; a master must sync before responding if the current operation is not commutative with any other existing (preceding) unsynced operations.

Now, we present proof sketches for the properties.

Durability: recovery of a master only completes after recovery from 1 backup and 1 witness, and the completed operation must exist in the backup or the witness by (Rule 1); thus, the completed operation must be recovered when the recovery is completed. \square

Consistency: Consider an individual completed operation α and its consistency. To prove that α 's result doesn't change even after crash recovery, we will think about the operation execution sequence before α , which we will call *history* of α (or H_α).

Case 1: the operation α has been synced to the backup used for recovery. This operation will be recovered from the backup (phase 1) and any replay from witnesses (phase 2) will be ignored (by RIFL). Since backup syncs preserve the execution order of operations, the H_α didn't change; so the post-recovery execution sequence should regenerate the original execution result of α .

Case 2: the operation α has not been synced to the backup used for recovery. α must have been recorded in all witnesses by (Rule 1) and will be recovered during phase 2. We can split the original execution history of α into two parts as in Figure 3: $\langle \text{synced} \rangle$ followed by $\langle \text{unsynced} \rangle$. The 1st phase of recovery will recover the exactly same execution history for the $\langle \text{synced} \rangle$ part. By (Rule 2), we know that losing any $\langle \text{unsynced} \rangle$ part of history after crash will not change the execution result of α . During phase 2 of recovery (from a witness), we may replay some other operations before replaying α , but the result of α doesn't change since all operations recorded in the witness must be commutative. \square

Linearizability: we assume that the underlying system before applying CURP guarantees linearizability for op-

erations that are replicated to backups. CURP may break the linearizability of the underlying system since masters in CURP return before syncing to backups. So, we will reason about how CURP recovers from master crashes without breaking linearizability.

The definition of linearizability can be reworded as following: if the execution of an operation is observed by the issuing client or other clients, no contrary observation can occur afterwards (i.e. it should not appear to revert or be reordered). Since we only care about what happens after recovery, we prove the following proposition: if the execution of an individual operation α is observed *before crash*, no contrary observation can occur *after recovery*.

Case 1: the execution of α was observed by other dependent operations (e.g. reads). By (Rule 2), the master must have synced α to backups since dependent operations don't commute with α . Since it was replicated to backups, α will be linearizable as long as the underlying system is.

Case 2: the execution was observed only by the completion of α . α must be recovered because of the Durability property. The only observation about α before crash was the returned execution result, and it must be still consistent even after recovery because of the Consistency property.

Case 3: no observation was made before crash. α may be lost if it didn't reach to either the backup or witness used for recovery. In CURP, the client keeps retrying until it can complete α . Regardless of whether α was recovered or not, RIFL ensures the retry will only execute α at-most once and return the result of the sole execution. \square

B Extra Discussions

B.1 Why Are Witnesses Separate from Backups?

By having witnesses separated from backups, CURP requires fewer changes to the existing systems and is more applicable to many wildly different backup mechanisms. Both of our two implementations leveraged this flexibility: in RAMCloud, a master keeps changing backups to which it replicates (to spread data over the entire cluster), so clients don't know which backups are currently used by the master; in Redis, operation logs are stored in local disks to ensure durability, so there are no separate backup servers to which CURP clients can record inputs. Thus, separating witnesses from backups improves CURPs applicability to many existing primary-backup systems.

On the other hand, when designing a new storage system, combining witnesses and backups can bring extra performance benefits. When they are combined, clients directly send requests to a master and backups, which now also serve as witnesses. The key change is masters now sync operation orders (by listing IDs as in witness **gc** RPCs) instead of full client requests; then backups lookup the matching requests from their witness storage and move them to backup logs. This approach will lower network bandwidth consumption. Also, most witness **gc** RPCs can be eliminated; immediately

after handling the sync, the requests in the witness storage can be deleted as they are now safe in the backup log. (For safety, the recovery protocol must pick 1 witness/backup combo and must not mix.) This saving of `gc` RPCs will improve masters’ throughput and will reduce the chance of commutativity conflicts.

B.2 Extending CURP to Consensus Protocols

This section illustrates how CURP can be extended to reduce the latency of consensus protocols. CURP can be integrated in most consensus protocols with strong leaders (e.g. Raft [25], Viewstamped Replication [24]). In such protocols, clients send requests to the current leader, which serializes the requests into its command log. The leader then replicates its command log to a majority of replicas before executing the requests and replying back to clients with the results. This process takes 2 RTTs, and CURP can reduce it to 1 RTT.

As in primary-backup replication, CURP on consensus allows clients to replicate requests to witnesses in parallel with sending requests to the leader; the leader then speculatively executes the requests and responds to clients before replicating the requests to a quorum of replicas. A client can complete an operation if it is accepted by a superquorum of witnesses or committed in a quorum of replicas.

To mask f failures, consensus protocols use $2f + 1$ replicas, and systems stay available with f failed replicas. For the same guarantee, CURP also uses $2f + 1$ replicas, but each replica also has a witness component in addition to existing components for consensus. Although CURP can proceed with $f + 1$ available replicas, it needs $f + \lceil f/2 \rceil + 1$ replicas (for superquorum of witnesses) to use 1 RTT operations. With less than $f + \lceil f/2 \rceil + 1$ replicas, clients must ask masters to commit operations in $f + 1$ replicas before returning result (2 RTTs).

Like masters in regular CURP, leader replicas execute operations speculatively if they are commutative with existing unsynced operations; for an incoming client request, the leader serializes it into its command log, executes it, and responds to the client before committing it in a majority of replicas.

For clients to complete an operation in 1 RTT, it must be recorded in a *superquorum* of $f + \lceil f/2 \rceil + 1$ witnesses. The reason why CURP needs a superquorum instead of a simple majority is to ensure commutativity of replays from witnesses during recovery. During recovery, only $f + 1$ out of $2f + 1$ replicas (each of which embeds a witness) might be available. If a client could complete an operation after recording to $f + 1$ witnesses, the completed operation may exist in only 1 witness out of available $f + 1$ witnesses during recovery (since intersection of two quorums is 1 replica). If the other f witnesses accepted other operations that are not commutative with the completed operation (since each witness enforces commutativity individually), recovery cannot distinguish which one is the completed one; executing all appearing in any $f + 1$ witnesses is also not safe since they are not commutative, so they must be replayed in a correct order.

For correctness, the client requests replayed from witnesses during recovery must be *commutative* and *inclusive* of all completed operations that are not yet committed in a majority of replicas. By recording to a superquorum, all completed operations (but not yet committed) are guaranteed to exist in a majority ($\lceil f/2 \rceil + 1$) of any quorum of $f + 1$ witnesses, and any operations that don’t commute with the completed operations cannot exist in more than $\lfloor f/2 \rfloor$ (less than majority of any quorum). Thus, during recovery, all requests that appear in a majority ($\lceil f/2 \rceil + 1$) from any quorum of $f + 1$ witnesses are guaranteed to be commutative and include all completed operations; so, recovery can replay requests that appear in more than $\lceil f/2 \rceil + 1$ witnesses out of any $f + 1$ witnesses.

When leadership changes (e.g. leader election in Raft [25] or view change in Viewstamped Replication [24]), the new leader must recover from witnesses before accepting new operations. To do so, the new leader must collect saved requests from at least $f + 1$ witnesses. This collection can be included in the existing data collection (e.g. Raft votes) that is required by most leadership change protocols. As mentioned in the previous paragraph, the new leader should only replay client requests that are recorded in at least $\lceil f/2 \rceil + 1$ witnesses to ensure commutativity.

After leadership changes, the state machine of the old leader could have diverged from other replicas due to speculatively executed operations that were not recovered from witnesses. To fix this, the old leader must reload from a checkpoint that does not have speculative executions. However, we can avoid reloading from checkpoints if the leadership change was not because of a crash or disconnect of the old leader; instead of requiring old leader to reload from a checkpoint, we can require the new leader to fetch and commit all uncommitted operations in the old leader’s command log.

The last problem introduced by speculative execution is that clients may use old zombie leaders (which believe they are current leaders). Zombie leaders were not possible before CURP since an operation must be committed in a majority before being executed and at least one replica would reject the operation. To prevent clients from completing operations with an old (possibly disconnected) leader, they tag record RPCs with a term number (e.g. a Raft term or a view-number in Viewstamped Replication), which increments every time when leadership changes. A witness checks the term number against the term used by its replica (recall that a witness is a part of a consensus replica); if the record RPC has an old term number, the witness rejects the request and tells the client to fetch new leader information.

CURP can use read leases like many consensus protocols so that read operations can be executed solely by leaders within 1 RTT without recording to witnesses. Optimizing read operations using read leases is common for consensus protocols with strong leaders. A leader replica with a valid read lease can safely execute read operations without committing the read operations through consensus. For the

optimization, each replica grants the read lease to the current leader, promising not to agree on a leader change for a lease period. With valid leases from a majority of replicas, the leader knows that no operations can be committed from other replicas, so it can safely execute read operations without consulting with other replicas. CURP does not interfere with this read lease mechanism.

B.3 Why Do Fast/Generalized Paxos require 1.5 RTTs?

There is a widespread misunderstanding that both Fast Paxos and Generalized Paxos already achieve 1 RTT operations. The confusion probably stems from the fact that both Fast and Generalized Paxos allow Paxos learners to know about acceptance of an operation in 1 RTT.

However, 1 RTT is sufficient to know only that an operation is committed but not enough to know the result: that requires another 0.5 RTT. The abstract for Generalized Paxos says that a server can *execute* the command in two message delays; however, it takes an additional message delay for the result to reach a client, for a total of three message delays (1.5 RTT). It doesn't help for the client to be a Paxos learner, because even learners don't know the result after 1 RTT.

For most operations, results are not trivial and clients must wait for the results from real executions before completing operations. Many writes, such as conditional writes or read-modify-writes, have results that clients cannot know before executions. Blind writes (those that don't return results) could potentially complete in 1 RTT. However, truly blind writes are rarely feasible because they can return exceptions, such as "table no longer on this server" or "permission denied"; clients must be aware of these exceptions.

As a result, Fast/Generalized Paxos are generally considered to have 1.5 RTT latency for clients to complete operations. [21, 28, 38]

C Implementation Details

C.1 Modifications to RIFL

RIFL [20] is a mechanism for detecting duplicate invocations of RPCs. With RIFL, masters make a durable *completion record* of each RPC that updates state, which includes the RPC result. The completion record survives crashes and can be used to detect duplicate invocations of the RPC. When a duplicate is detected, the master skips the execution of the RPC and returns the result from the completion record.

RIFL has two mechanisms for garbage collecting completion records: (1) on RPC requests, clients piggyback acknowledgments of the results of their previous requests (so servers can safely delete these completion records), and (2) clients maintain leases in a central server; if a client's lease expires, masters can delete all completion records for that client. Both of these must be modified to work with CURP.

Since both garbage collection mechanisms assume that retries always come from the same client that made the original request, RIFL must be modified to accommodate retries from witnesses. Firstly, once clients acknowledge

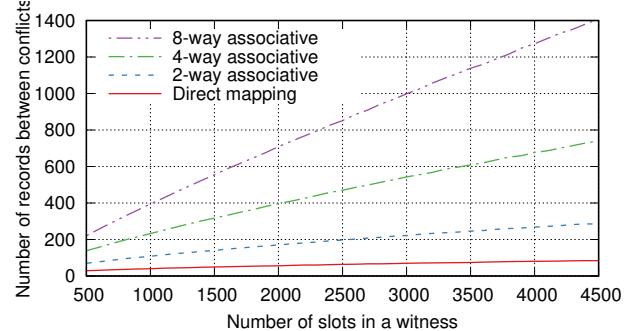


Figure 14: Simulation results for the expected number of recordings before a collision occurs in a witness' cache, assuming a random distribution of keys. Each data point is the average of 10000 simulations. Introducing associativity reduces the chance of collisions significantly.

the receipts of results, masters remove their completion records and start to ignore (not returning results) the duplicate requests. Since replays from witnesses happen in random orders, acknowledgements piggybacked on later requests can make masters to ignore the replay of earlier requests. Thus, clients' acknowledgments included in RPC requests must be ignored during recovery from witnesses.

Secondly, if a client crashes and its lease expires, masters remove all of the completion records for the client; then any requests from the expired client are ignored. This can be a problem in CURP since the replay of the expired client's requests will be ignored during witness-based recovery. To prevent this, masters must sync all operations to backups before expiring a client lease. In practice, the period of syncs is much smaller than the grace period between the time of a client crash and the time of its lease expiration; so, most systems are safe automatically.

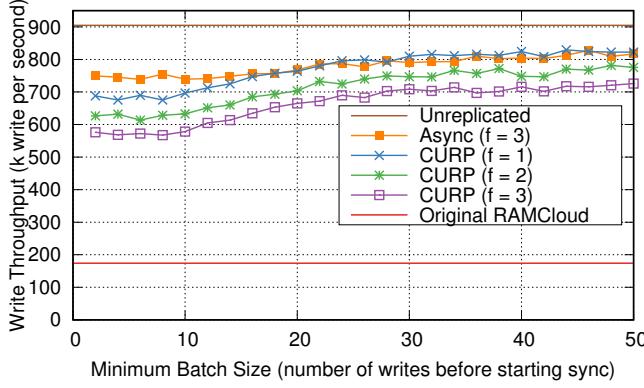
C.2 Why Use Set-associative Cache for Witnesses?

We initially used a direct-mapped cache instead of set-associative cache, but this resulted in a high rate of rejections because of conflicts (i.e. no slot is available for the mapped set). Figure 14 shows the expected number of recordings before a conflict occurs on a witness slot. Using a direct mapping and 4096 total slots, it is expected to have a false conflict after about 80 insertions. Thus, we switched to 4-way associative cache, to reduce witness rejections. We didn't need 8-way associativity (a bit slower than 4-way) since the number of requests in witnesses is already limited by commutativity. (Once a master hits a non-commutative operation and syncs to backups, all saved requests in the witness are garbage collected.)

D Additional Evaluations

D.1 RAMCloud's Throughput by Batch Size

Figure 15 shows the single-server throughput of write operations with CURP while varying the aggressiveness of syncs. After introducing CURP, RAMCloud can delay the sync to backups after responding back to clients; delaying and batching sync to backups makes the server more efficient and improves throughput about 4 times. Since RAMCloud



Minimum Batch Size (number of writes before starting sync)

Figure 15: The aggregate throughput for one server serving 100B RAMCloud writes with CURP, as a function of sync batch size. Each client repeatedly issued random writes back to back to a single server. “Original RAMCloud” refers to the base RAMCloud system before adding CURP. “Unreplicated” refers to RAMCloud without any replication. Each datapoint was measured 15 times, and median values are displayed.

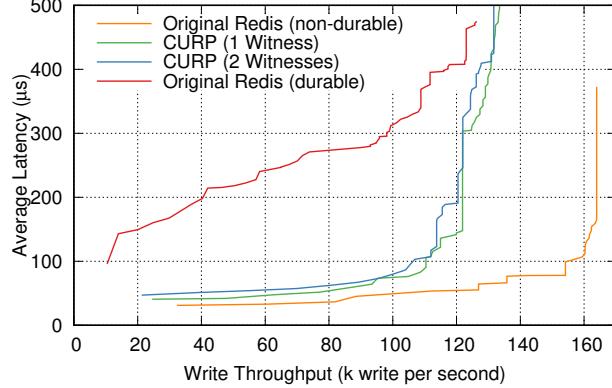


Figure 16: Observed latency at a specific throughput level for one server serving 100B Redis SET operations with CURP. “Original Redis (durable)” refers to the base Redis without CURP, but configured to invoke fsync before replying to clients. Original Redis processes requests from multiple clients, fsyncs once per eventloop, and replies to all clients.

allows only one outstanding sync, syncs are naturally batched for around 15 writes even at 1 minimum batch size.

D.2 Redis Latency vs. Throughput

Figure 16 shows observed latency during the throughput benchmark. Both CURP and non-durable Redis maintains latency low until it reaches 80% of max throughput. The latency of durable Redis increases almost linearly due to batching. The original Redis is designed to provide maximum throughput under high load and natively batches fsyncs; for each event-loop cycle, Redis iterates through TCP sockets for all clients and executes all requests from them; after the iteration, Redis fsyncs once and responds to the clients. This batching amortizes the cost of fsync, and throughput of durable Redis approaches that of non-durable Redis as the number of clients increases. However, this batching adds extra delay before responding back to clients, so latency increases linearly.

Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification

Assaf Eisenman¹, Asaf Cidon^{1,2}, Evgenya Pergament¹, Or Haimovich¹, Ryan Stutsman³, Mohammad Alizadeh⁴, and Sachin Katti¹

¹Stanford University, ²Barracuda Networks, ³University of Utah, ⁴MIT CSAIL

Abstract

As its price per bit drops, SSD is increasingly becoming the default storage medium for hot data in cloud application databases. Even though SSD’s price per bit is more than 10× lower, and it provides sufficient performance (when accessed over a network) compared to DRAM, the durability of flash has limited its adoption in write-heavy use cases, such as key-value caching. This is because key-value caches need to frequently insert, update and evict small objects. This causes excessive writes and erasures on flash storage, which significantly shortens the lifetime of flash. We present Flashield, a hybrid key-value cache that uses DRAM as a “filter” to control and limit writes to SSD. Flashield performs lightweight machine learning admission control to predict which objects are likely to be read frequently without getting updated; these objects, which are prime candidates to be stored on SSD, are written to SSD in large chunks sequentially. In order to efficiently utilize the cache’s available memory, we design a novel in-memory index for the variable-sized objects stored on flash that requires only 4 bytes per object in DRAM. We describe Flashield’s design and implementation, and evaluate it on real-world traces from a widely used caching service, Memcached. Compared to state-of-the-art systems that suffer a write amplification of 2.5× or more, Flashield maintains a median write amplification of 0.5× (since many filtered objects are never written to flash at all), without any loss of hit rate or throughput.

1 Introduction

Flash has an order of magnitude lower cost per bit compared to DRAM. Consequently, it has become the preferred storage medium for hot data that requires high throughput and low latency. For example Google [36] and Facebook [30] use it for storing photos, and databases like LevelDB [5] and RocksDB [9] are deployed on top of flash.

Key-value caches are an essential layer in modern web scale applications, and are widely used by almost all web services, including Facebook, Twitter and Airbnb. Large web service providers run their own key-value cache clusters,

	SSD+DRAM		DRAM only	
	Count	Cost	Count	Cost
Dell 2×10 core server with 256 GB DRAM	1	\$7,700	17	\$130,900
Samsung 1 TB enterprise SSD	4	\$4,800	0	0
Total		\$12,500		\$130,900

Table 1: The cost of a hybrid cache server with combined capacity of 4.25 TB, versus the cost of multiple DRAM-only cache servers with the same aggregate capacity. SSD’s superior cost per bit results in a 10× lower total cost of ownership for a hybrid cache server.

while smaller providers often utilize caching-as-a-service solutions like Amazon ElastiCache [1] and Memcached [7].

However, due to its limited endurance under writes, flash is typically not used for key-value caches like Memcached [6] and Redis [8]. This is all the more perplexing since these caches are typically deployed in a dedicated remote cluster [31] or remote data center [1, 7] or with client-side batching [31]. As a result, client-observed accesses times can be hundreds of microseconds to milliseconds, so flash would only increase delays by a small fraction when compared to using DRAM.

Furthermore, since the performance of caches is primarily determined by the amount of memory capacity they provide [13, 14], and the cost per bit of SSD is more than 10× lower than DRAM, flash promises significant financial benefits compared to DRAM. Table 1 demonstrates that the cost difference between DRAM-only cache and hybrid cache, both with 4.25 TB capacity, is more than 10×. The Total Cost of Ownership (TCO) difference would be even greater due to power costs, since flash consumes significantly less power than DRAM, and can be powered down when there are fewer requests without requiring re-warming the cache.

The reason flash has not been widely adopted as a key-value cache is that cache workloads wear out flash drives very quickly. These workloads typically consist of small objects, some of which need to be frequently updated [10, 31]. But, modern flash chips within SSDs can only be written a

few thousand times per location over their lifetime.

Further, SSDs suffer from write amplification (WA). That is, for each byte written by the application (e.g., the key-value cache), several more bytes are written to the flash at the device level. WA occurs because flash pages are physically grouped in large blocks. Pages must be erased before they can be overwritten, but that can only be done in the granularity of blocks. The result is that over time, these large blocks typically contain a mix of valid pages and pages whose contents have been invalidated. Any valid pages must be copied to other flash blocks before a block can be erased. This garbage collection process creates device-level write amplification (DLWA) that can increase the amount of data written to flash by orders of magnitude. Modern SSDs exacerbate this by striping many flash blocks together (512 MB worth or more) to increase sequential write performance (§2.1, [38]).

To minimize the number of flash writes, SSD storage systems are constrained to writing data in large contiguous chunks. This forces a second-order form of write amplification, which is unique to caches, that we name *cache level write amplification* (CLWA). CLWA occurs when the cache is forced to relocate objects to avoid DLWA. For example, when a hot object occupies the same flash block as many items that are ready for eviction, the cache faces a choice. It can evict the hot object with the cold objects, or it can rewrite the hot object as part of a new, large write. Therefore, in existing SSD cache designs, objects get re-written multiple times into flash.

To deal with this problem, the state-of-the-art system, RIPQ [38], proposes to store hot and cold objects together on flash, by inserting them in different physical regions. However, efficient data placement on flash is not sufficient to protect against high CLWA, and in fact, may further increase CLWA in certain scenarios. For example, consider an application, in which a large number of objects are infrequently accessed (or frequently updated). Since RIPQ admits all objects (hot and cold) into flash, infrequently accessed objects will get inserted into a “cold” insertion point, and will typically get evicted before it is accessed again. Therefore, these objects can get inserted and evicted multiple times. We show that under such workloads, RIPQ suffers from a CLWA of up to 150 (§5), which means it will wear out flash devices too quickly for many applications.

The flash reliability problem will become even greater over time, since as flash density increases, its durability will continue to decrease [20]. In particular, the next generation of flash technology (QLC), can endure $30\times$ fewer writes than the existing technology (TLC) [3, 29, 32].

We present Flashield, a novel hybrid key-value cache that uses both DRAM and SSDs. Our contribution is a novel caching strategy that significantly extends the lifetime of SSDs, such that it is comparable to DRAM by controlling and minimizing the number of writes to flash. Our main observation is that not all objects entering the cache are good

candidates for placement in SSD. In particular, the cache should avoid writing objects to flash that will be updated or that will not be read in the near future. However, when objects first enter the cache, it does not know which objects are good candidates for SSD and which are not.

Therefore, a key idea in Flashield’s design is that objects inserted into the cache always spend a period of time in DRAM, during which the cache learns whether they are good candidates for flash storage. If they indeed prove themselves as flash-worthy, Flashield will move them into flash. If not, they are never moved into flash, which reduces the resulting write amplification. Since the flash layer is considerably larger than DRAM (e.g., 10 \times larger), objects moved to flash on average will remain in the cache much longer than those that stay in DRAM.

To dynamically decide which objects are flash-worthy under varying workloads, we implement the admission control algorithm using machine-learning based Support Vector Machine (SVM) classification. We train a different classifier for each application in the cache. To train the classifiers, we design a lightweight sampling technique that uniformly samples objects over time, collecting statistics about the number of past reads and updates. The classifier predicts whether an object will be read more than n times in the future without getting updated, which is used to determine its suitability for storage on flash. We term this metric *flashiness*.

The second main idea in Flashield’s design is its novel DRAM-based lookup index for variable-length objects stored on flash that requires less than 4 bytes of DRAM per object. This is more than 5 \times less than RIPQ, which consumes 22 bytes per object. Since the flash layer’s capacity is much larger than the DRAM’s, a naïve lookup index for objects stored on flash would consume the entire capacity of the DRAM. Our index consumes a relatively small amount of memory by not storing the location of the objects and their corresponding keys. Instead, for each object stored on flash, the index contains a pointer to a region in the flash where the object is stored, and it stores an additional 4 bits that specify a hash function on the object key that indicates the insertion point of the object in its region on flash. The index leverages bloom filters to indicate whether the object resides on flash or not without storing full keys in DRAM. On average, Flashield’s lookup index only requires 1.03 reads from the SSD to return an object stored on it.

We implement Flashield in C and evaluate its performance on a set of real-world applications that use Memcached [7], a popular cloud-based caching service, using week-long traces. We show that compared with RIPQ [38], Flashield reduces write amplification by a median of 5 \times and an average of 16 \times , and the index size by more than 5 \times , while maintaining the same average hit rates. We show that when objects are read from SSD, Flashield’s read latency and throughput is close to the SSD’s latency and throughput, and when objects are written to the cache or read from DRAM,

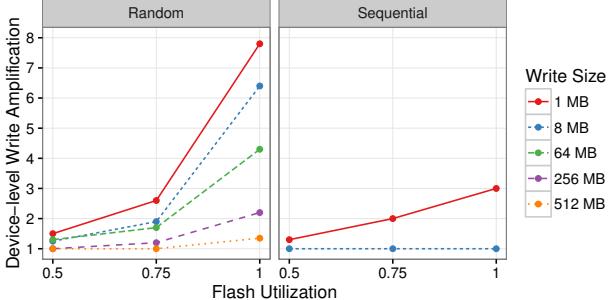


Figure 1: Device-level write amplification after writing 4 TB randomly and sequentially using different write sizes.

its latency and throughput are similar to that of DRAM-based caches like Memcached.

This paper makes three main contributions:

1. Flashield is the first SSD storage system that explicitly uses DRAM as an admission control filter for deciding which objects to insert into flash.
2. Flashield’s novel in-memory lookup index for flash takes up less than 4 bytes per object in DRAM, without sacrificing flash write amplification and read amplification.
3. Flashield is the first key-value cache that uses a machine-learning based admission control algorithm and lightweight temporal sampling to predict which objects will be good candidates for flash.

As new generations of flash technology can tolerate even fewer writes [3, 20, 29, 32], our dynamic admission control to flash can be extended to other systems beyond caches, such as flash databases and file systems.

2 The Problem

Designing an SSD-based cache requires solving two conflicting challenges. SSDs perform poorly and wear out quickly unless writes are large and sequential. This conflicts with the characteristics of cache workloads. Caches store small objects with highly variable lifetimes; this drives caches to prefer small random I/O for writes which will wear flash drives out quickly.

The lifetime of an SSD is defined by flash device manufacturers as the amount of time before a device has a non-negligible probability of producing uncorrectable read errors (e.g., a probability of 10^{-15} of encountering a corrupt bit). The lifetime of an SSD depends on several factors, including the number of writes and erasures (termed program-erase cycles), the average time between refresh cycles of the SSD cells, the cell technology, the error correction code and more. The typical lifetime of a flash cell is between 3-5 years assuming it is written 3-5 times a day on average.

The key metric for device wear is write amplification. Many write patterns force the SSD to perform additional writes to flash in order to reorganize data. Write amplification is the ratio of the bytes written to flash chips compared to the bytes sent to the SSD by the application. A write am-

plification of 1 means each byte written by the application caused a one byte write to flash. A write amplification of 10 means each byte written by the application caused an extra 9 bytes of data to be reorganized and rewritten to flash.

2.1 Device-level Write Amplification

Device-level write amplification (DLWA) is write amplification that is caused by the internal reorganization of the SSD. The main source of DLWA comes from the size of the unit of flash reuse. Flash is read and written in small (~8 KB) pages. However, pages cannot be rewritten without first being erased. Erasure happens at a granularity of groups of several pages called blocks (~256 KB). The mismatch between the page size (or object sizes) and the erase unit size induces write amplification when the device is at high utilization.

For example, when an application overwrites the contents of a page, the SSD writes it to a different, fresh block and maintains a relocation mapping called the Flash Translation Layer (FTL). The original block cannot be erased yet, because the other pages in the same block may still be live. When the flash chips are completely occupied, the SSD must erase blocks in order to make room for newly written pages. If there are no blocks where all of the pages have been superseded by more recently written data, then live pages from multiple blocks must be consolidated into a single flash block.

This consolidation or *garbage collection* is the source of DLWA. If a device is at 90% occupancy, its DLWA can be very high. Figure 1 shows DLWA under sequential and random writes. The measurements were taken on a 480 GB Intel 535 Series SSD using SMART, a system for monitoring the internal formation of the device. For each data point, 4 TB of randomly generated data is written either randomly or sequentially to the raw logical block addresses of the device with varying buffer sizes. Specifically, in the random workload the logical block space is broken into contiguous fixed buffer-sized regions; each write overwrites one of the regions at random with a full buffer of random data. The sequential workload is circular; regions are overwritten in order of their logical block addresses, looping back to the start of the device as needed. For both patterns, we varied the device space utilization by limiting writes to a smaller portion of the logical block addresses.

The results show that random, aligned 1 MB flash writes experience a nearly 8× DLWA. This is surprising, since flash erase blocks are smaller than 1 MB. The reason for this write amplification is because SSDs are increasingly optimized for high write bandwidth. Each flash package within an SSD is accessed via a relatively slow link (50-90 MB/s today); SSDs stripe large sequential writes across many flash packages in parallel to get high write bandwidth. This effectively fuses erase blocks from several packages into one logical erase block. A 1 MB random write marks a large region of pages as ready for erase, but that region is striped across several erase units that still contain mostly live pages. Others have

Avg Object Size	Read / Write / Update %	Unread Writes %
257 B	90.0% / 9.5% / 0.5%	60.6%

Table 2: Statistics of the 20 applications with the most requests in the week-long Memcached trace.

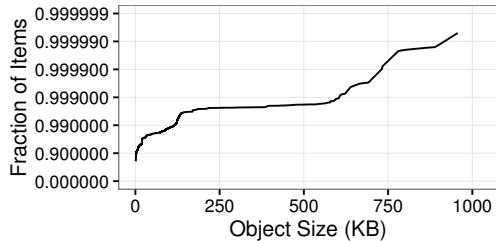


Figure 2: CDF of the object sizes written to memory by the top 20 applications in the Memcached trace.

corroborated this effect as well [38].

There are two ways to combat this effect. The first is to write in units of $B \cdot W$ where B is the erase block size and W is how many blocks the SSD stripes writes across. Our results show that a cache would have to write in blocks of 512 MB in order to eliminate DLWA. The second approach is to write the device sequentially, in FIFO-order at all times. This works because each $B \cdot W$ written produces one completely empty $B \cdot W$ unit, even if writes are issued in units smaller than $B \cdot W$. Figure 1 shows that 8 MB sequential writes also eliminate DLWA.

This means our cache is extremely constrained in how it writes data to flash. To minimize DLWA the cache must write objects in large blocks or sequentially. In either case, this gives the cache little control on precisely *which* objects should be replaced on flash.

2.2 Cache-level Write Amplification

Writing to flash in large *segments* (contiguous chunks of data) is a necessary but not sufficient condition for minimizing overall write amplification. The main side effect of writing in large segments is *cache-level write amplification* (CLWA). CLWA occurs when objects that were removed from the SSD are re-written to it by the cache eviction policy. If the size of the segments (MBs) is significantly larger than the size of objects (bytes or KBs), it is difficult to guarantee that high-ranking objects in the cache will always be stored physically separate from low-ranked objects or objects that contain old values. Therefore, when a segment that has many low-ranked objects is erased from the cache, it may also inadvertently erase some high-ranking objects.

Table 2 presents general statistics of a week-long trace of Memcached, a commercial Memcached service provider [13, 14], and Figure 2 presents the distribution of the sizes of objects written in the trace. The figure demonstrates that object sizes vary widely, and in general they are very small: the average size of objects written to the cache is 257 bytes, and 80.67% of objects are smaller than 1 KB. Therefore, even with a segment size of 8 MB using sequen-

	Hit Rate	CLWA
Victim Cache	69.72%	4.00
RIPQ	70.59%	2.59

Table 3: Hit rate and cache-level write amplification of RIPQ and the victim cache policy under the entire Memcached trace.

tial writes, which is the the smallest possible segment size that does not incur extra write amplification, each segment will contain on average over 32,000 unique objects.

In addition, 60.6% of writes (and 5.8% of all requests) are unread writes, which means they are never read after they are written, and 0.5% of all requests are updates. Both unread writes and updates contribute to write amplification. Ideally, unread writes should not be written to the cache. In the case of updates, to reclaim the space of an object after it was updated, the cache needs to erase and rewrite the object.

RIPQ [38] represents the state-of-the-art in minimizing CLWA; it is an SSD-based photo cache that minimizes CLWA by inserting objects that were read k times in the past together¹. When objects are first inserted into the cache, they are buffered in memory, and periodically they are moved into flash together as a segment with other objects that have been read the same number of times. The idea is that objects that were read k times in the past might share a similar future eviction rank. For example, an object that was read once is stored on flash in the same segment with other objects that were read once. Segments that contain objects that have been read fewer times will be evicted faster than segments with objects that have been read many times.

RIPQ works for photos, which are large and immutable, but it breaks down on web cache workloads where values are small and updated more frequently. To illustrate, we simulated the CLWA of RIPQ (the RIPQ implementation is not publicly available) with the Memcached traces using a segmented LRU with 8 queues. We also compared it with a victim cache policy, a naïve approach where the SSD simply serves as an L2 cache (i.e., every object evicted from DRAM is written to SSD). This policy is used by TAO [11], Facebook’s graph data store, which leverages a limited amount of flash as a victim cache for data stored in DRAM. The simulation assigns the same amount of memory for each application in the trace, with a ratio of DRAM to SSD of 1:7.

The results are presented in Table 3 and show that, while RIPQ considerably improves upon victim cache, it still suffers from a very high CLWA. Note that the victim cache would suffer from an even greater total WA, because it also suffers from DLWA (since it does not write to flash in large segments). RIPQ suffers from CLWA for two reasons. First, RIPQ has no admission policy and it writes *all* incoming objects to flash; even unread objects or objects that are frequently updated. Second, when the frequency of reads of a

¹Non-cache SSD key-value systems that store data persistently [5, 9, 25, 27] are not affected by CLWA, because they do not evict objects (all data fits in the database)

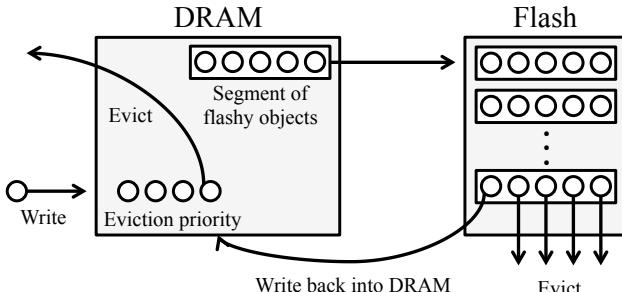


Figure 3: Lifetime of an object in Flashield. Objects always enter into DRAM. Objects that are a good fit for flash (*flashy* objects) are aggregated and moved into flash as a segment. The decision of whether to evict objects from DRAM or flash is based on a global eviction priority.

certain object changes, it creates additional writes. For example, if an object was read twice over a period of time after it was written, it is grouped with other objects that were read twice on flash. However, if it was read five more times, RIPQ needs to rewrite it to group it with other higher ranking objects. Since the objects are much smaller than the segment size, and there is a relatively high ratio of writes in the trace, RIPQ struggles to guarantee that objects that have been read around the same time will be stored in the same segment.

These results give two clues on how a cache should exploit DRAM differently to minimize CLWA for web cache workloads. First, not every object inserted into the cache by the application is a good candidate to be stored on SSD. For example, objects that are updated soon after they are first written or objects that have a low likelihood of being read in the future. However, the occurrence of such objects varies widely across different applications. For example, in some applications of the Memcached trace, more than half of written objects are never read again, and in some applications, a vast majority of objects are read many times and should be written to the cache. Second, due to the disparity between the segment size and the object size, it is difficult to guarantee that objects that were similarly ranked by the eviction policy will be stored in physically adjacent regions on SSD.

Both of these insights motivate Flashield, a cache that successfully minimizes CLWA with no DWLA.

3 Design

The design goal of Flashield is to minimize cache-level and device-level write amplification, while maintaining comparable hit rate. The key insights of Flashield’s design are to use DRAM as a filter, which prevents moving objects into flash that will be soon thereafter evicted or updated, and to maintain an efficient in-memory index which retains low write and read amplification.

Figure 3 illustrates the lifetime of an object in Flashield. Objects are first always written to DRAM. After the object is read for the first time, Flashield starts collecting features that describe its performance. These contain information about

when and how many times the object has been read and updated. An object may be evicted from DRAM by Flashield’s eviction algorithm.

Periodically, Flashield moves a segment (e.g., 512 MB) composed of many DRAM objects into flash. Flashield uses a machine learning classifier to rank objects based on their features. If an object passes a rank threshold, it will be considered as a candidate to move to flash. The candidates to flash are then ranked based on their score, which determines the order they are moved by Flashield into flash. This order is important when there are more flashy candidates than can fit in a single segment. After it gets moved to flash, an object will live in the cache for a relatively long duration. It will get moved out of flash once its segment is erased from flash, in FIFO order. At that point, the object will be evicted if it is low in terms of eviction priority, or it will get re-inserted into DRAM if it has a high eviction priority. Once the object is re-inserted into DRAM, it will have to prove itself again as flash worthy before it is re-written to flash. For more details, see §4.3.

In Flashield, DRAM serves three purposes. First, it is used as a filter to decide which objects should be inserted into SSD. Second, it stores the metadata for looking up and evicting objects on flash. Third, it serves as a caching layer for objects before they are moved to SSD and for objects that are not candidates for SSD.

3.1 DRAM as a Filter

In Flashield, DRAM serves as a proving ground for moving objects into flash. When objects are first written into DRAM, Flashield does not have a-priori knowledge whether they will be good candidates for flash. Furthermore, applications have unique workloads, so their access patterns need to be learned individually.

A strawman approach for determining which objects are flash-worthy is to rank them based on simple metrics like recency or frequency, as done by standard cache replacement policies like LRU or LFU. However, it is difficult to set a single threshold for flash-worthiness that will work for all applications. For example, the system can define a frequency-based threshold, requiring that an object will be read more than once before it enters flash. However, for some applications, such a threshold proves too stringent where the access patterns are long and reduces the hit rate due to premature evictions. It can also be too lenient for other applications, in which objects would be unnecessarily written to flash. Even for a single application, such a threshold is a heuristic that would have to be manually tuned (see the example described below and depicted in Table 4).

Instead of using a one-size-fits-all approach, machine learning can be used as a way to dynamically learn which objects are flash-worthy for each individual application.

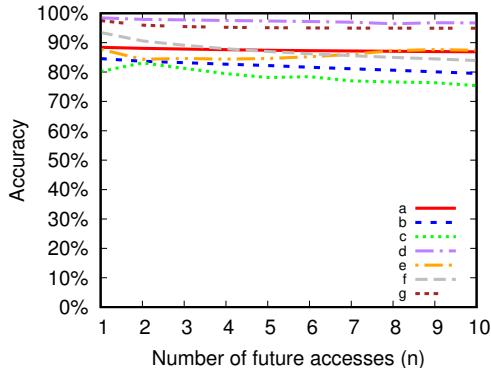


Figure 4: Accuracy of SVM classifier in different Memcached applications, for predicting whether an object will be accessed at least n times in the future without updates.

3.2 Flashiness

We define *flashiness* as a metric that predicts whether an object will be a good fit for flash. An object that has a high flashiness score is an object that meets two criteria. First, it is an object that will be accessed n times in the near future (where n is a configurable parameter). This guarantees that it will not be evicted by the cache’s eviction function. Second, it needs to be immutable in the near future, since updating an object in SSD requires an additional write.

Note that the threshold n , the number of times an object will be read in the future, can be used by the system to indicate how sensitive it is to write amplification. If the system is very sensitive to write amplification, it can set n to a relatively high number, which will ensure that Flashield will only move objects into flash that it predicts will be read many times in the future. On the other hand, if the system is more sensitive to hit rate, n will be set as a low number. In addition, Flashield allows the operator to set a fixed limit on the flash write rate to maintain a certain target lifetime.

Both of the above flashiness criteria can be captured by predicting the number of times an object will be read in the near future (e.g., one hour), and omitting objects that are predicted to be updated during this period.

Flashield uses a binary classifier using Support Vector Machine (SVM) to predict flashiness, by collecting two features: (1) number of past reads and (2) number of past updates. Figure 4 provides the accuracy of the classifier on different applications from the Memcached traces, with variable n values. Accuracy is defined as $\frac{tp + tn}{tp + tn + fp + fn}$, where tp is true positives, tn is true negatives, fp is false positives, and fn is false negatives. The classifier tries to predict whether an object will be accessed at least n times in the future without being updated, using a training time of one hour.

The accuracy of the prediction varies among the different applications (from 75% to 99%), due to their varying workloads. In addition, the accuracy generally decreases as n increases. This is because as n increases, the classifier is trying to predict more rare events, of which it has observed fewer

App	a	b	c	d	e	f	g
Num Accesses	5	4	5	2	4	4	6

Table 4: The threshold of the number of past accesses that predict whether an object will be accessed 5 times or more in the next hour.

training data points. For example, there are more objects that have been read more than once in the following hour, than objects that have been read five times or more.

To demonstrate why machine learning is more effective than having a fixed threshold of the number of past accesses for determining flashiness, consider the following example. We trained a simple classifier across the applications from the trace, which tries to predict flashiness with $n = 5$, utilizing a single feature (number of past reads), using a decision tree with a depth of 1. Table 4 presents the thresholds that the decision tree chose for each application, which would provide the highest prediction accuracy, based on its training samples. The results demonstrate that there is no one static threshold that would be optimal for all applications. This also shows that it is difficult to determine what this threshold would be a-priori. For example, for application d, only two reads occurring in the past is sufficient to predict that it will be read 5 more times or more in the future.

3.3 Flashiness Design Discussion

We experimented with several different features related to the number and frequency of reads and updates. We found that the only features that were impactful in the prediction and capture past information on reads and update are: (1) number of past reads and (2) number of past updates.

To our surprise, we found that across all the applications we measured, features related to recency (e.g., time between reads, time since the last read) had no positive impact on predictions, and in fact, in some instances reduced classifier accuracy. This supports our design choice to decouple the flashiness metric, which is based on number and type of past accesses, from the eviction policy, which is typically based on recency (e.g., LRU or one of its derivatives, see §3.4).

In addition, we experimented with several different classification algorithms. Initially, we tried predicting this number directly using a logistic regression. We ran this classifier on the Memcached trace and found the prediction was highly inaccurate. After trying different features and classifiers, we found it is difficult to accurately predict exactly how many times an object will be accessed in the future, which is why we use binary classification, which predicts whether the number of future reads is above n . We also tried using a different binary classifier, decision trees, which provided very similar accuracy to SVM. We decided to use SVM, because they provide a continuous score, which is used to provide a global flashiness rank for objects. With decision trees, the range of the score is limited to the number of leaves.

3.4 DRAM as an Index for Flash

Unlike log-structured merge trees (LSM), Flashield stores the index in DRAM (both for objects in DRAM and in flash). This allows Flashield to service requests at much lower latency, since the index is read from DRAM. More importantly, storing the index on flash requires LSMs to constantly update the index when objects get updated, which creates a large number of writes [24, 27, 39]. When the index is on DRAM, it is trivial to update it. However, since Flashield uses DRAM also as an admission control layer, we must ensure that indexes will consume a minimal amount of space on DRAM.

Similar to Memcached, Flashield stores its index in a hash-table to enable efficient lookups. A naïve index would contain the identity of the keys stored in flash, the location of the values, and their position in an eviction queue. However, such an index would be prohibitively expensive. If we take an example of a 6 TB flash device with an average object size of 257 bytes (equal to the average object size of the top 20 applications in the Memcachier trace), storing a hash of the key for each object that avoids collisions requires at least 8 bytes, storing the exact location of each object would be 43 bits, and keeping a pointer to a position in a queue would be 4-8 bytes. Storing 17 bytes per object on DRAM would require 406 GB of DRAM. This would take up (or exceed) all of the DRAM of a high end server. In RIPQ, for example, each in-memory index entry is 22 bytes. We design a novel in-memory lookup index for variable-sized objects that uses less than 4 bytes per object, without incurring additional flash write amplification.

Identities of keys. Rather than storing the identities of keys in the index, Flashield keeps them only in the flash device, as part of the object metadata. In order to identify hash collisions in the lookup hash-table, Flashield compares the key from flash. To limit the number of flash reads during key lookup and avoid complex table expansions, Flashield utilizes a multiple-choice hash-table without chains. During lookup, pre-defined hash functions are used one by one, such that if the key is not found, the next hash function is used. If all hash functions are used and the key was still not found then Flashield returns a miss. Similarly if a collision happens during insertion, the key is re-hashed with the next hash function to map it to another entry in the lookup table. If all hash functions are used and there is still a collision, the last collided object is evicted to make space for the new key.

To reduce the number of excess reads from the flash in case of hash collisions, Flashield utilizes an in-memory bloom filter for each segment, which indicates whether a key is stored in the segment. We decided to use a bloom filter per segment, rather than a global bloom filter, to eliminate the need of the bloom filter to support deletions (since each segment is immutable). We use bloom filters with a false positive rate of 1%. For the Memcachier trace, this translates to an average of 1.03 accesses to flash for every hit in

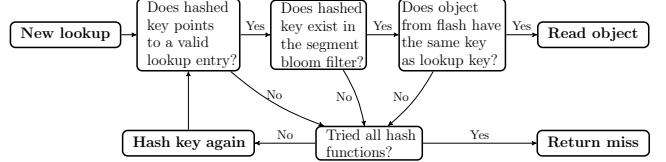


Figure 5: Algorithm for determining if an object exists in flash.

the flash and an extra memory overhead of 10 bits per item.

Object location. Instead of directly storing the location of the SSD object, the index contains two separate fields: segment number and the ID of a predefined hash function. The segment number represents a contiguous segment in flash where the object is stored. Hashing the object’s key using the predefined hash function provides the offset of the object within the segment. Using a hash function to indicate the object location in the segment may reduce flash utilization, because it limits the number of possible positions for placing an object within a segment. Note that these hash functions are orthogonal to the hash functions used for the hash-table lookup. We chose to utilize 16 pre-defined hash functions (i.e., up to 16 possible positions for an object) since increasing the number of hash functions beyond that provided negligible improvement in the flash utilization. We explore the flash utilization in §5.3. Note that since data is written to flash sequentially, segment sizes of 8 MB or larger achieves minimal DLWA. We use 512 MB segments in order to reduce the indexing overhead.

Eviction policy. To avoid the overhead of maintaining a full eviction queue composed of a doubly-linked list of pointers, Flashield uses the CLOCK algorithm [16], similar to other memory key-value caches [18]. CLOCK approximates the LRU policy, so to evaluate its impact we ran the top 5 applications in the Memcachier trace in a simulation and compared the results between CLOCK and LRU. The results show that by keeping just two bits per object for CLOCK timestamps, the hit rate decreases by an average of only 0.1% compared to LRU.

Figure 5 summarizes Flashield’s lookup process. The lookup key is first hashed to find the corresponding entry ID in the lookup hash-table, which provides the segment ID. Then, Flashield performs a key lookup in the segment’s bloom filter. If the key is found in the bloom filter, Flashield reads the object from the segment on flash. Since the bloom filter may cause a false positive, if the object that was read from flash does not have the same key as the object which is being looked up, the key will be hashed again and Flashield will look it up again in the lookup hash-table. Similarly, if the key is not found in the bloom filter, the key is hashed again and Flashield performs another lookup in the lookup hash-table. Flashield will attempt to lookup an object using all the configured hash functions (16 by default) until the object is found. If the object is not found after all attempts, the object does not exist in flash and Flashield returns a miss.

The hash-table entry format is summarized in Figure 6.

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ghost	Clock	Hash	Function ID																	Segment Number

Figure 6: Hash-table entry format for objects stored on flash.

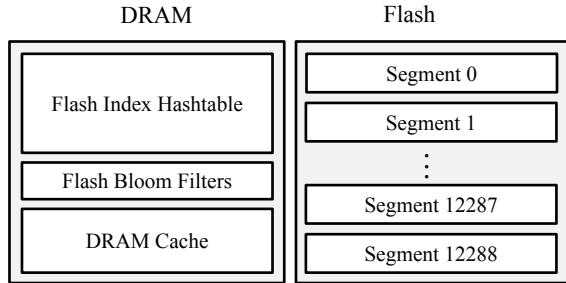


Figure 7: Flashield’s architecture. The flash index is an in-memory hash table. The bloom filters provide fast lookups for object existence in flash, and the rest of the DRAM is a cache. Most of the cache objects are stored on flash in segments.

The index contains an extra bit (*ghost*), that indicates whether the object is scheduled for deletion from flash. We describe the purpose of this flag at §4.3.

4 Implementation

This section presents the implementation of Flashield. We implemented Flashield in C from scratch, except for the transport, dispatch, request processing, and the hash table for DRAM objects, which are borrowed from Memcached 1.4.15. Flashield has four main functions: read, write, move data to flash and evict. Figure 7 depicts the high level components of Flashield’s architecture. It supports the generic Memcached protocol, so applications that deploy Memcached can transparently utilize Flashield.

For reads, Flashield first checks whether the object exists in the hash table for DRAM objects, which is based on Memcached’s hash table. If not, it checks whether the object exists in flash using a separate hash table for flash objects. If the object exists either in DRAM or flash, Flashield returns it, otherwise the request is counted as a miss. Incoming writes and updates are always stored in DRAM first. In the case of updates, the updated object is stored in DRAM, and the old version is invalidated. Flashield always maintains free space in the size of a segment in DRAM for incoming writes.

Flashield uses a configurable number of worker threads that process the client requests in parallel. To maintain enough free space on DRAM, Flashield uses a dedicated cleaner thread that works in the background and is not on the critical path for normal request (read/write) processing. In addition, Flashield let the operator configure a flash write limit to guarantee a certain target lifetime. When the free space on DRAM drops below a segment size, if there are enough objects that meet a threshold for their flashiness score and the flash write rate limit was not reached, the cleaner copies them into a segment buffer. When the buffer is full, the cleaner writes the segment to flash and then frees the space the objects occupied in DRAM. Objects are moved to

flash in an order based on their flashiness score. When the SSD is full, the cleaner will remove the last segment from flash based on FIFO order.

For eviction, Flashield maintains a global priority rank for all objects, whether they are stored in DRAM or flash. Objects are evicted from Flashield based on this global priority. By default the priority is an approximation of LRU using CLOCK. If the next object for eviction is in DRAM, Flashield simply evicts it. If the next object for eviction is in flash, Flashield marks it as a *ghost* object, and it will be evicted when its segment is removed from flash. Note that the movement of data from DRAM into flash is decoupled from eviction. They are conducted in parallel and use different metrics to rank objects. Objects that are moved between the flash and DRAM always keep their global priority ranking. When there are not enough objects in DRAM that meet a threshold for their flashiness score, or the flash write rate reached its limit, the cleaner will evict items from DRAM to maintain sufficient free space.

The rest of the section describes in detail how Flashield moves objects into flash, and the implementation of Flashield’s classifier and eviction algorithm.

4.1 Writing Objects to Flash

Flashield constructs a flash-bound segment in DRAM, by greedily trying to find space for the objects in the segment one-by-one. The output bits of the pre-determined hash functions provide different possible insertion points in the segment for each object. Flashield first assembles a group of objects that need to be moved to flash based on their flashiness. It then tries to insert the objects from this group based on their size. Larger objects go first, because they require more contiguous space than smaller objects. In this process, some objects will not have available space in the segment. Flashield skips these objects and tries to insert them again next time it creates a new segment. We evaluate the resulting segment utilization in § 5.3.

4.2 Classifier Implementation

Flashield’s flashiness score is computed based on two features for each object. Since these features depend on information across multiple object accesses, the features for an object are only generated after an object has been read at least once. If an object has never been read, its flashiness score is automatically equal to zero.

Flashield periodically trains a separate classifier for each application. For the commercial traces we used, we found that a training period of one hour at the beginning of the trace was sufficient.

The naïve way to train the classifier would be to update the features at each access to the DRAM. However, this approach may oversample certain objects, which can create an unbalanced classifier. For example, if a small set of objects account for 99% of all accesses, multiple sets of features would be created for these objects, and the flashiness esti-

mation would be biased towards popular objects.

To tackle this problem, we implemented a sampling technique that generates a single sample for each object, chosen uniformly over all of its accesses during the training period. Instead of updating the features at each object access, Flashield does it only with a probability of $\frac{1}{n}$, where n is the number of times the object was read and updated so far.

To illustrate this sampling technique, consider the following example. Suppose an object was written for the first time, and then read. Its feature vector is: $[1, 0]$ (number of past reads, number of past updates). Since the number of reads and updates is equal to 1, the feature vector generated by its first read will be the feature we use for training at a probability of 1. If the object is updated (feature vector is now: $[1, 1]$), Flashield will keep the second set of features with a probability of $\frac{1}{2}$, since the number of reads and updates is equal to 2. This is equal to uniformly sampling the features from the first or second access. Each subsequent access will be sampled at a uniform probability of $\frac{1}{n}$, and the probability of prior accesses to be sampled will also be uniform.

After collecting the samples for an hour, we measure the number of times each of the objects is hit in the subsequent hour. This number is used as the target function for the training. After these two periods, Flashield trains the classifier using these training samples and labels.

4.3 Eviction

Flashield uses the CLOCK algorithm to rank objects for eviction. Instead of keeping precise priority rank, each object has only two CLOCK bits in its hash table entry that signify priority. In order to approximate LRU, when the object is read, its bits are all set to 1. MFU (Most Frequently Used) is approximated by incrementing the bits by 1 at each read.

When a set operation inserts an object into the cache, it may trigger an eviction. On eviction, Flashield walks round-robin through each object entry in the index, decrementing its CLOCK value by one. It stops the walk when it reaches an entry that has a CLOCK value of zero. This object is chosen as the next victim for eviction. If the victim object is in DRAM, its space is freed and may be reused for the incoming value. In case there is sufficient space after freeing the victim, eviction stops, otherwise the process repeats as needed. If the object is in flash, Flashield cannot delete it immediately from flash, since fine-grained writes to the SSD would incur high DLWA. Instead, the entry is marked as a *ghost object*, which acts as a hint to the flash cleaning process. Later, when the on-flash segment that the object resides is about to be overwritten, the ghost object will not be preserved, effectively freeing the storage as part of the bulk flash cleaning process. Even so, a ghost object is still accessible if it is the most current value associated with a particular key, so long as the flash cleaning process has not yet overwritten its segment on flash. In a sense, ghost objects approximate the bottom of the global eviction rank (including both flash

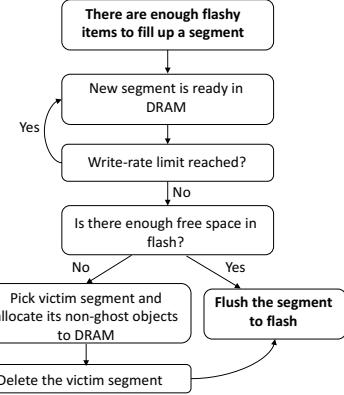


Figure 8: Flashield’s process of allocating and deleting a segment to and from flash.

and DRAM); non-ghost objects, are considered to be at the top of the global eviction rank and we call them *hot objects*.

Flashield triggers a segment deletion once a new segment is allocated and ready to be moved from DRAM to flash, given that the flash is full and the configured write rate limit was not exceeded. The cleaner removes the last segment from flash in FIFO order. During segment erasure, its ghost objects are removed from the cache, while hot objects are re-inserted into the DRAM. Figure 8 summarizes this process.

Moving objects from flash back to DRAM will trigger evictions; left unchecked this can create two issues. First, hit rates could suffer if objects are prematurely evicted from DRAM without proving they are flashy. Second, if too many flashy objects are evicted it can contribute to write amplification. Flashield guards against this with a *hot data threshold (HDT)*, which ensures that in the limit enough objects can be discarded during cleaning to free up sufficient space on flash, without placing too much pressure on eviction. Without HDT, the cleaner could re-allocate low ranked objects, at the expense of higher ranked objects residing in the DRAM.

The HDT is defined as $DRAM + SSD \cdot hot$, where *DRAM* is the available object storage in DRAM, *SSD* is the total size of the SSD, and *hot* is the percentage of SSD that is allocated for hot objects. Flashield strives to maintain the HDT, even when an incoming object has sufficient space in DRAM. To do so, whenever the amount of hot data exceeds the HDT, Flashield triggers a new eviction, which marks additional objects as ghost if they reside on flash. By default, *hot* is 70%, so about 30% of the objects on flash are ghost objects.

Ghost objects can still be accessed after they were marked as ghosts, since they are not immediately removed from flash. If a ghost object is accessed, it is not considered a ghost anymore and Flashield marks it as a hot object (the ghost bit is set to zero). Since Flashield always maintains the HDT, switching a ghost object from ghost to hot may trigger an eviction. To avoid unnecessary DRAM evictions, Flashield will not evict low ranking objects from DRAM in such case, but only walk through flash objects to mark ob-

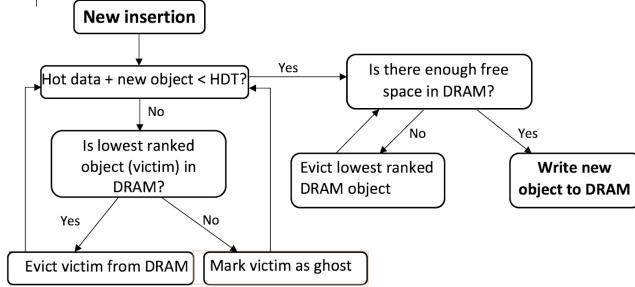


Figure 9: Flashield’s eviction process.

jects as ghosts.

Although the cleaner is responsible for maintaining enough free space in DRAM (by allocating new segments to flash), in rare occasions the DRAM may not have enough free space to accommodate an incoming write. This may happen when the flash write rate limit is reached, or if the number of objects with flashiness score above the threshold is not enough for forming a new segment. In such scenario, Flashield will trigger a special eviction where it will walk through the DRAM objects only, and will evict low ranking objects from DRAM to accommodate the incoming write.

Figure 9 demonstrates Flashield’s flow chart when a set operation inserts new object to the cache.

Delete operations in Flashield do not incur writes to flash. If the object is in DRAM, it is simply deleted. If it resides in flash, it is not immediately removed from flash, since that would incur DLWA. It is also not marked as a ghost, because ghost objects can still be accessed. Instead, Flashield deletes the object’s lookup entry. During segment eviction, the cleaning process identifies deleted objects by comparing the segment ID in their corresponding lookup entry with the evicted segment ID, and will not preserve them. Building on that, Flashield handles update operations as a delete operation followed by a new insertion.

5 Evaluation

In this section we evaluate the end-to-end performance of Flashield compared to existing systems. Unfortunately, to the best of our knowledge, there are no public traces of large-scale key-value caches. We use real-world traces of an entire week, provided by Memcached, a widely used Memcached service provider. Since the Memcached traces are fairly sparse in terms of their request rate, we ran a set of synthetic microbenchmarks to stress the performance of the system to measure its throughput and latency.

5.1 End-to-end Performance

We compare the end-to-end hit rate and write amplification of Flashield to RIPQ and the victim cache policy, by re-running real-world applications from the Memcached traces. Since no public implementation of RIPQ is available [38], we are forced to run and compare a simulation of the three systems. Each one of the policies uses the same amount of memory that was allocated in the Memcached trace, with a

App	Flashield		RIPQ		Victim Cache	
	Hit %	CLWA	Hit %	CLWA	Hit %	CLWA
a	98.8%	5.8	98.5%	151.9	99.3%	4536.3
b	98.6%	2.8	98.8%	4.4	98.9%	21.7
c	83.1%	0.4	83.1%	2.9	93.3%	3.7
d	98.1%	0.2	98.7%	12.4	99.3%	34.0
e	96.0%	0.8	96.0%	1.6	96.2%	1.3
f	90.1%	0.2	91.3%	1.8	94.4%	2.4
g	97.3%	0.5	97.3%	1.4	97.4%	1.0

Table 5: Hit rates and CLWA of Flashield using a threshold of one future read, RIPQ and victim cache.

App	Flashield 1		Flashield 10		Flashield 100	
	Hit %	CLWA	Hit %	CLWA	Hit %	CLWA
a	98.8%	5.8	99.0%	9.2	98.9%	5.0
b	98.6%	2.8	98.6%	2.7	95.2%	0.0
c	83.1%	0.4	83.1%	0.4	83.0%	0.4
d	98.1%	0.2	98.1%	0.2	98.1%	0.2
e	96.0%	0.8	95.9%	0.7	95.9%	0.7
f	90.1%	0.2	85.5%	0.0	85.2%	0.0
g	97.3%	0.5	97.3%	0.5	97.3%	0.5

Table 6: Hit rates and CLWA of Flashield using a flashiness prediction threshold of 1, 10 and 100 future reads.

ratio of 1:7 of DRAM and SSD. We run Flashield with a threshold of one future read. In other words, objects that are predicted to have at least one future read are deemed sufficiently flash-worthy. Since Flashield utilizes a separate SVM for each application, we compare the results of individual applications. To run RIPQ with 8 insertion points, and therefore at least 8 different segments on flash, we only run applications that were allocated a sufficient amount of memory by Memcached.

Table 5 presents the results comparing Flashield and RIPQ. The results show that Flashield achieves significantly lower CLWA than RIPQ and victim cache. The median CLWA of Flashield is 0.54, the median of RIPQ is 2.85 and the median of victim cache is 3.67. Even though Flashield uses a low threshold for flashiness of one future read, it still prevents a large number of writes that are not a good fit for SSD from being written to flash. Flashield and RIPQ have an almost identical hit rate. Both have a lower hit rate than victim cache, but victim cache suffers from significantly higher CLWA (and since it does not handle DLWA, also a much higher overall write amplification).

Table 6 compares Flashield with different flashiness prediction thresholds n . While the results vary from application to application, generally speaking, the higher the threshold the lower the CLWA and the lower the hit rate. Note that in some applications, such as in application a, this trade off does not hold, since we train the classifier individually on each application, and each application performs differently.

Table 7 depicts the results when we vary the ratio of DRAM and SSD, while keeping the total amount of memory constant for each application. The results show that if we reduce the amount of DRAM too much, the hit rate drops. This is due to the fact that when the DRAM is low, objects do not

App	DRAM 1:15		DRAM 1:7		DRAM 1:3	
	Hit %	CLWA	Hit %	CLWA	Hit %	CLWA
a	99.0%	5.1	99.0%	4.6	99.0%	2.6
b	98.3%	3.1	98.6%	4.1	98.8%	4.9
c	81.4%	0.4	83.2%	0.4	92.7%	0.8
d	97.6%	1.2	98.4%	0.9	98.9%	2.2
e	95.7%	0.7	96.0%	0.8	96.2%	0.9
f	89.0%	0.2	91.0%	0.3	94.3%	0.4
g	97.2%	0.5	97.3%	0.5	97.3%	0.5

Table 7: Hit rates and CLWA of Flashield using a threshold of 1, with varying ratios of DRAM and SSD. The results use a smaller segment size (2 MB).

	Flashield			Memcached	
	SSD Hits	DRAM Hits	Misses	Hits	Misses
Throughput (IOPS)	150K	270K	239K	275K	287K
Latency (μs)	106	13.5	19	13	12

Table 8: Throughput and latency of SSD hits, DRAM hits and cache misses for Flashield and Memcached.

have sufficient time to prove themselves as flashy enough to be moved to SSD before they are evicted from DRAM. Note that we used a smaller segment size in these runs, in order to be able to display results for a 1:15 ratio of DRAM.

5.2 Microbenchmarks

We drive Flashield’s implementation with microbenchmarks to stress the performance of the system, and compare its latency and throughput with Memcached. We use 4-core 3.4 GHz Intel Xeon E3-1230 v5 (with 8 total hardware threads), 32 GB of DDR4 DRAM at 2133 MHz with a 480 GB Intel 535 Series SSD. All experiments are compiled and run using the stock kernel, compiler, and libraries on Debian 8.4 AMD64. The microbenchmark requests are based on random keys, with an average object size of 257 bytes, which is the average object size of the top 20 application in the Memcachier trace. We disabled the operating system buffer cache to guarantee that SSD reads are routed directly to the SSD drive. Since the performance of SSD and DRAM is an order of magnitude different, we separately measured SSD and DRAM hits. Finally, we measured the latency and throughput of Memcached 1.4.15 as a baseline.

Table 8 presents the throughput and latency of the microbenchmark experiment. Note that in the case of both Memcachier and Facebook, Memcached is not CPU bound, but rather memory capacity bound [14, 15]. The latency and throughput of DRAM hits in Flashield are very similar to the latency and throughput of Memcached. While the average latency of SSD hits is significantly higher than DRAM, their latencies become similar when deploying over the network (network access times are typically 100 μs or more). The miss latency of Flashield is similar to the latency of DRAM hits, because all of Flashield’s lookup indices are stored in DRAM, and the only case it needs to access flash in a miss is when one of the in-memory bloom filters returns a

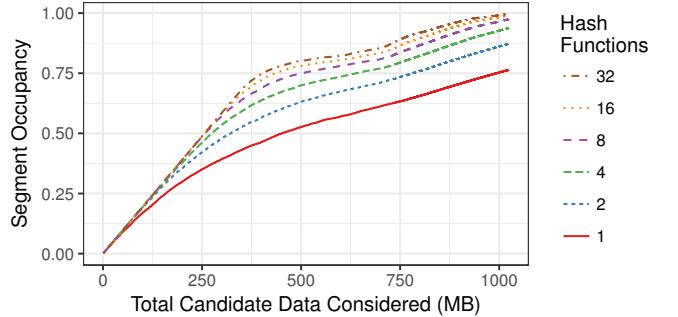


Figure 10: Utilization of a 512 MB segment on flash when Flashield tries to allocate space with a varying number of objects from the Memcachier trace. As Flashield tries to allocate more objects, it achieves higher utilization.

false positive. The write throughput and latency of Flashield were identical to Memcached, because writes always enter Flashield’s DRAM.

5.3 Utilization on Flash

When moving data from DRAM to flash, Flashield tries to allocate space for objects in different possible insertion points in the flash segment, using pre-defined hash functions. If none of the insertion point references to sufficient contiguous free space for the object, Flashield skips the object and will try to insert it during the next segment allocation.

Figure 10 depicts the utilization of Flashield’s flash allocation algorithm. To measure the utilization, we ran Flashield’s allocation algorithm on the Memcachier traces with different number of hash functions over a segment size of 512 MB. The allocation greedily tries to allocate space to more data and measures the resulting utilization. Note that after the segment reaches about 60% utilization, its utilization curve gradient decreases, since when Flashield tries to allocate objects there is a higher probability of collisions with other existing objects in the segment. Using 16 hash functions, it takes about 1 GB of objects to reach a 99% utilization, and on average each object needs to be hashed 8.2 times until it finds an insertion point with enough space.

6 Related Work

There are two types of prior research. There are several prior SSD-based key-value caches for specific workloads (e.g., photo cache, graph database), but all of them suffer from low flash lifetime under a general-purpose key-value workload with small keys and variable objects without leveraging specialized hardware. There is also a large number of prior SSD-based persistent key-value stores. Unlike caches, persistent stores do not maintain an admission control and eviction policies and do not suffer from CLWA, hence their write amplification problems are less severe.

SSD-based Key Value Caches Facebook’s flash-based photo cache evolved from McDipper [19] to BlockCache [2], and then to RIPQ [38], trying to improve hit rates while maintaining low write amplification. McDipper uses a simple FIFO policy, which causes it to suffer from low hit

rates. BlockCache improves cache hit rates by leveraging the SLRU policy which co-locates similarly prioritized content on flash, but incurs much higher write amplification than McDipper. RIPQ achieves even higher hit rates than BlockCache, while keeping its write amplification comparable to McDipper [2]. RIPQ performs insertions with priority-aware memory blocks, and uses virtual blocks to track the increased priority value when an item is accessed. However, in a general purpose key-value service like Memcached, RIPQ suffers from more than $5\times$ higher write amplification than Flashield, and up to $150\times$ on specific applications. Furthermore, RIPQ’s in-memory index map occupies 22 bytes per entry, consuming a very large amount of DRAM. Flashield’s novel index requires less than 4 bytes of DRAM per object. TAO [11], Facebook’s graph data store, uses a limited amount of flash as a victim cache for data stored in DRAM. Therefore, it suffers from a high rate of writes, because items which are not frequently accessed are written into flash and evicted soon after.

Twitter has explored SSD-based caching for its data center cache with Fatscache [21], a modified version of Memcached that buffers small writes and utilizes FIFO as an eviction policy. Flashield has better write amplification than Fatscache, since not all write requests are written to flash, and higher hit rates, because it uses eviction policies similar to LRU, which provide a higher hit rate than FIFO. Moreover, Fatscache’s in-memory index requires 32 bytes (or more) per entry, which is $8\times$ larger than Flashield.

A couple of systems try to support SSD-based caches by modifying the SSD’s Flash Translation Layer (FTL). Duracache [26] tries to extend the life of the SSD cache, by dynamically increasing the flash device’s error correction capabilities. Shen et al [37] allow the cache to directly map keys to the device itself, and remove the overhead of the flash garbage collector. Unlike these systems, Flashield addresses CLWA without any changes in the flash device.

Other than key-value caches, there are several systems that utilize flash as a block-level cache for disk storage [4, 22, 23, 33, 35, 40]. Unlike Flashield, storage blocks in these systems are always written to flash, and are fixed-sized (typically kilobytes in size). For this reason, they use a naive (inefficient) in-memory index to map from block’s key to a location in flash. These properties make them impractical for general purpose key-value workloads with a variable and on average small object sizes.

Cheng et al [12] present an offline analysis of the trade-off between write amplification and eviction policies in block-level caches. They generalize Belady’s MIN algorithm to flash-based caches, and demonstrate that LRU-based eviction is far from the optimal oracle eviction policy. However, they do not provide an online algorithm and an implementation that reduces write amplification of SSD-based caches.

SSD-based Key Value Stores Since these systems are persistent stores, all objects must be eventually written to flash,

and thus they do not maintain an admission control and eviction policies, which are necessary for cache systems like Flashield. Consequently, persistent key-value stores do not suffer from CLWA and its implications, so their lifetime constraints are less severe than in a cache workload. However, they still strive to minimize write amplification for performance, since they must still suffer write amplification costs to compact data and update their indexes.

Systems such as LevelDB [5] and RocksDB [9] store the entire dataset and index on flash using Log-structure Mergetrees (LSM), and buffer writes to flash in DRAM to avoid DLWA. To enable efficient lookups, LSM-trees continuously perform a background compaction process that sorts and re-writes key-value pairs to flash, creating a major write amplification, particularly for workloads like key-value caches. WiscKey [27] reduces write amplification by separating keys and values. Keys are kept sorted in the LSM-tree, while values are stored separately in a log, which is helpful for workloads with large value sizes. PebblesDB [34] aims to reduce write amplification during compaction by using Fragmented Log-Structured Merge Trees (FLSM), avoiding rewriting data in the same tree level. In addition, NVMKV [28] is a key-value store that relies on advanced FTL capabilities (advanced multi-block writes) to deliver higher performance and lower write amplification. SILT [25] is a flash key-value database that minimizes the index stored in memory by utilizing three basic key-value stores. Objects are inserted first to a write-optimized store, and then re-written and merged into increasingly more memory-efficient stores. The majority of the objects are stored in the most memory-efficient store, making the average index cost per key low. However, unlike Flashield, SILT is not optimized for write amplification, and assumes values are fixed-length.

7 Conclusions

SSD faces unique challenges to its adoption for key-value cache use cases, since the small object sizes and the frequent rate of evictions and updates create excessive writes and erasures. Flashield is the first key-value cache that uses DRAM as a filter for objects that are not ideal for SSD. Flashield profiles objects using lightweight machine learning, and dynamically learns and predicts which objects are the best fit for flash. It introduces a novel in-memory index for variable sized objects with an overhead of less than 4 bytes per object, without sacrificing the flash write and read amplifications.

The ideas in this paper can be extended to other use cases. For example, non-volatile memory (NVM) faces durability challenges too, especially when used as a replacement for DRAM, and may also require an admission policy [17]. This is also the case in multi-tiered storage systems, where cheaper storage layers offer more capacity at the expense of decreased performance. Finally, dealing with the durability of flash becomes an ever more pressing issue, as its density increases (and its ability to tolerate writes decreases).

References

- [1] Amazon ElastiCache. aws.amazon.com/elasticache/.
- [2] The evolution of advanced caching in the facebook cdn. <https://research.fb.com/the-evolution-of-advanced-caching-in-the-facebook-cdn/>.
- [3] Facebook asks for QLC NAND, Toshiba answers with 100TB QLC SSDs with TSV, author = Alcorn, P, note = http://www.tomshardware.com/news/qlc-nand-ssd-toshiba-facebook_32451.html.
- [4] Flashcache. github.com/facebookarchive/flashcache.
- [5] LevelDB. leveldb.org/.
- [6] Memcached. memcached.org/.
- [7] Memcachier. www.memcachier.com.
- [8] Redis. [http://redis.io/](https://redis.io/). 7/24/2015.
- [9] RocksDB. rocksdb.org/.
- [10] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.
- [11] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook’s Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 49–60.
- [12] CHENG, Y., DOUGLIS, F., SHILANE, P., WALLACE, G., DESNOYERS, P., AND LI, K. Erasing belady’s limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 379–392.
- [13] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Dynocache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, July 2015), USENIX Association.
- [14] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 379–392.
- [15] CIDON, A., RUSHTON, D., RUMBLE, S. M., AND STUTSMAN, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 321–334.
- [16] CORBATO, F. J. A paging experiment with the multics system. Tech. rep., DTIC Document, 1968.
- [17] EISENMAN, A., GARDNER, D., ABDELRAHMAN, I., AXBOE, J., DONG, S., HAZELWOOD, K., PETERSEN, C., CIDON, A., AND KATTI, S. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys ’18, ACM, pp. 42:1–42:13.
- [18] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi’13, USENIX Association, pp. 371–384.
- [19] GARTRELL, A. Mcdipper: A key-value cache for flash storage. <https://code.facebook.com/posts/223102601175603/mcdipper-a-key-value-cache-for-flash-storage/>.
- [20] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST’12, USENIX Association, pp. 2–2.
- [21] HOERNER, B., RAJASHEKHAR, M., YUE, Y., AND NYMEN, T. Fatcache. engineering.twitter.comopensource/projects/fatcache.
- [22] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC’14, USENIX Association, pp. 501–512.
- [23] LI, C., SHILANE, P., DOUGLIS, F., AND WALLACE, G. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware ’15, ACM, pp. 50–62.
- [24] LIM, H., ANDERSEN, D. G., AND KAMINSKY, M. Towards accurate and fast evaluation of multi-stage log-structured designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 149–166.
- [25] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP ’11, ACM, pp. 1–13.
- [26] LIU, R.-S., YANG, C.-L., LI, C.-H., AND CHEN, G.-Y. Duracache: A durable ssd cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference* (New York, NY, USA, 2013), DAC ’13, ACM, pp. 166:1–166:6.
- [27] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 133–148.
- [28] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, 2014), USENIX Association.
- [29] MELLOR, C. Toshiba flashes 100TB QLC flash drive, may go on sale within months. really. http://www.theregister.co.uk/2016/08/10/toshiba_100tb_qlc_ssdd/.
- [30] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2015), SIGMETRICS ’15, ACM, pp. 177–190.
- [31] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.
- [32] OHSHIMA, S., AND TANAKA, Y. New 3D flash technologies offer both low cost and low power solutions. <https://www.flashmemorysummit.com/English/Conference/Keynotes.html>.
- [33] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. Sdf: Software-defined flash for web-scale internet storage systems. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 471–484.
- [34] RAJU, P., KADEXODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium*

- on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).
- [35] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 267–280.
 - [36] SCHROEDER, B., LAGISSETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 67–80.
 - [37] SHEN, Z., CHEN, F., JIA, Y., AND SHAO, Z. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016).
 - [38] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 373–386.
 - [39] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 71–82.
 - [40] YANG, Q., AND REN, J. I-CASH: Intelligently coupled array of SSD and HDD. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture* (Washington, DC, USA, 2011), HPCA '11, IEEE Computer Society, pp. 278–289.

Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores

Diego Didona
EPFL

Willy Zwaenepoel
EPFL and University of Sydney

Abstract

This paper introduces the concept of size-aware sharding to improve tail latencies for in-memory key-value stores, and describes its implementation in the Minos key-value store.

Size-aware sharding distributes requests for keys to cores according to the size of the item associated with the key. In particular, requests for small and large items are sent to disjoint subsets of cores. Size-aware sharding improves tail latencies by avoiding that a request for a small item gets queued behind a request for a large item.

Minos uses hardware dispatch for all requests for small items, which form the very large majority of all requests, to achieve high throughput, and achieves load balancing by adapting the number of cores handling requests for small and large items to their relative presence in the workload.

We compare Minos to three state-of-the-art designs of in-memory KV stores. Compared to its closest competitor, Minos achieves a 99th percentile latency that is up to 20 times lower. Put differently, for a target 99th percentile latency equal to 10 times the mean service time, Minos achieves a throughput that is up to 7.4 times higher.

1 Introduction

Many distributed applications use in-memory key-value (KV) stores as caches or as (non-persistent) data repositories [3, 10, 13, 34, 41, 44, 51, 55]. Many of these applications exhibit a high fan-out pattern, i.e., they issue a large number of requests in parallel [55]. From the application’s standpoint, the overall response time is then determined by the slowest of the responses to these requests, hence the crucial importance of tail latency for KV stores [17].

The performance of KV stores has been the subject of much work, both in terms of software and hardware. Software optimizations include zero-copy network stacks, polling, run-to-completion processing, and sharding of requests among cores [37, 45, 57]. Hardware optimizations primarily rely on the use of RDMA [35, 36], programmable

NICs [38, 41] or GPUs [30, 64]. The work reported in this paper does not require any particular hardware support. We assume only commodity NICs with multiple queues and a hardware mechanism to direct requests to a particular queue.

Variable item sizes and tail latency. The workload observed for many KV stores consists of a very large number of requests for small items and a much smaller number of requests for large items [3, 9, 55]. Because of their higher service times, however, handling the requests for larger items consumes a significant share of the available resources. Processing these large items therefore increases the probability of head-of-line blocking, a situation in which a request for a small item ends up waiting while a large item is being processed. As a result of the wait, that request experiences additional latency, which in turn may increase the tail latency of the KV store. Even a very small number of requests for large items can significantly drive up tail latencies. As we show in Section 2.2, a percentage of large requests smaller than N% can lead to a substantial increase of the (100-N)th percentile.

Size-aware sharding. This paper introduces the notion of size-aware sharding to address this issue. In general, size-aware sharding means that requests for items of different sizes go to different cores. In its simplest form, it means that, for some cutoff value between small and large, small and large items are served by disjoint sets of cores. The intuition behind size-aware sharding is that by isolating the requests for small items, they do not experience any head-of-line blocking, and, given that they account for a very large percentage of requests, the corresponding percentile of the latency distribution is improved.

The implementation of size-aware sharding poses several challenges. A first challenge is how to use hardware dispatch of an incoming request to the right core. In general, a client of the KV store does not know the size of an item to be read, and moreover it does not know which cores are responsible for small or large items. Therefore, size-aware sharding would seem to necessitate a software handoff in which an I/O core reads incoming requests and dispatches them to the

proper core. Instead, we demonstrate a method by which software dispatch is required only for the very small number of requests for large items. Second, cutoff values between large and small items must be chosen and the proper number of cores must be allocated for handling small and large items. We show that, even in the presence of a workload that varies over time, this can be done by a simple control loop.

Minos. This paper describes the Minos in-memory KV store that implements size-aware sharding. We compare Minos to alternative size-unaware designs based on keyhash-based request sharding, software handoff and work stealing, implemented by state-of-the-art systems such as MICA [45], RAMCloud [57] and ZygOS [58].

We show that Minos achieves a 99th percentile latency that is up to two 20 times lower than the second best approach. Put differently, for a given value for the 99th percentile latency equal to 10 times the mean service time, Minos achieves a throughput that is up to 7.4 times higher.

Contributions. The contributions of this paper are:

- 1) the introduction of the notion of size-aware sharding for in-memory KV stores,
- 2) the design and implementation of the Minos KV store that implements size-aware sharding efficiently, and
- 3) the evaluation of Minos against state-of-the-art size-unaware designs.

Outline of the paper. Section 2 provides background on KV store workloads and discusses the shortcomings of existing approaches in achieving low tail latency. Section 3 presents Minos’ size-aware sharding approach. Section 4 discusses implementation details. Section 5 describes the experimental environment. Section 6 presents experimental results. Section 7 discusses related work. Section 8 concludes the paper.

2 Background

2.1 Item Sizes in Production KV Workloads

The sizes of the items stored and manipulated by KV stores in production environments can span orders of magnitude. For instance, large variations in item size have been reported in several deployments of the popular memcached KV store [51]. The Facebook ETC memcached pool stores items that vary in size from a handful of bytes to 1 Mbyte [3]. The size distribution is heavy-tailed: the 5th percentile in the regional pool is 231 bytes, while the 99th percentile is 381KB [55]. A similar degree of variability in item size has also been reported for other KV deployments such as Wikipedia [46] and Flickr [9], where item sizes span up to 4 orders of magnitude, from 500B to 1 MB.

Moreover, Atikoglu et al. report that in the ETC memcached pool at Facebook requests for large items, despite being rare, consume a large share of the computational resources, because service times are closely related

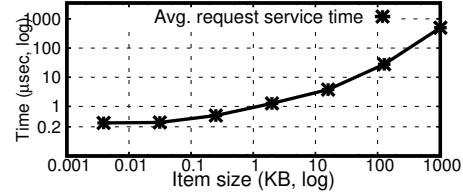


Figure 1: Service time of GET operations on items of different sizes on our platform (axes in log scale). The service time measures the interval from the reception of the client request on the server to the transmission of the reply. To avoid queueing effects, only one client performs operations. The time to process a large item can be up to almost four orders of magnitude higher than what is needed for a small one. This is due to the higher time needed to copy the content of the item to the network packets that are placed on the TX queue of the NIC.

to item size, and account for a significant fraction of the transferred data [3]. This dynamic is consistent with observations from similar application domains, such as, e.g., web servers [2, 15] and large-scale clusters [62].

2.2 Variations in Item Size and Tail Latencies

Variations in item size have profound implications for tail latencies. As anecdotal evidence, Nishtala et al. report that in the Facebook memcached servers the median response time is 333 microseconds, while the 95th percentile is 1.135 milliseconds [55]. In this section we show that this finding goes beyond the anecdotal, and that all common size-unaware sharding techniques exhibit high tail latencies for workloads in which even only a small fraction of requests targets large items. In particular, we show that, even under moderate loads, the (100-N)th percentile is affected dramatically by a fraction, much smaller than N%, of requests for large items. In the following we report on the 99th percentile, commonly used in Service Level Objective (SLO) definitions, but the results apply also to other high percentiles.

We simulate three common size-unaware sharding techniques on a server with 8 cores, each with a queue to store incoming requests¹:

- **Early binding:** requests are dispatched to a queue for a particular core, often based on a keyhash, similar to what is used, for instance, in the EREW version of MICA [45].
- **Late binding:** requests are kept in a single queue and dispatched to a core when it becomes idle, similar to what is used, for instance, in RAMCloud [57].

¹The goal of this simulation is *not* to predict quantitatively the performance differences between these strategies in any real implementation, as their performance is affected by factors such as locality, cost of synchronization, and cost of dispatching, which we do not simulate. Our goal is to demonstrate, for all three methods, the substantial increase in tail latency as a result of the presence of a small fraction of requests for large items.

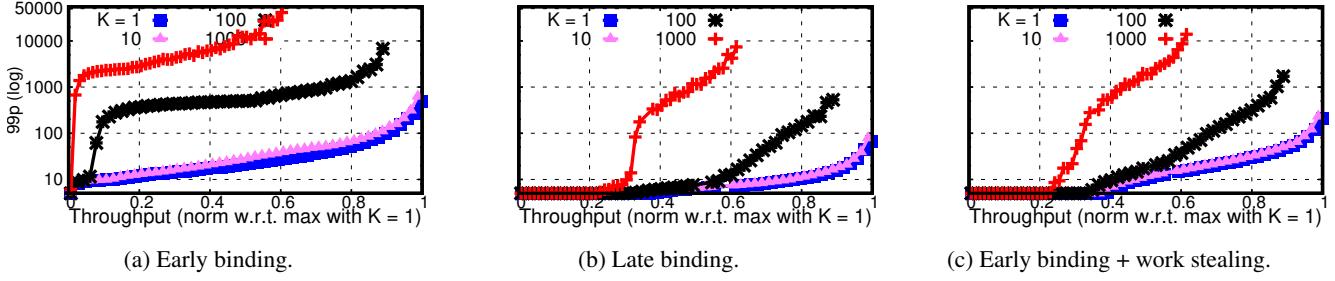


Figure 2: Throughput vs. 99th percentile of response times for different types of size-unaware sharding techniques (y axis in log scale). The workload distribution is bimodal: 0.125% of requests is for large items, whose service time is K time units; the remaining is for small ones, whose service time is 1 time unit. K is varied from 1 to 1,000. $K = 1$ corresponds to a baseline workload with only small requests. A small (<1%) fraction of large requests suffices to hamper greatly the 99th percentile of response times, and to considerably reduce the achievable throughput.

- **Early binding with work stealing:** requests are handled as in the early binding case, but in addition idle cores steal requests from the queues of other cores, similar to what is used, for instance, in ZygOS [58].

For simplicity, we use a workload with a bimodal size distribution. Small requests form 99.875% of the workload, and have a service time of 1 time unit. Large requests form the remaining 0.125%. We run different simulations in which the service time of large requests is, respectively, $K = 1, 10, 100$ and 1,000 time units. These values are in line with the order-of-magnitude differences in service time between small and large items observed on our platform (see Figure 1). We use $K = 1$ to establish a baseline where all requests are small. Inter-arrival times follow an exponential distribution.

Figure 2 shows the 99th percentiles for the three sharding strategies under the bimodal workload compared to a workload with an identical offered load, but with only requests for small items. Even though the fraction of large items requested is much smaller than 1%, all three strategies suffer from a considerable increase in the 99th percentile latency. For $K = 100$ and $K = 1,000$, at only 10% utilization the 99th percentile for the early binding design is two orders of magnitude higher than the 99th percentile in the workload composed only of small requests. Stealing and late binding are more resilient to service time variability at low load, but at higher loads they also suffer from one or two orders of magnitude degradation of the 99th percentile, with respect to the workload with only small requests.

The reasons for these increases in the 99th percentile latency are different from one strategy to the next. Early binding suffers from head-of-line blocking when a request for a small item ends up in a queue behind a request for a large item, or behind a request for a large item being executed by this core. The late binding of requests to cores is more resilient to head-of-line blocking, a well known result from queueing theory [28], but it does not avoid it. Late binding is vulnerable to cases in which the arrival of many large re-

quests in a short period of time leads many (or even all) cores to be busy serving large requests. Such an event temporarily reduces the amount of resources available to serve small requests, which impacts tail latency. Stealing improves the tail latency of the early binding design, as it steals some of the requests that would otherwise experience head-of-line blocking but it cannot completely avoid head-of-line-blocking. First, stealing only occurs when a core is idle, and the likelihood of a core being idle decreases as the load increases. Second, by the time a core becomes idle, a request that it steals is likely to have already experienced some head-of-line blocking in the queue from which it is stolen.

In light of these results, Minos processes requests for small and large items on disjoint set of cores, a technique we call *size-aware sharding*. This addresses the shortcomings of existing approaches, by avoiding that a small request waits for the completion of a large one.

3 Minos design

3.1 Size-aware sharding

Preliminaries. We consider a server with n cores. The server has a NIC with multiple receive (RX) and transmit (TX) queues. We configure the NIC to use n RX queues and n TX queues. At any time, there are n_l cores handling requests for large items and n_s cores handling requests for small items ($n_l + n_s = n$). With a slight abuse of language, we say that a request for a small (large) item is a small (large) request, and that a core handling small (large) requests is a small (large) core. In addition to an RX and a TX queue, each large core maintains a software queue.

In the following, we assume all n cores are within the same NUMA domain, so that KV item accesses and inter-core communication happen within the same NUMA domain. Minos can seamlessly scale to multiple NUMA domains by running an independent set of small and large cores

within each NUMA domain, and by having clients send requests to the NUMA domain that stores the target key [45].

We consider a KV store with the usual CRUD (Create, Read, Update, Delete) semantics. A client can perform a GET(key) and a PUT(key, value). Create and delete are considered special versions of PUT, and not discussed any further. When a client issues GET and PUT operations, the client software puts in the request the id of the RX queue in which the corresponding packets are deposited when they arrive at the server. The target RX queue is chosen at random for GET operations, and depends on the keyhash for PUT operations (as we describe in Section 4.2). A PUT request also includes the size of the item that is being written. The client does not know the size of an item to be read. Furthermore, the client does not need to know which or how many cores on the server handle small or large requests.

In the following discussion we initially assume that we know the threshold on the item size that separates small and large items. We explain later how the threshold is determined. We first explain size-aware sharding with a given number of small cores and one large core. Then, we show how the number of small and large cores is determined, and how the system operates with more than one large core.

Receiving incoming requests. Only the small cores read incoming requests from the RX queues. They do so in batches, to amortize the cost of communicating with the NIC. Each small core repeats the following sequence of actions w.r.t. the RX queues. First, it reads a batch of B requests from its own RX queue. Then it reads a batch of B/n_s requests from the RX queue of the large core. In this way, all RX queues are drained at approximately the same rate. The reason a large core never reads incoming requests from its RX queue is that, if it were to receive a small request, this request could experience head-of-line blocking behind large requests.

We start by explaining how GET operations are handled.

Operation of the small cores. For each request, a small core starts processing the request by looking up the item associated with the requested key. If its size is below the threshold, the small core continues the GET operation and replies to the client with the requested item (by putting the corresponding reply packet(s) on its TX queue). Else, the small core puts the request in the software queue of the large core.

Operation of a large core. For each request in its software queue, a large core finds the corresponding item, and replies to the client by putting the reply packet(s) on its TX queue.

The operation of a PUT is mostly similar, except that the size is present in the request. Hence, there is no need to do a lookup to find the size. Depending on the size, the request is handled either immediately by the small core or passed on by the small core to the large core, and handled there.

How to find the threshold between large and small requests. Each small core maintains a histogram of the number of requests that correspond to item sizes in certain ranges.

Each range corresponds to a size *class*. This histogram is updated on the receipt of every request according to the size of the target item. Periodically, core 0 aggregates these histograms, finds the size class corresponding to the Nth percentile of item sizes, declares that class to be the threshold for the next epoch, and resets the histograms.

To be resilient to workload oscillations, core 0 smooths the values in the aggregated histogram (noted H) according to a moving average that uses the histogram obtained in the previous epoch (noted H_{curr}). For each entry i , core 0 computes $H_{curr}[i] = (1 - \alpha)H_{curr}[i] + \alpha H[i]$, and uses the new H_{curr} to determine the Nth percentile. α is a discount factor in the range $[0, 1]$, and determines the weight of the new measurements over previous ones. Because Minos targets high throughput workloads, many requests are sampled during an epoch. Hence, H is highly representative of the current workload, and is assigned a weight equal to 0.9 [65].

How to choose the number of small cores. Minos maintains a cost function that gives us for a request of a given size a certain processing cost. Minos can use various cost functions, but currently uses the number of network packets handled to serve the request as cost, either the number of packets in an incoming PUT request or the number of packets in an outgoing GET reply. Alternatives could be the number of bytes or a constant plus the number of bytes. The number of small cores is then set to the ceiling of the fraction of the total processing cost for small requests times the total number of cores. The remaining cores are used as large cores.

Operating with a number of large cores different from one. If, as a result of the above calculation, there is more than one large core, then Minos distributes the large requests over the large cores such that each large core handles a non-overlapping contiguous size range of requests, and such that the cumulative processing cost of requests assigned to each large core is the same. By doing so, not only does Minos balance the load on large cores, but it also shards large requests in a size-aware fashion. That is, the smallest among the large requests are assigned to the first large core, and larger requests are progressively assigned to other cores. A small core that receives a large request puts the request in the software queue of the large core that is handling the size of the requested item.

If all cores are deemed to be small cores, then one core is designated a standby large core. In other words, it handles small requests, but if a large request arrives, it is sent to this core, which then becomes a large core.

3.2 Discussion

Design rationale. The goal of Minos is to improve the Nth percentile. To that end, Minos identifies the smallest N percent of the requests, and isolates the processing of these requests from the processing of larger requests, such that no

head-of-line blocking occurs. Furthermore, Minos assigns a number of cores to small/large requests proportionally to the expected load generated by requests of that size, so as to balance the load across cores.

The use of randomization and of the hashed value of the key to decide the target RX queue for a request leads to reasonable load balance among the RX queues. A similar observation was made in the context of MICA [45]. Since the small cores handle the requests that arrive in their own RX queue, and an equal portion of the requests that arrive in the RX queues of the large cores, overall the load is balanced among the small cores. By using purely hardware dispatch for the small requests we eliminate any unnecessary overhead in their processing, such as, for instance, software dispatches. We achieve these results while never dropping large requests, since there is always at least one core available for handling large requests.

The only overheads compared to a purely hardware dispatch solution such as MICA are then: 1) software dispatch for the very small number of large requests, 2) synchronization on the RX queue and the software queue of the large cores, for which we found contention to be low, and 3) some minor loss in locality for the small requests that arrive in the RX queues of large cores.

Not sharding small requests. Minos could implement size-aware sharding for small requests. This would allow for isolating requests of different sizes at a finer granularity. Minos eschews this design choice because it targets SLOs expressed in terms of a single response time percentile. Hence, it is less important to further improve the performance of smaller requests than to achieve the highest throughput with low target tail latency. Sharding small requests across multiple class sizes, instead, may result into a less efficient design because small cores would spend much of their resources in dispatching requests that are served by other cores, and may be idle while waiting to receive dispatched requests. We have experimented with a design in which we shard small requests, and it proved to perform poorly. Assigning all small requests to the same set of cores allows Minos to perform software handoff only for the few large operations, and to achieve high throughput and low Nth percentile latency.

Target percentile setting. The latency benefits brought by Minos naturally depend on the setting of the target percentile and the item size distribution. For example, an item size distribution could be such that the 95th percentile is 10B, the 96th percentile is 500KB and the 99th percentile is 1MB. Then, optimizing for the 95th percentile would benefit the latency of the smallest 95% of operations more than what would happen for the smallest 99% of operations if Minos was set to optimize for the 99th percentile. However, if the target SLO of the application using the key value store is expressed in terms of the 99th percentile, it is less important to achieve a very good 95th percentile by separating 10B requests from the rest, and Minos should be configured to tar-

get the 99th percentile. In this setting, Minos would improve the 99th percentile latency as much as possible by segregating 1MB operations and larger ones from the rest.

In the current design, Minos takes the target percentile as input. The system administrator may determine such percentile with the aid of workload traces collected offline, which are typically available in production systems [3, 5, 59]. Automatically determining a suitable percentile that results in high latency gains and high throughput is an orthogonal research issue that we are currently investigating.

Trade-offs. In Minos small and large operations each have access only to a subset of the processing power available on the machine. This may lead some requests to experience a longer queueing time than what they would experience if the request could be served by any core. The impact of this additional delay on short requests is outweighed by the benefits that stem from avoiding head-of-line blocking. This design, however, penalizes large requests. The rationale underlying this trade-off is that Minos aims to reduce a target Nth percentile of the response time distribution by favoring the smallest N% of the operations. Larger requests that fall out of such percentile, then, are processed in a best effort fashion –and, importantly, never dropped.

Penalizing larger request is an inevitable price to pay to favor smaller ones, as shown by the theoretical and quantitative analysis of scheduling policies similar to size-aware sharding [1, 6, 18]. We assess the effects of this trade-off on performance in Section 6.1, and we discuss the differences between size-aware sharding in Minos and related scheduling techniques in Section 7.

Alternative designs. We now discuss alternative designs to address item size variability, and why we do not adopt them. 1) *Use a dedicated set of machines to serve large requests*, as suggested in [45]. This solution may lead to waste of resources because the workloads of large and small requests cannot be consolidated. It also requires migrating items across machines in case an item changes size, and adds one network hop to redirect large requests.

2) *Splitting large operations in smaller chunks*. This allows interleaving the processing of such chunks with small requests. This design may lead to lower resource efficiency with respect to the run-to-completion model adopted by Minos. First, it may lead to worse data locality, by accessing memory regions corresponding to different requests, and by interleaving request processing with networking operations. Second, it requires the implementation of nontrivial scheduling mechanisms, whose costs may be not negligible with μ second scale SLOs. Instead, the run-to-completion model enables high efficiency [57], and allows us to re-use state-of-the-art techniques proposed for such model [45]. In addition, it allows Minos to avoid head-of-line blocking by implementing a simple FIFO scheduling policy within each core.

4 Implementation

4.1 Network stack

Minos relies on the availability of a multi-queue NIC with support for redirecting, in hardware, a packet to a specific queue on the NIC (e.g., RSS [32] or Flow Director [33]). This feature is now commonplace in commodity NICs.

To reduce packet processing overhead, Minos uses the Intel DPDK library [23] to implement a user-level zero-copy network stack. All memory for the DPDK library is statically allocated and accessible by all cores. Packets are received directly in memory, thus enabling zero-copy packet processing. Furthermore, Minos uses DPDK-provided lockless software rings to dispatch large requests from small to large cores without any copies [39]. Small cores check for incoming requests by means of polling, to avoid costly interrupts [57]. Similarly, large cores use polling to check for incoming requests on their software queue. Requests are moved in batches to further limit overhead.

Clients and servers communicate using UDP, implemented on top of Ethernet and IP. Clients use the UDP header to specify the target RX queue for a given packet. Requests that span multiple frames (large PUT requests and large GET replies) are fragmented and defragmented at the UDP level.

Retransmission is handled by the client. Similar to previous work [45], Minos does not support exactly-once semantics and assumes idempotent operations. Exactly-once semantics can be achieved by means of request identifiers.

4.2 KV store and memory management

Data structures. Minos employs the KV data structures used in MICA [45]. Keys are split in *partitions*. Each partition is a hash table, each entry of which points to a bucket, equal in size to a cache line. Each bucket contains a number of slots, each of which contains a tag and a pointer to a key-value item. A first portion of the keyhash is used to determine the partition, a second portion to map a key to a bucket within a partition, and a third portion forms the tag [22, 45], which is used to reduce the number of random memory accesses when performing a key lookup. Overflow buckets are dynamically assigned to a bucket when it has reached its maximum capacity.

Memory management. The current prototype of Minos employs the memory manager of the DPDK library to handle allocation of memory regions for key-value entries. Minos can be extended to integrate more efficient memory allocators, such as the one based on segregated fits of MICA, or a dynamic one as in Facebook’s memcached deployment [55].

Concurrency control. Minos uses a concurrency control scheme that is similar to Concurrent Read Exclusive Write (CREW) [45]. Each core is the *master* of one partition, and

each key can be written only by the master core of corresponding partition. This serializes write operations on a key.

The concurrency control scheme in Minos differs slightly from CREW, as a result of the distinction between small and large cores. PUTs on keys whose master core is a small core proceed along the lines of CREW. PUTs on keys whose master core is a large core may be served by any core (either because the request is small, or because it is dispatched to a large core different from the one which receives the request). In addition, two concurrent PUT operations on the same key may be assigned to two different cores (a small and a large one, or two large ones), depending on the size of the corresponding values. Hence, all PUTs are guarded by a spinlock.

We argue (and we experimentally show) that the corresponding overhead of spinlocks is largely outweighed by the benefits of size-aware sharding, especially for the read-dominated workloads that are prevalent in production environments [3, 10, 55, 56]. First, in such workloads PUTs are rare. Second, PUTs on large cores proceed mostly without contention, because large cores serve non-overlapping size ranges, so requests for the same large item are sent to the same core. Third, PUTs on small cores mostly proceed without contention because of the CREW nature of the concurrency protocol for keys whose master is a small core.

GETs can be served by any core, and are processed by means of an optimistic concurrency control scheme [45]. Each bucket has a 64-bit epoch, which is incremented when starting and ending a write on a key stored in that bucket. Upon reading, a core looks at the epoch. If it is odd, then there is an ongoing write on a key of the bucket, and the read is stalled until the epoch becomes even. If (or when) the epoch is even, the core saves the current epoch value and performs the read. After the read, the core re-reads the epoch of the bucket. If the value is the same as when the read started, the read is successful. Else, a conflicting write might have taken place, and the read is restarted. Because all memory is pre-allocated, a writer thread can safely modify/erase a KV entry that is concurrently accessed by a read.

5 Experimental Platform

5.1 Hardware

Our platform is composed of 8 identical machines equipped with an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz with 8 physical cores and 64 GB of main memory. The machines run Ubuntu 16.04.2 with a 4.4.0-72-generic kernel. One machine acts as server and the other 7 run the client processes. We disable hyperthreading and power-saving modes on all the machines. All the machines are equipped with a 40Gbit Mellanox MT27520 NIC (ConnectX-3 Pro), are located in the same physical rack, and are connected via a top-of-rack switch. The network stack relies on the Intel DPDK library (version 17.02.1), to which we allocate 50 1GB huge pages.

Our NIC supports only RSS to implement hardware packet-to-RX queue redirection [50]. RSS determines the RX queue for an incoming packet by performing the hash of the quintuplet composed of source and destination IP, source and destination port and the transport layer protocol. To allow the clients and the server to send packets to specific RX queues, we ran a set of preliminary experiments to determine to which port to send a packet so that it is received by a specific RX queue. More flexible hardware packet redirection methods can be used on NICs that support them. For example Minos can use Flow Director [33, 49] to set the target RX queue as UDP destination port of a packet.

5.2 Systems used in comparison

We compare Minos with three systems that implement state-of-the-art designs of KV store, and that are based on the queueing models that we have described in Section 2.

- **Hardware Keyhash-based sharding (HKh).** This system implements early binding of requests to cores, as done in MICA [45]. Requests are redirected in hardware to the target core, according to the CREW policy. This policy performs the best on skewed read-dominated workloads [45], such as our default workload.
- **Software hand-off (SHO).** This system implements the late binding of requests to cores, as in RAMCloud [57]. SHO uses disjoint sets of handoff and worker cores. Each handoff core has a software queue, in which it deposits the requests taken from its RX queue. Worker cores pull one request at a time from the handoff queues, process the corresponding KV request, and reply to the client. The best number of handoff cores depends on whether the workload is CPU or network bound. We have experimented with 1,2 and 3 handoff cores. We report experimental results corresponding to the best configuration for each workload.
- **HKh + work stealing (HKh+WS).** This system implements request stealing on top of HKh, as in ZygOS [58]. Each core has a software queue where it places the requests taken from its own RX queue. An idle core can steal requests from the software queues of other cores, and from their RX queues, if no request is found in any software queue.

All designs are implemented in the same codebase. This allows us to focus on the effects of item size heterogeneity on performance, and to factor out implementation differences (e.g., in the KV store data structure and concurrency control scheme) and limitations (e.g., leak of support for multi-frame packets and additional overheads to support richer APIs) of the existing systems that implement the designs we consider.

The internal parameters of Minos are set as follows. Workload statistics are collected by core 0 every second. The byte range corresponding to the i -th size class is $[2^{(i-1)}, 2^i]$, and i ranges from 1 to 10. The size of a batch of requests read from a RX queue is 32, and the same batch size is used

% large reqs (p_L)	Max size (s_L)	% data for large reqs
0.125	250 KB	25
	500 KB	40
	1000 KB	60
0.0625	500 KB	25
0.25		60
0.5		75
0.75		80

Table 1: Item size variability profiles.

for other systems as well.

5.3 Workloads

We use workloads characterized by different degrees of item size variability and GET:PUT ratios.

Item size variability. We use, as a starting point, the characterization of the ETC workload at Facebook [3]. Specifically, we consider a trimodal item size distribution, according to which an item can be tiny (1-13 bytes), small (14-1400 bytes) or large (1500-maximum size). The size of an item within a class is drawn uniformly at random. To generate workloads with different degrees of item size variability, we vary both the percentage of large requests, (noted p_L), and the size of items corresponding to large requests, by changing the maximum size of large items (noted s_L). We let s_L range from 250KB to 1MB. These values are consistent with the production workloads we discussed in Section 2.1. Similarly to what is seen for the ETC workload, we set $p_L < 1\%$, so that the 99th percentile of the requests service times corresponds to small and tiny items only. Specifically, we vary p_L from 0.0625 to 0.75. Table 1 reports the combinations of p_L and s_L we consider. It also reports the corresponding percentage of bytes that are exchanged because of large requests.

Key popularity. We consider a skewed workload that follows a zipfian distribution with parameter 0.99. This represents the default value in YCSB [14], is widely used in the evaluation of several KV stores [45, 35], and is representative of the strong skew of many production workloads [3].

We use the zipfian distribution on the sets of tiny and small items, because they are many and they exhibit small variability in size. Large items, instead, are much fewer and exhibit much higher variability, and are therefore chosen uniformly at random. This avoids pathological cases in which the most accessed large item is the biggest or the smallest item, thereby skewing the results.

We consider a dataset of 16M key-value pairs, out of which 10K are large elements. Of the remaining key-value pairs, 40% correspond to tiny items, and 60% to small ones. This setting is consistent with the item size distribution and the low access probability of individual large keys that characterize the ETC workload. Each large item has, in fact, a probability $p_L/100 \cdot 10K/16M$ of being accessed. For simplicity, we keep the size of the keys constant to 8 bytes.

Write intensity. We consider a read-dominated and a write-intensive workloads, corresponding, respectively, to a 95:5 and 50:50 GET:PUT ratio. These values are used as default values in YCSB and KV store evaluations [45, 35] (the ETC workload has a 97:3 GET:PUT ratio).

Default workload. We set a default value for each parameter, and generate different workloads by changing the value of one parameter at a time while keeping the other ones to their default values. The default workload is skewed with a 95:5 GET:PUT ratio, a percentage of large requests equal to 0.125 and a maximum large item size of 500 KB.

5.4 Benchmarking methodology

Load generation. We spawn 8 threads per client machine, each pinned to a separate physical core and to an RX queue. Client threads simulate an open system by generating requests at a given rate, which varies depending on the target throughput. The time between two consecutive requests of a thread is exponentially distributed.

Measurements. Each request is timestamped with the send time at the client, which is piggybacked by the server on the reply message. Client threads constantly check their own RX queues for replies, and compute the end-to-end latency of a request using the timestamp in the reply message.

A client thread can have multiple requests in flight, so for simplicity packet retransmission is not enabled. For this reason, we only report performance values corresponding to scenarios in which the packet loss rate is equal to 0.

Each workload runs for 60 seconds. The first and last 10 seconds are not included in the reported results.

Performance metrics. We focus on maximum achievable throughput (number of successful operations completed per second) and 99th percentile of end-to-end latencies, since large requests correspond to less than 1% of the total. We also measure the utilization of the server NIC to evaluate whether Minos is able to fully use the available bandwidth.

We consider SLOs in the form “The 99th percentile of latencies must be within $X \mu\text{sec}$ ”. We use $X = 50$ and $X = 100$ to evaluate the performance gains of Minos as a function of the strictness of the SLO. These values correspond to 10 and 20 times the mean service time for a GET request in our default workload (similarly to previous work [58]).

6 Evaluation

6.1 Default workload

Throughput vs. 99th percentile latency. Figure 3 shows the 99th percentile latency (99p) as a function of the throughput with the default workload. Minos achieves the highest peak throughput (6.2 Mops) and the lowest latency ($\leq 50 \mu\text{sec}$ up to 90% of peak throughput).

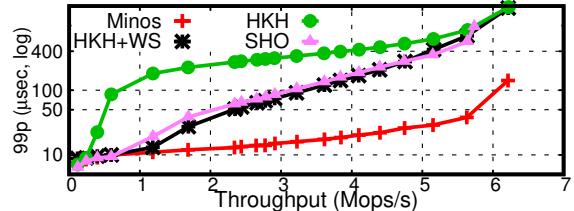


Figure 3: Throughput vs. 99th percentile latency (y axis in log scale) with the default workload. Minos matches the throughput of the purely hardware-based design and achieves the lowest latency.

Minos achieves the same peak throughput as HKH and HKH+WS, reflecting the fact that all three systems rely mostly or entirely on hardware handoff for request distribution (at very high load, stealing in HKS+WS rarely happens). SHO achieves 10% less peak throughput, because it is bottlenecked by the software handoff. In terms of 99th percentile, Minos does better than HKH at any load, with improvements reaching one order of magnitude as soon as the load exceeds 1 Mops. HKH+WS and SHO start out with similar 99th percentile latencies as Minos under loads below 1 Mops, but under high load their 99th percentile latencies rapidly deteriorate to reach values similar to HKH. For an SLO on the 99th percentile latency of 50 μsec Minos can perform 5.6 Mops, 2.4 times the throughput of its best competitor (HKH+WS). For an SLO of 100 μsec , Minos still achieves 1.75 times the throughput of its best competitor.

Minos achieves the best performance by overcoming the limitations of existing designs when dealing with variable-size items, and that we have discussed in Section 2.2. Interestingly, the performance curves of the competitor systems we consider follow the ones depicted in Figure 2, which portrays the behavior of the same systems in idealized conditions (i.e., without dispatching and synchronization costs). This indicates that the reason for the worse 99th percentile tail latency exhibited by such systems is primarily due to the shortcomings of their designs in presence of item size variability, and not to low level implementation details.

Latency of large requests. Minos leverages the insight that the latency of the largest N% of the requests should not impact the (100-N)th percentile. Minos restricts the N% largest requests to a subset of the cores (N=1 in our setting), which may result in increased latencies for such requests. We now evaluate the performance penalty incurred by large requests in Minos as a consequence of size-aware sharding. Figure 4 reports the 99th percentile latency of large requests in Minos and HKH+WS (the best alternative).

Inevitably, Minos imposes some penalty on the performance of large requests, reaching up to a factor of 2 for the 99th percentile latency of large requests before the system goes into saturation. We argue that moderately penalizing

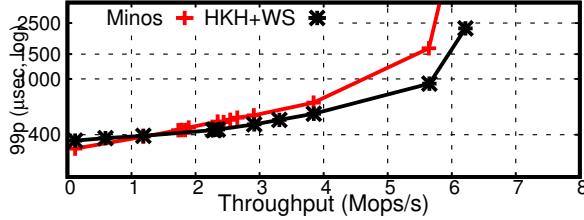


Figure 4: Throughput vs. 99th percentile latency of large requests with the default workload (y axis in log scale). Minos trades its large benefits in terms of the overall 99th percentile for a moderate penalty on the large requests, which represent a small fraction of the workload.

large requests is a reasonable price to pay for the order-of-magnitude improvement for the target (100-N)th percentile.

Minos can improve the latency of large requests by allocating more cores to them. Minos currently determines the number of small cores by taking the ceiling of the total number of cores times the fraction of load generated by small requests. For this workload, it allocates only one core to the large requests. This represents an over-allocation to small requests to completely isolate them from large requests, and hence an under-allocation for large requests. An alternative strategy is to allocate one more core to large requests, and let large cores steal from the RX queues of small ones to fully use any extra capacity. To avoid re-introducing head-of-line blocking, stealing can be done one request at a time, so that there is never a small request queued behind a large request. We are currently experimenting with this alternative design, which would improve performance for large requests, while only introducing a small degradation for small requests.

6.2 Write-intensive workload

We now investigate the effect of write intensity on Minos. Figure 5 reports the 99th percentile of response times with all four systems and a 50:50 GET:PUT workload.

Minos continues to deliver a 99th percentile latency one order of magnitude lower than alternative approaches, up to the saturation point at 6.3 Mops, but overall achieves a lower (by 10%) throughput than HKH and HKH+WS. Throughput values are in general higher than with the 95:5 workload, because replying to a PUT requires less network bandwidth, since the response message does not contain any item value payload. This behavior is consistent with that observed by previous work [45]. SHO is the only exception, as handoff cores represent the bottleneck.

Minos achieves a lower throughput with respect to HKH and HKH+WS because of the overhead stemming from profiling the workload and periodically aggregating them on core 0 to compute the 99th percentile of the item sizes. We are currently investigating techniques to reduce such over-

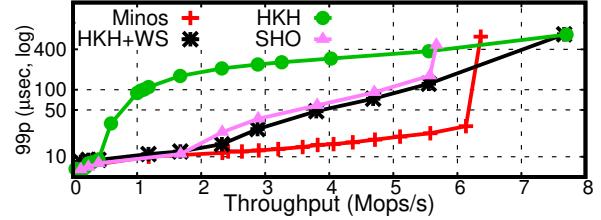


Figure 5: Throughput vs. 99th percentile latency for Minos vs. existing designs with the 50:50 GET:PUT workload (y axis in log scale).

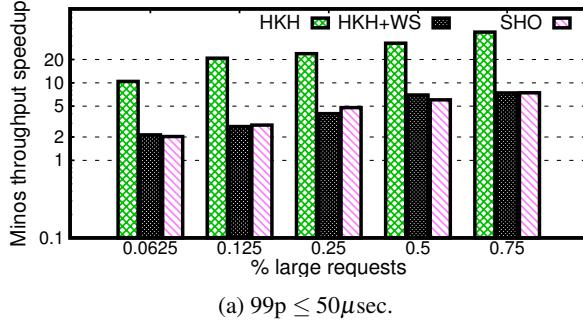
head, e.g., sampling only a subset of the requests. Alternatively, the threshold between large and small requests can be set statically if it does not vary over time and traces of the target workload are available for off-line analysis (as typical in production workloads [3, 55, 59]). With this variant, Minos is able to match the throughput of HKH and HKH+WS.

6.3 Sensitivity to item size distribution

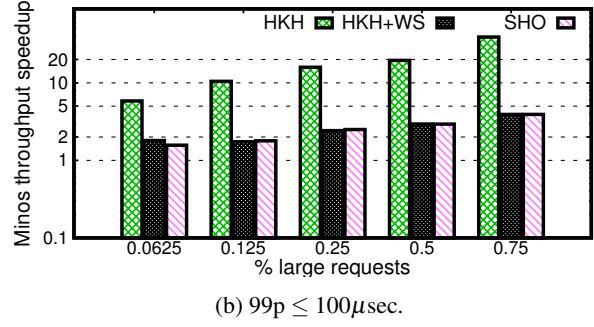
We vary the percentage of large requests in the workload (p_L) and the maximum size of large requests (s_L). When changing the value of one, the other parameter keeps the default value. We then measure the maximum throughput achievable under the two SLOs we consider.

Figure 6 and Figure 7 report the increase in throughput achieved by Minos compared to the other designs (y axis in log scale). Figure 6 shows the results of the experiments in which we change p_L . Figure 7 refers to changing s_L . The graph on the left uses an SLO of 50 μ sec, the one on the right 100 μ sec. When varying p_L , the maximum throughput achieved by Minos within the 50 μ sec (100 μ sec) SLO ranges from 6.2 to 1.7 Mops (6.9 to 2.3 Mops), corresponding to $p_L = 0.0625$ and $p_L = 0.75$. When varying s_L , the maximum throughput achieved by Minos within the 50 μ sec (100 μ sec) ranges from 6.2 to 4.7 Mops (6.9 to 4.7 Mops), corresponding to $s_L = 250KB$ and $s_L = 1000KB$.

Minos outperforms existing designs, achieving consistently higher throughput for a given workload and a given SLO. The throughput speedup grows with p_L and s_L , because the increased presence of large(r) requests negatively affects the latency of small requests, and hence the 99th percentile, in alternative designs. As expected, the throughput gains are higher with the stricter SLO: the looser is the performance target, the smaller is the impact of Minos' design. For the stricter SLO, Minos achieves a speedup of up to 7.4 w.r.t HKH+WS (corresponding to the $p_L = 0.75$ case), i.e., the second best design. For the looser SLO, the speedup ranges from 1.34 ($s_L = 250KB$) to 3.9 ($p_L = 0.75$).

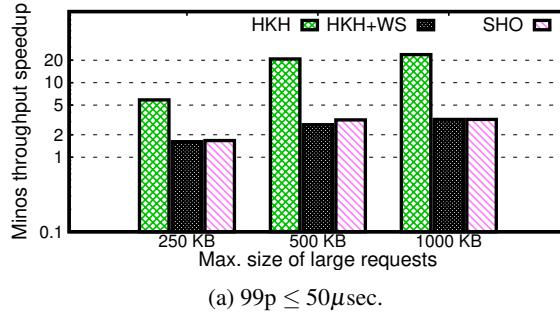


(a) $99p \leq 50\mu\text{sec}$.

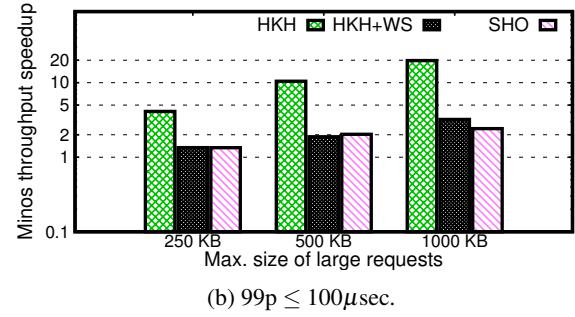


(b) $99p \leq 100\mu\text{sec}$.

Figure 6: Maximum throughput achievable for a given 99th percentile latency SLO with different percentages of large requests (y axis in log scale). Each bar represents the speedup of Minos over an alternative design (higher is better).



(a) $99p \leq 50\mu\text{sec}$.



(b) $99p \leq 100\mu\text{sec}$.

Figure 7: Maximum throughput achievable for a given 99th percentile latency SLO with different maximum sizes of large requests (y axis in log scale). Each bar represents the speedup of Minos over an alternative design (higher is better).

6.4 Higher network bandwidth

With the default workload, the NIC is 93% utilized. With higher percentages of large requests, the system becomes even more network-bound. In this section we investigate whether Minos can take advantage of larger network bandwidths. Because we cannot provision our machines with more bandwidth, we relieve the NIC bottleneck by sampling the number of replies that the server sends back to clients. That is, the server processes requests as before, up to the time at which it would otherwise send the reply to the client. Then, instead, it only sends replies to a percentage S of the total requests, and drops the remaining ones. We vary S from 100 to 25, and we measure the achieved performance (throughput and 99th percentile latency), as well as the utilization of the NIC. We choose the read-intensive workload with $p_L = 0.75$, as it quickly saturates the NIC with $S = 100$.

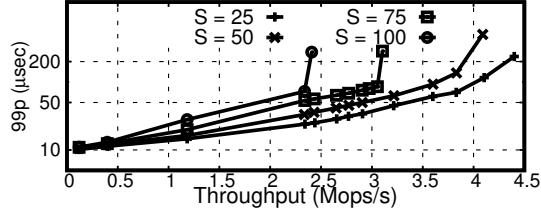
Figure 8 reports the results of the experiment. The left plot shows the throughput vs. 99th percentile latency (y axis in log scale). The right one shows the utilization of the NIC as a function of the throughput. As S decreases, Minos can sustain higher loads, because the bottleneck is increasingly shifted towards the CPU. Minos is able to fully utilize the available resources, by reaching throughput values that saturate (or almost saturate) the NIC ($S = 100, 75, 50$) except when the bottleneck is query processing ($S = 25$).

6.5 Load balancing

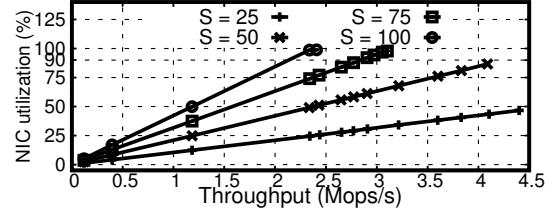
We now evaluate the ability of Minos to distribute the load evenly across cores according to the provided cost function. To this end, we measure the load sustained by each core with $p_l = 0.0625, 0.25, 0.75$, corresponding to low, medium and high load posed by large requests. Figure 9a reports the percentage of requests performed, and Figure 9b reports the percentage of packets processed by each core (y axis in log scale). Two conclusions can be drawn. First, all cores process roughly the same number of packets, and hence roughly perform the same amount of work. Small cores obviously process more requests per second, as these requests involve less work. Large cores process different requests per second among each other, as a consequence of the size-aware sharding that Minos implements also within large requests. Second, Minos varies the number of small and large cores as a function of the workload, such that enough resources are allocated to small and large requests.

6.6 Dynamic workload

We finally demonstrate the capability of Minos to adapt to changing workloads. To this end, we run a workload in which the percentage of large operations p_L varies every 20 seconds. It first grows gradually from 0.125 to 0.75, and

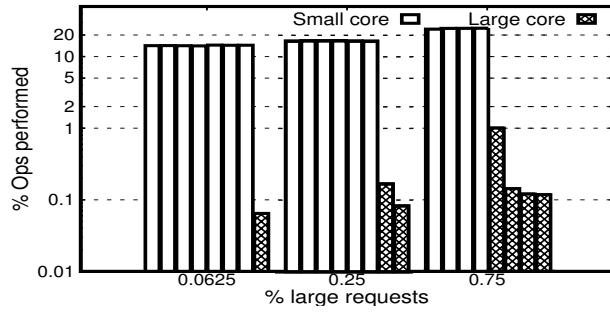


(a) Throughput vs. 99th percentile latency

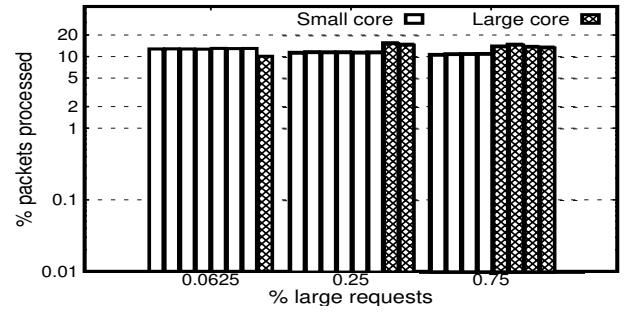


(b) Throughput vs. NIC utilization.

Figure 8: Scalability of Minos with more network bandwidth ($p_L = 0.75$). S is the sampling percentage used to simulate more network bandwidth. Minos processes and replies to $S\%$ of the requests. The remainder is processed, but the reply is dropped. Minos scales with more bandwidth (a) and saturates the NIC (b), except when query processing is the bottleneck ((b), $S = 25$).



(a) Operations per second.



(b) Packets per second.

Figure 9: Breakdown of the load per core in Minos (y axis in log scale). Large cores process fewer requests per second than small cores (a), but the number of packets processed per second is uniformly distributed across cores (b).

then shrinks back to 0.125. We keep the request arrival rate fixed at 2.25 Mops, corresponding to high load for $p_L = 0.75$. Figure 10(top) compares the performance achieved by Minos and HKH+WS, i.e., the second best design. Each point represents the 99th percentile latency as measured over a 1 second window (y axis in log scale). Figure 10(bottom)

shows how many cores Minos assigns to large requests over time. Minos achieves latencies up to 20 times lower than HKH+WS (70 μ sec vs ≈ 1.5 msec with $p_L = 0.75$). Minos achieves this result by programmatically allocating cores to small and large requests proportionally to their corresponding loads.

7 Related Work

To the best of our knowledge, Minos is the first KV store to introduce the concept of size-aware sharding to address the challenges of delivering μ sec-scale tail latency in presence of item size variability. We now discuss related systems.

In-memory KV stores. A plethora of in-memory KV stores have been proposed in the last years. These systems propose different designs based on new data-structures (CPHash [52], Masstree [48], MemC3 [22]) and lightweight network stacks (Chronos [37], MICA [43, 45], RamCloud [57], RockSteady [40]), or on the use of RDMA (Pilaf [53], Herd [35], FaRM [21], RFF [60], FaSST [36], TailWind [61]), FPGAs (KV-Direct [41]), GPUs (MegakV [64], MemcacheGPU [30]), HTMs (DrTM [11, 63]), or other specialized hardware ([38, 9]).

None of these systems addresses the problem of achieving low tail latency in presence of item size variability, which

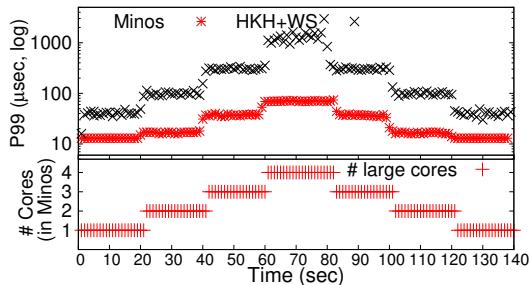


Figure 10: Evolution over time of the 99th percentile latency of Minos and HKH+WS with a dynamic workload (top, with y axis in log scale) and evolution over time of number of large cores in Minos (bottom). Every 20 seconds the percentage of large requests changes, first growing from 0.125 to 0.75 and then shrinking back. Minos adapts to changing workload conditions and delivers up to 20X lower 99th percentile latencies.

is the primary focus of Minos. In addition, Minos only assumes the availability of commodity hardware. Investigating the synergies between the design of Minos and specialized hardware is an interesting avenue for future work.

Size-aware data-stores. We are aware of a few data stores that take into account the size of items or requests to improve performance. Rein [59] supports multi-key get requests and processes them taking into account the number of keys involved in a request. Rein relies on the assumption that there is only a weak correlation between the size of an item and the service time of a request for that item. Minos, instead, targets workloads with high item size variability, for which the service time of a request strongly depends on the size of the corresponding item (see Figure 1).

AdaptSize [8] is a caching system that reduces the probability of caching large objects, so as to increase the hit rate of smaller, more frequently accessed ones. AdaptSize targets a problem that is orthogonal to Minos, which assumes the presence in memory of both small and large items.

The systems in [12, 29, 65] target static content and leverage a central component (the Linux kernel on a single-core architecture [29] or a scheduler in a distributed system [12, 65]) to implement request scheduling. By contrast, Minos deals with mixed read/write workloads and targets multi-core architectures with multi-queue NICs.

Operating systems. IX [7] and ZygOS [58] use lightweight network stacks to meet μ sec-scale SLOs. ZygOS uses work stealing to avoid core idleness and reduce head-of-line blocking. As we show by means of simulation (§ 2.2) and experimental data (§ 6), this approach cannot fully avoid head-of-line blocking as done by Minos, because work stealing *i*) is agnostic of the CPU time corresponding to serving a request; and *ii*) is only triggered by idle cores, whose presence becomes less likely as the load increases.

Scheduling systems. There is a vast literature on scheduling requests with heterogeneous service demands. Several approaches have been applied in the context of flow scheduling [1, 4, 24, 25, 31, 54], single-server request scheduling [26, 42, 47] and cluster request scheduling [18, 20, 19]. Proposed approaches include workload partitioning [16, 18, 29], preempting [6, 19] or migrating requests [26, 27], and stealing [20, 42]. One common result of these approaches is that favoring small requests inevitably comes at the expense of the performance of the largest requests.

Size-aware sharding draws from these techniques, and makes the same trade-off between the latencies of small and large requests. However, Minos substantially deviates from these systems, to apply size-aware sharding in an in-memory key value store efficiently. In particular,

- Minos does not rely on any *a priori* information on the size of a request. This contrasts with existing systems that rely on request runtime estimates, such as Hawk [20].
- Minos avoids head-of-line blocking by processing short

and large requests on disjoint sets of cores. This contrasts with systems like 2DFQ [47], where all resources are shared between short and large requests, and hence a burst of large requests may delay shorter ones.

- The design of Minos is tailored for the in-memory key value store domain. *i*) Minos integrates size-aware sharding with the run-to-completion model, which avoids interrupts and context switches, enhances locality, and reduces cache pollution [45, 57]. This is unlike the aforementioned systems, which target the classic multi-threaded approach and migrates requests across cores [26, 42, 44], or across servers [27]. *ii*) Minos leverages the hardware request-to-core dispatching enabled by multi-queue NICs to reduce the amount of software hand-offs. This allows Minos to achieve throughput values equal or close to those achievable by pure hardware request-to-core dispatching. *iii*) Minos co-designs size-aware-sharding and the concurrency control scheme to both achieve load balance and avoid head-of-line blocking.

These characteristics allow Minos to target μ scale tail latencies, whereas the aforesaid scheduling approaches reportedly support SLOs in the order of the milliseconds or higher.

8 Conclusion

This paper presents Minos, an in-memory key-value store designed to deliver μ sec-scale tail latency with workloads characterized by highly variable item sizes, as frequent in production workloads. Minos implements size-aware sharding, a new technique that assigns small and large requests to disjoint set of cores. This ensures small requests never wait due to the collocation with a long request. Minos identifies at runtime the size threshold between long and short requests, and the amount of cores to allocate to them. We compare Minos to three state-of-the-art designs and we show that, compared to its closest competitor, Minos achieves a 99th percentile latency that is up to 20 times lower. Put differently, for a given value for the 99th percentile latency equal to 10 times the mean service time, Minos achieves a throughput that is up to 7.4 times higher.

Acknowledgements

We thank the anonymous reviewers, Richard L. Sites and our shepherd Amar Phanishayee for their feedback. We also thank Hyeontaek Lim for his help in setting up the DPDK library. This research has been supported by an EcoCloud post-doctoral research fellowship.

References

- [1] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th*

- USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI’12, USENIX Association, pp. 19–19.
- [2] ARLITT, M. F., AND WILLIAMSON, C. L. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw.* 5, 5 (Oct. 1997), 631–645.
- [3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proc. of SIGMETRICS* (2012).
- [4] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI’15, USENIX Association, pp. 455–468.
- [5] BALMAU, O., DIDONA, D., GUERRAOUI, R., ZWAENEPOEL, W., YUAN, H., ARORA, A., GUPTA, K., AND KONKA, P. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 363–375.
- [6] BANSAL, N., AND HARCHOL-BALTER, M. Analysis of srpt scheduling: Investigating unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2001), SIGMETRICS ’01, ACM, pp. 279–290.
- [7] BELAY, A., PREKAS, G., PRIMORAC, M., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.* 34, 4 (Dec. 2016), 11:1–11:39.
- [8] BERGER, D. S., SITARAMAN, R. K., AND HARCHOL-BALTER, M. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 483–498.
- [9] BLOTT, M., LIU, L., KARRAS, K., AND VISSERS, K. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems* (Berkeley, CA, USA, 2015), HotStorage’15, USENIX Association, pp. 8–8.
- [10] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. Tao: Facebook’s distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC’13, USENIX Association, pp. 49–60.
- [11] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys ’16, ACM, pp. 26:1–26:17.
- [12] CIARDO, G., RISKA, A., AND SMIRNI, E. Equiload: a load balancing policy for clustered web servers. performance evaluation. In *Performance Evaluation* 46 (2001), 46–101.
- [13] CIDON, A., RUSHTON, D., RUMBLE, S. M., AND STUTSMAN, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 321–334.
- [14] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC ’10, ACM, pp. 143–154.
- [15] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Trans. Netw.* 5, 6 (Dec. 1997), 835–846.
- [16] CROVELLA, M. E., HARCHOL-BALTER, M., AND MURTA, C. D. Task assignment in a distributed system (extended abstract): Improving performance by unbalancing load. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1998), SIGMETRICS ’98/PERFORMANCE ’98, ACM, pp. 268–269.
- [17] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80.
- [18] DELGADO, P., DIDONA, D., DINU, F., AND ZWAENEPOEL, W. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC ’16, ACM, pp. 497–509.
- [19] DELGADO, P., DIDONA, D., DINU, F., AND ZWAENEPOEL, W. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the Ninth ACM Symposium on Cloud Computing* (2018), SoCC ’18.
- [20] DELGADO, P., DIDONA, D., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC ’15, USENIX Association, pp. 499–510.
- [21] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI’14, USENIX Association, pp. 401–414.
- [22] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi’13, USENIX Association, pp. 371–384.
- [23] FOUNDATION, T. L. Data plane development kit. <https://dpdk.org>, 2017.
- [24] GROSVENOR, M. P., SCHWARZKOPF, M., GOG, I., WATSON, R. N. M., MOORE, A. W., HAND, S., AND CROWCROFT, J. Queues don’t matter when you can jump them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI’15, USENIX Association, pp. 1–14.
- [25] GUO, L., AND MATTIA, I. The war between mice and elephants. In *Proceedings of the Ninth International Conference on Network Protocols* (Washington, DC, USA, 2001), ICNP ’01, IEEE Computer Society, pp. 180–.
- [26] HAQUE, M. E., HE, Y., ELNIKETY, S., NGUYEN, T. D., BIANCHINI, R., AND MCKINLEY, K. S. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 ’17, ACM, pp. 625–638.
- [27] HARCHOL-BALTER, M. Task assignment with unknown duration. *J. ACM* 49, 2 (Mar. 2002), 260–288.
- [28] HARCHOL-BALTER, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. Cambridge University Press, New York, NY, USA, 2013.
- [29] HARCHOL-BALTER, M., SCHROEDER, B., BANSAL, N., AND AGRAWAL, M. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.* 21, 2 (May 2003), 207–233.
- [30] HETHERINGTON, T. H., O’CONNOR, M., AND AAMODT, T. M. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC ’15, ACM, pp. 43–57.

- [31] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 127–138.
- [32] HUDEK, T. Introduction to receive side scaling. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>.
- [33] INTEL. Intel 82599 10 gigabit ethernet controller: Datasheet. <https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html>, 2014.
- [34] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 121–136.
- [35] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.
- [36] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 185–201.
- [37] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 9:1–9:14.
- [38] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81.
- [39] KIT, D. P. Ring library. http://dpdk.org/doc/guides/prog_guide/ring.lib.html, 2017.
- [40] KULKARNI, C., KESAVAN, A., ZHANG, T., RICCI, R., AND STUTSMAN, R. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 390–405.
- [41] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.
- [42] LI, J., AGRAWAL, K., ELNIKETY, S., HE, Y., LEE, I.-T. A., LU, C., AND MCKINLEY, K. S. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2016), PPoPP '16, ACM, pp. 14:1–14:13.
- [43] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 476–488.
- [44] LI, X., SETHI, R., KAMINSKY, M., ANDERSEN, D. G., AND FREEDMAN, M. J. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016).
- [45] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.
- [46] LIM, K., MEISNER, D., SAIDI, A. G., RANGANATHAN, P., AND WENISCH, T. F. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 36–47.
- [47] MACE, J., BODIK, P., MUSUVATHI, M., FONSECA, R., AND VARADARAJAN, K. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 144–159.
- [48] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 183–196.
- [49] MELLANOX. Mellanox connectx-3 product brief. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN.Card.pdf, 2013.
- [50] MELLANOX. Mellanox dpdk release notes (v 16.11.1.5). http://www.mellanox.com/related-docs/prod_software/, 2017.
- [51] MEMCACHED. memcached. <http://www.memcached.org>.
- [52] METREVELI, Z., ZELOVICH, N., AND KAASHOEK, M. F. Cphash: A cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 319–320.
- [53] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 103–114.
- [54] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 221–235.
- [55] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., McELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proc. of NSDI* (2013).
- [56] NOGHABI, S. A., SUBRAMANIAN, S., NARAYANAN, P., NARAYANAN, S., HILLA, G., ZADEH, M., LI, T., GUPTA, I., AND CAMPBELL, R. H. Ambry: LinkedIn's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 253–265.
- [57] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The ramcloud storage system. *ACM Trans. Comput. Syst.* 33, 3 (Aug. 2015), 7:1–7:55.
- [58] PREKAS, G., KOGIAS, M., AND BUGNION, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 325–341.
- [59] REDA, W., CANINI, M., SURESH, L., KOSTIĆ, D., AND BRAITHWAITE, S. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 95–110.

- [60] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 1–15.
- [61] TALEB, Y., STUTSMAN, R., ANTONIU, G., AND CORTES, T. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).
- [62] WANG, F., XIN, Q., HONG, B., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MCLARTY, T. T. File system workload analysis for large scientific computing applications. In *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)* (Apr. 2004), p. 139152.
- [63] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.
- [64] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.* 8, 11 (July 2015), 1226–1237.
- [65] ZHANG, Q., RISKA, A., SUN, W., SMIRNI, E., AND CIARDO, G. Workload-aware load balancing for clustered web servers. *IEEE Trans. Parallel Distrib. Syst.* 16, 3 (Mar. 2005), 219–233.

Monoxide: Scale Out Blockchain with Asynchronous Consensus Zones

Jiapeng Wang

ICT/CAS & Sinovation AI Institute

Hao Wang

Ohio State University

Abstract

Cryptocurrencies have provided a promising infrastructure for pseudonymous online payments. However, low throughput has significantly hindered the scalability and usability of cryptocurrency systems for increasing numbers of users and transactions. Another obstacle to achieving scalability is the requirement for every node to duplicate the communication, storage, and state representation of the entire network.

In this paper, we introduce the Asynchronous Consensus Zones, which scales blockchain system linearly without compromising decentralization or security. We achieve this by running multiple independent and parallel instances of single-chain consensus systems termed as *zones*. The consensus happens independently within each zone with minimized communication, which partitions the workload of the entire network and ensures a moderate burden for each individual node as the network grows. We propose *eventual atomicity* to ensure transaction atomicity across zones, which achieves the efficient completion of transactions without the overhead of a two-phase commit protocol. Additionally, we propose *Chu-ko-nu mining* to ensure the effective mining power in each zone to be at the same level of the entire network, making an attack on any individual zone as hard as that on the full network. Our experimental results show the effectiveness of our work: on a testbed including 1,200 virtual machines worldwide to support 48,000 nodes, our system delivers 1,000× throughput and 2,000× capacity over the Bitcoin and Ethereum networks.

1 Introduction

Since the peer-to-peer electronic cash system [37] was published in 2008, decentralized consensus systems continuously enlarged its community and exerted bigger impacts on our society. However, low transaction confirming throughput measured as transaction-per-second

(TPS) has significantly hindered the usability of such systems with increased amounts of users and transactions. Besides network latency, the root cause of the throughput issue is the sequential nature of block creation. In a blockchain, blocks are created sequentially with sufficient propagation time in between, which yields a fixed low TPS¹ regardless of how many full nodes and miners participate in the network.

Additionally, a consensus system can not scale out when every full node needs to duplicate the communication, storage, and state representation of the entire network, which are cases of Bitcoin and Ethereum. Even if a high throughput is achieved, workloads requiring fast communication, adequate storage and sufficient computing power will soon set a high barrier for full nodes to participate in, which in turn dramatically hinders decentralization in practice. Therefore, a scalable blockchain system needs to consider the scalability of its consensus protocol with the resource usage of communication, storage, computation and memory for state representation while preserving decentralization and security. Previously, the Ethereum community has investigated sharding in blockchain systems [39] (see Section 8 for more discussions). Motivation of the proposed method echoes many considerations discussed in this article regarding duplicated workload, low entry barrier, importance of efficient cross-shard transaction handling, and the security issue of diluted mining power.

A scalable Blockchain system is desired so that future applications at the Internet scale can be supported. VisaNet payment and clearance [46] takes roughly 4k TPS on average. Alipay mobile payment exceeded 256k TPS at peak traffic [3] in 2017. Rapid growth of DApps on blockchain [44] also exhibits huge demand for scalable Blockchain systems with high throughput and large capacity to support games and decentralized exchanges.

¹Roughly 7 TPS for Bitcoin with 1MB block and 10-minutes intervals; and 15 TPS for Ethereum with roughly 32KB block and 15-seconds intervals.

Those demands motivate our work.

Asynchronous Consensus Zones is the main idea in this paper, which aims to design a scalable blockchain system without weakening decentralization or security. We scale out blockchain systems by partitioning and handling workloads in multiple independent and parallel instances, or *Consensus Zones*. The state of the entire network are partitioned by zones, and each zone is responsible for its own piece. The core data structures, such as blocks and transactions, are zone-specific, and are replicated and stored only within their own zones. Mining competition, chain growth, and transaction confirmation are carried out separately and asynchronously in each zone.

Consensus zones exhibit natural linear scalability for capacity by having the amount of storage, computation power, and state-representing memory proportional to the total number of zones. There are two major challenges to design such a blockchain system: (1) high throughput should be ensured in a scalable fashion when handling cross-zone transactions; (2) security should be reinforced as honest mining power diluted due to independent growths of chains in individual zones.

Cross-Zone Atomicity is crucial to the correctness and robustness of the blockchain system. In consensus zones, a transaction might involve multiple parties in different zones. It is challenging since state-updating of those parties occurs independently in different zones. Efficient handling of such cases is the key to the throughput scalability and the performance of the entire system. We propose *Eventual Atomicity* to ensure transaction atomicity across zones. With this technique, all operations will complete and will eventually achieve the correct end-state instead of serializing transactions like the two-phase commit protocol [27] does. Eventual atomicity allows interleaving of transactions in an asynchronous and lock-free manner to keep zones concurrent and fully utilized. Eventual atomicity decouples a cross-zone transaction into multiple steps (relay transactions), each involves a single zone. Those steps are relayed and executed across zones by miners. Our system doesn't guarantee miners to handle all relay transactions, just like transaction handling is not guaranteed in Bitcoin or Ethereum. Instead, miners in each zone are incentivized to complete each step by handling relay transaction and ensuring the atomicity eventually.

Effective Mining Power Amplification is introduced to reinforce the security for consensus zones. A blockchain system relies on the majority of mining power to outpace attackers. However, when the mining power is distributed to different zones, an attacker can gather the

mining power toward a single zone and may easily exceed the 51% threshold within that zone. To address this problem, we propose *Chu-ko-nu*² *Mining* to ensure the per-zone security. With Chu-ko-nu mining, a miner is allowed to create multiple blocks in different zones by solving one proof-of-work puzzle. This greatly amplifies the effective mining power of honest miners who evenly distribute the mining power across zones. Such amplifications, on the other hand, doesn't apply to attackers because the amplified mining power is forced to be evenly distributed to multiple zones, which can not be gathered towards a single zone. In this way, the effective mining power in each zone will be at the same level of total physical mining power in the entire network, which makes attacking an individual zone as hard as attacking the entire network.

Contributions in this paper include the following:

1. A scale-out blockchain system that divides workloads of communication, computation, storage, and memory for state representation into independent and parallel zones. Our system keeps the burden of individual full nodes at a low level as the network grows.
2. An eventual atomicity technique for efficient handling of cross-zone transactions, ensuring correctness and robustness in zones that work asynchronously.
3. Chu-ko-nu mining, a novel proof-of-work scheme, to prevent lowering the attack bar when the mining power is dispersed into multiple zones.

To demonstrate the effectiveness of our system, we carry out a set of experiments on a testbed including 1,200 virtual machines worldwide with the historical data of ERC20 payments from the Ethereum network. In these experiments, our system has delivered 1,000 \times throughput and 2,000 \times capacity over Bitcoin and Ethereum networks.

2 Background

We now provide necessary background of this work including details of blockchain systems and a high-level comparison of two consensus mechanisms, i.e., Proof-of-Work (PoW) and Proof-of-Stake (PoS). Lastly, we discuss the differences between the UTXO and Account/Balance transaction models.

²Chu-ko-nu is a repeating crossbow shooting multiple arrows at once. It was invented by Zhuge Liang during the warring states period in ancient China.

2.1 Blockchain System

A blockchain system such as Bitcoin [37] and Ethereum [6] contains many compute nodes as miners and full nodes. Transactions are incremental updates of the state, which are confirmed and carried by blocks. Blocks are created by miners. The verification of a block is involved with a mathematical puzzle (also called the proof-of-work, PoW), which is moderately hard on the request side but easy to check for the network. Miners compete with each other, and the first miner who solves the puzzle will be given rewards and the one-time privilege for creating a new block. A newly-created block has to be sufficiently propagated among miners and full nodes before the next block can be created. Due to the network delay, other miners may still work on different blocks, and diverge the chain into different paths when appending blocks on the chain. The divergence of the chain is called a fork, and the blocks not in the main chain are called orphaned blocks. Enlarging the block size (higher propagation latency) or lessening the creation interval may lead to more orphaned blocks, and even prevent the system from converging to a single longest chain in extreme circumstances (e.g. orphan rate > 50%). We will provide detailed discussions on related studies in Section 8.

We summarize other aspects that affect the overall performance of a blockchain system with the consensus protocol as below:

- *Consensus*: The sequential nature of block creation and confirmation required by the consensus protocol is the major challenge of scalability. This is bound by the throughput as analyzed above.
- *Communication*: Information, including unconfirmed transactions and newly-created blocks, needs to be exchanged between all miners and full nodes. It is bound by the local bandwidth.
- *Storage*: All accepted blocks of the chain need to be stored persistently in every miners and full nodes. These are bound by the local disk space.
- *Representation*: The global states of the entire network, e.g. per-address balance and smart contract state, are maintained by every miners and full nodes. These are bound by the size of the host memory.

A scalable design of blockchain has to take all four aspects into consideration.

2.2 PoW and PoS

As discussed above, PoW in predominant cryptocurrencies, including Bitcoin and Ethereum, requires miners to

do a compute-intensive verification to maintain the consensus on each block. It yields huge electricity consumption, but it sets fundamental real-world values to the corresponding cryptocurrencies.

In contrast, PoS selects the creator of a block in a deterministic fashion that usually depends on the stake (wealth) of a node. Existing PoS systems adopt different methods to produce the randomness in the leader election to ensure decentralization and security [22, 7, 45, 18]. Although PoW is used in this paper, our technique is orthogonal to the actual consensus mechanism used per zone. Please refer to Section 8 for detailed discussions on state-of-the-art research of PoW and PoS.

2.3 UTXO and Account/Balance

There are two major types of transaction models in cryptocurrencies. The former is the Unspent Transaction Output (UTXO) model, where a transaction spends outputs from previous transactions and generates new outputs that can be spent in future transactions. In a UTXO-based system, a user or an account may have multiple UTXOs. When a user wants to spend money, she uses one or more UTXOs to cover the cost and may get some changes back as new UTXOs. This model is used by Bitcoin and many blockchain systems [11, 32, 23]. The latter is the account/balance model, which is similar to a bank account. Before approving a transaction, the bank needs to check if the account balance can cover the cost. This model is used by Ethereum and it is thought to be better than UTXO for supporting smart contracts [6].

Our system uses the account/balance model due to its simplicity since a transaction with an arbitrary amount can be performed with one sending account and one receiving account (instead of multiple UTXOs on both sides). Additionally, the balance can be extended to more complex state to support programmable application logic. Another important benefit offered by the account/balance model is allowing transactions to carry incremental updates of states, as oppose to the UTXO transactions that can only carry full states. This makes a significant savings of transaction size for applications like non-fungible tokens (e.g., Ethereum’s ERC-721 token), in which the state is a set of unique identifiers.

3 System Design

Before diving into details, let’s first check the high-level architecture of our system for handling a payment that involves two users from different zones, i.e., zones *A* and *B* as shown in Figure 1. In that case, the withdraw operation ρ that only involves the state in zone *A* is handled by a miner in zone *A*. If the account balance satisfies the cost of this withdraw operation, the corresponding

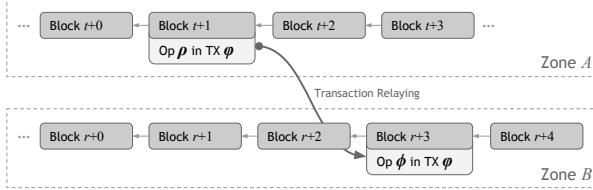


Figure 1: Message passing by relaying transaction across asynchronous zones.

block $t + 1$ carrying the transaction (*initiative transaction*) will be created by the miner and only be appended to the chain of zone A . After that, a *relay transaction* carrying the deposit operation ϕ is composed in zone A and forwarded to zone B . The deposit operation ϕ that only involves the state in zone B can always be executed, regardless of the balance of the target account in zone B . Once the relay transaction is picked up by another miner in zone B , operation ϕ will be executed, concluding the complete of the payment transaction.

3.1 Partitioning and Naming

In our system, an account, or a user, is represented by its address, i.e., a fix-sized hash value of its public key. Our system uniformly partitions the space of user addresses into 2^k zones in a fixed and deterministic way: a zone is identified by its *sharding scale* k and *zone index* s ($s \in \{0, 1, \dots, 2^k - 1\}$).

Given a sharding scale k , the zone index of a user can be easily derived, i.e., by calculating the first k bits of its address. The zone index of an initiative transaction is determined by the payer’s address, and the zone index of a relay transaction is determined by the payee’s address. A block is specified with the $\langle s, k, h \rangle$, with h being its height of the chain. Sharding scale k defines the number of zones and in turn determines the throughput and the capability of the entire network. The following discussion assumes a fixed k for simplicity.

In our system, full nodes join *swarms* to broadcast new transactions and receive blocks from other full nodes, including miners. A swarm is a group of nodes that participate in the replication of the same data set. In Bitcoin or Ethereum, there is only one swarm and every full node replicates the same data set, including all blocks and transactions. In our system, multiple swarms are established for different purposes. A distributed hash table (DHT) is employed for swarm addressing and peer discovery. Details are described in section 7.

Our system has a *global swarm* joined by all full nodes for replicating the minimum common information of all zones. On the other hand, most communication occurs in *zone-specific swarms* with full nodes belonging to spe-

cific zones only. In each swarm, the participating full nodes are sparsely connected, and use the gossip protocol to broadcast messages. Similar to zones, zone-specific swarms are also identified by zone index s and sharding scale k .

3.2 Isolated Intra-Zone Workload

A full node, or a miner, will have a persistent identifier that is initialized randomly; it determines a particular zone the node should work on. With address space partitioning, a blockchain is established within each zone independently. A miner only competes on PoW with other miners in the same zone and confirms transactions from its own zone. Full nodes will ignore any blocks or transactions received that do not belong to their zone, although those are unlikely to be received.

Therefore, the computation and storage related to transaction validation and chain formation are independent and isolated between zones: (1) a miner is only responsible for mining transactions that happen within the zone in which it has chosen to participate, and (2) any full node only records the chain for balances of users in its own zone. As the entire network grows, more zones will be created, ensuring that the burden of computation and storage on an individual node is always at a reasonable level. A low barrier of joining and operating in the network for a full node is essential to maintaining decentralization and robustness of a blockchain system.

3.3 Minimized Cross-Zone Overhead

In a blockchain system, most communication is for replicating unconfirmed transactions and for broadcasting new blocks carrying confirmed ones. In our system, such communication is performed only among nodes within the zone. Our system maintains a distributed hash table (DHT) on each node. After getting the zone index s of an unconfirmed transaction or a forwarding block, our system selects out nodes having the same zone index as s based on the local DHT routing table, and it sends the transaction and block to these nodes following the gossip protocol [12] as is used in Bitcoin and Ethereum. This isolates most communication within each zone.

For cross-zone transactions, our system sends relay transactions only to destination zones instead of the whole network. Additionally, minimized data for chain forming excluding actual confirmed transactions are replicated across all zones. We will discuss this in the next section.

4 Efficient Cross-Zone Atomicity

We divide a block into two parts: a *chaining-block* for the chain formation and the PoW verification, and a *transaction-block* carrying actual confirmed transactions. As shown in Figure 2, a chaining-block, e.g., Θ^a , carries block metadata, including a PoW nonce, a pointer to the precursor block, a Merkle tree [34] root of the list of confirmed transactions, etc. In addition, a chaining-block provides the Merkle tree root for the list of all relay transactions originated from initiative transactions in this block, which is used for the validation of relay transactions in other zones. A transaction-block, e.g., Φ^a , that records the transaction list (same as those in existing systems like Bitcoin and Ethereum) is only replicated and stored by full nodes in the zone. Compared to the hundreds of kilo-bytes used for transactions, a chaining-block has a fix-sized data structure that takes roughly 100 bytes, introducing negligible overhead for both communication and storage.

For an initiative transaction with a withdraw operation ρ from payer a and a deposit operation ϕ to payee b , it can be immediately handled within the zone if a and b belong to the same zone. When a and b are from different zones, we introduce a dual-stage transaction handling mechanism by deriving and forwarding a relay transaction that carries the deposit operation to its destination zone. Figure 2 illustrates the process of cross-zone payment with the data structures processed.

Transaction Validation and Forwarding at Zone A

1. An unconfirmed transaction $\langle \rho, a, \phi, b \rangle$ is picked up by a miner in the payer a 's zone, when the miner constructs a new block.
2. The initiative transaction is validated if the balance of a is not less than the transfer amount. If the balance is insufficient, the transaction will be marked as invalid, and be concluded and embedded in the block.
3. Otherwise, a chaining-block Θ^a and a transaction-block Φ^a are constructed. Φ^a has a list of validated transactions including the one from a to b .
4. The miner works on a PoW puzzle specific to the list of all confirmed transactions.
5. After the PoW puzzle is solved, immediately the chaining-block Θ^a is broadcast in the global swarm and the transaction block Φ^a is broadcast in a 's zone-specific swarm.
6. Intra-zone transactions are executed and concluded.
7. The withdraw operations ρ in all cross-zone transactions are then executed.

8. Each cross-zone transaction derives an outbound relay transaction $\psi := \langle \phi, b, \gamma \rangle$, which will be sent to the destination zone, i.e., the payee b 's zone B .

Relay Transaction Handling at Zone B

1. An inbound relay transaction $\psi := \langle \phi, b, \gamma \rangle$ is picked up by a miner in payee b 's zone when constructing a new block.
2. The miner verifies the inbound relay transaction against its originate block Θ^a . Skip this if invalid.
3. The miner constructed a new chaining-block Θ^b and a new transaction-block Φ^b . Θ^b including the inbound relay transaction $\langle \phi, b, \gamma \rangle$.
4. The block Φ^b will be broadcast in b 's zone after the PoW puzzle is solved.
5. The deposit operation ϕ is executed, concluding the transaction $\langle \rho, a, \phi, b \rangle$.

In Figure 2, the transaction-blocks are only propagated and stored in their own zone, i.e., Φ^a and Φ^b in zone A and zone B , respectively. The relay transactions, e.g., ψ , are generated at the originate zone, i.e., zone A , and only sent to the destination zone, i.e., zone B . The chaining-blocks, e.g., Θ^a and Θ^b , each of which has roughly 100 bytes, are replicated to all zones.

4.1 Verification

Transaction Verification

When a miner in zone B receives a relay transaction from zone A , it needs to verify this transaction in order to avoid the attack from a malicious peer. As shown in Figure 2, a forwarded transaction $\psi := \langle \phi, b, \gamma \rangle$ includes verification data γ , where

$$\gamma := \langle s, k, t, p, \{h_q\} \rangle, \quad (1)$$

position pointer p denotes the position in the list of outbound relay transactions in its originate block, Merkle tree path $\{h_q\}$ refers to hash values of all sibling nodes on the path from Merkle tree root to its entry; and zone index s , sharding scale k , and height t are used to identify its originate block.

The Merkle tree root will be recalculated using the Merkle tree path $\{h_q\}$ and the transaction $\langle \phi, b \rangle$ itself. It is verified if the recalculated Merkle tree root matches with that in its originate block Θ^a and Θ^a is on the chain in zone A . Note that the relationships to siblings (left or right) is not encoded in $\{h_q\}$, which is inferred from p instead.

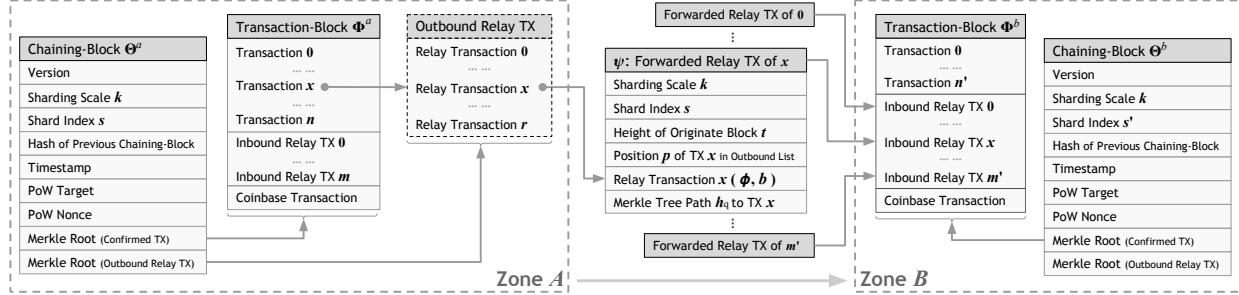


Figure 2: Data structure of the chaining-blocks and transaction-blocks. Outbound relay transactions are derived from confirmed transactions and forwarded with verification data (\mathbf{h}_q and etc.)

Block Verification

On receiving a block (pair of chaining-block and transaction-block) broadcasted by a miner, a full node needs to verify the block to defend malicious miners. In our system, a full node verifies three types of transactions. As shown in Figure 2, they are:

1. Confirmed initiative transactions in its own zone.
2. Inbound relay transactions previously forwarded from other zones.
3. Outbound relay transactions forwarded to other zones.

To save the storage space of full nodes, transactions of the first two types are actually embedded in the transaction-block; while outbound relay transactions are not, since they can be derived from the list of confirmed initiative transactions.

The confirmed transactions are verified against current user states. Any block containing illegal transactions or mismatched pairs of initiative/relay transactions will be rejected. The inbound relay transactions are verified against their originate blocks as described in section 4.1. This process also checks if all outbound relay transactions are created as expected, by double checking the Merkle tree root of the list of all outbound relay transactions. It is assumed that outbound relay transactions are ordered and precisely consistent with the order of confirmed initiative transactions. Note that only the Merkle tree root is embedded in the chaining-block, instead of the outbound relay transactions themselves.

4.2 Eventual Atomicity

A payment transaction involving withdraw and deposit operations, should be atomic to ensure correctness of the global ledger. In existing blockchain systems, e.g., RSCoin [11] and OmniLedger [23], the variants of two-phase commit (2PC) mechanism [36] are used to ensure the atomicity, with the known lock/unlock overhead.

In our system, for a cross-zone transaction, we allow the withdraw operation to execute first, interleaving with other transactions then the corresponding deposit operation to be settled later. What is achieved is that once the withdraw operation is confirmed, the deposit operation will be executed eventually. We call such an atomicity, **Eventual Atomicity**. We optimistically assume withdraw operations, carried by relay transactions, will be eventually picked as long as there are well-behaved miners that want to earn transaction fees. What our design ensures is that relay transactions will not be discriminated by sufficiently incentivizing with a fee split.

Theoretically, there could be bad-behaved miners creating empty blocks without confirming any transaction, neither for normal transactions and relay ones. In that case, throughput will be harmed and eventual atomicity will not be fulfilled until a well-behaved miner eventually gains the opportunity for block creation. In the history of the Bitcoin and Ethereum networks, creation of empty blocks is possible but rare [17, 33].

By default, the newly-derived outbound relay transactions are forwarded to their destination zones by the miner, who created the chaining-block and the transaction-block in the originate zone. Once the relay transaction gets replicated in the destination zone, it will never expire before being picked up by a miner, unless its initiative transaction is invalidated, e.g., being orphaned block due to the chain fork (we will discuss this case later). If a relay transaction is accidentally dropped, which is extremely unlikely, it can be reconstructed by any full node in the originate zone based on its originate block on the chain. No additional verification or consensus is required to restart the replication of the reconstructed relay transaction.

In the existing blockchain system, a payment transaction will be visible to its payee once it is packed in a block on the chain (first confirm). It will be secured after $n - 1$ successive blocks appended (n -th confirm, $n = 6$ in Bitcoin and 12 in Ethereum). In contrast, a cross-zone payment transaction in our system will be visible to

the payee, once its relay transaction is forwarded to the payee’s zone, and its originate block becomes available. With the eventual atomicity, the transaction is considered as eventually secured, once its initiative transaction gets n -confirmed, and the relay transaction gets a first confirm. Since miners of these two zones are working independently, n -confirmation of the initiative transaction is overlapped with the forwarding and first confirmation of the relay transaction. Thus, eventual atomicity introduces no additional delay, which is also demonstrated in our experiments in section 7.3. Theoretically, additional latency may occur, when the relay transaction waits too long to be picked up by a miner. This can take even longer than n -confirmation of its initiative transaction.

4.3 Fork Resolution

In a proof-of-work consensus system, it is possible for different miners to create two, or even more, blocks at the height, which are *forks*. Eventually, after a successful fork resolution, only one block will be accepted among those and the rest will be discarded as *orphan blocks*. The longest-chain rule [37] is proposed to resolve forks in Bitcoin and GHOST protocol [42] is also employed in Ethereum. We use GHOST protocol in consensus zones, which is reliable even when the fork rate is high.

In each consensus zone, fork resolution is performed independently in exactly the same way as previous single-chain consensus systems. A block can be one of the three states:

1. **Available**: no fork, or the block is on the winning path.
2. **Unsolved**: the block is on one of the equally competing paths.
3. **Orphan**: the block is on the losing path.

Fork resolution is a continuous procedure as more blocks being created and appended. A previously available block may lately become orphan or unsolved, and vice versa.

With eventual atomicity, the consequence of fork resolution in one zone may affect the validity of relay transactions forwarded and confirmed in another zone. Verification of relay transaction relies on the proof related to its originate block. If the originate block is no longer available after fork resolution, the proof will be invalidated and in turn invalidates all relay transactions that originated from it.

Pre-Confirmed

An unconfirmed relay transaction will not be considered in new block creation until its originate block is in available state. Relay transactions will stay in the uncon-

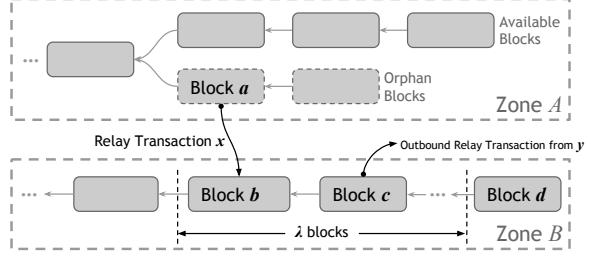


Figure 3: Relay transaction invalidated due to orphaned initiative block after fork resolution.

firmed transaction set regardless of the validity of their originate block and wait for originate blocks become available.

Post-Confirmed

As shown in figure 3 Zone A, a block a previously available lately become unsolved/orphan, and unfortunately, a relay transaction x originated from it has already been confirmed and embedded in a block b in zone B. In that case, block b will not be invalidated and just the relay transaction x will be invalidated. While, the ledger state in zone b is required to be rebuilt by executing all transactions of all historical blocks since the genesis block, skipping all invalidated relay transactions including x . In practice, state rebuilding is accelerated by state checkpoints so that only operations of recent blocks are re-executed.

Implicate Subsequent Transactions in Invalidating

After relay transaction x is invalidated, an even worse scenario is that a subsequent confirmed transaction become invalidated because it relies on the updated state by the relay transaction x . For example in Figure 3, the relay transaction x deposit u tokens in block b and the balance becomes $u_0 + u$. Lately a transaction y is confirmed in block c and it withdrawn v tokens ($u_0 < v \leq u_0 + u$).

If relay transaction x is invalidated and transaction y is a cross-zone transaction, block c will be invalidated and all subsequent blocks after that will be discarded as well. To avoid such a case, a miner will validate candidate transactions against a special state by delaying execution of inbound relay transaction for λ blocks. Thus, transaction y will be not be confirmed until block d , which makes such case unlikely since block a already received at least λ confirmations.

Let the latest block be b , the normal state \mathbb{S} is built by executing all operations in all transactions from the genesis block to block b . For miners, an additional state \mathbb{S}_λ is built by executing from the genesis block to block

$b - \lambda$ and then execute all operations except inbound relay transactions of blocks $b - \lambda + 1$ to b . An unconfirmed cross-zone transaction must be validated against both states \mathbb{S} and \mathbb{S}_λ to be considered as candidate when creating block $b + 1$.

5 Defense Per-Zone Security

In the early stage, miners are individuals and mining power is fragmented. The randomness of voluntary zone assignments based on the local identifier is safe, when no individual miner controls more than 50% mining power in any zone. Professional mining facilities will gradually dominate the mining power by owning great amount of mining power or aggregating mining power from individual miners, which delivers steady, frequent and divided mining rewards [28]. In our system with n zones, a rational mining facility will ideally distribute its total mining power in different zones to maximize the rewards (bias to zones with lower mining difficulty). Eventually, this makes the mining power of the entire network H converge to be evenly distributed across zones. Thus, per-zone mining power will be H/n . When a malicious mining facility gathers all its mining power T focuses on a single zone, the attack will success if $T > H/n \times 50\%$, which will be unacceptably low when with a large n .

To address this issue, we introduce a *Chu-ko-nu mining* mechanism that allows and encourages a miner to create multiple blocks in different zones with one PoW solution. This ensures the effective mining power in each zone is nearly equal to the total physical mining power in the entire network, raising the attack bar in each zone close to 50% when most miners participate in Chu-ko-nu mining. Also, Chu-ko-nu mining can save energy consumed in solving PoW by making it more productive of block creation. Like PoW mechanism itself, security based on Chu-ko-nu mining is driven by incentive.

Batch-Chaining-Block	
A	Version
	Sharding Scale k
	Shard Index s
	Hash of Previous Chaining-Block
	Timestamp
	Merkle Root (Confirmed TX)
	Merkle Root (Outbound Relay TX)
	PoW Target
B	Merkle Tree Path $\{h_i\}$
C	Base Shard Index b of the Batch
	Size of the Batch n
	Batch Sharding Scale k_b
	Batch PoW Nonce η_b

Chaining-Block	
A	Version
	Sharding Scale k_i
	Shard Index s_i
	Hash of Previous Chaining-Block
	Timestamp
	Merkle Root (Confirmed TX)
	Merkle Root (Outbound Relay TX)
	PoW Target
	PoW Nonce η_i

Figure 4: Comparison of a batch-chaining-block and a chaining-block

It will be taken down if the attacker controls more than 50% physical mining power of the entire network, which is also the case of PoW mechanism. While, the defense is all miners that acknowledge the incentive.

5.1 Chu-ko-nu Mining

Chu-ko-nu mining allows a miner use a single PoW solution to create multiple blocks at different zones simultaneously, but no more than one block per-zone. In such a case, a *batch-chaining-block* in Figure 4 will replace the chaining-block, as shown in Figure 2, and get replicated among all zones. Miners are allowed to perform Chu-ko-nu mining for n zones starting from zone index b or all zones ($n = 2^k, b = 0$) based on their capacities of IT resources.

A miner will perform the transaction validation for all involved n zones and collect n chaining-headers A_i , i.e., part A in Figure 4. Without Chu-ko-nu mining, as in Bitcoin or Ethereum, a miner is required to find n nonce η_i ($i \in [0, n - 1]$) that each fulfills

$$\text{hash}(\langle A_i, \eta_i \rangle) < \tau, \quad (2)$$

in which τ denotes the PoW target (a big integer) [37] determining the mining difficulty, and the $<$ operator takes place by regarding the hash value as a big integer. With Chu-ko-nu mining, a miner only needs to find a single nonce η_b that fulfills

$$\text{hash}(\langle h_0, C, \eta_b \rangle) < \tau, \quad (3)$$

in which C is the configuration of the batch (part C in Figure 4) and h_0 denotes the root of Merkle tree Υ_b over the list of all chaining-headers in this batch:

$$\langle A_0, A_1, \dots, A_{n-1} \rangle. \quad (4)$$

Note that we assume the PoW targets are the same in all involved zones. We will discuss the case if they are not in the next section.

Once η_b is found, zone-specific batch-chaining-blocks will be composed and sent to corresponding zones. As illustrated in Figure 4, for each zone, a batch-chaining-block carries a chaining-header (A), its Merkle tree path $\{h_j\}$ (B), batch configuration (C), and the found batch PoW nonce η_b , which are the minimum pieces of information for recalculating equation 3 and for verifying the PoW. Our system doesn't explicitly record the relationships to siblings (left or right) in the Merkle tree path $\{h_j\}$. Instead, relationships are inferred from bits of the offset in the batch list ($s - b$). By this means, the zone index s is coupled with the offset in the block list (Equation 4) in the batch, which guarantees that a miner is able to create only one block for each involved zone in one batch.

In each zone, full nodes, as well as miners, treat batch-chaining-blocks and chaining-blocks equally when accepting a new block. They follow the same approach to detect and resolve forks as described in Section 4.3. A chain of a zone is allowed to contain batch-chaining-blocks and chaining-blocks at different block heights.

Chu-ko-nu mining shares similar spirit with merged mining [4] in practices by allowing creating multiple blocks with a single PoW solution. Merged mining is designed for different motivation and scenario, which is for protecting blockchains with small mining power. Chu-ko-nu mining is designed for reinforcing mining power distribution across zones with amplified effective mining power. Chu-ko-nu mining works with multiple chains with equal role and equal mining power, instead of a parent chain and an auxiliary chain. Chu-ko-nu mining also enforce one-block per-zone which leads to a different data structure and implementation.

5.2 Independent Validation in Zones

Chu-ko-nu mining solves the PoW for chaining-headers in batch, while the batch-chaining-block with solved nonce are zone-specific and sent separately. Per-zone batch-chaining-blocks will be validated and accepted independently in each zone. One block can be orphaned or even invalidated, but this will not affect the validation or the acceptance of others.

Independent validation also allows efficient handling of mixed PoW targets of zones in one batch. The construction of batch-chaining-blocks in this case is the same as described in section 5.1, but it allows different values of PoW targets in each chaining-header. While, in probing the batch PoW nonce η_b , it is possible that some blocks with high PoW targets (easier to fulfill) are fulfilled but others are not. Batch-chaining-blocks for these fulfilled zones will be composed and sent to their zones immediately, regardless PoW targets of other zones are not fulfilled (thus not be sent out). The fulfilled blocks in the list of chaining-headers (Equation 4) will be removed and replaced with their successive candidate blocks. The probing of the batch PoW nonce will switch to a new puzzle with updated h_0 in Equation 3. Since the event of finding a fulfilled nonce in different zones is independent and random, such switching will not reduce the mining efficiency.

5.3 Redistributed Mining Power

This section explains why Chu-ko-nu mining can make an attack on a single zone as hard as the attack on the entire network, which can set the attack bar at the same security level of Bitcoin and Ethereum.

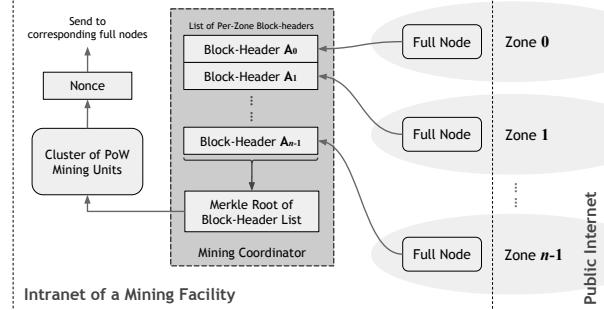


Figure 5: Internal architecture of a mining system

We use **hash rate** to describe the mining power, which is proportional to the speed of producing new blocks. In a network having 2^k zones, let m_p be the total physical hash rate of miners that participate in Chu-ko-nu mining and m_d be the total physical hash rate of miners that don't (as individual miners). The effective hash rate m_s distributed in each zone can be calculated as

$$m_s = \frac{m_d}{2^k} + m_p. \quad (5)$$

If a malicious node can obtain $> \frac{m_s}{2}$ hash rate in any zone, it can control this zone. Therefore, the attack bar in each zone, which can be calculated as the obtained hash rate of a malicious node divided by the total hash rate in the network, is

$$> \frac{m_s}{2 \cdot (m_d + m_p)} = 50\% - \frac{m_d \cdot (2^k - 1)/2^k}{2 \cdot (m_p + m_d)} \quad (6)$$

, which converges to 50% when the mining facility dominates the total hash rate. For example, if the mining facility contributes 99% hash rate in a 256-zone network, a successful attack requires 49.5% of total physical hash rate in the network. The current situation (September 2018) in Bitcoin network, 100% hash rate is contributed by mining facilities[5].

To maximize the incentive from a single PoW solution, professional mining facilities will participate in all zones and hopefully takes rewards from all zones. Chu-ko-nu mining actually amplifies miner's mining power, which is multiplied with the number of zones a miner participates in. The amplified mining power is evenly distributed to all involved zones. Such an effective amplification of mining power helps honest miners that distribute mining power to all zones but don't apply to attackers targeting on a single specific zone.

5.4 Scalable Mining System

A mining system of Bitcoin or Ethereum in a mining facility is a distributed system that consists of hundreds or even thousands PoW mining units, e.g., GPU and Specialized Application-Specific Integrated Circuit (ASIC)

just focusing on hash probing, plus a few PCs working on actual transaction validation and block construction.

In our system, a mining system is desired to monitor multiple zones. The partitioning scheme in the network naturally provides a scalable solution for a mining facility observing a large number of zones. As illustrated in Figure 5, besides the cluster of PoW mining units, in addition, there is a cluster of full nodes for observing all involved zones, independently discovering new blocks, validating transaction, and constructing candidate block-headers A_i . Once A_i is updated in any zone i , new A_i will be sent to *Mining Coordinator* in Figure 5. Merkle tree of block-header list (Equation 4) will be recalculated by the mining coordinator and the updated Merkle tree root h_0 (Equation 3) will be broadcast to all PoW mining units. Such design provides an example and demonstrate how a scalable mining system can be implemented and operated for professional mining facility.

6 Discussions

6.1 Single Address Hotspot

It is possible that a single address is involved in a great number of transactions, e.g., a deposit address of a large cryptocurrency exchange. In the workload described in Section 7, the address 0x3f5CE5FBFe3E9af3971dD833D26bA9b5C936f0bE, which is one deposit address of Binance (a top cryptocurrency exchange), is the payee of more than 2% total transactions. Since a single address is the finest unit in our partitioning scheme, right now our system can not further partition such workload into multiple zones.

In practice, such a "single address hotspot" issue can be easily resolved with the co-design of applications at the upper layer. For example, an application or a user can allocate multiple deposit addresses in different zones for load balancing. An institutional operator announces a list of addresses for deposit, and the wallet application automatically chooses a random address, or even an intra-zone address if available, for coin transferring. As a result, the transaction throughput of multiple zones can be leveraged.

6.2 Incentives and Fees

We follow Bitcoin's incentive model by rewarding the miners with phase down coinbase in every zone, which end up with a fixed total supply of cryptocurrency. The transaction fee is a parameter set by whoever issue the transaction, which is usually based on the average amount of the fee in recent confirmed transactions. A rational miner prioritizes unconfirmed transactions based on the transaction fee.

We introduce fee splitting for cross-zone transactions, which incentivize both miners working on initial step and relayed step of transaction handling so that relayed transactions will be equally prioritized with transaction fees at similar levels. For a simple payment, the transaction fee can be equally split. For complex transaction with programmable transaction logic, the transaction splitting should be based on the evaluation of workloads in each step similar to the gas calculation in Ethereum.

We don't introduce additional fees for propagating cross-zone transactions. Optimistically, we expect such task will be done voluntarily by all full nodes and miners throughout the network as voluntarily propagating of intra-zone transactions works well in Bitcoin and Ethereum networks. Block creation with cross-zone transactions is a bit more costly than intra-zone transactions. We recommend cross-zone transaction issuer set double or even triple the amount of transaction fees.

We encourage Chu-ko-nu mining by providing equal coinbase reward for blocks generated using Chu-ko-nu or not. Thus, given fixed physical mining power, the profit of Chu-ko-nu mining is proportional to the number of zones participating. It is rational for professionally-operated miners to work on all zones simultaneously, which enhances the overall security of the network.

We do not extend to the quantitative analysis on the incentives and economics of the ecosystem, as it is beyond the scope of this paper.

6.3 Generalization beyond Payment

So far, our discussions are based on the withdraw-deposit paradigm that serves well payment-centric scenarios, e.g., Bitcoin. It is also desirable to extend the model for complex transaction logic and go beyond payment-centric applications. To this end, a main challenge is to correctly handle generalized relay transactions, and guarantee the atomicity. We extend the withdraw-deposit paradigm to a more general model with programmable transaction logic, which is presented in the appendix.

The proposed programmable transaction logic supports programmable issuing and transferring of cryptocurrencies as well as user-defined fungible and non-fungible tokens similar to ERC20/ERC721 tokens on Ethereum. It also supports more complicated applications like a domain registration system. Our method requires that a cross-zone transaction can be verified in a single zone and be completed by one-step irrevocable relay transactions, which don't support applications like many-to-many payments. It can be extended to revocable relay transactions and multi-step relaying, we leave those improvements to future works.

7 Experimental Results

Our system is implemented using C++. Cryptography is implemented based on Botan cryptography library (v1.11) [30] and Intel IPP Cryptography library (v7.1) [21]. We use RocksDB (v4.11) [16] to store archived blocks and transactions. We implement Mainline DHT [31] for P2P routing and swarm formation facilities peer discovery for the global swarm and per-zone swarms.

We evaluate the proposed system by playing back the complete historical ERC20 payments in Ethereum from the beginning up to the block height 5867279, which includes 16.5 million unique addresses and 75.8 million transactions. We deploy our system on a distributed environment that includes 1,200 virtual machines, each of which has 8 cores and 32GB memory. These machines are uniformly distributed in 15 Availability Zones for testing cross-country latency in the real world. In the test network, we restrict the end-to-end peak bandwidth to 30Mbps and the measured average end-to-end latency is 102.48 msec. In every zone, we set a 32KB limit for the block size and a target of 15-second block creation interval, which yields around 15.6 TPS for one-to-one token transfer. The average orphan rate in every zone is 8.3%, which is independent of the number of zones. On such a testbed, we support 48,000 nodes of blockchain in our system.

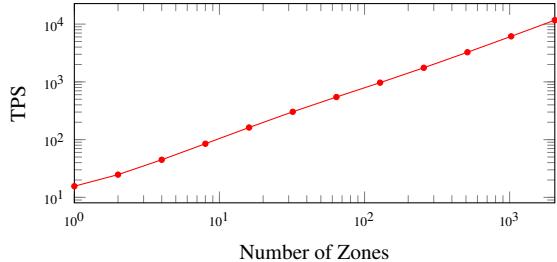


Figure 6: Linear scaling out with multiple zones.

7.1 Scalability

We first evaluate if our partitioning mechanism introduced in Section 3.1 can balance the number of payments between zones. We change the sharding scale k to generate different numbers of zones, i.e., setting k to 4, 5, 8 and getting 16, 32, 256 zones, respectively. Figure 7 illustrates transactions handled in each zone are balanced, with different sharding scale k . While, there exist single address hotspots, which can be further optimized as discussed in Section 6.1.

We evaluate the scalability by measuring the actual throughput of our system with a different number of zones. We fix the number of nodes in each zone, i.e., 24 nodes with 12 miners in average per zone, when increas-

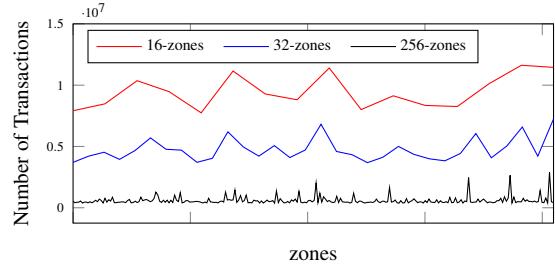


Figure 7: Transaction distribution across zones.

ing the number of zones. Figure 6 shows that the measured TPS scales out as the number of zones increase. The system exhibits the linear scalability, and achieves up to 11,694.89 TPS when there are 2,048 zones. The only exception occurs when increasing the number of zones from one to two. Due to the overhead of relaying transactions, the performance gain using two zones over one zone is 1.88 \times .

7.2 Overhead

Although the throughput of our system scales out linearly with the number of zones, relaying transactions across zones actually introduces the overhead, amplifying the number of transactions and total size of data to be stored and replicated. Figure 8 shows the percentage of cross-zone transactions grows when increasing the number of zones. Almost all transactions will lead to a relay transaction when there are more than 64 zones in our experiments. Such an amplification doubles the total number of transactions at most and don't weaken the scalability of throughput at all. As shown in Figure 6, throughput keep scaling out when there are more than 64 zones.

Figure 9 illustrates the amplified sizes of data replication and storage in the entire network. First, when increasing the number of zones, almost all transactions lead to a relay transaction, i.e., more than 64 zones in this case. Transactions are put into blocks and persistent storage in original zones; and relay transactions are those of destination zones, doubling the size of transaction-blocks. Second, compared to the transactions in a block, which takes hundreds of kilo-bytes, the chaining-block that has tens of bytes is much less significant. However,

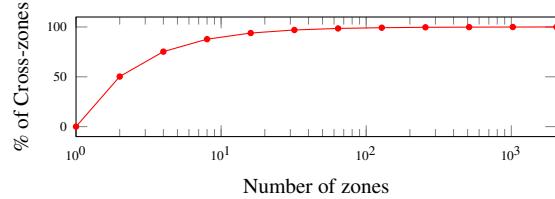


Figure 8: Percentage of cross-zone transactions, which approaches to 100%. Almost every original transaction produced a relay transaction.

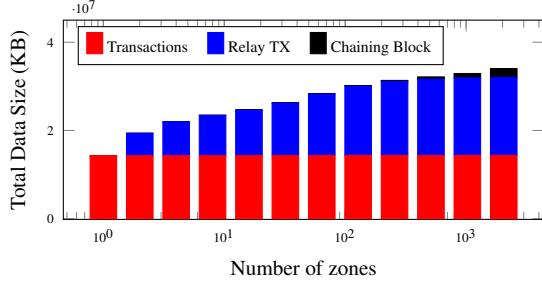


Figure 9: Sizes of the blockchain data in the entire network.

since the chaining-block is duplicated to all zones, their total size on storage in the entire network will be amplified with the number of zones. As shown in the figure, their total size is pushed up to tens of kilo-bytes with 2048 zones, 6.2% of the transaction size, which is acceptable in practice.

7.3 Confirmation Latency

The average number of connections per node affects the data propagation speed in the network, and in turn, the confirmation latency of blockchain. Figure 10 shows the cumulative distribution function of time elapsed and number of nodes reached. We configure the average number of connections per node to 16, 32, 64, 128, and observe that the fastest propagation speed is obtained at 128. In our experiments, each full node connects to 68.5 peers in average.

As presented in Section 4.2, a transaction is first confirmed when its block is replicated between nodes in the network; after n block, the transaction is secured. Bitcoin, which configures n to 6 with a 10-minute interval of block generation, requires 1 hour to secure a transaction. In our system, a relay transaction must be on-chain before the original transaction reaches n -confirms, so there is no additional confirmation latency. Figure 11 shows the first confirmation time of relay transactions. The latency is in a range of 13 to 21 seconds, when the number of zones is larger than 30 (almost all transactions have

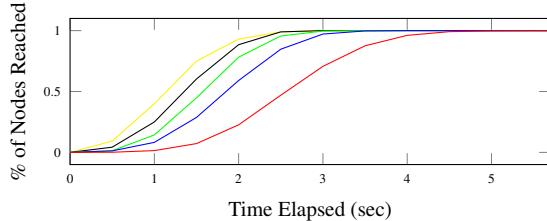


Figure 10: Transaction (<1KB) propagation speed with different average number of connections from each node to other peers. Fastest propagation shown here is 128-connected, slowest one is 8-connected. Ones in between are 64/32/16-connected.

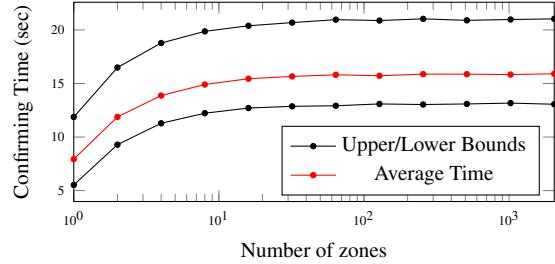


Figure 11: Average first confirming time of transactions.

relay transactions). Since we configure a 15-second interval of block generation, the original transaction is secured after 90 seconds ($n = 6$, same to Bitcoin), which is enough to cover the confirmation latency of relay transactions.

7.4 Throughput and Orphan Rate

We also evaluate the TPS and orphan rate of our system with different block sizes and block creation interval in Figure 12. In these experiments, we fix the number of zones to 256. As expected, enlarging block size or lessening block creation interval yields almost linear TPS in our system, since our system is neutral to actual configurations of Nakamoto consensus instances in each zone. However, a larger block size or a smaller block creation interval leads to a higher orphan rate, inducing blocks wasted. Such behavior matches well with the existing blockchain system. In a typical case, we configure a reasonable block size and a block creation interval, e.g., 32 KB and 15 seconds, for high TPS and low orphan rate.

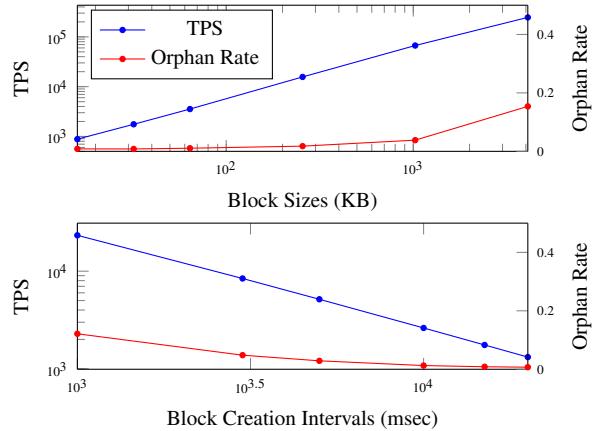


Figure 12: TPS and orphan rate with different block sizes (Upper) and with different block creation interval (Lower). (# zone = 256)

8 Related Work

Many research efforts have been put on improving PoW systems. Instead of the longest branch, GHOST protocol [42] chooses the block whose sub-tree contains most blocks as the main chain to prevent the attack by selfish mining at a fork. Bitcoin-NG [15] selects a leader in each epoch, and allows the leader to post multiple blocks, thus increasing the throughput. SPECTRE [41] and PHANTOM [43] increase Bitcoin’s throughput by replacing the chain-based structure to the Directed Acyclic Graph (DAG)-based structure and merging blocks from different branches to the ledger. SPECTRE provides the partial order between DAG blocks, while PHANTOM can keep the total order. Conflux [29] is another DAG-based protocol. It constructs the consistent total order of transactions over DAG through introducing parent and reference edges and combining ghost for pivot chain selection. It allows blocks outside the pivot chain to be able to contribute to the overall throughput. These proposals can enhance Bitcoin security by eliminating the selfish mining and improve the throughput by merging blocks from different branches to the main chain. However, as discussed in Section 1, the performance of these single chain systems is bound to the available network bandwidth of full nodes. Thus, they cannot scale out to thousands of nodes. Moreover, due to the requirement of replicating and processing forked blocks in these systems, it is unclear how full nodes address and resolve the capacity issue on the memory and storage in a large scale.

PoS systems reach the consensus with the majority of stake share. Ouroboros [22] is a PoS system with enhanced persistence and liveness. It uses a coin-flipping protocol to produce the randomness in the leader election. However, it cannot avoid the targeted attack [18] with the assumption that most leaders are incorruptible in an epoch. Tendermint [26] and Casper the Friendly Finality Gadget [7] adopt the Byzantine Fault Tolerance (BFT) protocol to select the committees, reach the consensus, and tolerant against up to one third malicious nodes. However, Practical BFT (PBFT) protocol [8, 40] used in these proposals has significant communication cost and only scales to dozens of compute nodes. Furthermore, on a public blockchain that anyone can participate in, PBFT has the security issues. The permissionless characteristics of PBFT makes the blockchain under the risk of Sybil attack [13], where an adversary can create an arbitrary number of pseudonyms. And, the absence of public-verifiable and unbiased randomness makes the elected committees under the risk of targeted attack [18], once an adversary can predict their identities.

The recent work [1] has discussed the relationship between BFT protocols and blockchain consensus, and highlighted how to optimize BFT for blockchain.

Zyzyva [25] uses speculation to simplify BFT state machine replication and can reduce replication overhead significantly. SBFT [19] is a scalable, trust, and decentralized infrastructure of blockchain. It can handle hundreds of active replicas and support smart contracts of Ethereum. HoneyBadgerBFT [35] proposes an atomic broadcast protocol to optimize the communication complexity of BFT, making the asynchronous BFT to be able to support hundreds of nodes. ByzCoin [24] leverages the collective signing and optimizes the transaction commitment of the BFT-based blockchain. RandHound and RandHerd [45] provide public-verifiable, unpredictable, and unbiased randomness. Algorand [18] grows the blockchain in asynchronous rounds. In a round, each node computes a verifiable random function to determine if it is a committee member. Once a validator sends a message to prove its membership with its vote, Algorand replaces participants immediately. It can avoid the Sybil attack and targeted attack. However, these proposals are reported to have either the security issue or the performance issue. For example, Algorand is susceptible to bias in the randomness of the variable random function (VRF) invocation [20], while ByzCoin may not reach the agreement on the committee election, as reported in the hybrid consensus system [38].

Many distributed systems, e.g., Google Spanner [9] and Slicer [2], use the sharding protocols to scale out; while these solutions are centralized and cannot be used directly in the decentralized blockchain systems. Elastico [32] is a decentralized sharding protocol. In each consensus epoch, the participants can solve a PoW puzzle to join a consensus committee. The committee of each shard runs PBFT to reach the agreement on a set of transactions and sends the agreement to a final committee. The final committee generates the final values from the received agreements, and broadcasts to the network. First, Elastico doesn’t ensure the transaction atomicity across shards. Second, in order to limit the overhead of running PBFT, it only supports a small number of committees, leading to a high failure probability [23]. Third, although each committee can verify transactions in their own shard, Elastico still broadcasts all blocks to all nodes and requires them to store the whole ledger. This leads to the capacity issue on full nodes inevitably. OmniLedger [23] is a distributed ledger based on a sharding protocol and tries to resolve the problems in Elastico. OmniLedger uses RandHoundm [45] to ensure the leader election bias-resistant and public-verifiable, and introduces Atomix, a two-phase atomic commit protocol, to guarantee the atomicity of cross-shard transactions. It only requires the validators to store the reference point of each shard, instead of the full transaction history, making full nodes more sustainable. The Ethereum community also introduces the beacon chain [14] to support the

sharding protocol. The beacon chain provides the distributed pseudorandomness for selecting committees of validators on each shard. Because the pseudorandomness randomness is susceptible to bias, the sharding-protocol based blockchain shouldn't assume a trusted randomness beacon. A recent study RapidChain [47] further optimizes these sharding protocols. It is resilient to Byzantine faults from up to a 1/4 fraction, e.g., in OmniLedger, to a 1/3 fraction of the participants. It also enhances the throughput via block pipelining and ensures the robustness of the setup for new participants to join the network. Compared to these systems, our work proposes the eventual atomicity to ensure the atomicity of cross-zone transactions without the lock/unlock overhead. Most importantly, a blockchain system that is partitioned to multiple zones/shards exposes a severe security problem, which is not addressed in these existing proposals. With partitioning, the honest majority of mining power or stake share or randomly selected committee is dispersed into individual zones/shards. This significantly reduces the size of honest majority on each zone/shard, thus dramatically lowering the attack bar on a specific zone/shard. Therefore, we introduce the Chu-ko-nu mining to ensure the security after partitioning, and make attacking any specific zone as difficultly as attacking the entire network.

In summary, the existing proposals targeting on the single chain and non-sharding solution [42, 41, 43, 15, 29, 24, 18] are bound to the network bandwidth and cannot scale out; while the sharding systems either do not support full sharding [32], or may lower the attack bar after sharding [23, 47]. To our best knowledge, Monoxide is the only scalable blockchain system implementing full sharding of the consensus protocol as well as the resource usage for scalability, preserving the guarantees of PoW for decentralization, and maintaining the same level of security of Bitcoin and Ethereum.

9 Conclusion

We proposed a scalable decentralized consensus system based on the blockchain mechanism. Without weakening decentralization and security, our technique offers a linear scale-out by partitioning the workload of all key components of a blockchain system including transaction broadcasting, mining competition, chain storage, transaction execution and state representation. We preserved the simplicity of the blockchain system and amplify its capacity by duplicating equal and asynchronous zones, which work independently with minimal coordination, in parallel. Additionally, Chu-ko-nu mining and eventual atomicity are key contributions proposed by our system, ensuring the efficiency and security of systems with thousands of independent zones. In our experiment, we demonstrate that our system delivers 1,000 \times throughput

and 2,000 \times capacity over Bitcoin and Ethereum.

10 Acknowledge

We thank Heung-Yeung Shum (Microsoft), Lidong Zhou (Microsoft Research Asia), Xiaobin Zhang (PPTV), Hongbo Zhang (Bloomberg), Xiao Sophia Wang (Uber), Shuang Zhao (University of California, Irvine), Minghao Pan (AMD) and Shumo Chu (University of Washington) for their suggestions, challenges, and encouraging comments along the way. We also thank our shepherd, Vijay Chidambaram (University of Texas at Austin), and all anonymous reviewers for their helpful feedback.

References

- [1] ABRAHAM, I., AND MALKHI, D. The blockchain consensus layer and bft. *Bulletin of the EATCS* 123 (2017).
- [2] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI’16, USENIX Association, pp. 739–753.
- [3] ALIPAY. World record!! we’ve processed 256,000 payment transactions per second (tps), 2017. <https://twitter.com/alipay/status/929123909970153472>.
- [4] BITCOIN WIKI. Merged mining specification, 2015. https://en.bitcoin.it/wiki/Merged_mining_specification.
- [5] BTC.COM. Pool distribution, 2018. <https://btc.com/stats/pool>.
- [6] BUTERIN, V., AND ET AL. A next-generation smart contract and decentralized application platform, 2013. <https://github.com/ethereum/wiki/wiki/White-Paper>.
- [7] BUTERIN, V., AND GRIFFITH, V. Casper the friendly finality gadget, 2017. <https://arxiv.org/pdf/1710.09437.pdf>.
- [8] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461.
- [9] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google’s globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI’12, USENIX Association, pp. 251–264.
- [10] CRYPTOKITTIES. Cryptokitties, 2017. <https://www.cryptokitties.co>.
- [11] DANEZIS, G., AND MEIKLEJOHN, S. Centrally banked cryptocurrencies. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium* (2016), NDSS ’16.
- [12] DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1987), PODC ’87, ACM, pp. 1–12.
- [13] DOUCEUR, J. R. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, UK, 2002), IPTPS ’01, Springer-Verlag, pp. 251–260.
- [14] ETHEREUM. Ethereum 2.0 phase 0 – the beacon chain, 2019. https://github.com/ethereum/eth2.0-specs/blob/master/specs/core/0_beacon-chain.md.
- [15] EYAL, I., GENCER, A. E., SIRER, E. G., AND VAN RENESSE, R. Bitcoin-ng: A scalable blockchain protocol. In *NSDI* (2016), pp. 45–59.
- [16] FACEBOOK OPEN SOURCE. Rocksdb: A persistent key-value store, 2014. <https://rocksdb.org/>.
- [17] GAUTHIER, P. Why do some bitcoin mining pools mine empty blocks?, 2016. <https://bitcoinmagazine.com/articles/why-do-some-bitcoin-mining-pools-mine-empty-blocks-1468337739/>.
- [18] GILAD, Y., HEMO, R., MICALI, S., VLACHOS, G., AND ZELDOVICH, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP ’17, ACM, pp. 51–68.
- [19] GUETA, G. G., ABRAHAM, I., GROSSMAN, S., MALKHI, D., PINKAS, B., REITER, M. K., SEREDINSCHI, D.-A., TAMIR, O., AND TOMESCU, A. Sbft: a scalable decentralized trust infrastructure for blockchains, 2018. <https://arxiv.org/abs/1804.01626>.
- [20] HANKE, T., MOVAHEDI, M., AND WILLIAMS, D. Dfinity technology overview series, consensus system, 2018. <https://arxiv.org/abs/1805.04548>.
- [21] INTEL. Developer Reference for Intel®Integrated Performance Primitives Cryptography 2018, 2018. <https://software.intel.com/en-us/ipp-crypto-reference>.

- [22] KIAYIAS, A., RUSSELL, A., DAVID, B., AND OLIYNYKOV, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference* (2017), Springer, pp. 357–388.
- [23] KOGIAS, E. K., JOVANOVIC, P., GASSER, L., GAILLY, N., SYTA, E., AND FORD, B. Omnipledger: A secure, scale-out, decentralized ledger via sharding. In *Security and Privacy (SP), 2018 IEEE Symposium on* (2018), Ieee.
- [24] KOKORIS-KOGIAS, E., JOVANOVIC, P., GAILLY, N., KHOFFI, I., GASSER, L., AND FORD, B. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2016), SEC’16, USENIX Association, pp. 279–296.
- [25] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP ’07, ACM, pp. 45–58.
- [26] KWON, J. Tendermint: Consensus without mining, 2014. <https://tendermint.com/static/docs/tendermint.pdf>.
- [27] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 9 (1979), 690–691.
- [28] LEWENBERG, Y., BACHRACH, Y., SOMPOLINSKY, Y., ZOHAR, A., AND ROSENSCHEIN, J. S. Bitcoin mining pools: A cooperative game theoretic analysis. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2015), AAMAS ’15, pp. 919–927.
- [29] LI, C., LI, P., XU, W., LONG, F., AND YAO, A. C.-C. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870* (2018).
- [30] LLOYD, J. Botan: Crypto and tls for c++11, 2015. <https://botan.randombit.net/>.
- [31] LOEWENSTERN, A., AND NORBERG, A. Dht protocol (bep-0005), 2008. http://www.bittorrent.org/beps/bep_0005.html.
- [32] LUU, L., NARAYANAN, V., ZHENG, C., BAWEJA, K., GILBERT, S., AND SAXENA, P. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS ’16, ACM, pp. 17–30.
- [33] MCFARLANE, G. Mining empty blocks is spiking on ethereum — that could be a problem, 2018. <https://ethereumworldnews.com/mining-empty-blocks-ethereum-could-be-a-problem/>.
- [34] MERKLE, R. Method of providing digital signatures, 1979. <https://patents.google.com/patent/US4309569>.
- [35] MILLER, A., XIA, Y., CROMAN, K., SHI, E., AND SONG, D. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS ’16, ACM, pp. 31–42.
- [36] MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 378–396.
- [37] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system, 2008. <http://bitcoin.org/bitcoin.pdf>.
- [38] PASS, R., AND SHI, E. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPICS-Leibniz International Proceedings in Informatics* (2017), vol. 91, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [39] RAY, J. Sharding faqs, ethereum wiki, 2019. <https://github.com/ethereum/wiki/wiki/Sharding-FAQs>.
- [40] SHI, R., AND WANG, Y. Cheap and available state machine replication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2016), USENIX ATC ’16, USENIX Association, pp. 265–279.
- [41] SOMPOLINSKY, Y., LEWENBERG, Y., AND ZOHAR, A. Spectre: Serialization of proof-of-work events: Confirming transactions via recursive elections, 2016. <https://eprint.iacr.org/2016/1159.pdf>.
- [42] SOMPOLINSKY, Y., AND ZOHAR, A. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security* (2015), Springer, pp. 507–527.

- [43] SOMPOLINSKY, Y., AND ZOHAR, A. Phantom, ghostdag: Two scalable blockdag protocols, 2018. <https://eprint.iacr.org/2018/104.pdf>.
- [44] STATEOFTHEADAPPS.COM. Dapp statistics, 2019. <https://www.stateofthedapps.com/stats>.
- [45] SYTA, E., JOVANOVIC, P., KOGIAS, E. K., GAILLY, N., GASSER, L., KHOFFI, I., FISCHER, M. J., AND FORD, B. Scalable bias-resistant distributed randomness. In *Security and Privacy (SP), 2017 IEEE Symposium on* (2017), Ieee, pp. 444–460.
- [46] VISA. Visa acceptance for retailers, 2018. <https://usa.visa.com/run-your-business/small-business-tools/retail.html>.
- [47] ZAMANI, M., MOVAHEDI, M., AND RAYKOVA, M. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS ’18, ACM, pp. 931–948.

Appendix

A Programmable Transaction

In Section 3, our discussions are based on a withdraw-deposit paradigm that serves well for payment-centric applications such as Bitcoin. However, it is desirable to have a transaction logic that is more complex and goes beyond payment-centric applications. To this end, the proposed system is required to correctly handle (generalized) relay transactions and guarantee eventual atomicity. We extend the withdraw-deposit paradigm to a more general model with programmable transactions logic.

A.1 World State

We extend per-user balance to per-user state with user-customizable data types and structures. The world state Ω only contains per-user states:

$$\Omega := \{\psi_{\mu_i}\}, \quad (7)$$

where μ_i donates a user identified by its address. Each per-user state ψ_* can be simple integers (that describe account balance) or be arbitrarily complex data structures like strings, lists, sets or maps.

A.2 Transaction

We extend the fixed logic of withdraw and deposit to programmable logic with a few restrictions as follows. A transaction φ is an atomic updates over the state of users. It modifies states of one or multiple users with a condition ρ :

$$\varphi := \langle \rho(\psi_{\mu_c}), \{\psi_{\mu_i} \leftarrow \phi_i(\psi_{\mu_i}; \psi_{\mu_c})\}, \kappa \rangle, \quad (8)$$

where κ is the argument of the transaction available to condition and operation logic, ρ denotes the condition for validating the transaction (a binary function of state ψ_{μ_c}). $\{\phi_i\}$ contains operations that modify states of users $\{\psi_{\mu_i}\}$ if ρ is true. Note that we restrict the condition ρ to only be related to a single user μ_c , so that a transaction can be validated solely in the user μ_c 's zone. If the condition ρ is true, all operations in $\{\phi_i\}$ are required to be executed successfully without failure or throwing exceptions. Each operation ϕ_i may have arbitrarily complex logic but restricted access pattern. Specifically, ϕ_i updates the state ψ_{μ_i} only with access to the previous state of ψ_{μ_i} and readonly access to ψ_{μ_c} which is involved in the condition ρ (and nothing else). Modifying ψ_{μ_c} is allowed if $\mu_c \in \{\mu_i\}$, which is a frequent case.

A.3 Operation

When a transaction is validated (i.e., ρ is true), operations $\{\phi_i\}$ modifying cross-zone users' states will be carried by relay transactions and executed in an asynchronous manner. As described in Section 4, relay transactions from different originate transactions might be interleaved and disordered during execution. To ensure a consistent end result without requiring serialization, we restrict all operations $\{\phi_i\}$ to be order-independent with any arguments κ_a and κ_b :

$$\forall \kappa_a, \kappa_b : \phi_i(\phi_i(\psi_i; \kappa_a); \kappa_b) \equiv \phi_i(\phi_i(\psi_i; \kappa_b); \kappa_a). \quad (9)$$

A.4 Smart Asset

The pre-user state ψ_{μ_i} in our system includes the balance of the native fungible token and a dictionary of states of the *smart asset* defined by all 3rd-parties.

$$\psi_{\mu_i} := \langle \gamma, \{v \rightarrow \hat{\psi}_{\mu_i}\} \rangle \quad (10)$$

Native fungible token is the platform currency like Ether in Ethereum, which is consumed in the transaction handling as gas fee. It is issued as the coinbase reward when a miner finds a new block. Smart asset are conceptually borrowed from smart contract in Ethereum.

The definition of a smart asset for standard tokens includes an issuing transaction $\check{\varphi}$ and a payment transaction φ as in the equation (8). Issuing transaction $\check{\varphi}$ is

designed for extending the coinbase logic for rewarding miners. It defines the 3rd-party mineable token. More types of transaction can be defined and invoked when issue a transaction.

Fungible Token

The state of a fungible token is a signed big integer representing the balance $\hat{\psi}_\mu := \beta_\mu$. An example of bitcoin-like issuing transaction can be:

$$\beta_{\mu_m} \leftarrow \beta_{\mu_m} + \left\lfloor \frac{50/n}{2^{[h_b/210000]}} \right\rfloor, \quad h_b > 0 \quad (11)$$

, in which n is the number of zones, μ_m denotes the miner and h_b is the *asset block height*, the number of block appended since the block for deploying the specific smart asset.

Issuing transaction will also be invoked when it is deployed with h_b . For a pre-allocated 1 billion token, it can be:

$$\beta_{\mu_m} \leftarrow \lfloor 1000000000/n \rfloor, \quad h_b = 0 \quad (12)$$

Usually issuing transaction contains one unconditional operation and can be extended to multiple ones (e.g. given additional reward to certain users).

Payment transaction from μ_a to μ_b consists of a condition ρ and multiple operations:

$$\rho : (\kappa_v \leq \beta_{\mu_a}) \wedge (\kappa_e \leq \gamma_{\mu_a}) \quad (13)$$

$$\{\phi_i\} : \beta_{\mu_a} \leftarrow \beta_{\mu_a} - \kappa_v \quad (14)$$

$$\gamma_{\mu_a} \leftarrow \gamma_{\mu_a} - \kappa_e \quad (15)$$

$$\beta_{\mu_b} \leftarrow \beta_{\mu_b} + \kappa_v, \quad (16)$$

$$\gamma_{\mu_m} \leftarrow \gamma_{\mu_m} + \kappa_e \quad (17)$$

, in which κ_v is a positive big integer indicating the amount of the transfer and κ_e is a non-negative big integer denoting the transaction fee charged from the platform currency.

Non-Fungible Token

The state of a non-fungible token is an set, $\hat{\psi}_\mu := \{\tau_i\}$ indicates the ownership of each non-fungible token. Non-fungible tokens usually don't need any issuing transaction, which can not be mined.

Payment transaction from μ_a to μ_b consists of a condition ρ and multiple operations:

$$\rho : (\kappa_\tau \in \hat{\psi}_{\mu_a}) \wedge (\kappa_e \leq \gamma_{\mu_a}) \quad (18)$$

$$\{\phi_i\} : \hat{\psi}_{\mu_a} \leftarrow \hat{\psi}_{\mu_a} - \{\kappa_\tau\} \quad (19)$$

$$\gamma_{\mu_a} \leftarrow \gamma_{\mu_a} - \kappa_e \quad (20)$$

$$\hat{\psi}_{\mu_b} \leftarrow \hat{\psi}_{\mu_b} \cup \{\kappa_\tau\} \quad (21)$$

$$\gamma_{\mu_m} \leftarrow \gamma_{\mu_m} + \kappa_e \quad (22)$$

, in which κ_τ is the non-fungible token to be transferred.

A customized transaction can be defined and allow invocation only by a hard-coded issuer μ_u (e.g., the game developer [10]). The example issues and releases a non-fungible token κ_τ to a specific user κ_μ .

$$\rho : (\mu_a = \mu_u) \wedge (\kappa_e \leq \gamma_{\mu_u}) \quad (23)$$

$$\{\phi_i\} : \gamma_{\mu_u} \leftarrow \gamma_{\mu_u} - \kappa_e \quad (24)$$

$$\hat{\psi}_\mu \leftarrow \hat{\psi}_\mu \cup \{\kappa_\tau\} \quad (25)$$

$$\gamma_{\mu_m} \leftarrow \gamma_{\mu_m} + \kappa_e \quad (26)$$

FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds

Daehyeok Kim¹, Tianlong Yu¹, Hongqiang Harry Liu³, Yibo Zhu², Jitu Padhye²
Shachar Raindel², Chuanxiong Guo⁴, Vyas Sekar¹, Srinivasan Seshan¹

¹Carnegie Mellon University, ²Microsoft, ³Alibaba, ⁴Bytedance

Abstract

Many popular large-scale cloud applications are increasingly using containerization for high resource efficiency and lightweight isolation. In parallel, many data-intensive applications (*e.g.*, data analytics and deep learning frameworks) are adopting or looking to adopt RDMA for high networking performance. Industry trends suggest that these two approaches are on an inevitable collision course. In this paper, we present *FreeFlow*, a software-based RDMA virtualization framework designed for containerized clouds. *FreeFlow* realizes virtual RDMA networking purely with a software-based approach using commodity RDMA NICs. Unlike existing RDMA virtualization solutions, *FreeFlow* fully satisfies the requirements from cloud environments, such as isolation for multi-tenancy, portability for container migrations, and controllability for control and data plane policies. *FreeFlow* is also transparent to applications and provides networking performance close to bare-metal RDMA with low CPU overhead. In our evaluations with TensorFlow and Spark, *FreeFlow* provides almost the same application performance as bare-metal RDMA.

1 Introduction

Developers of large-scale cloud applications constantly seek better performance, lower management cost, and higher resource efficiency. This has lead to growing adoption of two technologies, namely, *Containerization* and *Remote Direct Memory Access (RDMA)* networking.

Containers [7, 11, 6] offer lightweight isolation and portability, which lowers the complexity (and hence cost) of deploying and managing cloud applications. Thus, containers are now the de facto way of managing and deploying large cloud applications.

RDMA networking offers significantly higher throughput, lower latency and lower CPU utilization than the standard TCP/IP based networking. Thus, many data-intensive applications, *e.g.*, deep learning and data analytics frameworks, are adopting RDMA [24, 5, 18, 17].

Unfortunately, the two trends are fundamentally at odds with each other in clouds. The core value of containerization is to provide an efficient and flexible management to applications. For this purpose, containerized clouds need containers to have three properties in networking:

- *Isolation*. Each container should have its dedicated network namespace (including port space, routing table, interfaces, etc.) to eliminate conflicts with other containers on the same host machine.
- *Portability*. A container should use virtual networks to communicate with other containers, and its virtual IP sticks with it regardless which host machine it is placed in or migrated to.
- *Controllability*. Orchestrators can easily enforce control plane policies (*e.g.*, admission control, routing) and data plane policies (*e.g.*, QoS, metering). This property is particularly required in (multi-tenant) cloud environments.

These properties are necessary for clouds to freely place and migrate containers and control the resources each container can use. To this end, in TCP/IP-based operations, networking is fully virtualized via a software (virtual) switch [15].

However, it is hard to fully virtualize RDMA-based networking. RDMA achieves high networking performance by offloading network processing to hardware NICs, bypassing kernel software stacks. It is difficult to modify the control plane states (*e.g.*, routes) in hardware in shared cloud environments, while it is also hard to control the data path since traffic directly goes between RAM and NIC via PCIe bus.

As a result, several data-intensive applications (*e.g.*, TensorFlow [24], CNTK [5], Spark [18], Hadoop [17]) that have adopted both these technologies, use RDMA only when running in dedicated bare-metal clusters; when they run in shared clouds, they have to fundamentally eschew the performance benefits afforded by RDMA. Naturally, using dedicated clusters to run an application is, however, not cost efficient both for providers or for customers.

Thus, our goal in this paper is simple: we want cloud-based, containerized applications to be able to use RDMA as

Property	Native	SR-IOV [21]	HyV [39]	SoftRoCE [36]
Isolation	✗	✓	✓	✓
Portability	✗	✗	✓	✓
Controllability	✗	✗	✗	✓
Performance	✓	✓	✓	✗

Table 1: RDMA networking solutions that can be potentially used for containers.

efficiently as they would in a dedicated bare-metal cluster; while at the same time achieving the isolation, portability and controllability requirements in containerized clouds.¹

Currently, there is no mature RDMA virtualization solutions for containers.² Table 1 summarizes some important options that can potentially be extended to support containers, although they fail to achieve the key requirements or have to do so at a substantial performance cost.

For instance, hardware-based I/O virtualization techniques like SR-IOV [21] have fundamental portability limitations [39, 28], since they require reconfiguration of hardware NICs and switches to support migrations of containers. Control path virtualization solutions, such as HyV [39], only manipulate the control plane commands for isolation and portability, and they do not have the visibility or control of the data traffic. Because of this, they cannot flexibly support data plane policies needed by cloud providers. Software-emulated RDMA, *e.g.*, SoftRoCE [36], can easily achieve isolation, portability, and controllability by running RDMA on top of the UDP networking stack and use existing virtual IP networking solutions, but its performance will be limited by UDP.

In this paper, we present *FreeFlow*, a software-based virtual RDMA networking framework for containerized clouds, which simultaneously achieves isolation, portability and controllability and offers performance close to bare-metal RDMA. At the heart of *FreeFlow* is a software virtual switch running on each server to virtualize RDMA on commodity RDMA NICs. *FreeFlow* does not require any specialized hardware or hardware-based I/O virtualization. The software virtual switch has the full access to both control path (*e.g.*, address, routing) and data path (*e.g.*, data traffic) of the communications among containers. This design philosophy is similar to existing software virtual switches used for TCP/IP networking in the containerized cloud, *e.g.*, Open vSwitch (OvS) [15] although *FreeFlow*'s actual design is dramatically different from OvS due to RDMA's characteristics.

The design of *FreeFlow* addresses two key challenges. First, we want *FreeFlow* to be completely transparent to the application. This is challenging because RDMA requires a NIC to manipulate memory buffers and file descriptors, while applications inside containers do not directly inter-

¹Indeed, our primary motivation to start this work is to enable a large-scale AI application at a leading cloud provider to be migrated from a dedicated cluster to clouds, and yet continue to use RDMA.

²There are some recent proposals from industry [35, 26] but these have limitations as we discuss in §9.

act with the NIC due to network virtualization. Our key insight to address this challenge is that containers are essentially processes, and they can easily share resources like memory and file descriptors with *FreeFlow*. If *FreeFlow* and a container share the same memory (§4.3) and file descriptor (§4.4), any operations on the underlying physical RDMA NIC will automatically take effect inside the container. A further problem is that sharing resources transparently to applications is not straightforward, given that applications do not cooperatively create resources that are shareable. We design methods to convert resource from non-shareable to shareable with no or minimal modifications on application code.

Second, *FreeFlow* must offer throughput and latency that is comparable to bare-metal RDMA. We identify the performance bottlenecks in throughput and latency as memory copy and inter-process communication respectively. We leverage a zero-copy design for throughput (§4.3), and a shared memory inter-process channel with CPU spinning for latency (§5.2). We also optimize *FreeFlow* for bounding CPU overhead.

We evaluate the performance of *FreeFlow* with standard microbenchmarking tools and real-world data-intensive applications, Spark and TensorFlow without any or with minimal modification on them. *FreeFlow* achieves the performance comparable to bare-metal RDMA without much CPU overhead. We also show that *FreeFlow* significantly boosts the performance of real-world applications by up to 14.6 times more in throughput and about 98% lower in latency over using conventional TCP/IP virtual networking. *FreeFlow* has drawn interests from multiple RDMA solution providers, and is open sourced at <https://github.com/Microsoft/Freeflow>.

2 Background

This section provides a brief background on container and RDMA networking, to motivate the need for software-based RDMA virtualization for containers.

Containers and container networking: Containers are becoming the de facto choice [30, 27, 25] to package and deploy data center applications. A container bundles an application's executables and dependencies in an independent namespace using mechanisms such as chroot [4]; thereby offering a lightweight isolation and portability solution.

Most containerized applications use microservices architecture, and are composed of multiple containers. For example, each mapper and reducer node in Spark [2] is an individual container; each parameter server node or worker node in TensorFlow [22] is also an individual container. The containers exchange data via a networking solution. The design of the networking solution affects the degree of isolation and portability.

For instance, in the *host mode* networking, containers use their host's IP and port space, and communicate like an ordinary process in the host OS. This mode has poor isolation

#Machines / #GPUs	Transport layer	Normalized speed
1 / 8	-	1.00×
2 / 16	TCP/IP (host)	0.45×
2 / 16	RDMA (host)	1.38×

Table 2: Speeds of a RNN job over TensorFlow on a single machine and multiple machines with TCP and RDMA networking. Speeds are normalized to the single machine case.

(e.g., port conflicts) and portability (e.g., must change IP addresses and ports after migrating to another host).

Thus, many applications use *virtual mode* networking. In this mode, the network namespaces of containers are fully isolated, and containers communicate via a virtual (overlay) network composed of software virtual switches on host machines. The virtual IPs of the containers are highly portable, given that the routes to the virtual IPs can be controlled in the software virtual switches. Since all data traffic must go through the virtual switches, they have access to the traffic, which provides the full controllability to the container networks. Such isolation and portability give orchestrators full flexibility in container placement and migrations, and such controllability offers cloud providers the power to enforce their policies on both control and data plane.

Indeed, orchestrators like Kubernetes [11] mandate the use of virtual networking mode [12]. A number of software solutions are available to provide virtual networking fabrics for containers, such as Weave [23], and Docker Overlay [7].

RDMA networking: Many modern applications (e.g., deep learning and data analytics frameworks) have adopted RDMA networking [18, 17, 22, 5] to get higher throughput, and lower latency than the traditional TCP/IP stack. RDMA offers these gains by offloading most of the networking functionality to the NIC, effectively bypassing the OS kernel.

Table 2 shows measured performance improvements of using RDMA for a deep learning application – training a Recurrent Neural Network (RNN) speech recognition model. The application was first benchmarked on a single machine with 8 GPUs. When the application run on two machines with 16 GPUs, traditional TCP/IP networking becomes a bottleneck, and the performance degrades. With RDMA, however, the extra GPUs offer performance gains.

The reason is that this RNN training task consists of thousands of steps. In each step, all GPUs must shuffle the training model parameters, and the total traffic volume ranges from 100 MB to 10 GB. The time spent on communication is essentially wasting GPU’s time, since GPUs are idle during shuffling. TCP performs badly in these frequent and bursty workloads, while RDMA can instantaneously climb to full bandwidth at the beginning of each shuffle.

Need for software-based RDMA virtualization: We have noted the benefits of *virtual mode networking* for containerized applications – namely, enhanced isolation, portability, and controllability. We have also noted that RDMA can of-

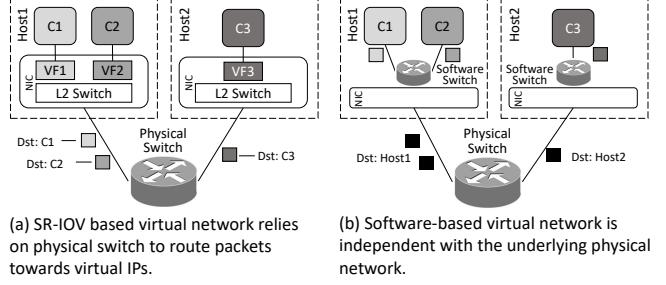


Figure 1: Comparison between hardware-based (SR-IOV) and software-based virtual networking solutions.

fer significant performance boost to many applications that have a microservice architecture.

The question then is, how do we use RDMA networking with containerized applications that require virtual mode networking, especially in a cloud environment.

RDMA networking, as we saw earlier, relies on offloading most of the networking functionality to a NIC. One possible approach to “virtualize” RDMA networking is to use hardware-based solutions such as SR-IOV [21]. However, this would limit the portability offered by the virtual mode networking. As an example shown in Figure 1(a), with SR-IOV, the NIC runs a simple layer-2 switch that merely performs VLAN forwarding. Hence, all packets generated from and destined to a virtual network have to be directly routed in the underlying physical network. Thus, migrating container C1 to Host2 requires reconfiguring the physical switch to route C1’s packets to Host2 rather than Host1. Also, in production, physical switches need to maintain a huge size of routing table to manage routes for all containers in virtual networks, which can be infeasible in a large-scale cloud environment.

Thus, we believe that the right approach to virtualizing RDMA network for containers is to use a software switch – just like it is done for virtualizing traditional TCP/IP networking. As shown in Figure 1(b), the physical network is only in charge of delivering packets targeting on different hosts, and virtual networking routing is completely realized in software switches inside each host, which is independent with the physical network. The software switch can control all addressing and routing, thereby providing good isolation and portability for control plane. It can also be used to implement network functions on data plane such as QoS and metering.

3 Overview

The goal of *FreeFlow* is to provide an virtual interface inside each container, and applications can use RDMA via a virtual network on top of the virtual interface in an unmodified way. Ideally, the performance of the virtual network should be close to bare-metal RDMA, and policies on both control and data path are flexible to be configured purely in software. In this section, we present the system architecture and key challenges in the design of *FreeFlow*.

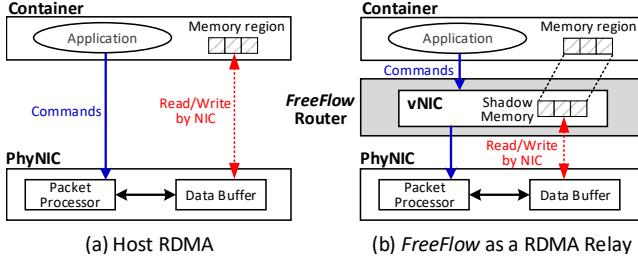


Figure 2: Design overview: *FreeFlow* router directly accesses NIC(s) and serves as a RDMA relay for containers. Blue and red lines are control and data path, respectively.

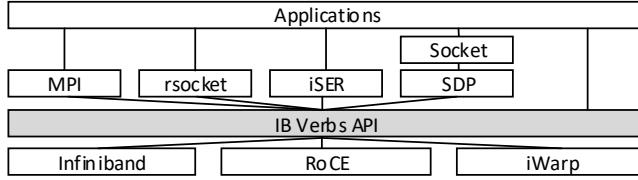


Figure 3: IB Verbs is the de facto “narrow waist” of various RDMA-based network offloading solutions.

3.1 Overall Design

In native RDMA, as shown in Figure 2(a), applications leverage RDMA APIs to directly send commands to the hardware NICs for both control and data path functions. *FreeFlow* intercepts the communication between applications and physical NICs, and performs control plane and data plane policies inside the software *FreeFlow* router which runs as another container on the host machine. In particular, for controlling the data path, *FreeFlow* router only allows the physical NIC to directly read and write from its own memory (the shadow memory in Figure 2(b)) and take the charge of copying data from and to the applications’ memory. Note that the memory inside container and the shadow memory in the *FreeFlow* router can be the same piece of physical memory for zero-copy (§4.3).

3.2 Verbs: the “narrow waist” for RDMA

There are multiple ways to intercept the communications between applications and physical NICs, but we must choose an efficient one. A number of commercial technologies supporting RDMA are available today, including Infiniband [9], RoCE [8] and iWarp [19]. Applications may also use several different high-level APIs to access RDMA features, such as MPI and rsocket [20]. As shown in Figure 3, the de facto “narrow waist” of these APIs is the IB Verbs API (Verbs). Thus, we consciously choose to support Verbs in *FreeFlow* and by doing so we can naturally support all higher-level APIs.

Verbs uses a concept of “queue pairs” (QP) for data transfer. For every connection, each of two endpoints has a send queue (SQ) and a receive queue (RQ), together called QP. The send queue holds information about memory buffers to be sent, while the receive queue holds information about which buffers to receive the incoming data. Each endpoint

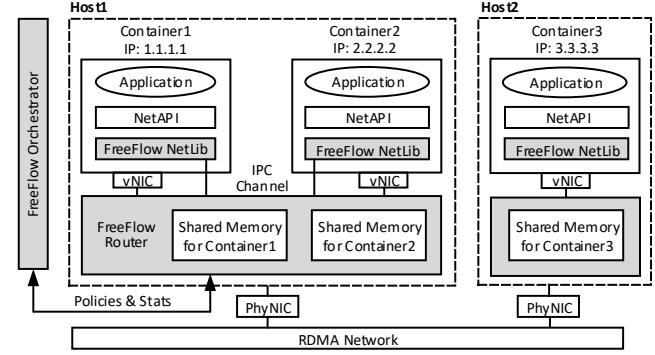


Figure 4: *FreeFlow* architecture.

also has a separate completion queue (CQ) that is used by the NIC to notify the endpoint about completion of send or receive requests. The Verbs library and associated drivers allow applications to read, write and monitor the three queues. Actual transfer of the data, including packetization and error recovery, is handled by the NIC.

To transparently support Verbs, *FreeFlow* creates virtual QPs and CQs in virtual NICs and relates the operations on them with operations on real QPs and CQs in the physical NICs.

3.3 FreeFlow Architecture

The architecture of *FreeFlow* is shown in Figure 4. The three components of container networking stack that we modify or introduce are shown in gray: (i) the *FreeFlow* network library (*FFL*), (ii) the *FreeFlow* software router (*FFR*), and (iii) the *FreeFlow* network orchestrator (*FFO*).

FFL, located inside the container, is the key to making *FreeFlow* transparent to applications. From application’s perspective, it is indistinguishable from the standard RDMA Verbs library [16]. All applications and middleware built atop the Verbs API can run with no (or negligible) modification. *FFL* coordinates with *FFR*.

FFR runs a single instance on each host and works with all containers on the same host to provide virtual networking. In the data plane, *FFR* shares memory buffers with containers on the same host and isolates the shared memory buffers for different containers. *FFR* sends and receives data in the shared memory through the NIC, relying on *FFL* to sync data between application’s private data buffers and the shared memory buffers. *FFR* implements the data-plane resource policies, e.g., QoS, by controlling the shared-memory channel between containers and *FFR*. It also works with *FFO* to handle bookkeeping tasks such as IP address assignment.

FFO makes control-plane decisions for all containers in its cluster based on user-defined configurations and real-time monitoring of the cluster. It also maintains centralized memory maps, as we shall discuss in §4.3.

3.4 Challenges

In designing *FreeFlow*, we need to address two key challenges. First, *FreeFlow* should provide an RDMA interface

which *transparently* supports all types of existing RDMA operations. There are various types of RDMA operations including one- and two-sided operations for data transfer, poll-and event-based mechanisms for work completion notification, and TCP/IP and RDMA-CM for the connection establishment. We observe that it is not straightforward to support them transparently due to the complexity of RDMA operations. Second, *FreeFlow* should provide near bare-metal RDMA performance while minimizing CPU and memory overhead. Since *FFR* intercepts the Verbs calls from applications via *FFL*, we need to carefully design the communication channel between *FFR* and *FFL*.

We will present our approach for each challenge in §4 and §5, respectively.

4 Transparent Support for RDMA Operations

Verbs supports multiple types of operations and mechanisms. With one-sided operations such as WRITE and READ, a writer (reader) can write (read) data to (from) a specific memory address in the remote side, without the latter aware of this operation. With two-sided operations such as SEND and RECV, the receiver must first get ready to receive before a sender sends out the data. Also, applications can use either poll-based or event-based mechanisms to get work completion notifications. Different applications use different operation types as their needs, and we see all of them used in popular applications [32, 18, 17, 22].

FreeFlow completely and transparently supports such different types of RDMA operations. The primary challenge is to support one-sided operations and event-based completion notifications, in which RDMA NIC can modify memory or file descriptors in *FFR* silently. *FFR* cannot know about the modifications immediately unless it keeps busily polling the status of the memory or file descriptor, so that it is hard to convert the operations from physical NICs to virtual NICs inside containers as soon as possible. We solve this challenge taking advantage of the fact that containers are essentially processes, so that *FFL* and *FFR* can share memory and file descriptors, and physical NIC's modifications can automatically be passed into containers. Sharing memory between *FFL* and *FFR* is also not straightforward for application transparency, because applications inside containers do not allocate memory in IPC shared memory space, and we need to convert the memory to shared memory transparently.

4.1 Connection Establishment

Two RDMA communication endpoints need to first establish a connection. They create a QP in each one's NIC, registering a buffer of memory to the QP and pairing local QP with remote QP. After a connection is established, the application can ask the NIC to send the content in the registered memory to the remote end or put received data into the local buffer.

Steps 1–7 in Figure 5 show the typical process of connection establishment using Verbs. The left column shows the sequence of Verbs calls made by the application. The two

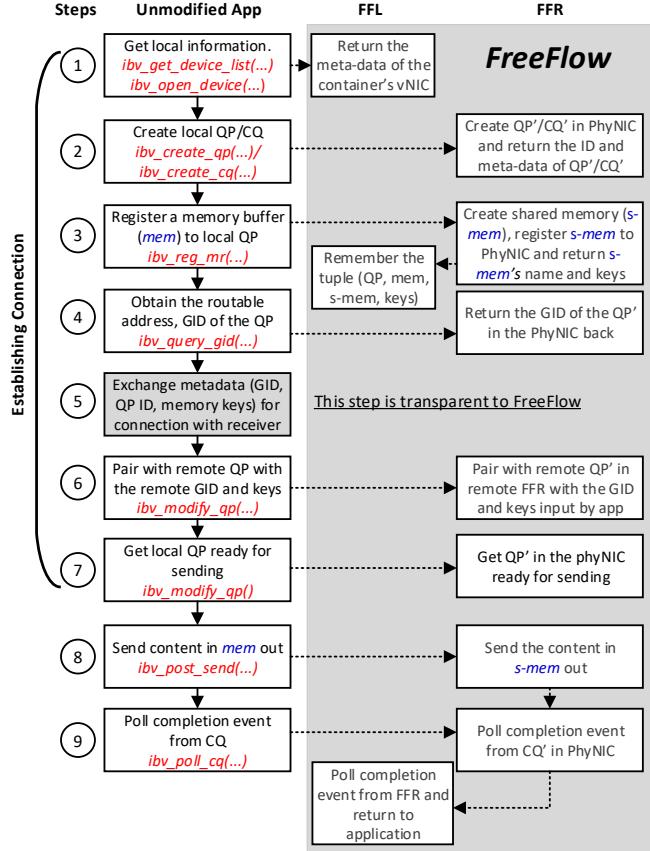


Figure 5: The workflow of a RDMA SEND operation.

columns in the blue/shaded area shows how *FreeFlow* traps the Verbs calls from the application, and to establish a connection between the sender's *FFR* and the receiver's *FFR*.

Step 1: The application queries for the list of NICs whose drivers support Verbs. *FFL* intercepts the call and returns the context data object of the virtual NIC of the container.

Step 2: The application creates a QP and a CQ on its virtual NIC, while *FFR* creates the corresponding queues (QP' and CQ') on the physical NIC. The QP-IDs and other metadata information of the queues will be forwarded to the application by *FFR* after *FFR* finishes the creations of the queues.

Step 3: The application registers a block of memory (*mem*) to the QP. *FFR* allocates a corresponding block memory (*s-mem*) in its shared memory inter-process communication (IPC) space with the same size as *mem*, registers *s-mem* to QP'. *FFR* returns the ID (a host-wide unique name of the IPC memory) it used to create *s-mem*. With this ID, *FFL* can map *s-mem* into its own virtual memory space.

Step 4: The application queries the address (so-called GID in RDMA) of the local QP. This address information will be shared with the other side for pairing the local QP and remote QP together. At the end of this step, *FFR* returns the *actual* GID of QP'.

Step 5: The application exchanges GID and QP-ID with the remote end. Applications can exchange this information via any channels such as TCP/IP or RDMA-CM.³

Step 6: The application pairs its local QP with the remote container's QP using the receiver's GID. *FFL* forwards this GID to *FFR*. *FFR* pairs QP' with this GID.

Step 7: The application modifies the state of local QP to Ready to Send/Receive state, while *FFR* modifies the state of QP' accordingly.

After Step 7, from the application's point of view, it is ready to send or receive data – it has created a QP and a CQ, registered *mem* to the QP, paired with the remote QP and established a connection with the remote QP.

From *FreeFlow*'s point of view, it has created QP' and CQ' which are associated with the QP and CQ in the application, registered *s-mem* as the shadow memory of *mem*, and paired with the QP' in the remote *FFR*. It is also ready to get and forward Verbs calls from the application.

FreeFlow may increase the latency for connection establishment due to the additional interactions between *FFR* and *FFL*. However, it does not much affect the overall latency of *FreeFlow* since it is a one-time cost; many RDMA applications re-use pre-established connections for communications.

4.2 Two-sided Operations

Each sender or receiver needs to go through two steps to perform a data transfer. The first step is to use QP to start sending or receiving data, and the second step is to use CQ to get completion notifications. Steps 8–9 in Figure 5 shows this process.

Step 8: The application invokes the SEND call, and supplies pointer to *mem*. *FFL* first copies data from *mem* to *s-mem*, and *FFR* then invokes its own SEND call to send *s-mem* to the remote *FFR*. We avoid the memory copies from *mem* and *s-mem* by applying our zero-copying mechanism described in §4.3. Note that the remote router would have posted a corresponding RECV call by this time.

Step 9: The application either polls the CQ or waits for a notification that indicates the completion of the send. *FFR* also polls/waits-on CQ' associated with QP' and forwards it to *FFL*.

For subsequent SEND operations on the same QP, the application only needs to invoke Step 8 and 9 repeatedly. The workflow of a RECV operation is similar, except that at Step 9, *FFL* will copy data from *s-mem* to *mem* after the QP' finishes receiving data, which is the opposite of Step 8 in SEND operation.

The presence of *FFL* and *FFR* is completely transparent to the application. To the application, it appears that it is performing normal verbs operations on its vNIC. The steps in Figure 5 are standard way of writing Verbs programs. The

FreeFlow behavior illustrated here is sufficient to fully support SEND and RECV operations.

4.3 One-sided Operations

In one-sided operations, a client needs not only the GID of a server, but also the address of the remote memory buffer, and the security key for accessing the memory. This information is exchanged in Step 5 in Figure 5 and becomes available to *FreeFlow* in Step 8 (where WRITE or READ can be called).

Compared to two-sided operations, it is more challenging to transparently support one-sided operations. There are two problems to support one-sided operations in *FreeFlow*.

First, the target memory address *mem* is in the virtual memory of the remote container. However, the local *FFR* does not know the corresponding *s-mem* on the other side. For example, in Figure 6(a), when the sender tries to write data in *mem-1* to remote memory *mem-2*, it fails at stage 3) because the target memory address *mem-2* is not accessible for *FFR* on the receiver side.

To solve this problem, *FreeFlow* builds a central key-value store in *FFO* for all *FFRs* to learn the mapping between *mem*'s pointer in application's virtual memory space and the corresponding *s-mem*'s pointer in *FFR*'s virtual memory space. Updating this table adds latency to Step 3 in Figure 5, when applications register memory to their virtual NIC. However, data plane performance is not impacted because *FFR* can cache the mappings locally.

Second, even if we know the memory mapping on the remote side, WRITE and READ can remotely modify or copy data without notifying the remote side's CPU, so that *FFR* does not know when to copy to or from application's memory. For instance, in Figure 6(b), the sender finds the correct address of *s-mem-2* and send the data to it. However, after the data is available in *s-mem-2*, there is no notification for the *FFR* in the receiver side to know when to copy *s-mem-2* to *mem-2*. One way to solve this is to continuously synchronize *s-mem-2* and *mem-2*. This would consume a lot of CPU and memory bus bandwidth.

To address this, in *FreeFlow*, we design a *zero-copy based mechanism* to efficiently support one-side operations. The high-level idea is to make *mem* and *s-mem* the same physical memory, so that *FFR* does not need to do any copy, and the data will be naturally presented to the application. Figure 6(c) illustrates this design. By getting rid of memory copies, we can also improve *FreeFlow* performance.

The key here is to make applications directly allocate and use shared memory with *FFR* for data transfers. For this, *FreeFlow* provides two options:

Option 1—Allocating shared buffers with new APIs: We create two new Verbs functions, `ibv_malloc` and `ibv_free`, to let applications delegate the memory creation and deletion to *FreeFlow*. This allows *FFL* to directly allocate these buffers in the shared memory region (shared with *FFR*), and thus avoid the copy. The drawback of this option is the need

³*FreeFlow* also has an extension to support RDMA-CM with similar a design to support IB Verbs, while we omit the details due to space limit.

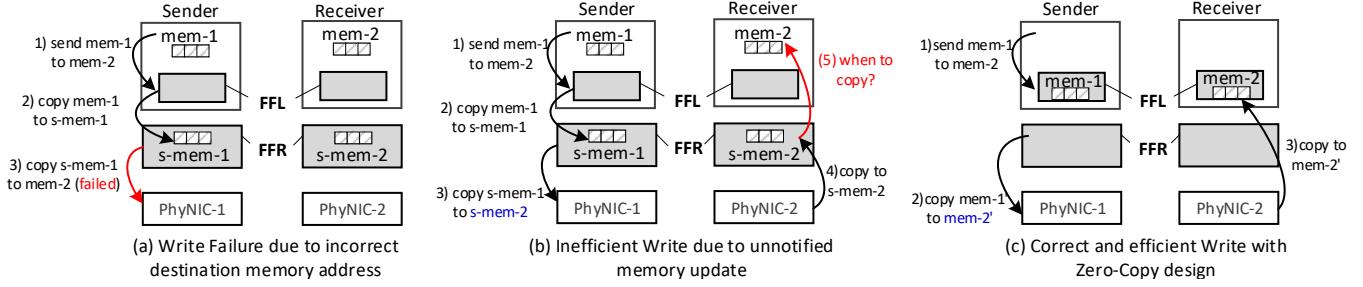


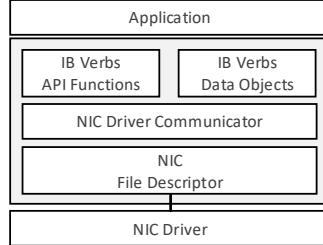
Figure 6: Zero-copy design enables FreeFlow to address the challenges to support one-sided operations efficiently.

```
// From <infiniband/verbs.h>
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr,
                  struct ibv_send_wr **bad_wr);

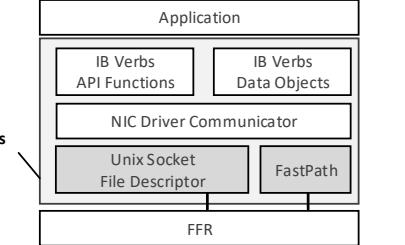
struct ibv_qp {
    struct ibv_context    *context;
    void                 *qp_context;
    struct ibv_pd         *pd;
    struct ibv_cq         *send_cq;
    struct ibv_cq         *recv_cq;
    struct ibv_srq         *srq;
    uint32_t              handle;
    uint32_t              qp_num;
};

};

(a) A typical function and data structure definition in IB Verbs
```



(b) The structure of Verbs Library



(c) FreeFlow ‘hijacks’ the communication between Verbs and NIC driver

Figure 7: The structure of Verbs API and library. *FFR* intercepts the calls between Verbs library and NIC drivers.

to modify application code, despite the modification should be only several lines on the data buffer creation.

Option 2—Re-mapping applications’ virtual memory address to shared memory: When an application registers a private memory piece with virtual memory address va as a data buffer (*e.g.*, Step 3 in Figure 5), *FFL* releases the physical memory piece behind va and assign a shared physical memory piece from *FFR* to va . In Linux, this operation is only valid when va is an address at the start of a memory page. To force the application to allocate memory always at the start of a page, *FFL* intercepts the calls like `malloc` in C language and makes it always return page aligned memory addresses. While this option can achieve zero memory copy without modifying application code, it forces all memory allocations in the application to be page aligned, which can result in lower memory efficiency on the host.

In practice, we recommend the first option since it is cleaner and efficient. However, since many RDMA applications already make their data buffer page aligned for better performance (*e.g.*, RDMA-Spark [18]), we can directly use the Option-2 without intercepting `malloc`, so the side-effect is limited. Note that if a developer chooses to modify an application using the option 1 or an application originally supports page-aligned buffers, in either case, *FreeFlow* will not incur any overhead in actual memory usage.

4.4 Event-based Operations

There are two options to get notified from CQs (Completion Queue). The first option is to let application poll the CQs periodically to check whether there are any completed operations. The second option is event-based, which means the

application creates an event channel and add CQs into the channel. The channel contains a file descriptor which can trigger events when operations are completed.

In *FreeFlow*, since the raw file descriptor is created from physical NIC, *FFR* needs to pass the file descriptor to *FFL*, otherwise the latter cannot detect any events associated with the file descriptor. We take advantage of the fact that *FFL* and *FFR* are essentially two processes sharing the same OS kernel, and leverage the same methodology to pass file descriptors between processes [41] to pass event channels from *FFR* to *FFL*.

5 Communication Channel between *FFL* and *FFR*

Since *FreeFlow* intercepts every Verbs calls via *FFL*, translates, and forwards them to physical NICs via *FFR*, it is crucial to have an efficient channel between *FFL* and *FFR* that provides high RDMA performance while minimizing system resource consumption. In this section, we present two designs of such communication channels, which allows trade RDMA performance for resource consumption and vice versa depending on the requirements of applications.

5.1 Verbs Forwarding via File Descriptor

A straightforward way to pass Verbs calls between *FFL* and *FFR* is to use RPC: *FFL* passes API name and parameters to *FFR*, and *FFR* modifies the parameters properly, executes the API and returns the result of the API call back to *FFL*. Nevertheless, this simple RPC approach does not work well in *FreeFlow* because of the complexity of input data structures of the Verbs calls. As shown in Figure 7(a), a typical function call in Verbs, *e.g.*, `ibv_post_send`, has inputs (`qp`,

wr) and outputs *bad_wr* that are pointers to complex data structures. Since *FFL* and *FFR* are in two different processes, the pointers of *FFL* will be invalid in *FFR*.

One may advocate “deep copy” which traces down the complex input/output data structures and transfer the data objects under all pointers between *FFL* and *FFR*. However, this approach has two severe drawbacks. First, data structures in Verbs are quite deep (*i.e.*, multiple levels of pointers and nesting) and such deep copies can hurt the performance. Second, there are customized data structures that are defined by user code whose deep copy methods cannot be predefined by *FreeFlow*.

To address this issue, we take advantage of the structure of the current Verbs library. As shown in Figure 7(b), the Verbs library consists of three layers. The top layer is the most complicated one and hard to be handled as described above. However, when it comes down to the middle layer that communicates with the NIC file descriptor, Verbs library must prepare a simple enough (no pointers) data structure that the NIC hardware can digest.

Therefore, instead of forwarding the original function calls of Verbs, we forward the requests to be made for the NIC file descriptor. We replace the NIC file descriptor in the container with a Unix socket file descriptor whose the other end is *FFR*, as shown in Figure 7(c). By doing this, *FFR* can learn the command sent by the application and the supplied parameters. *FFR* will map the operations to virtual queues in the container to the same operations to the actual queues in the physical NIC. It then converts the replies from the physical NIC to replies from the virtual NIC for the virtual queues, and returns the new reply to *FFL* via the Unix socket. The NIC driver communication layer in *FFL* will process the reply normally without knowing about the operations behind the Unix socket file descriptor.

While this Unix socket based approach consumes little CPU, it can incur additional latency due to the inherent delay from communicating via the socket. Our measurement shows that the round trip time over Unix socket (and shared-memory with semaphore) can easily be $\geq 5 \mu\text{s}$ in a commodity server. Because of this, the Unix socket communication channel in Figure 7(c) can become a performance bottleneck for latency sensitive applications that expects ultra low latency (*e.g.*, $< 5 \mu\text{s}$).

For applications requiring low latency communication, we will describe the design of *Fastpath*, which optimizes the communication delay by trading CPU resources, in the next section.

5.2 Fastpath between *FFL* and *FFR*

To accelerate the communication between *FFR* and *FFL*, we design a *Fastpath* in parallel with the Unix socket based channel between them. As shown in Figure 8, *FFL* and *FFR* co-own a dedicated piece of shared memory. With Fastpath, *FFR* spins on a CPU core and keeps checking whether there is a new request from *FFL* got written into the shared mem-

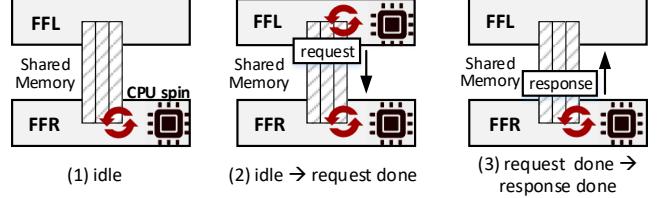


Figure 8: Fastpath channel between *FFR* and *FFL*.

ory piece. Once a request is detected, *FFR* will immediately execute it, while *FFL* starts to spin on a CPU core to check whether the response is ready. After reading the response, *FFL* will stop the CPU spinning on its side.

As we will see in § 8.1.2, Fastpath can significantly reduce the latency. However, the price is the CPU cycles spent on spinning for reading requests and responses. To limit the CPU overhead brought by Fastpath, we make two design decisions: (1) *FFR* only spins on one CPU core for all Fastpath channels with *FFL* on the same host; (2) Fastpath is only used for functions which are on data path and are non-blocking, so that the CPU spinning time on *FFL* to wait for a response will be short (few microseconds). Overall, Fastpath only consumes one CPU core per host on average to significantly shorten the latency of message passing (§8.1.2). In addition, if *FFO* knows there is no latency sensitive application on a host machine (according to running container images), it can disable Fastpath and the CPU spinning.

6 Implementation

We implement *FFL* by modifying `libibverbs` (v1.2.1), `libmlx4` (v1.2.1) and `librdmacm` (v1.1.0).⁴ We add about 4000 lines of C code to implement *FreeFlow*’s logic. We have implemented *FFR* from scratch in about 2000 lines of C++ code. For *FFO*, we use ZooKeeper to store the user defined information; *e.g.*, IP assignment, access control, resource sharing policies, and memory mapping information for one-sided operations. Due to space limits, we only show three representative implementation details next.

Control & data plane policies: Since *FreeFlow* can control both control and data plane operations requested by containers, it can support common control and data plane policies including bandwidth enforcement, flow prioritization, and resource usage enforcement.

As an example of control plane policy, in our prototype, *FreeFlow* enforces a quota for the number of QPs each container can create, since large number of QPs is a major reason of the performance degradation of RDMA NICs [32]. This control plane policy prevents a container from creating too many QPs which can impact other containers on the same host machine.

Also, as an example of data plane policy, *FreeFlow* enables per-flow rate limiting with little overhead. We imple-

⁴`libibverbs` and `librdmacm` are libraries that allow userspace processes to use InfiniBand/RDMA Verbs and RDMA communication manager interfaces, respectively. `libmlx4` is a userspace driver for `libibverbs` that allows userspace processes to use Mellanox hardware.

ment a simple token-bucket data structure in *FFR*. When an application creates a new QP, we check the policies that are stored in *FFO*, and associate a token-bucket with pre-set rate limit to the QP. Upon every application’s send request, the router checks whether the QP has enough tokens to send out the requested message size. If so, the send request is forwarded to the real NIC immediately. Otherwise, *FFR* will notify *FFL* and delay it until there are enough tokens. Note that it is only an example of implementing QoS policies. *FreeFlow* provides flexible APIs for implementing sophisticated QoS algorithms in *FFR*, while we omit the details due to space limit.

Memory management in Fastpath: In Fastpath implementation, we use assembly codes to explicitly force the cache lines of requests and responses written by *FFL* and *FFR* to be flushed into main memory immediately. This is necessary because otherwise, the CPU will keep the newly written lines in cache for a while to wait more written lines, slowing down the message exchanging speed on Fastpath.

Supporting parallelism: Since applications can create multiple QPs and use multiple threads to transfer data in parallel, each Unix domain socket between the *FFL* and *FFR* needs a lock. To improve performance, we create multiple Unix domain sockets between the *FFL* and *FFR*. We avoid “head of the line blocking” by dedicating more of these sockets to data plane operations and event notifications and only a few of sockets to creation, setups and delete operations. On *FFR*, we use a dedicated thread for each incoming Unix domain socket connection. We also create a dedicated data structures for each container and a dedicated shared memory region for each registered memory buffer to keep the data path lock free.

7 Discussion

In this section, we discuss about some primary concerns and potential extensions in the current design of *FreeFlow*.

CPU overhead: Similar to software-based TCP/IP virtual networking solutions, *FreeFlow* incurs CPU overhead. In particular, *FreeFlow* uses a CPU core for polling control messages between *FFL* and *FFR* to support low latency IPC channel (§5.2). We admit that this is a cost for network virtualization on top of current commodity hardwares. One possible approach to address this is to utilize hardwares that support offloading CPU tasks, such as FPGA, ARM co-processor, or RDMA NICs [1]. We leave it as a future work to eliminate the CPU overhead in Fastpath.

Security: One concern is that since *FFR* shares its memory with containers, whether one container can read the communications of other containers on the same host by scanning the IPC space. This is not a concern for *FreeFlow* because *FFR* creates a dedicated shared memory buffer for each individual QP. Only those shared memory buffers that belong to a container will be mapped into the container’s virtual memory space. Another concern is the security of the memory keys. If one can see the keys by wiretapping, subsequent

communications can be compromised. This problem is inherent in the way one-sided operations in raw RDMA work, and is not made worse by *FreeFlow*.

Working with external legacy peers: Containers in *FreeFlow* can naturally communicate with external RDMA peers, since each *FFR* works independently. *FFR* does not distinguish whether the remote peer is another *FFR* or an external RDMA peer.

Container migration: *FreeFlow* supports offline migrations naturally. If a container is captured, shutdown, moved and rebooted in another host machine, its IP address is not changed, so that its peers re-establish RDMA connections with it as if it is just got rebooted. Nowadays, offline migrations are commonly used in container clusters for resource packing or fail-over. *FreeFlow* does not support live migration, since RDMA has poor mobility nowadays [39].

VM host: Our prototype (and evaluation) is based on containers running on bare-metal host machines. But *FreeFlow* can be directly used on containers deployed inside VMs if the VMs use SR-IOV to access the physical NIC.

Congestion control: RDMA NICs already have congestion control mechanisms, and *FreeFlow* relies on them.

8 Evaluation

We evaluate the performance and overhead of *FreeFlow*. We start from microbenchmarks (§8.1) and then the performance of real-world applications on *FreeFlow* (§8.2).

8.1 Microbenchmarks

Setup: We run microbenchmarks on two testbeds. One testbed runs InfiniBand, which is a traditional RDMA-dedicated fabric. The servers are equipped with two Intel Xeon E5-2620 2.10GHz 8-core CPU, 64GB RAM, and 56Gbps Mellanox FDR CX3 NIC. The OS is Ubuntu 14.04 with the kernel version 3.13.0-129-generic.

The other testbed runs RoCE (RDMA over Converged Ethernet). As the name indicates, RoCE only requires conventional Ethernet switches (in our case, Arista 7050QX as the ToR switch). The servers in this testbed cluster have Intel Xeon E5-2609 2.40GHz 4-core CPU, 64GB RAM, 40Gbps Mellanox CX3 NIC and Ubuntu 14.04 with the kernel version 4.4.0-31-generic.

We run containers using Docker (v1.13.0) [7] and set up a basic TCP/IP virtual network using Weave (v1.8.0) [23] with Open vSwitch kernel module enabled. Unless otherwise specified, we run Fastpath (§5.2) enabled *FreeFlow*.

We mainly compare *FreeFlow* with bare-metal RDMA, which is a stand-in for the “optimal” performance. We will show that *FreeFlow* enables virtual RDMA networking for containers with minimal performance penalty. In §8.1.4, we will also demonstrate the performance of translating TCP socket calls into RDMA on top of *FreeFlow*, so that conventional TCP applications can also benefit from *FreeFlow*. There we also compare *FreeFlow* with bare-metal TCP and Weave which supports virtual TCP/IP virtual networks for containers.

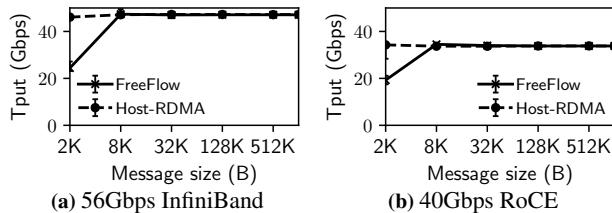


Figure 9: RDMA SEND throughput between a pair of containers on different hosts. *FreeFlow* enables container virtual networks with minimal performance penalty.

8.1.1 Throughput and Latency

We focus on two basic performance metrics, throughput and latency. We use the benchmark tools provided by Mellanox perftest [13]: `ib_send_lat` and `ib_send_bw` to measure latency and throughput of two-sided operation (SEND), `ib_write_lat` and `ib_write_bw` for one-sided operation (WRITE). These tools can run on *FreeFlow* without any modification, as explained in §4.3. In general, *FreeFlow* does not differentiate the inter-host setting (sender and receiver run on different hosts) and the intra-host setting. Here we just show inter-host performance values.

Throughput: We measure the single thread RDMA SEND/WRITE throughput on two testbeds, and show the RDMA SEND results in Figure 9. Each run transmits 1GB data with different sizes of messages ranging from 2KB to 1MB. *FreeFlow* RDMA WRITE results are in fact slightly better than SEND, and omitted for brevity. We see that with message size equal or larger than 8KB, *FreeFlow* gets full throughput as bare-metal RDMA (46.9Gbps on InfiniBand and 34.5Gbps on RoCE). In addition, when we increase the number of concurrent container pairs (flows) to up to 512, the aggregated throughput of all flows is still close to optimal (Figure 11). We also verify that the bandwidth is fairly distributed among different flows by calculating Jain’s fairness index [31] (0.97 on average).

In general, the bandwidth-hungry applications tend to use larger message sizes than a few KB. For example, in one of our internal storage clusters that uses RDMA, typical message sizes are 1MB or more. *FreeFlow* will have no throughput penalty in this case (see §8.1.2 for CPU overhead).

Even when the message sizes are small, like 2KB, *FreeFlow* still achieves more than half of the full throughput. We verified that, in this case, the throughput is bounded by the single *FFR* Fastpath thread (§5.2). This bottleneck can be easily removed by assigning one more CPU core to the *FFR* and balancing RDMA request loads across the two cores. While we leave this option open, developers usually do not expect to saturate the full bandwidth with small messages. Instead, for small messages, developers usually care about latencies.

Latency: We measure the latency of sending a 64B, 256B, 1KB, and 4KB message, respectively. Like the throughput benchmark, the two containers run on different hosts con-

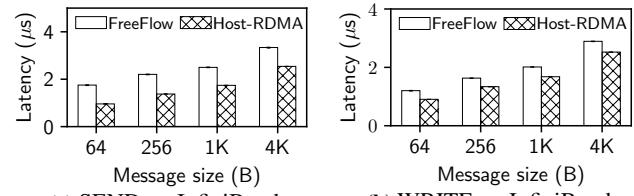
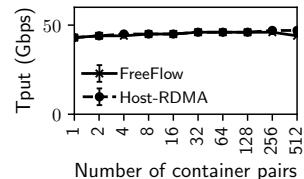


Figure 10: RDMA latency between a pair of containers on different hosts. SEND is a typical two-sided operation, while WRITE is one-sided.



	Host RDMA	Fastpath	LowCPU
1.8μs		2.4μs	17.0μs

Table 3: 2-byte message latency of two *FreeFlow* modes.

Figure 11: Aggregate throughput when scaling up the number of container pairs.

nected via the same ToR switch. For each message size, we measure the latency 1000 times. We plot the median, 10- and 99th-percentile latency values.

Figure 10 shows the one-way latency reported by the perftest tools. We can see that one-sided WRITE operation have lower latency than two-sided SEND operation, and also smaller gap between *FreeFlow* and bare-metal RDMA. However, even with the two-sided operation, *FreeFlow* causes less than $1.5\ \mu s$ extra delay. The extra delay is mainly due to the IPC between the *FFL* and *FFR*. One-sided operation will trigger IPC only one time, while two-sided operations will trigger two times and one time memory copy. This explains the larger latency gap of two-sided operations.

To put these latency values into perspective, one hop in network, i.e., a hardware switch, has $0.55\ \mu s$ latency [3]. Thus, *FreeFlow* latency overhead is comparable to an extra switch hop in the network. In comparison, host TCP stack latency is at least $10\ \mu s$ (§8.1.4) and then TCP/IP virtual network latency is even larger (more than $40\ \mu s$ in our test). This means *FreeFlow* preserves the latency advantage of RDMA while enabling virtual network for containers.

8.1.2 CPU Overhead and Trade-off

FreeFlow achieves good performance with low CPU overhead. *FreeFlow* has two modes: Fastpath and non-Fastpath (or LowCPU, in §5.1). By default, Fastpath is enabled and provides the best performance in terms of latency. In this mode, *FFR* spins on one CPU core and serves Verbs requests as soon as possible. One CPU core is capable of serving all the containers on one host, thanks to the fact that *FFR* only handles message-level events, instead of at packet-level like in Open vSwitch. On a commodity server with many CPU cores, this is acceptable.

In addition, users may choose the LowCPU mode, which uses a Unix socket as the signal mechanism instead of core

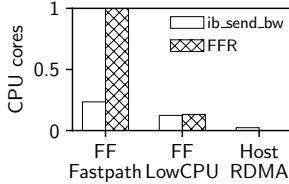


Figure 12: CPU usage of `ib_send_bw` and *FFR* when measuring the throughput with 1MB messages. 100% CPU means one fully utilized CPU core.

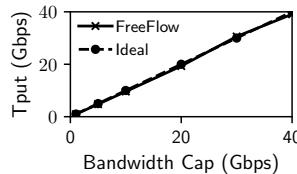


Figure 13: *FreeFlow* can accurately control the rate of traffic flows from containers.

spinning. This hurts latency performance (increase from $2.4\mu\text{s}$ to $17.0\mu\text{s}$), as shown in Table 3. In Figure 12, we record the per-process CPU utilization when measuring inter-host throughput. The throughput of all three cases in the figure are the same (full bandwidth). It shows the CPU benefit of LowCPU mode, especially on the *FFR*. In LowCPU mode, *FFR* CPU overhead scales with the actual load.

We recommend choosing the mode according to the workload requirement. Latency-sensitive or non-CPU heavy (*e.g.*, GPU-heavy) applications should be run with Fastpath mode while the rest can be run with LowCPU mode. However, even with Fastpath, *FFR* consumes at most one CPU core, and the extra overhead due to *FFL* is less than 30% for full bandwidth throughput.

8.1.3 Rate Limiter and Performance Isolation

We demonstrate the performance of rate limiter mentioned in §6. In Figure 13, we start a single flow between two containers on different hosts, on Infiniband testbed. We limit the flow rate and set different bandwidth caps from 1Gbps to 40Gbps. We see that the controlled bandwidth (y-axis) is close to the bandwidth cap we set (x-axis). *FreeFlow* achieves this with only 6% CPU overhead.

FreeFlow can isolate performance (*i.e.*, throughput) for different containers using the rate limiter. To demonstrate this, we ran 10 concurrent flows between container pairs and applied the different rate limits to each flows (from 1 to 10Gbps). We verified that the throughput of each flow is accurately capped.

8.1.4 TCP Socket over RDMA

Enabling virtual RDMA can also benefit the performance of socket-based applications. Below we show that *FreeFlow* provides better performance than conventional TCP/IP virtual networks with the help of `rsocket`, an existing socket-to-Verbs translation layer.

We run the experiments on both InfiniBand and RoCE clusters. By dynamically linking with `rsocket` during runtime,⁵ application socket calls are transparently translated into RDMA Verbs calls. We run `iperf` [10] for measuring

⁵This can be easily configured by setting an environment variable called `LD_PRELOAD` in Linux.

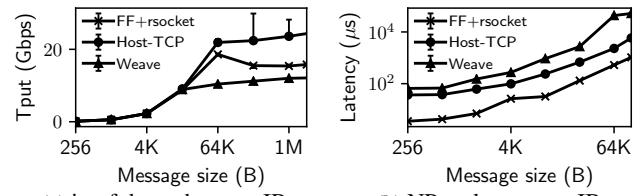


Figure 14: TCP throughput and latency between a pair of containers on different hosts. We compare native TCP with *FreeFlow* + `rsocket` (socket-to-Verbs translation).

TCP throughput, and NPtcp [14] for TCP latency *without any modifications on these tools*. We compare against the same tools running on the virtual and host mode network.

As Figure 14 shows, *FreeFlow* always outperforms Weave. Especially for small message latency, *FreeFlow* is consistently lower than even host TCP/IP, by up to 98%. For throughput, *FreeFlow* is sometimes worse than host TCP and cannot achieve full throughput like raw RDMA, due to the overhead of socket-to-Verbs translation. However, it is still 6.8 to 13.4 times larger than Weave with large messages.

The are two reasons for *FreeFlow*'s good performance. First, the RDMA stack and *FreeFlow* architecture works only in the userspace and avoids the context switching in kernel TCP stack. This advantage is not unique; customized userspace network stacks can also achieve this. The second reason *FreeFlow* outperforms Weave is fundamental. The existing TCP/IP virtual networking solutions perform packet-by-packet address translation from virtual network to host network. However, *FreeFlow* performs message-based translation from virtual connection to physical connection. Thus, *FreeFlow* always outperforms Weave, though `rsocket` introduces some socket-to-Verbs translation overhead.

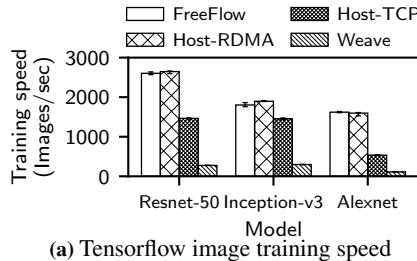
8.2 Real-world Applications

In this section, we show the performance of TensorFlow and Spark, a representative machine learning and data analytics framework, running in containers. We compare the application performance on *FreeFlow* against Host-RDMA, Host-TCP, and Weave.

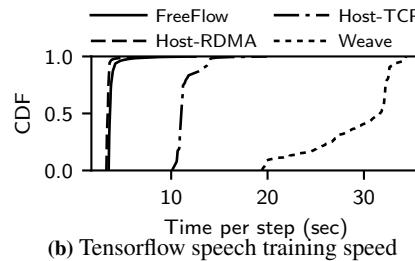
Since TensorFlow requires GPUs that our RoCE cluster does not have, we run all the experiments on our InfiniBand cluster. Based on the microbenchmarks, we believe RoCE clusters will have similar trends if equipped with GPU.

8.2.1 Tensorflow

We run RDMA-enabled Tensorflow (v1.3.0) on three servers in the InfiniBand cluster. We modified a single line of the source code of Tensorflow to replace the original memory allocation function with our custom memory allocator (§4.3). Each server has eight NVIDIA GTX 1080 Ti GPUs. One of the servers is a master node and also a parameter server, while the other two servers are workers. We run two main types of training workloads for deep learning, namely, image recognition based on Convolutional Neural Network (CNN),



(a) Tensorflow image training speed



(b) Tensorflow speech training speed

Figure 15: TensorFlow performance on *FreeFlow*.

and speech recognition based on Recurrent Neural Network (RNN).

For image recognition, we run three specific models, ResNet-50 [29], Inception-v3 [42] and AlexNet [33]. We use synthetic ImageNet data as training data. Figure 15(a) shows the median training speed per second with 10-percentile and 99-percentile values. From the results of all three different models, we conclude, first, the network performance is indeed a bottleneck in the distributed training. Comparing host RDMA with host TCP, host RDMA performs 1.8 to 3.0 times better in terms of the training speed. The gap between *FreeFlow* and Weave on container overlay is even wider. For example, *FreeFlow* runs 14.6 times faster on AlexNet. Second, *FreeFlow* performance is very close to host RDMA. The difference is less than 4.9%, and *FreeFlow* is sometimes even faster. We speculate that this is due to measurement noise.

For speech recognition, we run one private speech RNN model consisting of a bi-directional encoder and a fully-connected decoder layers, with a hidden layer dimensionality of 1024 and a vocabulary size of 100k. The dataset is 4GB large including 18.6 millions samples. In each training step, GPUs “learn” from a small piece and communicate with each other for synchronization. Figure 15(b) shows the CDF of the time spent for each training step, including the GPU time and networking time. Again, *FreeFlow* is very close to host RDMA. The median training time is around 8.7 times faster than Weave.

8.2.2 Spark

We run Spark (v2.1.0) on two servers. One of the server runs a master container that schedules jobs on slave containers. Both of the servers run a slave container. The RDMA extension for Spark [18] is implemented by is closed source. We download the binary from their official website and did not make any modification.

We demonstrate the basic benchmarks shipped with the Spark distribution – GroupBy and SortBy. Each benchmark run on 262,144 key-value pairs with 2 KB value size. We set the number of Spark mappers and reducers to 8 and each of them is a single thread. Figure 16 illustrates the result. We conclude similar observations as running TensorFlow. The performance of network does impact the application end-to-end performance significantly. When running with *FreeFlow*, the performance is very close to running on host

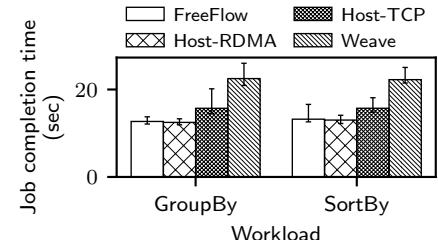


Figure 16: Spark performance on *FreeFlow*.

RDMA, better than host TCP, and up to 1.8 times better than running containers with Weave.

9 Related Work

RDMA virtualization for containers: There is an ongoing effort from Mellanox to extend network namespace and cgroup in Linux kernel to accommodate RDMA for networking isolation [34, 35]. It uses MACVLAN to split a physical interface to multiple virtual interfaces, inserts one or multiple interfaces to each container, and relies on VLAN routing to deliver traffic to the correct virtual interface. Apparently, it has portability issues for cloud environments, since moving an IP means updating VLAN routing in hardware. Also, it does not offer a flexible controllability, because it allows containers to directly access physical NICs.

Another approach is using programmable hardware to handle the RDMA virtualization for containers, such as smart NICs [26] or FPGA [38]. *FreeFlow*’s advantages compared with such hardware-based solutions are its lower cost by using commodity hardware and better flexibility to customize network features.

RDMA virtualization for VM: HyV [39] is the closest solution to *FreeFlow*. It also intercepts the communication between applications and NIC driver and provides address translation, QP/CQ mapping, and memory mapping. The key difference between HyV and *FreeFlow* is that HyV does not control data path to provide bare-metal performance in private clusters, while *FreeFlow* does for fitting in cloud environments. This creates more challenges to *FreeFlow*, such as making the performance still close to bare-metal quality while maintaining transparency to applications in data path. VMM-bypass I/O [37] has a similar design and issues as HyV. VMware has been working on para-virtualizing RDMA devices called vRDMA [40]. vRDMA is designed for VMware’s hypervisor and VMs, so it does not inherently work for containers.

10 Conclusion

In this paper, we presented *FreeFlow*, a virtual RDMA networking solution that provides the isolation, portability and controllability needed in containerized clouds. *FreeFlow* is transparent to applications and achieves close-to bare-metal RDMA performance with acceptable overhead. Evaluations with real-world applications and microbenchmarks show that *FreeFlow* can support performance comparable to bare-metal RDMA and much better than the existing TCP/IP virtual networking solution. We open source the prototype of *FreeFlow*.

11 Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd, Hakim Weatherspoon for their helpful comments. This work was funded in part by NSF awards 1700521 and 1513764.

12 Availability

FreeFlow is open sourced at <https://github.com/Microsoft/Freeflow>.

References

- [1] Mellanox coredirect. http://www.mellanox.com/page/products_dyn?product_family=61&mtag=connectx_2_vpi/, 2010.
- [2] Apache spark. <https://spark.apache.org/>, 2018. Accessed on 2018-01-25.
- [3] Arista 7050x & 7050x2 switch architecture. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7050X_Switch_Architecture.pdf, 2018. Accessed on 2018-01-25.
- [4] chroot(2) - Linux man page. <https://linux.die.net/man/2/chroot>, 2018. Accessed on 2018-01-25.
- [5] CNTK. <https://github.com/Microsoft/CNTK/wiki>, 2018. Accessed on 2018-01-25.
- [6] CoreOS. <https://coreos.com/>, 2018. Accessed on 2018-01-25.
- [7] Docker. <http://www.docker.com/>, 2018. Accessed on 2018-01-25.
- [8] Infiniband architecture specification release 1.2.1 annex a16: Roce. <https://cw.infinibandta.org/document/dl/7148>, 2018. Accessed on 2018-01-25.
- [9] Introduction to infiniband. <https://en.wikipedia.org/wiki/InfiniBand>, 2018. Accessed on 2018-01-25.
- [10] Iperf - the TCP/UDP bandwidth measurement tool. <http://iperf.fr>, 2018. Accessed on 2018-01-25.
- [11] Kubernetes. <http://kubernetes.io/>, 2018. Accessed on 2018-01-25.
- [12] Kubernetes networking. <https://kubernetes.io/docs/concepts/cluster-administration/networking/>, 2018. Accessed on 2018-01-25.
- [13] Mellanox perfest package. <https://community.mellanox.com/docs/DOC-2802>, 2018. Accessed on 2018-01-25.
- [14] netpipe(1) - linux man page. <https://linux.die.net/man/1/netpipe>, 2018. Accessed on 2018-01-25.
- [15] Open vswitch. <http://openvswitch.org/>, 2018. Accessed on 2018-01-31.
- [16] Openfabrics, libibverbs release. <https://www.openfabrics.org/downloads/libibverbs/>, 2018. Accessed on 2018-01-25.
- [17] Rdma-based apache hadoop. <http://hibd.cse.ohio-state.edu/>, 2018. Accessed on 2018-01-25.
- [18] Rdma-based apache spark. <http://hibd.cse.ohio-state.edu/>, 2018. Accessed on 2018-01-25.
- [19] Rdma-iwarp. <http://www.chelsio.com/nic/rdma-iwarp/>, 2018. Accessed on 2018-01-25.
- [20] rsocket(7) - linux man page. <https://linux.die.net/man/7/rsocket>, 2018. Accessed on 2018-01-25.
- [21] Single root I/O virtualization. http://pcisig.com/specifications/iov/single_root/, 2018. Accessed on 2018-01-25.
- [22] Tensorflow. <https://www.tensorflow.org/>, 2018. Accessed on 2018-01-25.
- [23] Weave Net. <https://www.weave.works/>, 2018. Accessed on 2018-01-25.
- [24] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *USENIX NSDI* (2016).
- [25] DATADOG. 8 surprising facts about real Docker adoption. <https://www.datadoghq.com/docker-adoption/>, 2016.
- [26] DEIERLING, K. Ensuring both high performance and security for containers. In *Flash Memory Summit* (2017).
- [27] DOCKER. Docker community passes two billion pulls. <https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/>, 2016.
- [28] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI* (2018).
- [29] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *IEEE CVPR* (2016).
- [30] IRON.IO. Docker in production – what we’ve learned launching over 300 million containers. <https://www.iron.io/docker-in-production-what-weve-learned/>, 2014.
- [31] JAIN, R., CHIU, D. M., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC Technical Report*.
- [32] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 295–306.
- [33] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *NIPS* (2012).
- [34] LISS, L. Containing RDMA and high performance computing. In *ContainerCon* (2015).
- [35] LISS, L. RDMA container support. In *International OpenFabrics Software Developer’s Workshop* (2015).
- [36] LISS, L. The Linux SoftRoce Driver. In *OpenFabrics Annual Workshop* (2017).
- [37] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX ATC* (2006).
- [38] MOUZAKITIS, A., PINTO, C., NIKOLAEV, N., RIGO, A., RAHO, D., ARONIS, B., AND MARAZAKIS, M. Lightweight and Generic RDMA Engine Para-Virtualization for the KVM Hypervisor. In *High Performance Computing & Simulation (HPCS), 2017 International Conference on* (2017), IEEE, pp. 737–744.
- [39] PFEFFERLE, J., STUEDI, P., TRIVEDI, A., METZLER, B., KOLTSIDAS, I., AND GROSS, T. R. A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces. In *ACM VEE* (2015).
- [40] RANADIVE, A., AND DAVDA, B. Toward a paravirtual vRDMA device for VMware ESXi guests. *VMware Technical Journal, Winter 2012 1, 2* (2012).
- [41] STEVENS, W. R., AND RAGO, S. A. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.
- [42] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *IEEE CVPR* (2016).

Direct Universal Access: Making Data Center Resources Available to FPGA

Ran Shu¹, Peng Cheng¹, Guo Chen^{1,2}, Zhiyuan Guo^{1,3}, Lei Qu¹, Yongqiang Xiong¹, Derek Chiou⁴, and Thomas Moscibroda⁴

Microsoft Research¹, Hunan University², Beihang University³, Microsoft Azure⁴

Abstract

FPGAs have been deployed at massive scale in data centers. Using currently available communication architectures, however, it is difficult for FPGAs to access and utilize the various heterogenous resources available in data centers (DRAM, CPU, GPU, ...). In this paper, we present Direct Universal Access (DUA), a communication architecture that provides uniform access for FPGA to these data center resources.

Without being limited by machine boundaries, DUA provides global names and a common interface for communicating across various resources, the underlying network automatically routing traffic and managing resource multiplexing. Our benchmarks show that DUA provides simple and fair-share resource access with small logic area overhead (<10%) and negligible latency (<0.2μs). We also build two practical multi-FPGA applications—deep crossing and regular expression matching—on top of DUA to demonstrate its usability and efficiency.

1 Introduction

Large-scale FPGA deployments in data centers [1–9] has changed the way of FPGA-based distributed systems are designed. Instead of a small number of FPGAs and limited resources (*e.g.*, only the DRAM on each FPGA board), modern FPGA applications can use heterogeneous computation/memory resources, such as CPU, GPU, host/onboard DRAM, SSD *etc.*, across large-scale data centers. The scale and diversity of resources enables the building of novel FPGA-based applications, such as cloud-scale web search ranking [10,11], storage systems [4,6], or deep learning platforms [12].

Building large-scale and diverse FPGA applications requires communication capabilities between any pair of FPGAs and other components in the data center. However, with today’s technology such *FPGA communication* is highly impractical and cumbersome, posing severe challenges to designers and application developers. There are three main problems barring FPGA applications to conveniently and ef-

ficiently use data center resources (see Fig. 1(a)):

First, different resources at different locations (local/remote) are connected in different ways (*e.g.*, PCIe, network) requiring different communication stacks. This greatly increases programming complexity. For example, an FPGA application may use a custom communication stack [2] to access a local (same server) FPGA, a networking stack [11] to access a remote (different server) FPGA, GPU/FPGA Direct [13] to access a GPU, DMA to access system DRAM, DDR IP to access local DRAM, *etc.* Each of these communication stacks has a different interface (different I/O ports, functional timings, *etc.*), making it hard to understand, program, optimize, and debug.

Second, most resources (*e.g.*, host DRAM, SSD) in a data center are organized in a server-centric manner. Each resource uses a dedicated name space that can only be accessed from within a host (*e.g.*, a PCIe address.) The lack of global names for resources is inefficient for FPGAs when accessing remote resources, since they first need to communicate with the remote host, and the host first has to perform the access on behalf of the requesting FPGA. If an FPGA wants to write a remote SSD, for example, it first has to transfer the data to a daemon process running on its local CPU, which passes the data to the remote CPU, which then finally writes the data to the targeted SSD. To make matters worse, developers manually write dedicated logic for each type of FPGA-to-resource communication.

Third, although FPGAs have been deployed at data center scale, current FPGA communication does not deal well with resource multiplexing. Though various resources are accessed through the same physical interface (*e.g.*, DMA and GPU/FPGA Direct both through PCIe), we are not aware of any general resource multiplexing scheme. FPGA developers need to manually handle each specific case, which is tightly coupled with the application logic. Moreover, problems become more severe when there are multiple FPGA applications using the same resource (*e.g.*, applications on two FPGAs accessing the same SSD).

Instead, FPGA developers would like an FPGA commu-

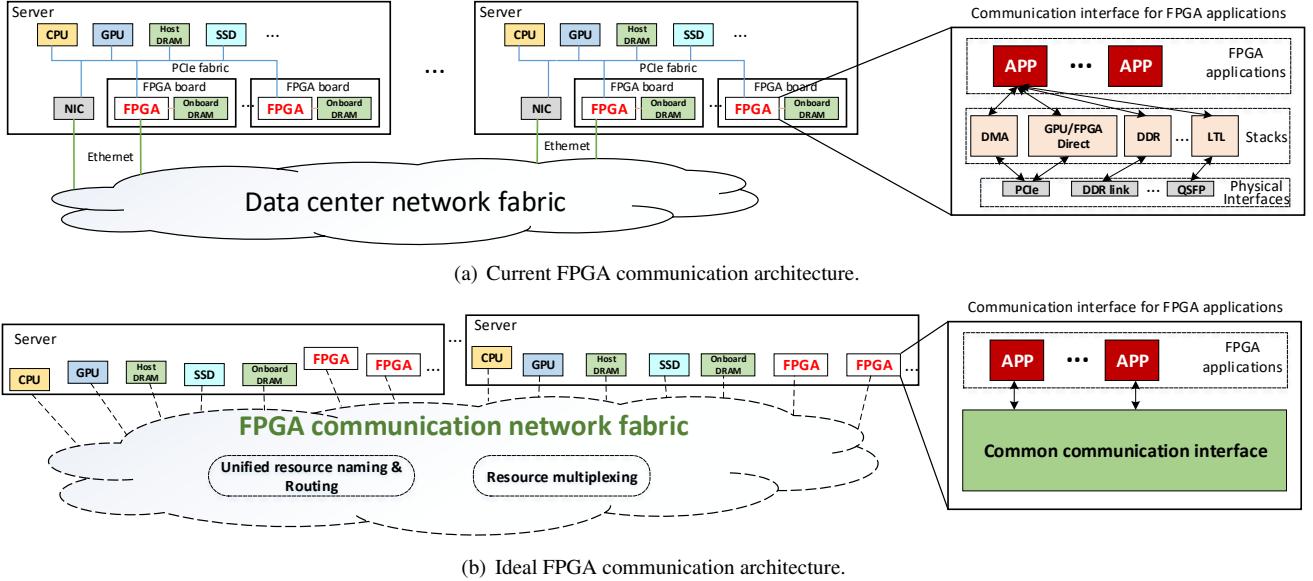


Figure 1: Comparison between a current FPGA communication architecture and an ideal communication architecture that enables FPGAs to easily access data center resources.

nication architecture as shown in Fig. 1(b). This architecture has the following desirable properties: 1) a *common communication interface* regardless of what the communication endpoints are and where they reside; 2) a *global, unified naming scheme* that can address and access all resources regardless of their location; 3) an *underlying network service* that provides routing and resource multiplexing. With such a communication architecture, FPGA applications could easily access a diverse set of resources across the data center, using the common programming interface with each resource’s global name. Indeed, such a communication architecture is what developers and architects expect and is used in other distributed systems. For example, such a design has been proven successful in IP networks.

In this paper, we propose Direct Universal Access (DUA) to bring this desirable communication architecture to the FPGA world. Doing so is challenging in multiple ways, especially considering that very little, if anything, can be changed when we seek real-world deployment in existing data centers. It is impractical to require all manufacturers to support a new unified communication architecture. To circumvent this challenge, DUA chooses to abstract an overlay network on top of the existing communication stacks and physical interconnections, thereby providing a unified FPGA communication method for accessing all resources. Moreover, performance and area is often crucial in FPGA-based applications, so DUA was designed to minimize performance and area overheads. Inspired by ideas in software defined networking [14, 15], we architect DUA into a *data plane* that is included in every FPGA, and a hybrid *control plane* including both CPU and FPGA control agents. Needless to say, designing and implementing this novel communication architecture also brings about numerous specific tech-

nical challenges, design choices and implementation problems. We discuss these challenges alongside our solutions in Sections 4, 5, 6 and 7.

In summary, we make the following contributions in this paper. We introduce DUA, a communication architecture with unified naming and common interface to enable large-scale FPGA applications in a cloud environment. We implement and deploy DUA on a twenty FPGA testbed. Our testbed benchmarks show that DUA introduces negligible latency ($<0.2\mu s$) and small area ($<10\%$) overhead on each FPGA. Moreover, we demonstrate that using DUA, it is easy to build highly-efficient production-quality FPGA applications. Specifically, we implement two real-world multi-FPGA applications: Deep Crossing and regular expression matching, and show the vastly superior performance of these applications based on DUA. In addition, we also design and implement a new communication stack that supports high-performance communication between local FPGAs through PCIe, which can be integrated into DUA data plane as a highly efficient underlying stack.

2 Background

2.1 FPGA Deployments in Data Centers

The left side of Fig. 1(a) shows an overview of current FPGA data center deployments. FPGA boards connect to their host server motherboard through commodity communication interfaces such as PCIe. Each hosting server can contain one [11] or more FPGAs [2]. Each FPGA board is typically equipped with gigabytes of onboard DRAM [2, 9, 11]. Recent deployments [11] directly connect each FPGA to the data center networking fabric, enabling it to send and receive packets without involving its host server. To meet the high data rate of physical inter-

Table 1: FPGA programming efforts to connect different communication stacks.

Resource	Communication stack	LoC
Host DRAM	DMA	294
Host CPU	FPGA host stack	205
Onboard DRAM	DDR	517
Remote FPGA	LTL	1356

faces, FPGAs use Hard IPs (HIPs) to handle physical layer protocols for each interface. Above these HIPs, FPGAs provide communication stacks that abstract access to different resources. Communication stacks may share the same HIP, e.g., DMA [16] and GPU [13] stacks both need to use the PCIe fabric. Although a server may contain multiple boards connected through PCIe [2], there are no PCIe-based stacks that support efficient and direct communication between FPGAs.

Data center FPGAs often contain a *shell* [10, 11] that contains modules that are common for all applications. For example, shells typically include communication stacks for accessing various resources (e.g., PCIe, DMA, Ethernet MAC). In this way, developers only need to write their FPGA application logic and connect using these communication interfaces. The FPGA shell is similar to an operating system in the software world.

FPGAs in data centers are widely used to accelerate different applications. Applications like deep neural networks [17, 18] and bioinformatics [19] have high demand on communications between FPGAs. FPGAs for web search ranking applications [10, 11] rapidly exchange data with host and other FPGAs to generate the ranking as quick as possible. Key-value store acceleration [20] requires FPGAs to access remote FPGA’s on board DRAM or even remote servers host memory. Big data analytics [21] not only require rapid coordination between computation nodes, but also need to directly fetch data from database [4, 5]. The demand of high throughput and extra low latency require FPGAs to access heterogeneous resources directly which challenges the design of FPGA communication architecture in data centers.

2.2 Existing Problems

Current FPGA communication architecture pose multiple severe problems:

Complex FPGA Application Interface: FPGA-based systems are hard to develop and deploy. One of the major reasons is that communication interfaces are hard to implement. Interfacing requires significant programming expertise and effort by application developers. To make things worse, existing stack interfaces are highly implementation specific, with substantial incompatibilities and differences between different vendors. This makes building the communication system of the application alone a major undertaking (e.g., KV-Direct [20]).

To convey a concrete sense of the programming difficulties involved, consider a simple FPGA application that uses

different communication interfaces to access four different resources: host DRAM, host CPU, on board DRAM, and remote FPGA. Table 1 shows the lines of Verilog code for application developers to connect each stack’s interface.

Poor Resource Accessibility: Although data centers provide many computation/memory resources that potentially could be used by FPGA applications, most of these resources (even the homogeneous ones) are only named in server-local name space and work with their own software device driver/stack. There is no unified naming scheme for accessing remote resources. Without unified naming, most PCIe-based resources (e.g., DRAM, SSD, GPU) can only be accessed within a server’s PCIe domain, making it difficult for remote FPGAs to use. Even with latest technology like RDMA that FPGAs can use to access specific remote resources, the software driver/stack is still needed for remote communication, impacting performance.

Fixed Network Routing: In current communication architectures, FPGA applications can only access resources through limited and fixed paths. In [2], for example, FPGAs communicate with other local FPGAs through the dedicated PCIe fabric and can not access remote resources through networking. In [11, 22], FPGAs are directly connected to Ethernet through top-of-rack (ToR) switches, i.e., a pair of FPGAs can only communicate through Ethernet even when they are in the same PCIe domain. Both of these examples cannot make full use of all available bandwidth.

Also, such fixed communication architectures limit the system’s scalability. For example, deploying large FPGA applications via a network-based communication architecture increases the port density of ToR switches and is a challenge to data center networking, even if most FPGAs are used for compute-intensive tasks and need only little networking bandwidth.

Poor Resource Multiplexing: To support accessing data center resources as a pool, resource multiplexing is one of the key considerations of an FPGA communication architecture. Current architectures do not handle stack multiplexing well. For example, if two applications both access local host DRAM through DMA, they need to collaboratively write a DMA multiplexer and demultiplexer. From our experience, even a simple multiplexer/demultiplexer requires 354 lines of HDL code. Moreover, currently there is no general physical interface multiplexing scheme, and it is therefore hard for current FPGA applications to simultaneously access local host DRAM and local SSD without modifying the underlying shell, since these two resources are both connected through the PCIe bus.

The Elastic Router proposed in [11] tries to solve the multiplexing problem in an FPGA environment. Currently, however, it only addresses the problem of multiplexing between multiple applications which use a common networking stack, without handling multiplexing between other stacks and between physical interfaces. Later we will see that DUA ex-

tends this with a general resource multiplexing scheme.

Inefficient Communication Stack: Existing communication architectures implement resource accessing for FPGAs in an indirect and inefficient way. Typically, FPGA applications use DMA to access local resources, which results in significant latency and low bandwidth. We note that while [23] provides a direct FPGA-to-FPGA communication mechanism through PCIe, it is inefficient. Specifically, the receiver FPGA acts as host DMA driver and first issues an DMA request. The sender FPGA treats the request as a normal DMA request and sends data. After sending data they need to synchronize the queue pointers. Since a data transmission crosses the PCIe fabric 3 times, it wastes bandwidth and has higher than necessary latency.

Furthermore, an FPGA can only access remote host DRAM by relaying data between the two sides' CPUs, reducing performance and consuming cores. We measured the performance of doing so in our testbed (see §8). Specifically, we ran two daemon processes on the local and remote server's CPU that relayed data between the local FPGA and remote DRAM through a TCP socket. Results show that for writing 256B data to the remote DRAM, the average end-to-end latency is $\sim 51.4\mu s$. The tail latency is in the milliseconds. Using remote DMA instead of TCP may improve the performance, but in our measurement the average latency is still $\sim 20\mu s$ due to the CPU involvement (e.g., initiate request, interrupt). We note that in such an application it is possible to leverage remote FPGA as a data relay for accessing remote host DRAM, through direct communication between FPGAs (e.g., using LTL [11]).

3 Desired Communication Architecture

To overcome the problems outlined in the previous section, we design DUA using a familiar and concrete model: *Global names* and a *common communication interface* for FPGAs and resources regardless their location, where the underlying network automatically *routes* communication traffic according to the global name and manages the *resource multiplexing* with full utilization of *existing stacks*.

This communication architecture supports pooling data center resources for FPGAs to access. Specifically, FPGA applications can access any resource in data center using a global name and a common programming interface. The network provides a globally unified name for various kinds of resources. When getting a message, the network either routes the access to the targeted resource if it is available, or notifies the application if the resource is not available, automatically without the application being involved. Also, there is no need for applications to implement multiplexing between communication stacks or physical interconnections. DUA utilizes underlying communication when appropriate. The network automatically manages sharing and contention according to the desired policy (e.g., fair sharing or priority scheduling).

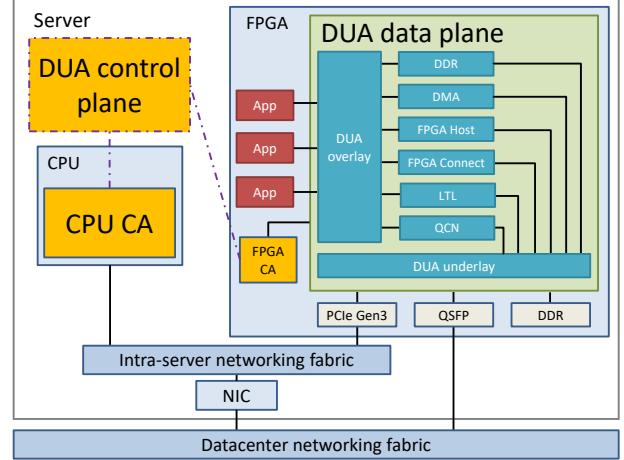


Figure 2: DUA architecture.

Networked systems communicate in exactly this way. In computer networking systems, programmers use IP addresses with TCP/UDP ports as a global name to identify a communication endpoint, and access them using a unified BSD socket interface. The network (both networking stack and fabric) automatically route the traffic through the paths calculated from routing protocols. The network also deals with resource multiplexing through techniques such as congestion control and flow control.

Of course, the communication architecture must also carefully consider security mechanisms, such that the universal access of FPGAs to resources within the data center does not damage or crash other hardware/software systems. Since FPGA-based applications often have high performance requirements, performance and resource overhead of the unified communication method must be kept low.

4 DUA Overview

Overall, DUA abstracts an overlay network for FPGA on top of existing data center network fabric. DUA provides a new communication architecture that has the desired properties mentioned before, which makes data center resources a shared pool for FPGA.

In detail, we provide an communication architecture with such overlay network, including the common communication interface and the naming scheme suitable for various applications to access different resources, and the routing, multiplexing, and resource management scheme correspondingly provided by the network.

Fig. 2 shows the system architecture of DUA. Specifically, DUA consists of a low-cost hardware *data plane* residing in every FPGA's shell, and a hybrid *control plane* including both CPU and FPGA control agents.

The DUA control plane is the brain of the overlay network. It manages all resources, assigning addresses and calculating the routing paths to them, and manages the multiplexing of resources and the network. DUA supports both connection-based and connectionless communication. The connection

UID (serverID:deviceID)	Address /port	Resource description
192.168.0.2:1	0x00000001CFFFF000	1st block of host DRAM
192.168.0.2:1	0x000000019FFFF000	2nd block of host DRAM
192.168.0.2:2	0x80000000	1st block of FPGA onboard
192.168.0.2:3	8000	1st application on FPGA
192.168.0.2:3	8001	2nd application on FPGA

Figure 3: Example resources address on server 192.168.0.2.

setup and close are processed and managed in the DUA control plane.

The DUA data plane is the actual executer of FPGA communication traffic. It stays between the FPGA applications and physical interfaces. The data plane can efficiently reuse existing communication stacks, as well as support new stacks, providing the same communication interface for various applications to access different resources.

5 DUA Communication Interface

We first describe the unified communication interface provided by DUA for accessing various resources.

5.1 Resource Address Format

DUA provides each resource a unified address, which is globally unique in the data center. Devising a totally new address format is not a wise option, since it would require both a complicated system for managing data-center-scale address spaces and changes to the existing network fabric to use that new address format. Instead, DUA leverages the current naming schemes of various resources, and combines them into a new hierarchical name.

Specifically, DUA assigns each device a unique name (UID) that extends the IP address into the intra-server network. A UID consists of two fields, *serverID:deviceID:resourceINST*. *serverID* is a globally unique ID for each server. In an Ethernet-based data center network, we leverage the server IP address as *serverID*, which is already uniquely assigned by the network fabric. *deviceID* is a unique ID for each resource within the server (e.g. FPGA on-board DRAM, GPU, NVMe and etc.), which is newly assigned by DUA. In our current implementation, UID is designed to be 48 bits in total (32b *serverID* (length of IPv4 address in current data centers) and 16b *deviceID*).

Within each device, DUA leverages the existing addressing scheme of each resource. For example, it can be the memory address if the targeted resource is host/onboard memory, or the port number if the targeted resource is a remote FPGA application. Fig. 3 provides some examples of different resources' addresses in DUA. In §6.1 and §6.2 we will describe how it is easy to manage addresses and do routing using such an UID format.

5.2 API

DUA supports both connection-based and connectionless communication. Connectionless communication is less efficient than connection-based communication because it fa-

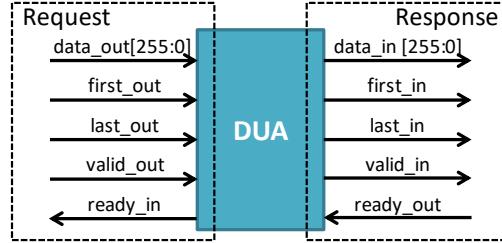


Figure 4: I/O interface for DUA.

cilitates management issues (e.g., access control) and network resource multiplexing (e.g., multiplex underlying stack tunnels for the messages that have common routing paths). More details of routing and connection management will be discussed in §6.2 and §6.3.

5.2.1 Semantic Primitives

For each FPGA communication connection, applications generate communication primitives similar to BSD socket. There are two types of primitives:

1. **Connection setup/close primitives:** These primitives include CONNECT/LISTEN/ACCEPT and CLOSE, which are used by applications to setup and close a connection, respectively.
2. **Data transmission primitives:** These primitives include SEND/RECV and WRITE/READ. SEND/RECV are for sending/receiving data to/from computation resources such as other FPGA applications and CPU processes, which works as a FIFO between the two sides. Additionally, DUA supports message-based communication by adding a PUSH flag in each DUA message header. WRITE/READ are one-sided primitives for write/read data to/from memory/storage resources, which are different from WRITE/READ in BSD sockets.

5.2.2 I/O interface

We implement DUA in both Verilog and OpenCL. OpenCL is a high-level programming language for FPGAs (and GPUs.). DUA implemented in OpenCL can provide socket-like interface. However, it cost much more FPGA logic and degrade performance. Thus we only use the Verilog implementation and add a wrapper on top of it to support OpenCL. See Appendices A and B for a sample usage of the DUA interface.

Fig. 4 shows the physical I/O interface of DUA, which is full-duplex. The *request interface* is for applications to issue primitives (§5.2). The *response interface* is for applications to get primitive responses (completion information or response data).

The DUA I/O interface is the same in both direction. For each DUA message, the message header and payload use the same data wires for transmission. Note that the data signal has only 256 bits. Although a wider interface could

increase the amount of data transmitted per hardware cycle, it would increase the switch fabric logic complexity and thus decrease the available clock frequency of corresponding modules. Consequently, a DUA message may be transmitted in several cycles. The first 1 or 2 cycles (depends on the message type) of data bus is message header, followed by the payload. The first/last bit indicate the first and last cycle of a message transmission. For each cycle with data to be sent or received, the valid signal is set. Note that valid can only be raised when the receiver side set the ready signal.

Next in §6 and §7, we introduce the design and implementation of control and data plane.

6 DUA Control Plane

The DUA control plane manages all resources, routing and connections. Since FPGAs are not suitable for implementing complicated control logic, we put the main control logic in the CPU control agent (CPU CA). We also implement a control agent in the FPGA (FPGA CA) which monitors local resources on its board and delivers control plane commands to the data plane.

6.1 Resource Management

Logically, the entire DUA control plane maintains the information of all available resources in data center, and assigns each resource with a UID. Thanks to the hierarchical address format (§5.1), each DUA control agent only needs to handle its local resources.

Specifically, each FPGA CA monitors all available resources on its FPGA board (*e.g.*, onboard DRAM, FPGA application). The FPGA CA does not assign addresses for those resources. Instead, each FPGA CA uploads its local resource information to the CPU CA in its host server, and the CPU CA assigns the UIDs all together. The host CPU CA gathers the resource information from all FPGA CAs, as well as all other resources such as host memory and GPU in this server.

It is straightforward to assign UIDs to local resources. As mentioned, the first UID field, *serverID*, is the server IP. Then CPU CA assigns a unique *deviceID* for each resource within this server. The CPU CA maintains the mapping from *deviceID* to different local resources, and updates the mapping once when are any resource changes, plug-in/plug-out or failures. DUA does not manage the address within each resource instead letting each device control it.

Currently, DUA does not provide a naming service. Applications directly use UID to identify resources without a name resolution service. The design of a naming service is future work.

6.2 Routing Management

To offer routing capabilities, the DUA control plane calculates the routing paths. The data plane forwards the traffic to the target resource (directly or through other FPGAs' data plane) fully transparent to applications.

Src Resource (UID)	Dst Resource (UID) / Stack
FPGA 1 (192.168.0.2:1)	FPGA 2 (192.168.0.2:2) / FPGA Connect
	Host DRAM (192.168.0.2:3) / DMA
	Onboard DRAM (192.168.0.2:4) / DDR
FPGA 2 (192.168.0.2:2)	FPGA 1 (192.168.0.2:1) / FPGA Connect
	Host DRAM (192.168.0.2:3) / DMA
Resources on other servers (*:*) / LTL	

Figure 5: Example interconnection table on server 192.168.0.2.

Designing and implementing data center scale routing is challenging [24]. Benefiting from the hierarchical UID format, we leverage existing data center network routing capabilities. Each DUA control agent only needs to maintain interconnection information and calculate routing paths within each server.

Specifically, each CPU CA independently maintains an interconnection table for all local resources, as shown in Fig. 5. The interconnection table records the neighborhood information between FPGAs or FPGA and other resources. The first column records a source FPGA, and the second column records the local/remote resources that can be directly accessed from this FPGA through which underlying communication stack.

The interconnection table's information is updated as follows. Besides the resource information, each FPGA CA uploads the information about its communication stacks and physical interfaces to the CPU CA in its own server. Based on the uploaded information, CPU CA determines the interconnection between different FPGAs and updates the interconnection table. If an FPGA reports that it has connectivity through the data center networking fabric, the CPU CA will insert an entry for this FPGA, with an entry for each legal destination to any resources on other servers (the last row of Fig. 5).

According to the interconnection table, it is easy to calculate a routing path to targeted resources. Specifically, if an FPGA wants to communicate with some resource, DUA first checks the *serverID* and *deviceID* field in the destination resource UID, to see if this resource has a direct connection from this FPGA. If yes, DUA uses the stack recorded in the interconnection table to access the resource. If not, DUA looks up the interconnection table to find a routing path through other FPGAs.

For example, in Fig. 5, if FPGA 1 (UID 192.168.0.2:4) wants to communicate with a remote application on FPGA 3 located on another server (say, UID 192.168.11.5:3), the calculated routing path is from FPGA 1 to FPGA 2 via FPGA Connect, and then to FPGA 3 via LTL.

6.3 Connection Management

In DUA, every FPGA communication is abstracted as a connection. A connection is uniquely identified by a *<src UID:dst UID>* pair. The DUA control plane is in charge of managing all connections.

At the connection setup phase, to ensure security, DUA first checks the access control policy to see if the source FPGA application is allowed to access the destination resource. If so, the CPU CA will check the *dst UID* with the interconnection table to calculate the routing path (§6.2) and then delivers the forwarding table to the FPGA data planes along the routing path, so the data plane will forward the application traffic to the right stack. Depending on the type of routing path, CPU CA will deliver different actions to the data plane and underlying stacks. Specifically:

- 1) If the destination resource is directly connected, CPU CA simply delivers the corresponding forwarding table to the data plane.
- 2) If the destination resource is not directly connected, but still within the same server, CPU CA calls the stacks in the local FPGAs along the routing path to setup a stack connection. For example in Fig. 5, if FPGA 2 initiates a connection to access the onboard DRAM of FPGA 1, CPU CA first sets up an FPGA Connect connection between FPGA 2 and FPGA 1.
- 3) If the destination resource is on a different server, CPU CA first calls the remote CPU CA to collaboratively setup a connection tunnel between the two remote FPGAs (*e.g.*, LTL connection). If necessary, CPU CA also sets up stack tunnels between each sides' local FPGAs.

If the above procedures all succeed, the DUA connection is established and the application is notified. Also, the active DUA connection is maintained in the control plane. Note that some underlying stacks do not support a large number of concurrent connections (*e.g.*, LTL currently only supports 64). For multiple DUA connections with common routing paths, DUA supports connections multiplexing the same tunnel connection (*e.g.*, two DUA connections share an LTL tunnel connection) to solve this problem. Moreover, DUA sets up multiple tunnels for each traffic class to simplify traffic scheduling.

When an application closes a connection, the DUA control plane closes the stack tunnel connections along the path (if no one is multiplexing them), and deletes the corresponding forwarding tables in data plane. If any failures of the data path (*e.g.*, targeted resource, physical interface, communication stack) is detected, the control plane immediately disconnects all affected DUA connections, and notifies the application.

7 DUA Data Plane

As shown in Fig. 2, the DUA data plane resides between FPGA applications and the physical interfaces. It consists of three components: *overlay*, *stack*, and *underlay*. DUA overlay acts as a router, transferring data between different applications and communication stacks. Below the overlay, DUA leverages all existing (or future new) stacks to efficiently access target resources. DUA underlay connects between the stacks and physical interfaces, which provides efficient multiplexing on physical interfaces for different stacks.

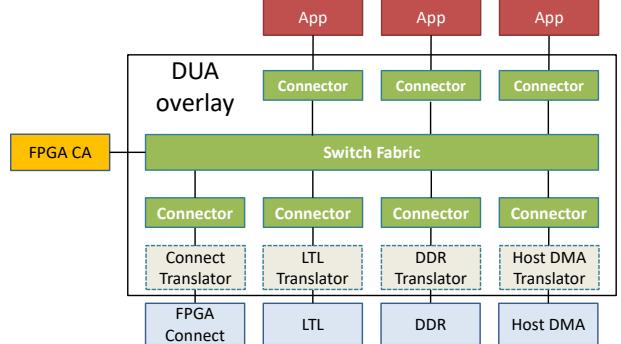


Figure 6: DUA overlay components.

Src UID (6B)		Dst UID (6B)		Sequence (4B)
IP (4B)	devID (2B)	IP (4B)	devID (2B)	
Flag (1B)	Length (2B)	Type (1B)	Param (12B)	

Figure 7: DUA message header format.

7.1 DUA Overlay

To efficiently transfer data between multiple different applications and stacks, we use an “Internet-router-like” architecture to implement the DUA overlay module. Specifically, there are three components inside the overlay, *connector*, *switch fabric* and *stack translator*, as shown in Fig. 6.

7.1.1 Connector

Connectors reside between application/stack and switch fabric, playing a role similar to line cards in Internet routers. Specifically, connector performs the following tasks:

1) Translating data (from application or stack) from/to I/O interface to/from DUA messages: The I/O interface described in §5.2 is actually implemented in connectors, receiving data both from applications or stacks. A DUA message is the data transmission unit inside the overlay (like IP packets). Its header format is shown in Fig. 7. Connector encapsulates data into DUA messages in cut-through mode with the corresponding header fields filled. Also, when connector receives a DUA message from the switch fabric, it translates it back to the I/O interface signals. One thing to note is that for connection setup/close primitives passed from the I/O interface, connector encapsulates a special message and passes it to the FPGA control agent, notifying the control plane to setup/close the connection.

2) Maintaining and looking up the forwarding table: The forwarding table stores the mapping of destination UID to the switch output port. After message encapsulation, the connector needs to lookup the forwarding table to determine the switch output port to forward the message to the destination connector through the switch fabric. The forwarding table is computed by the control plane and delivered to DUA connector (§6.2). To eliminate the contention between connectors during forwarding table lookup, each connector only maintains its own forwarding table and performs lookups independently. Note that only entries for active connections and permitted connectionless message routes are delivered to the data plane, so this table is not large. In our current im-

plementation, the table can store 32 forwarding entries and the area cost is very low (see §8.1).

3) Access control: Connector is also responsible for security checks whenever there is data coming in. Specifically, for connection-based communications, after a connection has passed the security check and has been successfully setup by the control plane (see §6.3), the control plane adds this connection to the routing control table in the connectors along the path. For connectionless communications, the control plane sets the routing table according to policies that determine which path is allowed for these messages. Only data from these legal connections or paths are transmitted by DUA connectors.

4) Transport Control: DUA adopts an end-to-end transport control design. Thus this feature is only enabled for connectors that attach to applications. We do not reimplementing TCP or RDMA on FPGA, instead, DUA leverages LTL [11] as the transport protocol.

7.1.2 Switch Fabric

The switch fabric performs the same role as its counterpart in Internet routers, switching messages from the incoming connector to the destination connector. Specifically, DUA overlay adopts a crossbar switch fabric. To minimize memory overhead, we do not buffer any application or stack data in the switch fabric. Instead, we make our switch fabric lossless, and utilize the application data buffer or stack data buffer to store data that is going to be transmitted. Once the output is blocked, the switch fabric will back pressure to the input connector, and unset the *Ready* signal of I/O interface.

In our current implementation, the underlying stacks (LTL, FPGA Connect, DMA, FPGA Host and DDR) are all reliable, as such, the lossless fabric ensures the DUA data transmission primitives (§5.2) are also reliable. Note that in real hardware implementations, although we do not buffer data, in order to achieve full pipelining, we need to cache 256b data (one cycle of data from the I/O interface) at each input port. And to remove head-of-line blocking among different ports, we implement a Virtual Output Queue (VOQ) as in elastic router [11] at each input port of the switch fabric.

7.2 Communication Stacks

In our current implementation, we integrate four existing communication stacks (LTL, DMA, FPGA-Host and DDR) into the DUA data plane using *stack translators*. Note that DUA leverages LTL as its end-to-end transport protocol. LTL here only provides reliable communication in data center network, its end-to-end congestion control protocol is disabled. In addition, we design and implement a new stack called *FPGA Connect* that provides high-performance intra-FPGA communication through PCIe for improving the communication efficiency mentioned in §2.2.

7.2.1 Stack Translators

Stack translators translate between DUA interface (§5.2) and the actual interface of underlying stacks. After control

plane sets up the connection, it delivers the corresponding translation tables to the stack translators along the routing path. Translation tables record the mapping of DUA message header and underlying stack header. Whenever receiving data from connector, the translator encapsulates the data into stack interface according to the translation table. If control plane decides to multiplex stack tunnels, stack translator encapsulates multiple DUA connections' data into the same stack connection. On the other end, when receiving data from stacks, translator translates it back into DUA interface and passes it to the connector for further routing.

Taking stack translator for memory stacks DDR/DMA as example, it converts DUA operations to memory operations. For instance, when the stack translator receives data with *Type READ* from DUA connector (*i.e.*, DMA/DDR read initiated by applications), it calls the DMA/DDR stack to issue a read request, with the memory address set accordingly to the address in DUA message header. Also, the DUA message header is stored for sending the READ response back to the application through DUA. After it gets the response, stack translator calls DUA interface to send data back.

Similar to the forwarding table, the translation table also only stores entries for active connections. Currently we implement a table with size for 32 entries.

7.2.2 FPGA Connect Stack

FPGA Connect Stack enables direct communication between multiple applications on different FPGAs through PCIe within a single server. Here we introduce the design and implementation details of FPGA Connect Stack.

Challenge: There are three major challenges. 1) *Differentiating different inter-FPGA connections*: One naive solution is to use a large number of physical memory addresses to receive packets of each connection. That not only needs a large amount of memory address space but also introduces address management overhead. 2) *Head-of-line (HOL) blocking*: PCIe is a lossless fabric and back pressure is adopted. Due to application processing limitations and PCIe bandwidth sharing, the available rates of each connection can be different. Without an explicit flow control, the slower connection will saturate the buffer on both sender HIPs and receiver which will delay other transmissions. 3) *Bufferbloat*: If packets are sent to PCIe HIP in a best-effort manner, the buffer inside HIP will quickly fill up which causes further delays.

Design: FPGA Connect provides SEND and RECV operations to users. In order to differentiate different connections and minimize physical memory address waste, FPGA Connect adds a packet header in PCIe packet's payload containing both sender's and receiver's port, and uses one identity single-packet-size¹ physical memory address as receive address for each board.

¹The packet size mentioned in this paper is the PCIe TLP layer payload size.

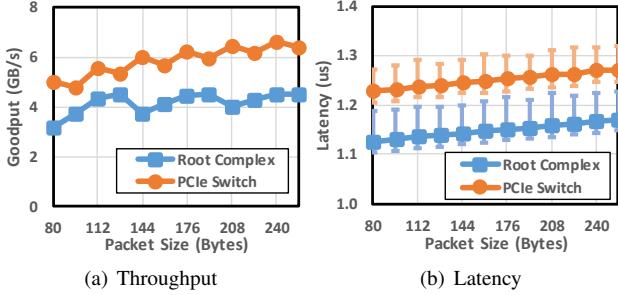


Figure 8: Performance of FPGA connect stack using different TLP packet sizes.

We provide a simple token-based flow control to avoid HOL blocking and bufferbloat. FPGA connect sends one token request, which is a normal data packet with a special bit, in each RTT to ask the receiver for the available token if it keeps sending. Receiver responds with an ACK that includes the available token for this connection once it receives this request. The sender uses this token to set the size of the sending window for the next RTT. The receiver only keeps the available connection number and assigns the available token based on the algorithm of [25] to keep low buffer in HIPs.

PCIe provides WRITE and READ operation primitives for data transmission. According to our measurements, PCIe peer-to-peer READ throughput is 20%-40% lower than WRITE because of hardware limitations. Therefore, FPGA Connect only uses WRITE as the data transmission primitive for performance reason.

Implementation: In our implementation, FPGA Connect has a 16 bits header including 8 bits destination connection ID, 2 bits type and 6 reserved bits. Packets with $Type = 0x01$ are token requests, and those with $Type = 0x11$ are ACKs. Other packets are normal data packets. We leverage CPU software for connection setup, release and fail-over.

Evaluation: The typical PCIe network topology is a tree structure. The root node is called root complex which is in the CPU. Devices (e.g. FPGA, GPU and NVMe) are directly connected to it. As the number of PCIe lanes provided by root complex is limited, the number of devices connected to root complex and the peer to-peer bandwidth among devices is limited. Thus, some data center servers use PCIe switch to support more devices and improve peer-to-peer performance which provides high density of heterogeneous computation capacity. Thus we test the performance of FPGA Connect on two platforms. One provides connection through the PCIe switch and the other through the root complex. For our testbed, the maximum packet payload size is 256B. Although the FPGA can send 256B packets, Root complex forces segmentation of packets to align in 64B units (the PCIe switch does not segment packets). We have measured the performance of FPGA Connect with different packet sizes using our testbed described in §8.1. Fig. 8 shows the results. FPGA Connect achieves 6.66 GB/s peak throughput when

the packet size is 240B (the peak throughput is limited by TLP implementation issues in our FPGA shell). Consequently, in order to reduce the header overhead and achieving higher throughput, we choose 240B as our maximum packet size. As for latency, FPGA Connect provides latency as low as 1.1~1.33 us. In our testbed, PCIe switch offers better throughput but slightly higher latency than root complex.

7.3 DUA Underlay

The DUA underlay resides between the stacks and physical interfaces, managing hard IPs and resource sharing among stacks, protecting DUA stacks against outside attacks and avoiding failed stacks sending packets outside the FPGA. All these features are managed by policies configured by the control plane. Each physical interface has a separate underlay module. The upstream and downstream interface are the same to provide seamless integration with stacks. Therefore, existing stacks need no modification when attached to DUA underlay.

The DUA underlay achieves these goals by setting up a virtual transaction layer, which provides multiplexing and security protection without proscribing a stack interface abstraction. The virtual transaction layer works by checking, modifying, and, if necessary, dropping traffic generated by or routed to the stacks to prevent causing a physical interface (or even the whole network) into an error condition.

When data flows into DUA underlay from stacks, all packages are passed through a filter which validates them as well-formed per the rules configured by the control plane. If stack traffic violates any security rules or physical interface restrictions, the packet is dropped and the violation is reported to FPGA CA. Then, when data flows from virtual transaction layer to physical interfaces, the DUA underlay works as a multiplexer, take the responsibility of managing multiple connections for supporting multiple users. DUA underlay scheduling the data to the physical interface using policies like fair-sharing, weighted sharing, strict priority etc. In our implementation, we use fair-sharing. To avoid wasting bandwidth, we implement a shallow input FIFO for each stack in the underlay. The scheduler fairly schedules data from non-empty FIFOs only.

When receiving data from a physical interface, the DUA underlay works as a demultiplexer. It demultiplexes the incoming data to the corresponding stack through virtual transaction layer according to the data header.

8 Evaluation

Testbed Setup: As shown in Fig. 9, we build a testbed consisting of two Supermicro SYS-4028GR-TR2 servers, 20 FPGAs and one Arista 7060X switch. Every 5 FPGAs are inserted under the same PCIe switch and only one FPGA under each PCIe switch is connected to the Arista switch. All FPGAs in the testbed are the same as in [11], which is an Altera Stratix V D5, with 172.6K ALMs of programmable logic, one 4 GB DDR3-1600 DRAM channel, two PCIe Gen 3 x8

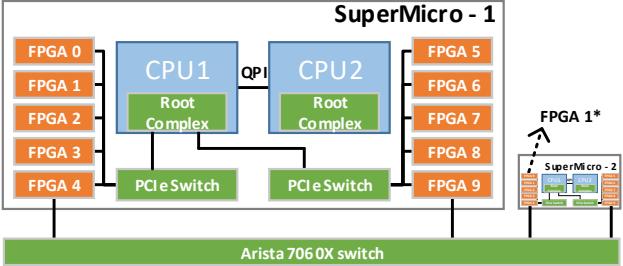


Figure 9: DUA experiment testbed

Component			ALMs	
DUA overlay	Switch fabric	2 ports	1272	0.74%
		4 ports	3227	1.88%
		8 ports	9366	5.45%
	Connector		3011	1.75%
	Stack translator	FPGA Connect	138.4	0.08%
		LTL	255.4	0.15%
		DMA	115.7	0.07%
		DDR	190.3	0.11%
DUA underlay	Stacks: FPGA Connect, LTL, DMA, DDR PHY interfaces: PCIe, DDR, QSFP		431.7	0.25%

Figure 10: FPGA area cost of different components in DUA data plane.

HIPs and two independent 40 Gb Ethernet QSFP+. Note that in all the following experiments, we only enable one HIP and one QSFP+ in each FPGA shell. In addition, because SuperMicro does not provide two PCIe slots directly connected to the root complex, we use a Dell R720 server to test the performance under root complex (experiments in §7.2.2). Other experiments are on the SuperMicro servers. Servers’ OS is Windows Server 2012 R2, and each server has a 40Gbps NIC connected to the switch.

8.1 System Micro Benchmark

We first show that DUA only consumes little FPGA area. Then we show that the DUA switch fabric and routing table achieve high throughput and low latency. Finally we show that DUA incurs little latency overhead and handles the multiplexing of communication stacks and applications well.

8.1.1 FPGA Area Cost

Fig. 10 shows the FPGA resource consumption for implementing DUA. Here we only list logic resource overhead (in ALMs) since DUA does not buffer data and BRAM cost is negligible. When connecting four stacks and no application, the total ALMs consumed by DUA overlay (including 4-port switch, 4 connectors and 4 stack translators) is only 9.29%. When increasing the number of switch ports to 8 (4 ports for applications), the overlay still only costs 19.86% logic area in our middle-end FPGA. The underlay consumes only 0.25% logic resources when connecting 4 stacks and 3 physical interfaces. Compared to the logic resources consumed by the existing underlying communication stacks and physical interfaces (in total >17%), such overhead incurred by

Table 2: Throughput and latency of switch fabric.

Number of ports	Latency (ns)			Throughput (GBps)
	min	avg	max	
4	53	326	1086	9.6
8	56	649	1903	9.6

Table 3: Area cost and max frequency of routing table.

Number of entries	ALMs (per port)	Fmax (MHz)
32	1435	0.83%
64	2810	1.63%
128	5571	3.23%

DUA is moderate and acceptable. With more advanced FPGAs, such logic area cost will become negligible (e.g., latest Stratix 10 5500 FPGA has 10x logic resource [26]).

8.1.2 Switch Fabric Performance

We conduct an experiment to evaluate the performance of the switch fabric in DUA overlay. In this test, all switch ports are attached to an application which acts as traffic generator and result checker. All applications send messages to a single port to construe a congested traffic scenario. Message length is varied from 32B to 4KB. The switch fabric works at 300 MHz, thus the ideal throughput is 9.6 GBps. We measure the latency and output throughput during a test lasting for 2 hours. Table 2 shows the result with different number of switch ports. Throughput achieves the theoretical maximum and Latency is low.

8.1.3 Routing Table Performance

We implement parallel matching engines for each table entry. Each entry comparison takes 1 cycle, and the matching result is calculated by a priority selector in another cycle. Note that this implementation has a two cycles constant latency. On the other hand, the routing table is well pipelined, so in every cycle it can accept a message and look up its output port. Thus, the message-per-second throughput is the same as the clock frequency. Table 3 shows the area cost and max frequency of the routing table with different number of entries. Our implementation with typical 32 entries consumes 0.83% of ALMs, that is 3.3%-6.6% for a typical 4-8 port implementation. The max frequency is high enough for serving the shortest DUA message at a rate of over 10GBps per port. As the number of entries increases to 64 and 128, area cost increases linearly and the frequency dose not decrease much.

8.1.4 Latency Overhead

We use FPGA 1 to send data through DUA to FPGA 1* in Fig. 9. Specifically, DUA first transmits data from FPGA 1 to FPGA 4 through FPGA Connect, and then to FPGA 4* (the 4th FPGA on Server 2) through LTL, and then to FPGA 1* through FPGA Connect. We measure the end-to-end communication latency including DUA and all the traversing stacks, as well as the break-down latency for each stack.

Fig. 11(a) shows the average latency of each part. Under various packet sizes, DUA only incurs less than 0.2μs

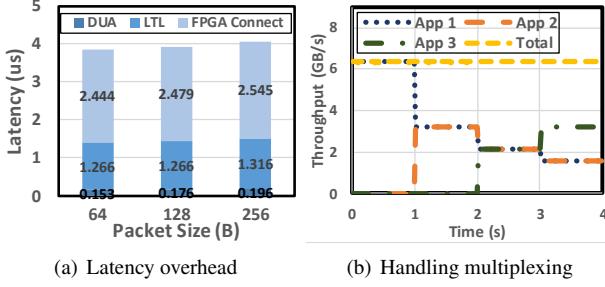


Figure 11: DUA only adds little latency overhead and handles multiplexing well.

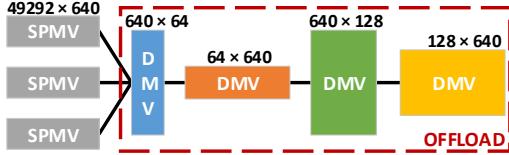


Figure 12: Deep Crossing Model in our experiment

latency, which is a negligible overhead compared with the other stacks’ latency in the end-to-end communication. Note that DUA is fully pipelined and can achieve line rate, incurring no throughput overhead.

8.1.5 Handling Multiplexing

We build three applications (App 1,2,3) on the same FPGA all using DUA, to test whether DUA can handle multiplexing well. Specifically, App 1 starts from second 0, keeps writing data to host DRAM. At second 1, App 2 also starts to write data to host DRAM. At second 2, App 3 starts to send data to another local FPGA through FPGA Connect. In this scenario, all three applications multiplex the same PCIe physical interface, App 1 and 2 multiplex the same DMA stack, and DMA stack and FPGA Connect stack multiplex the same physical PCIe interface. DUA adopts fair scheduling policy for all DUA connections, and changes to weighted share scheduling (share ratio 1:1:2) starting from second 3.

Fig. 11(b) shows the results. During the experiments, the total throughput of all applications always achieves the maximum throughput of the PCIe physical interface (§7.2.2). When new applications join, DUA successfully balances the throughput between them. Specifically, App 1 and 2 each achieve ~ 3.2 GBps between second 1 and 2, all three applications get ~ 2.2 GBps between second 2 and 3. Also, when we change the scheduling policy to 1:1:2 weighted share at second 3, the three applications quickly get their respective expected throughput.

8.1.6 Deep Crossing

8.2 Applications Built With DUA

In this section, we present two applications built on DUA, demonstrating that DUA can ease the process of building high-performance multi-FPGA applications.

Deep crossing, a deep neural network, was proposed in [27] for handling web-scale application and data sizes. The model trained from the learning process is deployed in Bing

Table 4: ALMs cost (%) of different dense part

	Base	D1	D2	D3	D4	All
Parall = 32	26.0	11.0	9.4	11.0	9.8	67.2
Parall = 64		20.8	19.1	20.8	19.6	106.3

Table 5: Latency (μ s) of the dense part in FPGA.

	D1	D2	D3	D4	Comm.	E2E
Single FPGA	7.27	6.58	13.30	12.96	N/A	40.12
Two FPGAs	4.17	3.48	7.22	7.09	1.419 (FC)	23.38
					3.354 (LTL)	25.32

to provide web services. The critical metric for the deep crossing model is latency since it impacts service response time.

Fig. 12 shows the structure of the deep crossing model in our experiments. There are three sparse matrix-vector (SPMV) multiplications and four dense matrix-vector (DMV) multiplications. Each input data to the whole model is a 49,292 dimensional vector, as in [27]. The sparse part is a memory-intensive task while the dense part is computation-intensive, and the vector between each dense part is small. Therefore, we offload the dense parts to FPGA to reduce latency.

We implement all dense parts inside FPGA using OpenCL. In our implementation, for each matrix multiplication, there is an adjustable parameter called parallel degree (Parall), which determines the number of concurrent multiplications being done in one cycle. The larger Parall, the fewer cycles are needed to complete this matrix multiplication; meanwhile, the larger parall, the more FPGA logic resources are consumed. As shown in Tab. 4, if we implement the whole four DMVs in a single FPGA board, we can only offload the model with Parall = 32 because of FPGA resource limitations.

To achieve better latency, we use DUA to build a two-FPGA deep crossing accelerator. Specifically, we implement all the DMVs in the model with Parall = 64, and download the first two DMVs on one FPGA, and the other two DMVs into another FPGA. The two FPGAs are physically connected through both the intra-server PCIe network and Ethernet, with underlying stack FPGA Connect / LTL enabled. We use DUA interface to connect the DMVs logic on the two FPGA boards.

It only incurs 26 extra lines of OpenCL code to call the DUA interface to connect the two FPGA boards. Moreover, changing the communication method only requires changing the routing table, without any change to OpenCL code and thus eliminates hours of hardware recompilation time. Table 5 shows the latency results of the FPGA offloading (counting only the FPGA-related latency). The results show that the two-FPGA version built with DUA reduces the latency by $\sim 42\%$ (through FPGAConnect, FC for short in table) or $\sim 37\%$ (through LTL) compared to the single-FPGA version.

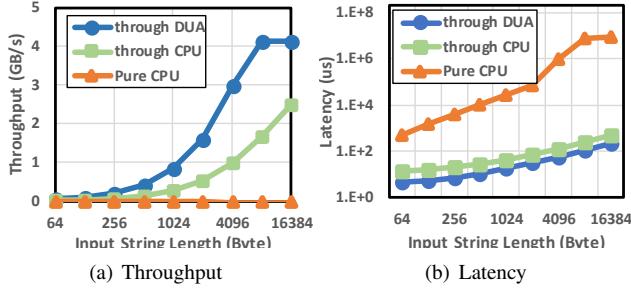


Figure 13: Performance of multi-regular-expression matching system.

8.2.1 Fast Multi-Regular-Expression Matching

A network intrusion detection system (IDS) [28–31] lies between the network and the host, intercepting and scanning the traffic to identify patterns of malicious behaviour. Typically, these patterns in IDS are expressed as regular expressions. Each pattern is denoted as a *rule*, and all patterns in the different stages together are called the *rule set*.

We have built a fast multi-regular-expression matching prototype which consists of three FPGAs over DUA. The boards are physically connected through PCIe within the same server. Each regular expression is translated into one independent NFA, and the NFA is translated into matching circuit logic using methods in [32]. We use DUA with underlying FPGA Connect stack to transfer data between these three FPGAs. Connecting the DUA interface only costs less than 30 lines of OpenCL code on each FPGA. We implement the whole core rule set [33] of ModSecurity in our system. The rule set contains 219 different regular expression rules in total. We randomly divide these 219 rules into three pattern stages with each stage containing 73 rules, and implement each stage in a single FPGA.² For each stage in each FPGA, we implement 32 parallel matching engines, with each engine matching one 8-bit character in one cycle. An input string goes to the next FPGA only after it finishes the matching in this stage.

For comparison, we also let these three FPGAs exchange data through CPU, without the direct communication method provided by DUA FPGA Connect. Also, we compare with the baseline performance, which uses a single 2.3GHz Xeon CPU core and the widely used PCRE [34] regular expression engine.

We generate different length of input strings for matching, to evaluate the throughput and latency of the whole regular expression matching system. String lengths vary from 64 to 16K byte, with contents randomly generated. In our experiment, we use a single CPU core to DMA the input string into the matching system instead from the network. We get the matching results back using DMA and count the performance in the same CPU core.

²Note that the number of rules in each FPGA does not affect the matching speed, since all rules are matched in parallel.

Fig. 13 shows the result. Enabled by direct communication through DUA, our regular expression matching system (denoted as “through DUA”) achieves about three times higher throughput and lower latency compared to FPGAs exchanging data through CPU (denoted as “through CPU”). Benefiting thus from the direct communication through pure hardware, our system almost reaches the maximum possible throughput of input string DMA when the string length exceeds 8KB. On the contrary, when exchanging data through CPU, the CPU needs to frequently read matching results from former-stage FPGAs and send strings to next-stage FPGAs, which becomes the performance bottleneck. Also, we can see that for such a complex rule set, pure CPU can only achieve very low performance. Note that our throughput is slightly lower than the maximum FPGA Connect speed (§7.2.2) due to the following reasons: 1) software libraries for DMA incur overhead compared with pure physical interface; 2) although the data paths through PCIe are different for DMA input/output data to CPU and exchanging data between FPGAs, they all use the same PCIe HIP to issue operations which incurs some contention.

9 Related Work and Conclusion

While prior work has aimed to provide abstractions and simplified FPGA communications, to the best of our knowledge, DUA is the first unified communication architecture for FPGAs to access all available data center resources, regardless of their location and type. Catapult shell [10, 11], Amazon F1 [2] and its derivatives provide an abstract interface between FPGA logic, software and physical interfaces, but it remains far from being the unified communication architecture provided by DUA. The Altera/Intel Avalon bus [35] and the AXI bus [36] used by Xilinx provide a unified interface for FPGA applications, but they are designed solely for on-chip buses, not the scale of data center network. TMD-MPI [37] provides a general MPI-like communication model for multi-FPGA systems, but it only targets communication between FPGAs rather than general resource access. Also, it is implemented in software and requires the CPU. The recent LTL [11] work targets the communication between FPGAs in data center through Ethernet. DUA can leverage and support all these works as communication stacks to improve connectivity.

Providing a communication architecture with unified naming and common interface has proven widely successful in IP networks. In this paper, DUA takes a first step to bring this communication architecture into the FPGA world. Our experiments show that DUA has negligible impact on performance and area, and greatly eases the programming of distributed FPGA applications that access data center resources. Note that though this work is targeted to FPGAs, there is no reason why it cannot be applied to other devices as well.

References

- [1] Microsoft Goes All in for FPGAs to Build Out AI Cloud. <https://www.top500.org/news/microsoft-goes-all-in-for-fpgas-to-build-out-cloud-based-ai/>.
- [2] Amazon EC2 F1 Instances. <https://aws.amazon.com/ec2/instance-types/f1/>.
- [3] Intel, Facebook Accelerate Datacenters With FPGAs. <https://www.enterprisetech.com/2016/03/23/intel-facebook-accelerate-datacenters-fpgas/>.
- [4] Data Engine for NoSQL - IBM Power Systems Edition White Paper. <https://www-01.ibm.com/common/ssi/cgi-bin/sialias?htmlfid=POW03130USEN>.
- [5] Baidu Takes FPGA Approach to Accelerating SQL at Scale. <https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/>.
- [6] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. *ACM SIGPLAN Notices*, 49(4):471–484, 2014.
- [7] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–23. IEEE, 2014.
- [8] Alibaba, Intel introduce FPGA to the Cloud. <https://luxeelectronicscomblog.wordpress.com/2017/03/13/alibaba-intel-introduce-fpga-to-the-cloud/>.
- [9] Tencent FPGA Cloud Computing. <https://www.qcloud.com/product/fpga>.
- [10] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.
- [11] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A Cloud-Scale Acceleration Architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.
- [12] Microsoft demonstrates the world’s ‘first AI supercomputer,’ using programmable hardware in the cloud. <https://www.geekwire.com/2016/microsoft-touts-first-ai-supercomputer-using-programmable-hardware-cloud/>.
- [13] Ray Bittner, Erik Ruf, and Alessandro Forin. Direct gpu/fpga communication via pci express. *Cluster Computing*, 17(2):339–348, 2014.
- [14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [15] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.
- [16] Jian Gong, Tao Wang, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, and Jason Cong. An efficient and flexible host-fpga pcie communication library. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.
- [17] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.
- [18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.
- [19] Michael Johannes Jaspers. Acceleration of read alignment with coherent attached fpga coprocessors. 2015.
- [20] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017.
- [21] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. Accelerating big data analytics using fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 164–164. IEEE, 2015.
- [22] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling fpgas in hyperscale data centers. In *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on*, pages 1078–1086. IEEE, 2015.
- [23] Malte Vesper, Dirk Koch, Kizheppatt Vipin, and Suhaib A Fahmy. JetStream: an open-source high-performance PCI express 3 streaming library for FPGA-to-host and FPGA-to-FPGA communication. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–9. IEEE, 2016.
- [24] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy M Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network. *SIGCOMM*, 45(4):183–197, 2015.
- [25] Jiao Zhang, Fengyuan Ren, Ran Shu, and Peng Cheng. Tfc: token flow control in data center networks. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 23. ACM, 2016.
- [26] Stratix 10 - Overview. <https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.htm>.
- [27] Ying Shan, T Ryan Hoens, Jian Jiao, Hajjing Wang, Dong Yu, and JC Mao. Deep crossing: Web-scale modeling without manually crafted combinatorial features. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 255–262. ACM, 2016.
- [28] Snort - Official Site. <https://www.snort.org/>.
- [29] The Bro Network Security Monitor. <https://www.bro.org/>.
- [30] ModSecurity: Open Source Web Application Firewall. <https://www.modsecurity.org/>.
- [31] Application Layer Packet Classifier for Linux. <http://17-filte.r.sourceforge.net/>.
- [32] Reetinder P S Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. pages 227–238, 2001.
- [33] OWASP ModSecurity Core Rule Set (CRS). <https://modsecurity.org/crs/>.
- [34] PCRE - Perl Compatible Regular Expressions. <http://www.pcre.org/>.
- [35] Avalon Interface Specifications. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

- [36] AXI Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf.
- [37] Manuel Saldana and Paul Chow. TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6. IEEE, 2006.

Appendices

A OpenCL sample code to use DUA API

Figure 15 shows an example Verilog code for application using DUA interface to initiate a connection and send data. It first generates a CONNECT primitive and then waits for the response. If the connection is successfully established, it records the source UID and port number, then enters the sending data state. If the connection setup fails, it will generate another CONNECT primitive. Note that the source address and port is not available before the connecting setup, thus this field is reserved when issuing a CONNECT command. And the response of CONNECT command will contain the corresponding fields.

B OpenCL sample code to use DUA API

Fig. 14 shows an OpenCL sample code pieces to use DUA API. *DUA_Msg* is a union storing DUA messages to be sent in current clock (see Fig. 7). *dua_tx* is a reserved channel that automatically connects to the request interface of our DUA I/O interface (Fig. 4). *simple_write ()* function writes 32B data to address *DST_ADDR* of the resource whose UID is *DST_UID* through DUA interface.

```
void simple_write () {
    DUA_Msg msg;
    bool is_header = true;
    while (1) {
        if (is_header) {
            msg.header.length = 32;
            msg.header.type = WRITE;
            msg.header.src_uid = SRC_UID;
            msg.header.dst_uid = DST_UID;
            msg.header.dst_addr = DST_ADDR;
            is_header = false;
        }
        else {
            msg.raw = data;
        }
        write_channel_altera(dua_tx, msg.raw);
    }
}
```

Figure 14: OpenCL sample code to use DUA API

```
assign rx_header = rx_data_in;

assign connect_header.type = CONNECT;
assign connect_header.dst_uid = dst_uid;
assign connect_header.dst_port = dst_port;

assign send_header.dst_UID = dst_UID;
assign send_header.src_UID = src_UID;
assign send_header.type = SEND;
assign send_header.length = 48;
assign send_header.src_port = src_port;
assign send_header.dst_port = dst_port;

always @ (posedge clk) begin
    tx_valid_out <= 1'b0;
    tx_first_out <= 1'b0;
    tx_last_out <= 1'b0;
    case (state)
        SETUP_CONNECTION: begin
            if (tx_ready_in) begin
                tx_data_out <= connect_header;
                tx_valid_out <= 1'b1;
                state <= WAITING_RESPONSE;
            end
        end
        WAITING_RESPONSE: begin
            if (rx_valid_in
                && rx_data.type == CONNECT) begin
                if (rx_data_in.status == SUCCESS) begin
                    src_UID <= rx_data_in.src_UID;
                    state <= SENDING_HEADER;
                end
                else begin
                    state <= SETUP_CONNECTION;
                end
            end
        end
        SENDING_HEADER: begin
            if (tx_ready) begin
                tx_data_out <= send_header;
                tx_valid_out <= 1'b1;
                tx_first_out <= 1'b1;
                state <= SENDING_DATA_0;
            end
        end
        SENDING_DATA_0: begin
            if (tx_ready) begin
                tx_valid_out <= 1'b1;
                tx_data_out <= data_0;
                state <= SENDING_DATA_1;
            end
        end
        SENDING_DATA_1: begin
            if (tx_ready) begin
                tx_valid_out <= 1'b1;
                tx_data_out <= {128'h0, data_1}; // 128bits
                tx_last_out <= 1'b1;
                state <= CLOSE_CONNECTION;
            end
        end
        CLOSE_CONNECTION: begin
            // close connection logic
        end
    endcase
end
```

Figure 15: DUA API usage example

Stardust: Divide and Conquer in the Data Center Network

Noa Zilberman
University of Cambridge

Gabi Bracha
Broadcom

Golan Schzukin
Broadcom

Abstract

Building scalable data centers, and network devices that fit within these data centers, has become increasingly hard. With modern switches pushing at the boundary of manufacturing feasibility, being able to build suitable, and scalable network fabrics becomes of critical importance. We introduce Stardust, a fabric architecture for data center scale networks, inspired by network-switch systems. Stardust combines packet switches at the edge and disaggregated cell switches at the network fabric, using scheduled traffic. Stardust is a distributed solution that attends to the scale limitations of network-switch design, while also offering improved performance and power savings compared with traditional solutions. With ever-increasing networking requirements, Stardust predicts the elimination of packet switches, replaced by cell switches in the network, and smart network hardware at the hosts.

1 Introduction

For the last ten years, cloud computing has relentlessly grown in size [28]. Nowadays, data centers can host tens of thousands [75] of servers or more. The complexity of the data center network (DCN) has grown together with the scaling of data centers. While scale has a direct effect on the bisection bandwidth, it also affects latency, congestion, manageability and reliability. To cope with these demands, network switches have grown in capacity by more than two orders of magnitude in less than two decades [85].

The computing community faced the end of Dennard’s scaling [35] and the slowdown of Moore’s law [60] over a decade ago, prompting a move to multi-core and many-core CPU design [68]. Similar challenges are faced by the networking community today. In this paper, we discuss limitations on network device scalability, and assert that in order to continue to scale DCN requirements, data center network devices need to be significantly simplified. We introduce Stardust, a DCN architecture based on the implementation of network-switch systems on a data center scale.

In Stardust, we divide the network into two classes of devices: top-of-rack (ToR) devices maintain classic packet-switch functionality, while any other device in the network is a simple and efficient cell switch. We refer to the part of the network created by these simple switches as the *network fabric*. Devices within the network fabric do not require complex header processing and large lookup tables, have minimal queueing and reduced network interface overheads. Their data path requires minimum clock frequency, regardless of packet size. Creating a network fabric that is made completely of a single type of a simple switch is a radical move from common approaches, whereby each network device is a full-fledged, autonomous packet switch. To conquer the DCN scalability and performance requirements, we build upon a number of insights, drawn from silicon through the network level:

- The DCN can be compared to a large-scale network-switch system, where complex routing decisions are taken at the edge, and simple forwarding is applied within the fabric. By using only basic routing mechanisms within the core of the DCN, significant network-switch resources can be saved, while maintaining functionality.
- The increase in port rates, utilizing an aggregation of N serial links per port as a norm, limits the scalability of the DCN. By using discrete, rather than aggregated links, the scale of Fat-tree networks can improve by $O(N^2)$.
- Unnecessary packet transmissions can eventually lead to packet loss. Credit-based egress scheduling can prevent packet loss due to congestion, and increase fairness between competing flows.
- Congestion can happen within underutilized networks due to imperfect balancing of flows across available paths. By evenly distributing traffic across all links, optimum load balancing can be achieved.
- Network-switch silicon is over-designed: it is underutilized for small packet sizes, and packet sizes not aligned to the internal data path. By batching packets together, and chopping them into cells that perfectly match internal data-path width, maximum data path utilization can be achieved.

Stardust has no sense of flows, and it does not require route installation. The network fabric is completely distributed, with no need for complex software-defined networks (SDN) management, central control or scheduling. Stardust is also protocol agnostic, reducing silicon level requirements by 33%. The resulting network provides full bisection bandwidth. It is lossless, load balanced and provides significant power and cost savings.

Switch-silicon vendors have produced, and shipped in volume, chipsets such as we exploit in Stardust (e.g., [20, 61]); we do not claim these designs as a contribution. Similarly, switch vendors have produced single-system switches that internally use such chipsets to produce the illusion of a single many-port packet switch (e.g., [14, 43, 26]). Our contribution, in this paper, is to architect and implement our approach on a data center scale, using such commodity chipsets.

To summarize, we make the following contributions:

- We explore performance and scalability limitations in current DCN network devices (§2), motivating the need for simpler network devices.
- We introduce Stardust, an architecture offering a better approach to realizing DCN (§3, §4), and discuss its advantages on a data center scale (§5).
- We evaluate Stardust (§6): experimental measurements on $O(1K)$ s of port environments, event simulations of $O(10K)$ s port systems and an analysis of large scale $O(100K)$ s port networks, illustrating the advantages of Stardust.
- We propose a future vision of DCNs (§8), where ToR functionality is reduced to fit within network interface cards, with ToR switches turned into cell switches.

2 Motivating Observations

The design of scalable network devices has hit a wall. If a decade ago the main question was “*how can we implement this new feature?*”, now the question is “*is this feature worth implementing given the design constraints?*”. We discuss three types of limitations to network device scalability: resources, I/O, and performance. Our observations are orthogonal, providing trajectories to improving scalability.

2.1 The Resource Wall

Every network device is limited in its die size and its power consumption. Chip architects need to balance the design to avoid exceeding these resources limitations; die size is not just a function of cost, but also of manufacturing and operating feasibility, and reliability.

Specializations and simplifications enabled DCNs to evolve considerably from being an *Internet-in-miniature* using complex router and switch architectures. The data center’s unique administrative environment allows Stardust to present a *dispersed switch* design, eliminating the need to support in each network hop a full suite of protocols, traffic-control, or full interface flexibility.

Previous large switches have been custom-made machine-

room-wide HPC interconnects, or multi-chassis systems interconnecting interface line-cards and control processors. Such platforms, (e.g., [27, 43]) implement advanced packet operations on the line-cards, interconnected using a simple fabric consisting of minimal packet queuing or processing. In DCN, the ToR switches take the role of the line-cards, while the “interconnect fabric” is the spine and leaf switches interconnecting all ToR devices. This model of the DCN equally applies to Stardust.

Fig. 1 illustrates the difference between the two approaches: In both cases, the first switch (the ToR) provides packet processing, queueing, and full network interfaces. However, in the typical data center network approach, traversing a Fat-tree will include going through all functions (ingress and egress processing, queueing, scheduling,...) in each and every hop. In contrast, with Stardust, scaled up from the single system approach, these functions will be used only at edge devices. By eliminating stages, logic and memory, consuming considerable silicon area [19], significant resources can be saved. While the I/O remains the same, the network interface is considerably smaller; there is no need to support multiple standards (e.g., 100GE, 400GE), and a single MAC functionality is sufficient. Stardust benefits include reducing both network-wide latency and network-management complexity as it practically presents as a single (large) switch. Shallow buffering is not entirely eliminated (see §6).

On the order of microseconds [16, 69] rather than milliseconds, the reduced fabric latency of a Stardust DCN behaves as a single hop (albeit across a larger *backplane* fabric). This allows applications to continue operating unchanged

Observation: *Significant resources can be saved by simplifying the network fabric and adopting a scaled-up single-system approach.*

2.2 The I/O Wall

The maximum number of network interfaces on network devices has grown by an order of magnitude over the last decade, climbing from 24 ports [31] to 260 ports [64]. Device packages are big, $55mm \times 55mm$ [79] or more, bigger than high-end CPUs [44]. Such big packages raise concerns about manufacturing feasibility, cost and reliability. It is unlikely the I/O number will continue to scale at the same rate.

The second part of the I/O problem is that a network port is not necessarily a single physical link. A *Link Bundle* (l) is the number of serial links connecting two hops. For example, connecting two switches using a port of 100GE CAUI4, four lanes at 25Gbps each, is a *link bundle of four*. Link bundling is a common practice in high-speed interfaces, used to increase the bandwidth of a logical port through the multiplexing of information from multiple physical links. This practice overcomes the signaling and physical limitations of the media (e.g., copper), and also applies to integrated photonics devices for network switches [17].

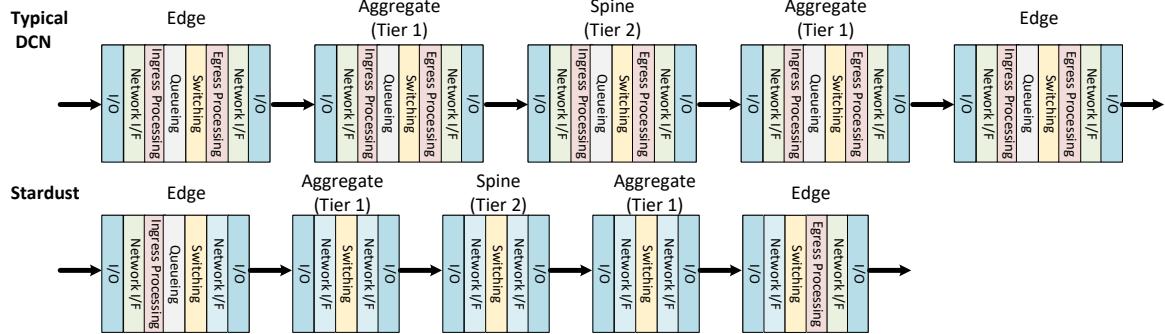


Figure 1: Traversing through a Fat-tree network, in a typical data center network (top) and in Stardust (bottom). In a network with n tiers, a typical DCN requires $n \times (\text{Ingress processing} + \text{Queueing} + \text{Scheduling} + \text{Egress Processing})$. Stardust requires just $1 \times (\text{Ingress processing} + \text{Queueing} + \text{Scheduling} + \text{Egress Processing})$.

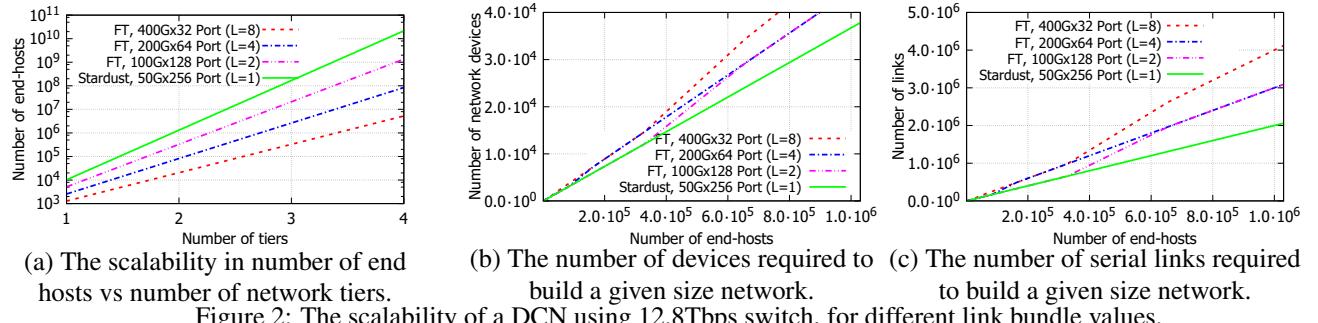


Figure 2: The scalability of a DCN using 12.8Tbps switch, for different link bundle values.

For the rest of this paper, we refer to each switch port as a link bundle, and a switch radix indicates the number of ports (link-bundles) in a switch. We use a Fat-tree topology to demonstrate the effect of switch radix and link bundling on DCNs scalability, and provide in Appendix A the mathematical justification. As each end host connects to a single ToR, link bundling from the host has no effect on scalability.

Assume a network that consists of edge network devices (e.g., ToR) and a network fabric connecting all edge devices. The number of layers within the network fabric is the number of *Tiers*. A network built from only ToR and aggregation devices is a 1-Tier network, whereas the common Fat-tree Edge-Leaf-Spine structure is a 2-Tier network. A network can be extended to include more edge devices by introducing additional tiers. Fig. 2(a) demonstrates the size of the DCN required to connect a given number of end hosts, as a function of the link-bundle. In this figure, we assume a network device of 12.8Tbps, using different link bundling configurations, ranging from $256 \times 50\text{Gbps}$ ($L = 1$) to $32 \times 400\text{Gbps}$ ($L = 8$), and assuming 40 servers connected to each edge device using 100Gbps ($L = 2$) links. A link bundle of one enables a 1-Tier network of over ten thousand servers, whereas a 1-Tier network with a link bundle of eight is limited to an eighth of this number of hosts. For a 2-Tier network, a link bundle of eight allows connecting only 20K hosts, compared with $\times 64$ the number of hosts using a link bundle of one. The link bundle affects not just the overall scalability of the network, but also the number of network devices and links required to build a given size network (Fig. 2). §5.1 and Appendix A expand on link bundling.

Observation: For a given switch bandwidth, a link bundle of one will allow the optimum number of switches in a DCN.

2.3 The Performance Wall

Network-silicon devices support throughput in the order of 12.8Tbps [21]. This is equivalent to $32 \times 400\text{Gbps}$ ports or 19.05Gpps (19.05 billion packets per second) for 64B packets, and 5.8Gpps for 256B packets. The clock frequency of switch silicon is in the order of 1GHz, meaning that even if a new packet can be processed every clock cycle, between 6 (for 256B) and 20 (for 64B) packets need to be processed in parallel. Even for 1500B packets, more than a single packet needs to be processed every clock cycle. To address this problem, network silicon today either does not support full line rate for all packet sizes, or implements multiple data-plane cores within each device, connected through an on-chip switching module or a shared memory [39, 62]. See Appendix B for a detailed analysis.

Observation: To support full line rate for all packet sizes, network devices need to process multiple packets each and every clock cycle.

In Fig. 3 we explore the scaling requirements for a switch’s pipeline. We assume two 12.8Tbps devices: one device implements standard packet switching, and the other device (Stardust) optimally packs data into the pipeline, in units that are equal to its data path’s width. The internal data path frequency is 1GHz, and the data path is 256B wide¹. Fig. 3 shows the number of parallel buses, or processing

¹Data path width is upper-bounded by timing and resources limitations, and lower-bounded by performance requirements.

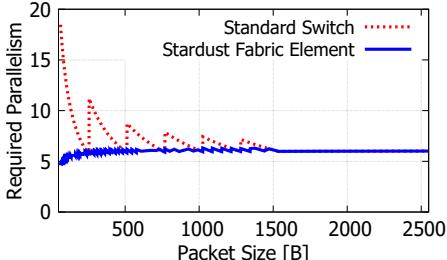


Figure 3: The required parallel processing in a standard switch and in a Stardust Fabric Element. We assume 12.8Tbps switch, 256B wide bus, 1GHz data path frequency.

cores, required to support the device’s 12.8Tbps. For small packets, a design optimally packing data (Stardust) outperforms a packet-based design by a factor of $\times 4$. Packing data provides 41% improvement for 513B packets, and 18% for 1025B packets. Increasing the data path width eases the requirements for large packets, but not for small ones.

Packets smaller than the internal data path width (which can be over 50% of the traffic, assuming a 256B wide bus [74]), benefit from data packing even more, as the on-wire inter packet gap (IPG) and preamble are applied only once per packed unit, rather than once per packet, increasing the overall packet rate. From a silicon design standpoint, expecting standard switches within the DCN fabric to process all packets at wire rate requires a huge amount of over design for the task performed, with substantial penalties.

Observation: *Packing data to optimally fit data path width achieves higher throughput, for all packet sizes, at lower clock frequencies. This simplifies switch design and enables better scalability over time.*

3 Architecture

Stardust is a new architectural paradigm for DCNs, addressing the resource, I/O and performance limitations discussed in §2. Stardust scales up the switch-system to the DCN.

In order to support a scalable network, we define two types of devices: the Fabric Adapter device, and the Fabric Element device. The Fabric Adapter device, such as StrataDNX Jericho [22], plays a similar role to a ToR device. The Fabric Element device, such as StrataDNX Ramon [23], is used as the building block of the network fabric. The Fabric Adapter and Fabric Element used in the architecture differ from the commonly used packet switches in at least one important aspect: the Fabric Element is not a packet switch but a simple cell switch, and the Fabric Adapter takes every incoming packet and chops it into data units, which we refer to as cells. Cells are still “packet” units, but with payload and header sizes optimized to fit the Fabric Element pipeline.

3.1 Constructing a Stardust Based Network

The Stardust architecture is Clos based [30]. Fabric Adapters are used as the first and last stage devices, whereas the middle stages are composed of Fabric Elements. For the rest of

this paper we limit the discussion to Fat-tree topologies [51], a special case of a folded Clos topology.

The building of a network starts by connecting host machines, (e.g., servers) to a Fabric Adapter device that acts as a ToR switch. Fabric Elements are used in the aggregation and spine layers. The number of uplink connections from the Fabric Adapter to the aggregation layer of the network fabric depends on the desired uplink bandwidth and over-subscription ratio. In Stardust every physical link is an independent entity and is not bundled.

The network fabric is made only of Fabric Elements. Each Fabric Element has k full duplex links. In the aggregation layer, half of the links are connected to the Fabric Adapter and half of the links are connected to a spine layer. In the spine layer, all the Fabric Element’s k links are connected to the aggregation layer.

We define the fabric speed up, fs , as the ratio between links’ capacity to the network fabric and links’ capacity to the hosts. $1/fs$ is the link *utilization*. A network fabric may be under-subscribed with $utilization < 1$ or fully utilized, from any device to any device, though some redundancy is typically applied. Long-term over-subscription from the hosts to the Fabric Adapter is handled as in any ToR, i.e., packets will be dropped in the Fabric Adapter. Short-term over-subscription is absorbed by the Fabric Adapter’s buffers. Similarly, over-subscription is allowed between tiers in the network fabric.

3.2 Dynamic Cell Forwarding

Packets arriving from the host to a Fabric Adapter are parsed, and the destination is identified. Every destination is mapped to a destination Fabric Adapter and port. Each Fabric Adapter holds a reachability table indicating which links can be used to reach the destination. The Fabric Adapter collects multiple packets and chops them into bounded-size (e.g., 256B) cells. The cells hold a small header including the destination and a sequence number that allows reassembling cells into packets.

A pivotal idea in Stardust is dynamic cell forwarding: each cell is sent over a different link, while the Fabric Adapter arbitrates over all the links in the reachability table marked as leading to the destination. At each Fabric Element stage, the same process is repeated by load balancing among all available paths converging on the destination Fabric Adapter. This dynamic cell forwarding is a radical departure from traditional static routing architectures, where a flow (or flowlet) is bound to a specific path according to its headers, leading to complex issues of provisioning, congestion avoidance, failure recovery, routing table configuration and more. Load balancing on the links is nearly perfect and enables an optimal use of fabric resources. The fabric behaves as a uniform transport media, agnostic to incoming traffic patterns, or to flow granularities. Dynamic forwarding may introduce out-of-order fabric traversal. However, the load-balancing cre-

ates a limit on the extent of out-of-order mismatching, and reordering is managed at the destination Fabric Adapter (§4).

3.3 Buffering And Scheduling

Stardust is based on the combination of packet buffering at the edge and a distributed scheduled fabric. The architecture uses virtual output queues (VOQs) to queue packets arriving to the Fabric Adapter. Each destination port (and priority) has an assigned VOQ. The Fabric Adapter uses a buffering memory, in the order of megabytes to gigabytes per Fabric Adapter, storing the queued packets. Empty VOQs do not consume buffering resources.

Distributed traffic scheduling is used in Fabric Adapter’s Egress. Non-empty VOQs request from the destination port permission to send traffic, and the port scheduler at the destination Fabric Adapter is responsible for generating credits without exceeding the port’s capacity, as it has a view of all of the VOQs toward its ports. Stardust supports multicast and broadcast, but this support is beyond this paper’s scope.

A credit entitles a VOQ to release an amount of data, (e.g., 4KB) to the fabric. The VOQ dequeues packets up to the credit size. The amount of surplus data is stored for later accounting. The next VOQ is selected by the order of credit arrival and a priority. The ingress packet buffer and sufficient egress memory enable absorbing long bursts of data at the ingress and momentary bursts of data at the egress.

3.4 Packet Packing

Using cells can be quite wasteful: sending packets that are just one byte bigger than a cell size can lead to 50% waste of throughput. *Packet packing* avoids such waste, effectively increasing the packet processing rate, but operating on packed packets rather than on individual ones (Appendix B). When a VOQ receives a credit to send packets, it chops the packets in the queue into cells while treating the entire burst of data as a unit. As a consequence, a cell may include multiple packets or multiple packet fragments (Fig. 4). Packet packing is feasible only within the same VOQ.

4 Device Architecture

4.1 Fabric Adapter

The Fabric Adapter (Jericho) is the edge unit interconnecting hosts with the network fabric. The Fabric Adapter resembles a ToR switch, providing (programmable) header-processing, scheduling and switching functionality. The size of its tables resembles a ToR. However, it has additional functionalities:

- Chopping incoming packets from the hosts into cells, and sending them to the network fabric
- Reassembling cells arriving from the fabric into packets
- Providing scheduling services by sending credits

Fig. 5 depicts an architecture of a Fabric Adapter device. On the Fabric Adapter ingress, packets arrive from the host through multiple interfaces, (e.g., 40GE, 100GE). The packet header is parsed and packets are stored in VOQs, ac-

cording to the destination Fabric Adapter and port. The number of VOQs is determined by the total number of downlink ports on Fabric Adapters and the number of traffic classes (rather than the number of routable IP addresses).

Non-empty VOQs send special control messages to an egress scheduler at the destination Fabric Adapter. The egress scheduler is consequently aware of all VOQs toward its ports. It sends to the ingress Fabric Adapter a “credit”, (e.g., 4KB). The total rate of credits matches the egress port’s rate. The egress scheduler implements various QoS policies, typically a combination of round-robin, strict priority and weighted among VOQs of different Traffic Classes, allowing a flexible allocation of fabric and egress port capacities. While Jericho uses a proprietary scheduler, similar schedulers have been described before, e.g. in [56, 24]. To keep the egress buffer full, compensating for propagation and processing delays, the credit rate is set slightly above the egress port bandwidth (e.g., 2%) and slightly below the fabric speed-up, to avoid congestion (§ 6.2). When the egress buffer is close to full, the scheduler stops sending credits to the VOQs and resumes as packets are drained.

Packets are dequeued from the ingress VOQ when a credit arrives. The VOQ dequeues a number of packets, matching the credit size. Packets are allocated to cells of a limited size (e.g. up to 256B), distributed across multiple links toward multiple Fabric Elements. The cell size matches the width of the Fabric Element pipeline, and therefore fabric performance is minimally affected by packet size distribution (§6).

At the network’s egress, cells arrive from Fabric Elements over multiple links and are reassembled into packets. Cells and packets can arrive out of order, and many hardware-based solutions exist (e.g., [33, 15, 32]). However, as credits are spread evenly over time, the number of concurrent interleaved packets at the egress reassembly is bound by Fabric Element queue size and does not scale with network size, or the number of VOQs. If a packet reassembly timer expires, e.g., due to a link error, the packet is discarded.

The minimum credit size is set by the Fabric Adapter’s output bandwidth and the credit generation rate of its scheduler. For example, for a 10Tbps Fabric Adapter, using 1GHz clock frequency and generating a credit every two clocks, the minimum credit size will be $10Tbps/(1GHz/2) = 2000B$. The maximum credit size dictates the amount of memory required at the devices. Consider a flow control to a port: the egress memory will need to absorb the data in flight from source Fabric Adapters, which is a function of credit size and latency across the network (from last credit generated to last bit of data arriving). To minimize the amount of Fabric Adapter’s egress memory, credit size should be at the same scale of the minimum credit size, as we validate (§6.2).

4.2 Fabric Element

The network fabric is composed only of Fabric Elements (Ramon). Fabric Elements connect to either Fabric Adapters

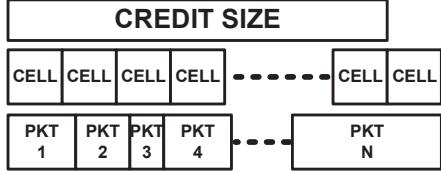


Figure 4: The relations between packets, cells and credits.

or other Fabric Elements. A Fabric Element handles only fixed size cells, and does not parse packets, while providing bare minimum buffering and scheduling. A possible implementation of a Fabric Element device is shown in Fig. 6.

The Fabric Element is a radical simplification of DCN switches. It eliminates the logic and large tables associated with protocols. There is also no need for complex software configuring routing tables, handling bandwidth provisioning or protection recovery. Additional logic handling congestion avoidance and queue management is redundant as well.

In Stardust, a cell is transmitted on a single link in its entirety (rather than over a link bundle) with a bounded cell size and a small header that indicates the destination Fabric Adapter. Packing packets into cells guarantees that only a very small fraction of the cells are smaller than the maximum cell size (e.g., 256B). Thus, the Fabric Element data path can be throughput optimized, regardless of packet size, only bounded by technology and resource constraints.

Every Fabric Element holds a simple forwarding table that relates every destination Fabric Adapter to an outgoing Fabric Element link. If a Fabric Adapter is accessible through multiple links, the Fabric Element load-balances the traffic among them. As the destination is a Fabric Adapter and not an end-host, the size of the table can be two orders of magnitude smaller than a typical routing table. Beyond this, the Fabric Element is essentially two $k \times k$ crossbars (e.g., 256×256), one for data cells and one for control messages.

The forwarding table is automatically maintained by hardware exchanging special reachability control messages, where each device, Fabric Adapter and Fabric Element, advertises itself to all directly connected network-fabric devices. The reachability messages are sent periodically. If no reachability messages are received on a link periodically, it is considered failed. Topology changes are incorporated in the forwarding tables of the entire fabric in order of hundreds of microseconds (see §5.9). This automatic setup in hardware of forwarding tables, and the relative low latency of the process, sets it apart from past proposals of extending the internal structure of a packet switch to DCN scale.

As cells from different sources may compete on an output link, shallow buffering is required within the device (§4.2.1). If the link's queue is above a configured threshold, then

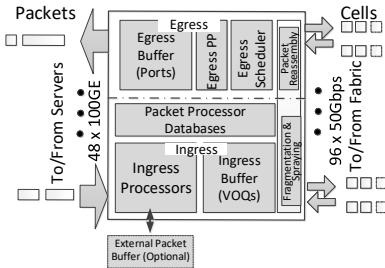


Figure 5: An architecture of a Fabric Adapter device

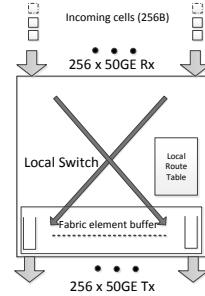


Figure 6: An architecture of a Fabric Element device.

Fabric-Congestion-Indication (FCI) bits are piggybacked on all cells. The FCI reaches the destination Fabric Adapter and throttles the credit rate. The shallow queuing within the Fabric Element guarantees low latency and jitter within cells' travel time of the network fabric. The shallow buffering within the fabric guarantees shallow buffering at the egress of the Fabric Adapter, and easier packet reassembly.

4.2.1 Fabric Element Queuing Analysis

The probability distribution of queue size per link is a vital characteristic of Stardust. First, it determines latency and jitter characteristics for traffic through the network fabric. Second, it determines the amount of memory the egress Fabric Adapter requires for absorbing in-flight data and packet reassembly. Third, the queue size statistics determine the maximal practical link utilization. Last, it affects the die size of the Fabric Element.

A discrete queuing model represents the Fabric Element queue behavior, as a function of link utilization. The time unit is “fabric cell time”, which is the time it takes to transmit a cell on a serial link. Consider a link’s queue at the last stage of a network fabric: on average, a cell will be added to a link with probability $1/fs$. However, the cells arrival process is bursty, as multiple Fabric Adapters may be sending cells simultaneously.

Cell arrival is bound by a Poisson arrival process with parameter $1/fs$. The discharge process is a constant one cell every “fabric cell time”. With M/D/1 queuing model, the probability of queue build-up on a link of size N can be approximated by $o(fs^{-2N})$ (as corroborated in §6.2). The Poisson arrival model is a worst-case scenario, as it assumes an infinite number of Fabric Adapters, and ignores the distribution effect of credits. As a limited number of Fabric Adapters generates traffic, both the burstiness of the arrival process and queue size’s tail probabilities decrease.

5 Stardust on a data center scale

5.1 Scaling, Tiers Reduction and Longevity

The Fabric Element utilizes a lean functionality, allowing it to pack more interfaces into the same area and enables higher radix devices. As Stardust eliminates the need for link bundling logic, then when, e.g., 400GE is used (link-bundle

of 8), the n^{th} tier of a Stardust based network can support $\times 8^n$ more ToR devices than a typical DCN, while still being non-blocking.

Since the traffic from a source to a destination Fabric Adapter is load balanced over all upstream links, the network fabric behaves like a single large pipe whose bandwidth is the bisection bandwidth (for example, 12.8Tbps pipe for 256 links of 50Gbps). Consequently, the fabric is agnostic to port rate or flow rate.

Stardust supports a gradual growth of a DCN. A data center operator will typically start with a small instance of a network fabric, then expand it over time. When designing a Stardust fabric, it is not necessary to populate the entire fabric from the start. The auto-construction of reachability tables, combined with dynamic forwarding, enables starting with a partially populated fabric, and adding Fabric Elements over time within a live network.

Stardust scales from switch-systems to data centers with 10K's to 100K's of nodes as the Fabric Elements, queueing and reassembly processes depend on device radix and are independent of network scale. Scheduling scales with the amount of on-chip memory. The number of VOQs in Jericho [22], as well as in other high-end switches [13, 29], is in the order of 100K, and increasing the number of VOQs is feasible (see §C).

5.2 Push Fabric vs. Pull Fabric

The following example shows the fundamental difference between a network fabric based on a “standard” Ethernet switch and a Stardust scheduled fabric. Consider a network fabric as in Fig. 7. On a single device there are two 100GE ports, A and B. From one input device 100Gbps are injected toward A (marked red), and additional 100Gbps are injected toward B (marked green). Additional 100Gbps are injected towards A from a second device (marked red).

In a fabric based on standard Ethernet switches, both of A’s 100Gbps flows are pushed into the fabric. Even with multi-path routing and load balancing, local congestion occurs in the middle stage switch and both A’s and B’s traffic is dropped. At the egress device only 66% of B traffic passes, despite not being congested. The problem is exacerbated if we consider Traffic Classes, where the throughput of the standard Ethernet switch based fabric may be half of Stardust (Appendix F).

Ethernet switch fabric drops are due to local optimization of the ToRs. The network fabric, made of autonomous Ethernet switches, makes local decisions using local congestion, oblivious to the end-to-end traffic. Thus, congested ports can block non-congested ports. While on a longer time scale packet losses may be reduced by congestion control mechanisms (e.g. ECN marking), this is not the case for short bursts.

In Stardust, the egress port scheduler of B sends 100Gbps worth of credit toward B’s flows, and the egress port sched-

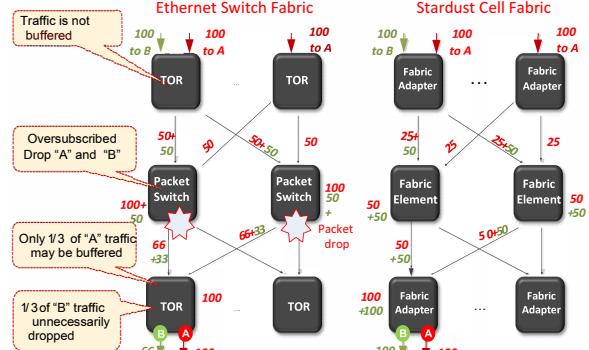


Figure 7: Packet drops in Ethernet switch fabric vs. Stardust.

uler of A sends 50Gbps worth of credits toward each of A’s sources. These flows, distributed through the fabric, reach the egress, and transmit 100Gbps at B and 100Gbps at A. The surplus 100Gbps of A’s traffic is stored at the ingress buffer, and from there it may be sent (if over subscription is momentary) or dropped (if persistent).

5.3 Optimal Load Balancing

Network fabric architectures based on typical Ethernet switches use flow hashing to load balance traffic across a network. This has been shown to be highly inefficient, allowing only 40%-80% utilization of the bisection bandwidth [5].

Recent research [41] started investigating packet level load balancing, giving rise to more uniform distribution, lower queues, higher utilization [8], lower latency and jitter, and therefore a much shallower buffer. However, even with packet level load balancing, there is $\times 144$ disparity between the smallest 64B packet and the largest 9KB packet, leading to substantial variance in queue length, and hence latency.

Stardust load balances fixed size cells, effectively achieving a perfect fluid model. At each Fabric Adapter, each packet is segmented to fixed size cells that are distributed in a round robin manner across all links leading to the destination port. Thus, the same amount of data is sent down each link. There are only two aspects that cause insignificant variations in link loads. First, cells at the credit-worth tails may be shorter. Their relative amount is negligible as multiple packets from the same VOQ are accumulated up to a threshold (e.g., 4KB), before fragmentation. Second, recurrent synchronization between packet transmit times and the load balancing mechanism bias load on some links for some destinations. However, the round robin arbiter traverses the Fabric Element links in a random permutation order, that is replaced every few rounds. Thus, the probability of a persistent synchronization is negligible.

5.4 Effective Buffer and Incast Absorption

Stardust provides a large and distributed packet buffer, enabling it to absorb incast. These packet buffers may be managed by smart queue management (SQM). When a destination port is oversubscribed, the data accumulates only at

source Fabric Adapters and the packet buffers available for storing data bursts are large. Even if the packet buffers are not sufficient, the source Fabric Adapter can avoid packet loss by sending flow control messages back to the host, as in a standard ToR.

Let us compare a “standard” unscheduled network fabric and a Stardust network, and assume a configuration of 128 ToR switches/ Fabric Adapters with $50\text{Gbps} \times 128$ ports, 64 switches/ Fabric Elements with $50\text{Gbps} \times 128$ ports, and 32MB packet buffer per device. Consider an incast event of 1MB from each ToR toward a single 50Gbps port (total of 128MB). In the Ethernet switch fabric, all traffic will reach the egress ToR, and whatever is not transmitted at 50Gbps, will fill the ToR’s packet buffer or be dropped. This happens regardless of the load balancing granularity (flow, flowlet, packet). Activating PFC flow control at the destination may propagate and block the entire fabric. In Stardust, the incast traffic from all Fabric Adapters will be admitted to the fabric at an aggregate rate of 50Gbps. The rest (99%) accumulates in the ingress Fabric Adapters, occupying 0.99MB of each packet buffer. No packet is lost. The available packet buffer memory per destination is effectively $\times 128$ larger, distributed across all Fabric Adapters. The destination’s egress scheduler distributes bandwidth (credits) to incast sources evenly, guaranteeing an even draining rate of the VOQs, and an optimal flow completion time of the incast event.

5.5 Lossless Transmission

The Fabric Element is lossless. As shown in §6, when under-subscribed, the probability of a queue reaching its maximal level is very low. The FCI reduces this probability further, as credit rate is reduced when a queue starts building. This control loop is activated with a very low probability, and affects only traffic heading to the congested destinations. When the network fabric is intentionally over-subscribed, the FCI mechanism is activated only when the fabric is actually over-subscribed with traffic. Queue overflow is further reduced using a shared queue-memory pool, permitting dynamic balancing between congested and non-congested links. As a last resort, the Fabric Element pipe can be paused to avoid dropping a cell. The probability of this event is infinitesimal, thus the net effect on the entire fabric throughput is negligible.

5.6 Latency and Jitter

The packet latency and jitter through the network fabric are governed by the queue size probability distribution within the Fabric Element. The latency is at the scale of 0.5-4 microseconds per hop. In a multi-tier network fabric, the latency is governed by fiber length as much as it is by the Fabric Element latency, as every 100m of fiber translates to a half microsecond propagation delay.

First packet latency depends on VOQ configuration. Receiving a credit will take about a microsecond, but a *low*

latency VOQ starts transmitting immediately. We assume a limited aggregate bandwidth of all low latency VOQs, bounded by Egress Fabric Adapter and Fabric Element memory resources, else packets may be dropped (as in a ToR).

5.7 Traffic Pattern Agnosticism

A valuable property of Stardust fabric is its agnosticism to the traffic pattern at the edges (ToR/Fabric Adapter). The single parameter that affects the fabric performance and latency is the fabric speed-up f_s^2 . If we consider the worst-case scenario for latency and drop probability, it is for many (e.g., 1000) sources sending traffic to a single output port. This is bound by the M/D/1 queuing model. Other scenarios would have a shorter-tailed queue size (hence latency) distribution. If the total bandwidth toward a port exceeds its bandwidth, as in incast, it does not matter; the egress distributes credits at the egress port’s rate and no more, and excess traffic remains queued in the source Fabric Adapters, while the fabric experiences the same cell loads, and hence, will have the same or better queue size distributions than that of the M/D/1 queuing model.

5.8 Dynamic Routing

Routing within Stardust’s fabric requires no routing protocol or SDN controller inputs, and is dynamic per cell. When a cell needs to be sent to a destination, there are multiple paths that can be taken. Cells are load balanced on all available paths, even if they are part of the same flow or packet. Consequently, a large set of paths is taken between every pair of source and destination Fabric Adapters. There is no set of fabric routes per destination that is saved in a memory: for each destination Fabric Adapter there is only the set of links outgoing from the current device that can be used. This forwarding database has a negligible size of *Number-of-Fabric-Adapters* entries of size *Number-of-Links*.

Each device periodically transmits its Fabric Adapter reachability table to its upstream neighbors, leading to an automatic update of routing tables. Any topology change due to link failure is transmitted from its source to the routing tables of all upstream devices, and affects their forwarding decision.

5.9 Improved Resilience, Self-Healing Fabric

The use of non-bundled links improves the resilience of the network. While for a 400GE port, a link failure means that $8 \times 50\text{Gbps}$ links go down, in the Stardust, a link failure affects only this single link. Therefore, the improvement in resilience equals the link bundling between stages.

The reachability messages exchange, used for building the destination’s reachability table, is also used to establish a link’s status and health, making the network self-healing and increasing its resilience. When a link is down, the reacha-

²And even then, as long as the fabric is < 90% utilized the dependence of latency distribution on fabric speed-up is very weak

bility tables in all devices are automatically updated, and the load balancing is adjusted to exclude dependencies on this link. The load balancing, starting at the first stage’s Fabric Adapter, eliminates hot spots of congestions. Recovery time is determined by the rate allocated for reachability messages, i.e., interval between successive reachability cells. The recovery time is around hundreds of microseconds with negligible reachability cell overhead (Appendix E). While the values are configurable, assume for example, a Fabric Element reporting the reachability of 128 Fabric Adapters on a link every $1\text{ns} \times 10,000$ clocks. It takes the Fabric Element seven messages to report the status of a network connecting 32K hosts (40 hosts per Fabric Adapter). It takes 3 hops in a Fat-tree to reach the ingress Fabric Adapter from the last Fabric Element, amounting to $1\text{ns} \times 10000 \times 7 \times 3 = 210\mu\text{s}$. Updating the status based on multiple (e.g, three) consecutive updates will amount to roughly $630\mu\text{s}$. The mechanism precedes BFD [47] and, as implemented in a dedicated hardware, has a lower overhead.

5.10 Handling Failures

Load balancing cannot compensate for a link’s failure; if the link is connected to a destination Fabric Adapter, the bandwidth capacity toward the destination is reduced. Resilience can be improved by adding redundant links. With traffic automatically spread across all links, and as there is no management overhead for adding links, contrary to typical networks, only the physical cost applies.

Stardust provides multiple mechanisms to handle failures, from the link level to end-to-end detection and isolation. For example, packets with Ethernet CRC errors are dropped at the Fabric Adapters. On the link level, traditional mechanisms are applied to protect against errors (e.g., Forward error correction). If the error rate on a link crosses a threshold, the link marks itself as faulty on reachability cells, and is excluded from cell forwarding. A link is declared valid only after the number of good reachability cells received crosses a threshold. The probability of an error on a link is unaffected by the use of cells, as the number of bits on the wire is approximately maintained. As Stardust minimizes the number of hops in a network, by saving tiers, the probability of an error on a packet is reduced.

When a device fails, it stops sending reachability messages. Consequently, links leading to the device are removed from the reachability tables, and the traffic bypasses the faulty device. In a multi-tier network fabric, this information is also propagated upstream to the Fabric Adapters. If the load per link increases due to a device’s removal, the FCI mechanism readjusts the credit rate toward affected devices.

Unlike scenarios, such as a faulty Fabric Adapter sending unlimited credits, will not collapse the network. Instead, it will degrade the network to the performance of a standard “Push” Ethernet network. While switch-systems are mature solutions, where failure mechanisms have been thoroughly

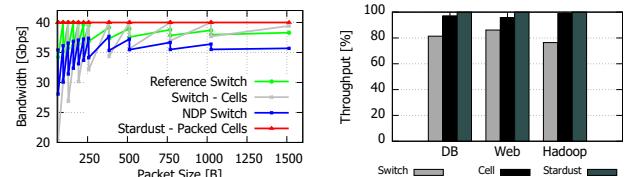


Figure 8: Throughput of switches running at 150MHz and using (a) single packet size, (b) traces from [74].

debugged over the years, DCNs face additional reliability challenges, which we intend to study in future work.

6 Evaluation

We take a three stage approach to the evaluation. First, running experiment-driven measurements on stand-alone devices and in an existing single-tier Stardust system. Second, we conduct a precise simulation of a two-tier network. Last, in §7 we analyze a large scale network, attending to aspects of cost, power and scalability. The simulation and analysis are based on the results of our measurements.

6.1 Experiment-Driven Measurements

6.1.1 Packet Packing

Measuring the effect of packet packing on throughput is not possible on existing silicon devices. Instead, we use a programmable platform, NetFPGA SUME [84] to demonstrate this effect. We compare four different architectures, all originating from the same source code: NetFPGA $4 \times 10\text{GE}$ Reference Switch (Release 1.7.1), NDP switch [41, 40], Stardust, and Stardust-based using non-packed cells. The NDP design originated from the NetFPGA Reference Switch, and treats non-NDP packets similar to it. The Stardust design is similar to the Reference Switch, but instead of switching packets it switches packed cells. The Stardust-based non-packed cells design is identical in implementation to the Stardust design, yet we inject non-packed cells. The credit size is 2KB, and each cell is 64B, as the NetFPGA data path is 32B wide with 2 clock cycles per table lookup. We compare the designs’ throughput at a data path clock frequency of 150MHz (The NetFPGA platform is limited at lower frequencies [63]). The traffic, generated using OSNT [12], is a stream of TCP/NDP packets of various sizes. As a reminder, our goal is to explore device-level performance and scalability, rather than protocol-level properties.

As Fig. 8(a) shows, Stardust achieves full line rate for all packet sizes, up to 15%, 30% and 49% better than the Reference Switch, NDP, and non-packed cells, respectively. The Reference Switch achieves full line rate for all packet sizes only at a clock frequency of 180MHz, while NDP fails to reach full line rate for some packet sizes (65B, 97B, 129B) even at 200MHz³. Using real world traces [74], Stardust

³NDP’s TCP performance is identical to the Reference Switch, and only NDP packets experience performance loss. NDP’s authors confirmed this performance loss is expected in their design.

maintains its performance edge, as shown in Fig. 8(b). NDP is omitted as it performs worse than the standard switch.

These results indicate the power of packet packing: achieving higher throughput, for all packet sizes, at lower clock frequencies, simplifying network silicon design and enabling better scalability over time.

6.1.2 Throughput and Latency

A number of relevant experimental results from a wider evaluation of Fabric Adapters and Fabric Elements within a single-tier Arista 7500E system are reported here. An extended set of results appears in [53]. The test platform contained $1152 \times 10GE/288 \times 40GE$ ports, connecting 24 Fabric Adapters with a single tier of 12 Fabric Elements, and using Ixia XG12 as a traffic source. The Fabric Adapter devices used, older generation Arad, do not support packet packing.

In a configuration of 960 ports, the platform achieves full line rate on all ports, for all packet sizes. In a configuration of 1152 ports, full line rate is achieved for packets of 384B or more (as packing was not supported), and no packet loss occurred within the network fabric. In the same 1152 port configuration, under full load, the minimum latency is almost independent of packet size (between $2.8\mu s$ and $3.5\mu s$), and for the average ($3.3\mu s$ to $9.1\mu s$) and maximum ($5.6\mu s$ to $13.5\mu s$) it increases with packet size, as the Fabric Adapter used store-and-forward architecture. The maximal latency for small packets decreases with packet size until reaching the cell size. The average latency variance is in the order of nanoseconds (1ns-11ns).

6.2 Simulation of a 2-Tier Network

To understand the queuing intricacies of Stardust in a two tier network, we use a proprietary packet-level C++ simulator, constructing a network of Fabric Elements and Fabric Adapters, generating traffic and emulating scheduler and control traffic exchange. We use this experiment to discuss aspects of load balancing, queue size distribution and memory requirements.

The network is evaluated using 128K simultaneously active flows. The number of Fabric Adapters is 256, each of them connected using $t = 32$ links to the network fabric. There are 128 Fabric Elements in the first fabric tier, each with 64 links connected to the Fabric Adapters and 64 links connected to the second tier. In the second tier, there are 64 Fabric Elements, each connected using all 128 links to Fabric Elements in the first tier. The length of each link is 100m. We simulate two flows from each Fabric Adapter to every other Fabric Adapter, creating 128K simultaneous flows. The number of connected hosts does not affect the simulation, but assuming the number of uplinks is no more than the number of downlinks, it is in the range of 8K to 64K hosts. The setup size is limited by simulator run times.

Fig. 9(left) presents the latency distribution of network fabric traversal for different fabric utilizations. The latency distribution is tight, and even at 95% utilization, the latency

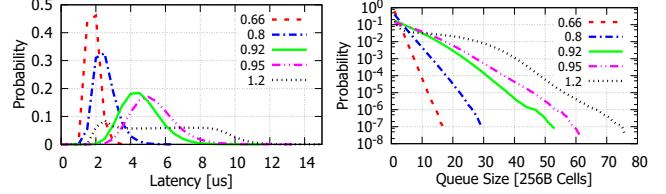


Figure 9: Probability Distribution of latency (left) and link queue size (right) under different loads.

is bound by 13 microseconds. Fig. 9(right) presents queue size distribution in the last stage of the network fabric, for different fabric utilizations. The fabric utilization refers to raw data utilization, after deducting physical protocol overheads, cell headers and control cell overheads. A distribution of an oversubscribed fabric (120%) is included, using FCI to control credit rate. In all runs no cells were lost with the network fabric. With $utilization < 1$, queue size probability is an exponential function of fabric utilization, conforming to the theoretical M/D/1 model. In the oversubscribed case, the FCI reduced the effective fabric utilization to 0.9.⁴

We extrapolate the simulation's results to recent network devices, equipped with $256 \times 50Gbps$ links. For a cell size of 256B and a speed up of 1.05 the respective memory will be $128 \times 256B \times 256$, i.e. only 8MB. Given the 50Gbps links, this stands for at most $5\mu s$ latency within the Fabric Element.

6.3 Comparisons to Existing Works

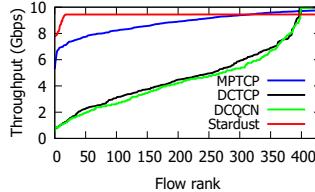
We compare Stardust with Multipath TCP [72], DCTCP [7] and DCQCN [82]. We use *htsim* [72] for our evaluation, and reproduce the experimental environment of [40, 41], implementing a Stardust network, and running TCP. The simulated environment uses a 432-node Fat-tree (see Appendix G).

We run a permutation experiment, where each node in a Fat-tree continuously sends traffic to one node and receives from another, fully loading the data center. Stardust uses 512B packets, 4KB credit size, and 3% credit speed up. Other solutions use 9000B packets. Fig. 10(a) shows the throughput achieved by each node in the network, in an increasing order. Stardust achieves 9.44Gbps on 96% of the flows, and a mean utilization of 94%, compared with 49%, 47% and 90% for DCTCP, DCQCN and MPTCP, respectively.

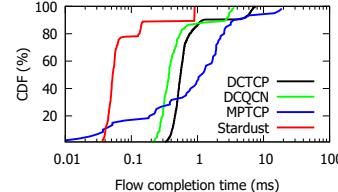
We use the Web workload from [74] to exchange traffic between a pair of nodes and measure the flow completion time (FCT). The same 432-node setup is used, and all other nodes source four long-running connections to a random destination, thus testing the effect of queuing within the network on short flows. Fig. 10(b) shows that Stardust significantly outperforms all other schemes, as the fabric is scheduled. Even flows of 1MB have a FCT of less than a millisecond.

Next, we test an incast traffic pattern. A frontend server

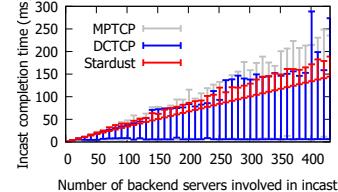
⁴Representing a single trade-off point between latency and throughput. Higher utilization can be gained using less aggressive FCI reaction.



(a) Throughput



(b) FCT



(c) Incast

	B/A
Header Processing	13%
Network Interface	30%
Other logic	60%
I/O	87.5%
Relative area/Tbps	66.6%
Relative power/Tbps	64.8%

(d) Relative Area

Figure 10: Performance comparison, 432-node FatTree (a) Per-flow throughput (b) FCT in an over-subscribed network (c) Incast performance vs number of senders (d) Relative device area of a Fabric Element device (B) and a standard switch (A)

fans out work to many backend servers and then receives their replies, creating an incast scenario. We use a constant response size (450K) and vary the number of backend servers, measuring the FCT. Fig. 10(c) shows the first and last FCT, a measure both of performance and “fairness”. Stardust’s last FCT is the same as DCTCP and better than MPTCP, but its fairness is considerably better. Furthermore, no packets are dropped within the Stardust fabric.

7 Cost Analysis

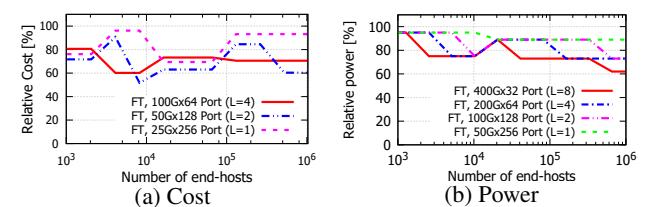
The absolute cost of a Stardust based system is difficult to calculate, as it depends on multiple factors, such as manufacturing costs and platform implementation. Thus, we take a comparative approach and analyze the complexity of Stardust and closely related Fat-tree networks, showing that Stardust is more cost efficient.

Silicon cost Our silicon cost comparison is based on two Broadcom devices, manufactured using the same process. Device A is a standard Ethernet switch (ToR), whereas device B is a Fabric Element [23]. The table in Fig. 10(d) compares the area of these two devices, normalized to their bandwidth, accounting for a difference in the number of I/O (which is favorable for the ToR), detailed further in Appendix C. The relative area/Tbps of a Fabric Element is only 66.6% of that of a ToR. The overall area of a Fabric Adapter is very similar to device A, and is referred as identical henceforth. The area is a good indicator of silicon cost, dominating yield, packaging costs and others factors.

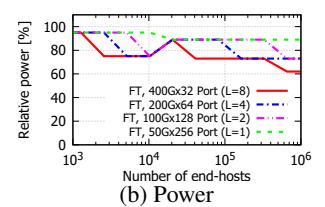
Switch-Platform Cost A Fabric Adapter based switch-platform will cost the same as a standard ToR, while a Fabric Element box will be less expensive. A switch platform is composed of many components, yet the high-end switch silicon dominates the cost. In a Fabric Element platform, the board complexity is reduced and a lower-cost CPU can be used, as only basic control is needed⁵. The reduced complexity also means that more than a single Fabric Element can be used within a single platform, with two being the common case for emerging OCP designs. While in switch-systems the cost ratio between Fabric Adapter and Fabric Element modules of the same generation is high⁶, we use the conser-

⁵From discussions with leading cloud providers, some will prefer the cost-driven CPU option, whereas others will prefer the same type of CPUs across all platforms, for better manageability

⁶Comparison of modules from same manufacturer and platform, based



(a) Cost



(b) Power

Figure 11: The (a) cost and (b) power of building a Stardust DCN relative to Fat-tree.

vative silicon-area ratio as a cost indicator.

Transceivers and Fibers Stardust always opts for the minimal number of transceivers in a DCN. If a network can be constructed with a similar number of tiers using, e.g., 400G and 100G transceivers, then Stardust supports building the network using the least expensive option. The devices are oblivious to whether bundling was used in the transceiver. Furthermore, breakout cables can be used to connect transceivers of different bundling, easing a gradual growth of a network. As every additional tier doubles the number of links required ($\times 4$ transceivers, Table 2), a Stardust based solution becomes more attractive based on cost.

System Cost The cost of a large scale DCN can be cut in half using Stardust. We analyze the relative cost of building a DCN based on current generation data center components, detailed in Appendix D. We assume 25G as the link speed and its link bundles (50G, 100G). ToR and Fabric Adapter platforms are of an equal cost. We use the relative silicon area (0.67) as a relative cost indicator for Fabric Element platforms (See switch-platform cost). Each ToR/Fabric Adapter is assumed to connect conservatively to 40 servers using direct-attach cables, with no over-subscription from the ToR/Fabric Adapter to the network fabric. 100m fibers are used in the last tier (except for a 1-tier network), and 10m fibers within any other tier. The number of devices and links is calculated as detailed in Appendix A. Fig. 11(a) shows the overall cost of building the network. Stardust is always the most cost effective solution.

Power Power consumption significantly varies between switch vendors, e.g., from 150W to 310W [57]. Fig. 11(b) examines the relative power consumption of Stardust, compared with Fat-tree networks with different link bundlings. The calculation is as in Appendix A, using the power ratio indicated in Fig. 10(d). Power per link and equal cross-

on e.g., [USENIX Association](https://h22174.www2.hpe.com/SimplifiedConfig>Welcome</p>
</div>
<div data-bbox=)

section bandwidth are considered. The biggest power saving is in networks of up to ten thousands nodes: up to 25% of the entire network’s power, and 78% saving within the network fabric, a result of both the reduction in the number of devices, and power saving per device.

The number of network tiers and devices required to build a network (Fig. 2b), and the power consumption of each device, affect the physical space required. The amount of physical space also impacts performance: smaller networks require shorter fibers, reducing the latency, and in turn improving throughput. In Stardust, Fabric Element based chassis can still be used within the network, further reducing space requirements.

8 The Case for Future Data Centers

Using current Fabric Adapter and Fabric Element devices, Stardust naturally fits within regional data centers, and can similarly be used to build efficient clusters of 10K end-hosts (see §6). However, while Stardust reduces the complexity within the network’s fabric, the Fabric Adapter maintains the complexity and resource consumption of current ToRs (§6). The challenge becomes the scalability of ToR devices.

We make the case for future Stardust-based data centers, made entirely of Fabric Element devices within the network and of Fabric Adapter-like network interface cards (NICs) at the edges. Here, the concept of a simple network-core is further extended, and the complex network-edge diminished.

The divide-and-conquer approach adapts the end-host’s NIC to Stardust through a *reduction* of the Fabric Adapter: combining VM-facing functionality with the handling of cells and scheduling, but at a smaller scale. The number of VOQs will match host-scale requirements [34, 77]. The host’s memory will be used for further buffering [54, 55, 58]. The NIC’s MAC will become lighter: a fabric interface. A programmable header processor, as in the Fabric Adapter, will also support acceleration in the NIC. Connecting a NIC to a Fabric Element is the same as to a ToR, while the reachability table required is smaller than in a Fabric Adapter, by a factor of *Num-of-Fabric-Adapter-Uplinks/Num-of-NIC-ports*, or can be entirely eliminated if the NIC connects to a single Fabric Element.

This is a natural evolution, supporting trends advocating smarter edge devices [52, 85, 34] and moving scheduling close to the host [71, 78]. We estimate the power consumption of such a NIC to be on-par with current smart NICs [34]. As Jericho [22] already supports a PCIe interface and a direct memory access (DMA) engine to send and receive host traffic, this vision is realizable.

9 Related Work

Switch systems today scale to thousands of ports and to multiple chassis [27, 43, 14, 65]. Their chipsets vary from commodity devices [20] to custom-made solutions [66, 61]. The

closed nature of the system, has allowed building upon concepts that have failed in wide area networks (WAN), and in particular in cell based solutions such as ATM [46, 50, 11], while optimizing for system-scale needs (e.g., [48, 24, 10]). While DCNs differ from supercomputing in many ways, supercomputing research also explored aspects of credit-based scheduling and cell-switched cores on a medium scale (e.g., [49, 3, 1]). Stardust is deeply rooted in switch systems, which inspired supercomputing networks.

Stardust is not a circuit switch. Circuit switching (e.g., recent [70, 36, 58]) allows a single path at a time between source-destination pairs and has strong (and sometimes high-latency) scheduling requirements, whereas Stardust allows any-to-any communication at all times and applies distributed end-to-end scheduling.

This paper focused on Clos [30], and Fat-tree topologies [51, 4, 75]. Our link bundling observation is also applicable to other topologies (e.g., [2, 38, 18]). Unlike other solutions, Stardust makes no assumptions about traffic patterns or network utilization, and its routing is trivial [18, 80, 76]. It also remains a single network [59], providing better manageability and resilience. Aspects of network reconfigurability combining different topologies (e.g., [58, 81]) are beyond the scope of this contribution.

Congestion control and load balancing research has tried to optimize performance, from protocol and flow level (e.g., [5, 8, 73, 9, 41]), through flowlets (e.g., [45, 6]) and flow-cells [42] to packets (e.g., [25, 37]). Stardust uses evenly distributed dynamic forwarding, combined with end-to-end scheduling. Stardust is refined by combining ingress buffering with congestion avoidance mechanisms (e.g., ECN), providing a lossless solution for short-term congestion.

10 Conclusion

We presented Stardust, a scalable fabric architecture for data center networks, separating the simple network-fabric from the complex network-edge. Stardust applies system-switch architecture on a data center scale, while attending to the scalability limitations of network devices: resources, I/O and performance. The resulting network fabric devices are a radical change from commodity Ethernet switches, eliminating significant overheads in DCNs. Our demonstrated *divide and conquer* approach is practical, power-efficient, cost-effective, scalable, and, critically, a deployable approach.

11 Acknowledgement

We would like to thank the many people who contributed to this work, particularly to the team at Broadcom Yakum. We would like to thank Andrew W Moore and Jeff Mogul for their useful advice. We thank the anonymous reviewers and our Shepherd, Xin Jin, who helped us improve the quality of this paper. This work was partly funded by the Leverhulme Trust (ECF-2016-289) and the Isaac Newton Trust.

References

- [1] ABTS, D. Cray xt4 and seastar 3-d torus interconnect. In *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 470–477.
- [2] AHN, J. H., BINKERT, N., DAVIS, A., MCLAREN, M., AND SCHREIBER, R. S. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 41.
- [3] AJIMA, Y., INOUE, T., HIRAMOTO, S., TAKAGI, Y., AND SHIMIZU, T. The tofu interconnect. *IEEE Micro* 32, 1 (Jan 2012), 21–31.
- [4] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 63–74.
- [5] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010), vol. 10, pp. 19–19.
- [6] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., MATUS, F., PAN, R., YADAV, N., VARGHESE, G., ET AL. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 503–514.
- [7] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., , AND SRIDHARAN, M. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM* (Aug. 2010).
- [8] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *ACM SIGCOMM computer communication review* (2010), vol. 40, ACM, pp. 63–74.
- [9] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 435–446.
- [10] ALVERSON, R., ROWETH, D., AND KAPLAN, L. The gemini system interconnect. In *IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)* (2010), IEEE, pp. 83–87.
- [11] ANSI. T1.105-1991, digital hierarchy optical interface rates and formats specifications (SONET)”. *American national standard for telecommunication* (1991).
- [12] ANTICHI, G., SHAHBAZ, M., GENG, Y., ZILBERMAN, N., COVINGTON, A., BRUYERE, M., MCKEOWN, N., FEAMSTER, N., FELDERMAN, B., BLOTT, M., ET AL. OSNT: Open source network tester. *IEEE Network* 28, 5 (2014), 6–12.
- [13] ARISTA. 7500R Series Data Center Switch Router, *Datasheet*. <https://www.arista.com/assets/data/pdf/Datasheets/7500RDataSheet.pdf>.
- [14] ARISTA. Arista 7500 Switch Architecture (“A day in the life of a packet”), 2016. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7500E_Switch_Architecture.pdf.
- [15] ARMITAGE, G. J., AND ADAMS, K. M. Packet reassembly during cell loss. *IEEE Network* 7, 5 (1993), 26–34.
- [16] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the killer microseconds. *Communications of the ACM* 60, 4 (2017), 48–54.
- [17] BARWICZ, T., TAIRA, Y., LICOULAS, T. W., BOYER, N., MARTIN, Y., NUMATA, H., NAH, J.-W., TAKENOBU, S., JANTA-POLCZYNSKI, A., KIMBRELL, E. L., ET AL. A novel approach to photonic packaging leveraging existing high-throughput microelectronic facilities. *IEEE Journal of Selected Topics in Quantum Electronics* 22, 6 (2016), 455–466.
- [18] BESTA, M., AND HOEFLER, T. Slim fly: A cost effective low-diameter network topology. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)* (2014), IEEE, pp. 348–359.
- [19] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2013), SIGCOMM ’13, ACM, pp. 99–110.
- [20] BROADCOM. *StrataDNX Switch Solutions*. <https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx>.
- [21] BROADCOM. *BCM56980 series, 12.8 Tbps StrataXGS Tomahawk 3 Ethernet switch series*, dec 2017. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series>.
- [22] BROADCOM. *BCM88690, StrataDNX 10 Tb/s Scalable Switching Device, Product Brief*, 2018. <https://docs.broadcom.com/docs/88690-PB100> [Online, accessed January 2019].
- [23] BROADCOM. *BCM88790, Scalable Fabric Element 9.6 Tbps Self-Routing Switching Element, Product Brief*, 2018. <https://docs.broadcom.com/docs/88790-PB00> [Online, accessed January 2019].
- [24] CHRYSOS, N., AND KATEVENIS, M. Scheduling in non-blocking buffered three-stage switching fabrics. In *INFOCOM* (2006), vol. 6, pp. 1–13.
- [25] CHRYSOS, N., NEESER, F., CLAUBERG, R., CRISAN, D., VALK, K. M., BASSO, C., MINKENBERG, C., AND GUSAT, M. Unbiased quantized congestion notification for scalable server fabrics. *IEEE Micro* 36, 6 (2016), 50–58.
- [26] CISCO. *Cisco Nexus 9000 Series Switches*. <https://www.cisco.com/c/en/us/products/switches/nexus-9000-series-switches/index.html>.
- [27] CISCO. *Cisco CRS Carrier Routing System Multishelf System Description*, jul 2014. <http://www.cisco.com/c/en/us/td/docs/iosxr/crs/hardware-install/crs-1/multishelf/system-description/b-crs-multishelf-sys-desc.html>.
- [28] CISCO. *Cisco Global Cloud Index: Forecast and Methodology, 2015–2020*, nov 2016.
- [29] CISCO. *Cisco Nexus 9500 R-Series Line Cards and Fabric Modules White Paper*, 2018. <https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-738392.pdf>.
- [30] CLOS, C. A study of non-blocking switching networks. *Bell Labs Technical Journal* 32, 2 (1953), 406–424.
- [31] CUMMINGS, U., DALY, D., COLLINS, R., AGARWAL, V., PETRINI, F., PERRONE, M., AND PASETTO, D. Fulcrum’s focalpoint fm4000: A scalable, low-latency 10GigE switch for high-performance data centers. In *IEEE Symposium on High Performance Interconnect (HOTI)* (2009), IEEE, pp. 42–51.
- [32] DHARMAPURIKAR, S., AND PAXSON, V. Robust tcp stream reassembly in the presence of adversaries. In *USENIX Security* (2005).
- [33] ESCOBAR, J., AND PARTRIDGE, C. A proposed segmentation and reassembly (sar) protocol for use with asynchronous transfer mode (atm). *High Performance Network Research Report* (1990).
- [34] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018).
- [35] FRANK, D. J., DENNARD, R. H., NOWAK, E., SOLOMON, P. M., TAUR, Y., AND WONG, H.-S. P. Device scaling limits of Si MOSFETs and their application dependencies”. *Proceedings of the IEEE* 89 (2001), 259–288.

- [36] GHOBADI, M., MAHAJAN, R., PHANISHAYEE, A., DEVANUR, N., KULKARNI, J., RANADE, G., BLANCHE, P.-A., RASTEGARFAR, H., GLICK, M., AND KILPER, D. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the ACM SIGCOMM Conference* (2016), ACM, pp. 216–229.
- [37] GHORBANI, S., YANG, Z., GODFREY, P., GANJALI, Y., AND FIROOZSHAHIAN, A. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the ACM SIGCOMM Conference* (2017), ACM, pp. 225–238.
- [38] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VI2: a scalable and flexible data center network. *Communications of the ACM* 54, 3 (2011), 95–104.
- [39] GUREVICH, V. Barefoot networks, programmable data plane at terabit speeds. In *DXDD* (2016), Open-NFP.
- [40] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. *NDP Switch NetFPGA*, 2017. Repository, <https://github.com/nets-cs-pub-ro/NDP> [accessed December 2017].
- [41] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM Conference* (2017), ACM, pp. 29–42.
- [42] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J., AND AKELLA, A. Presto: Edge-based load balancing for fast datacenter networks. In *SIGCOMM '15* (2015), ACM, pp. 465–478.
- [43] HUAWEI. *CloudEngine 12800 Series Data Center Switches*. <http://e.huawei.com/us/products/enterprise-networking/switches/data-center-switches/ce12800>.
- [44] INTEL. *Intel Xeon Processor E7 v4 Family*, 2017. <https://ark.intel.com/products/series/93797/Intel-Xeon-Processor-E7-v4-Family> [accessed January 2018].
- [45] KANDULA, S., KATABI, D., SINHA, S., AND BERGER, A. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review* 37, 2 (2007), 51–62.
- [46] KATEVENIS, M., SIDIROPOULOS, S., AND COURCOUBETIS, C. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on selected Areas in Communications* 9, 8 (1991), 1265–1279.
- [47] KATZ, D., AND WARD, D. *Bidirectional forwarding detection (BFD)*, 2010. RFC 5880.
- [48] KESLASSY, I., CHUANG, S.-T., YU, K., MILLER, D., HOROWITZ, M., SOLGAARD, O., AND McKEOWN, N. Scaling internet routers using optics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (2003), ACM, pp. 189–200.
- [49] KUMAR, S., SABHARWAL, Y., GARG, R., AND HEIDELBERGER, P. Optimization of all-to-all communication on the blue gene/l supercomputer. In *2008 37th International Conference on Parallel Processing* (Sept 2008), pp. 320–329.
- [50] LE BOUDEC, J.-Y. The asynchronous transfer mode: a tutorial. *Computer Networks and ISDN systems* 24, 4 (1992), 279–309.
- [51] LEISERSON, C. E. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers* 100, 10 (1985), 892–901.
- [52] LI, H. Introducing "Yosemite": the first open source modular chassis for high-powered microservers. <https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-high-powered-microservers/>, 2015. [Online; accessed January 2017].
- [53] LIPPIS REPORT. *Arista 7500E Software-Defined Cloud Network Switch Performance Test*, 2014. <https://www.arista.com/assets/data/pdf/7500E-Lippis-Report.pdf> [Online, access January 2019].
- [54] LIU, H., LU, F., FORENCICH, A., KAPOOR, R., TEWARI, M., VOELKER, G. M., PAPEN, G., SNOEREN, A. C., AND PORTER, G. Circuit switching under the radar with reactor. In *NSDI* (2014), vol. 14, pp. 1–15.
- [55] MANIHATTY BOJAN, N., ZILBERMAN, N., ANTICHI, G., AND MOORE, A. W. Extreme data-rate scheduling for the data center. In *Proceedings of the ACM SIGCOMM Conference* (2015), pp. 351–352.
- [56] McKEOWN, N., IZZARD, M., MEKKITTIKUL, A., ELLERSICK, W., AND HOROWITZ, M. Tiny tera: a packet switch core. *IEEE Micro* 17, 1 (1997), 26–33.
- [57] MELLANOX. *Mellanox Spectrum Ethernet Switch*. http://www.mellanox.com/page/products_dyn?product.family=218&mtag=spectrum_ic.
- [58] MELLETTE, W. M., MCGUINNESS, R., ROY, A., FORENCICH, A., PAPEN, G., SNOEREN, A. C., AND PORTER, G. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the ACM SIGCOMM Conference* (2017), ACM, pp. 267–280.
- [59] MELLETTE, W. M., SNOEREN, A. C., AND PORTER, G. P-fattree: A multi-channel datacenter network topology. In *HOTNETS* (2016), ACM, pp. 78–84.
- [60] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics* 38 (April 1965), 114–117.
- [61] MORGAN, T. P. *Homegrown 25G Chips Help Cisco Hold The Switch Line*, 2016. <https://www.nextplatform.com/2016/03/01/homegrown-25g-chips-help-cisco-hold-the-switch-line/>.
- [62] MORGAN, T. P. *Flattening Networks And Budgets With 400G Ethernet*, jan 2018. <https://www.nextplatform.com/2018/01/20/flattening-networks-budgets-400g-ethernet/> [accessed January 2018].
- [63] NETFPGA. *NetFPGA-SUME-live, issue 36: 10G port - Attachment unit - Rx side - inefficiency*, 2017. <https://github.com/NetFPGA/NetFPGA-SUME-live/issues/36>.
- [64] NETWORKS, B. *Barefoot Tofino, World's fastest P4-programmable Ethernet switch ASICs*, 2016. <https://barefootnetworks.com/products/brief-tofino/>.
- [65] NETWORKS, J. *QFabric System, Scalable Fabric Switching System*. <https://www.juniper.net/uk/en/products-services/switching/qfabric-system/>.
- [66] NETWORKS, J. *Juniper Networks Unveils Industry's Most Powerful Core Routing Platform*, 2015. <http://investor.juniper.net/investor-relations/press-releases/press-release-details/2015/Juniper-Networks-Unveils-Industries-Most-Powerful-Core-Routing--Platform/default.aspx>.
- [67] OHRING, S. R., IBEL, M., DAS, S. K., AND KUMAR, M. J. On generalized fat trees. In *Proc. Ninth International Parallel Processing Symposium* (1995), IEEE, pp. 37–44.
- [68] PATTERSON, D. A., AND HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. Newnes, 2013.
- [69] POPESCU, D. A., AND MOORE, A. W. Ptpmesh: Data center network latency measurements using ptpt. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2017 IEEE 25th International Symposium on* (2017), IEEE, pp. 73–79.
- [70] PORTER, G., STRONG, R., FARRINGTON, N., FORENCICH, A., CHEN-SUN, P., ROSING, T., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. *Integrating microsecond circuit switching into the data center*, vol. 43. ACM, 2013.
- [71] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. Senic: Scalable nic for end-host rate limiting. In *NSDI* (2014), vol. 14, pp. 475–488.

- [72] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving datacenter performance and robustness with Multipath TCP. In *Proc. ACM SIGCOMM* (Aug. 2011).
- [73] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 266–277.
- [74] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network’s (datacenter) network. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 123–137.
- [75] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network. *ACM SIGCOMM Computer Communication Review* 45 (2015), 183–197.
- [76] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking data centers, randomly. In *NSDI* (2012), vol. 12, pp. 17–17.
- [77] SOLARFLARE. *Ultra Scale, Scaling from two to two thousand containers on a single server*, 2018. <https://www.solarflare.com/ultra-scale>.
- [78] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue nics. In *2017 USENIX Annual Technical Conference (ATC’17)* (2017), pp. 431–444.
- [79] TECHNOLOGIES, M. *Mellanox Spectrum-2 Ethernet Switch*, 2017. http://www.mellanox.com/related-docs/prod_silicon/PB_Spectrum-2.pdf.
- [80] VALADARSKY, A., DINITZ, M., AND SCHAPIRA, M. Xpander: Unveiling the secrets of high-performance datacenters. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (2015), ACM, p. 16.
- [81] XIA, Y., SUN, X. S., DZINAMARIRA, S., WU, D., HUANG, X. S., AND NG, T. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proceedings of the ACM SIGCOMM Conference* (2017), ACM, pp. 295–308.
- [82] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *SIGCOMM* (2015), pp. 523–536.
- [83] ZILBERMAN, N. An evaluation of NDP performance . Tech. Rep. UCAM-CL-TR-933, University of Cambridge, Computer Laboratory, Feb. 2019.
- [84] ZILBERMAN, N., AUDZEVICH, Y., COVINGTON, G., AND MOORE, A. W. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (September 2014), 32–41.
- [85] ZILBERMAN, N., MOORE, A. W., AND CROWCROFT, J. A. From photons to big-data applications: terminating terabits. *Phil. Trans. R. Soc. A* 374, 2062 (2016), 20140445.

A The Math Behind Network Size

In this section, we provide the mathematical justification for the number of elements used in a Fat-tree based network. The number of devices required to build a Fat-tree network is long known [51, 67], and we build upon it to demonstrate the dependency on link bundling. For the clarity of the discussion, Table 1 lists the parameters used to describe the network topology, and Table 2 provides the number of different elements within a multi-tier, Fat-tree based, DCN and the parameters used to describe the network topology. Note that Table 2 shows the number of network elements, and not the number of connected end hosts, and that unless noted otherwise, downlinks from a ToR to end hosts are not counted.

In a 1-Tier network, the number of switches equals, at most, the number of uplink ports from a ToR (t). The number of ToRs equals, at most, the number of downlinks from a switch, which is in this case the number of ports on the switch (k). The number of link bundles in this network will equal the number of uplink ports from a ToR times the number of ToRs, which equals the number of downlinks from a switch times the number of switches ($t \times k$). As this is a 1-Tier network, there is only one layer of links within it, therefore the number of links per ToR will be the same as the number of uplinks ($t \times l$).

In a 2-Tier network, the number of downlink ports from a switch in a second tier will still be k , but a switch in the first layer will have $k/2$ downlink ports and $k/2$ uplink ports. The number of uplink ports from a ToR is the same. This leads to a network of $k^2/2$ ToRs (at most). The number of link bundles in the first tier will equal the number of ToRs times the number of uplink ports per ToR ($t \times k^2/2$). With a naive construction, assuming a fully provisioned network, this will equal the number of link bundles between the switches in the first and the second tier, bringing the overall number of link bundles in the network to $t \times k^2$. The number of links per ToR (excluding ToRs’ downlinks) equals the number of link bundles in the network, multiplied by the number of links per link bundle and divided by the number of ToRs ($2 \times t \times l$). The same logic is used to continue and build networks of 3, 4 and n tiers.

So far we have limited the discussion to the network devices within the DCN. Next, we consider the number of end hosts connected to this network. The number of end hosts will always be $O(k^n/2^{n-1})$ in a fully provisioned n tier network. The exact number of end-hosts will depend on the number of downlink ports from the ToR switch. This number is not necessarily t : the link bundle from a host to the ToR can be different than the link bundle from the ToR to the first switch. Furthermore, a ToR can be over-subscribed, in which case the number of uplink and downlink ports will be different. Given d downlink ports from a ToR, the number of end hosts connecting to an n tier network will be $d \times k^n/2^{n-1}$.

While over-subscription is most common at the ToR level,

Parameter	Description
k	switch radix
t	number of ToR uplink ports
l	number of links in a link-bundle

Table 1: The parameters used to describe a Fat-tree based network.

it is also possible within the network. This does not change the math of a 1-Tier network, but in a two (or n) tier network, each switch will use u uplink ports, and $k-u$ downlink ports, and the math will be adjusted accordingly: the maximum number of ToRs will be $k \times (k-u)$, and the number of switches will be $t \times (k+u)$.

B Parallel Processing of Packets

§2.3 discussed the number of processing cores required in a device. In this appendix we explain this calculation. Here we focus not on bandwidth, but rather on packet processing rate.

Let us assume a packet of size S bytes, running through an Ethernet switch of bandwidth B bits/sec. As this is Ethernet, there are preamble, start of frame delimiter (SFD) and inter frame gap between packets on the wire, G . The packet rate R required from a device is therefore:

$$R = B / (8 \times (S + G)) \quad (1)$$

Note that Equation 3 makes no assumption on the architecture and design of the switch.

Most packet switches today are pipelined (e.g. [19]) as a way to increase performance. The length of the pipeline does not affect the rate at which packets are being processed, rather the latency through the pipeline. Let us assume that each stage through the pipeline takes c clock cycles, and that the clock frequency of the device is f . This means that the number of packets r processed in the pipeline every second is:

$$r = f/c \quad (2)$$

Under optimal conditions, $c = 1$, as in a pipelined design the minimum time required per stage is one clock cycle. Note that an *operation* within a stage may take less than one clock cycle, but a stage will always be clocked (and sampled) in order to avoid metastability and close timing. This leads to $r = f$.

We define the amount of parallelism required in a switch P as:

$$P = R/r \quad (3)$$

P defines the ratio between the number of arriving packets that needs to be processed every second, and the rate at which the pipeline can process packets. Where $P > 1$ it means that

more packets need to be processed every second by the device than a single pipeline can process.

For example, let us assume packet size $S = 64B$, switch bandwidth of $B = 12.8Tbps$, gap $G = 12B + 8B = 20B$, clock frequency of $f = 1GHz$ and $c = 1$ clock cycle per packet. The parallelism required is 19.047:

$$\begin{aligned} R &= B / (8 \times (S + G)) = 19.047E + 9 \\ r &= f/c = 1E + 9/1 = 1E + 9 \\ P &= R/r = 19.047 \end{aligned} \quad (4)$$

In a similar manner, a packet size of 256B will require $P = 6.06$.

The way to handle $P > 1$ may vary between chip architectures. One common solution is to use more than one pipeline within a switch [39, 62]. Another solution is to process more than one packet within the pipeline every clock (in a manner not unlike CPU’s pipelines) - however such a solution is much more complex. Packet packing effectively increases the packet processing rate, but it operates on packed packets rather than on individual ones.

Recent devices have used four pipelines as a way to increase parallelism [39, 62], while processing one packet every clock ($c = 1$). This means that such devices likely support full line rate only above a certain packet size ⁷.

C Silicon Area

§7 briefly introduced the silicon area comparison of Stardust devices and a standard ToR switch. In this section we extend this discussion, based on the Table in Fig. 10(d).

Device A is a standard Ethernet switch (ToR), whereas device B is a Fabric Element [23]. A significant difference between devices A and B is in the area consumed by header processing. Device A supports advanced (programmable) header processing, whereas device B supports only simple cell header parsing. There is also a notable difference in the resources required for lookup tables: Device B only requires a reachability table, whereas device A requires standard forwarding tables, such as for IPv4 lookup. Assuming a network with N end hosts, and only 40 servers per rack, device A will require an exact match IPv4 table size $N \times (32 + \log_2 k)$, where k is the device’s radix, whereas Device B will only need to point to Fabric Adapters, thus its table size will be $(N/40) \times \log_2 k$. Using hashing will reduce the required memory size, but this simplified explanation gives a notion of the difference between the two devices.

In terms of I/O, both devices use the same libraries and therefore have a relatively close I/O area ratio⁸. However, the network interface modules beyond the I/O, e.g. the MAC, are significantly different. Device A requires a full-blown MAC

⁷This is an estimation based on current manufacturing processes, as f and c of switches are not published by vendors.

⁸The difference is for practical implementation reasons.

Tiers	Max ToRs	Max of Switches	Switches per ToR	# of Link Bundles	Links per ToR
1	k	t	t/k	$t \times k$	$t \times l$
2	$k^2/2$	$3/2 \times t \times k$	$3 \times t/k$	$t \times k^2$	$2 \times t \times l$
3	$k^3/4$	$5/4 \times t \times k^2$	$5 \times t/k$	$3/4 \times t \times k^3$	$3 \times t \times l$
4	$k^4/8$	$7/8 \times t \times k^3$	$7 \times t/k$	$7/8 \times t \times k^4$	$7 \times t \times l$
n	$k^n/2^{n-1}$	$(2 \times n - 1)/2^{n-1} \times t \times k^{n-1}$	$(2 \times n - 1) \times t/k$	$(1 - 1/2^{n-1}) \times t \times k^n$	$(2^{n-1} - 1) \times t \times l$

Table 2: The number of different elements within a multi-tier, Fat-tree based network. The maximum size of a network of n tiers using a switch with port radix k is $O((k/2)^n)$.

supporting different Ethernet standards, whereas Device B requires a simpler module to extract the cells. The gain in area per port is 70%.

For the remaining functionality of the device, the savings using a Fabric Element amounts to 40%. While in a standard switch, the amount of packet buffering is a compromise between silicon area and packet drop under different traffic scenarios, in the Fabric Element, cell buffering is agnostic to traffic scenarios.

In a Fabric Adapter [22], functionality supporting Stardust (e.g., cell generation, load balancing, and credit generation) takes about 8% of the device’s area. This area is largely compensated by the saving on network fabric facing interfaces, a gain of 70% per port. The number of VOQs supported is directly mapped to memory resources, where 128K VOQs consume roughly 4MB, an order of magnitude less than consumed by a header processing module [19]. The overall area of the Fabric Adapter is very similar to device A.

The area of a device can vary based on a number of factors. For example, an Ethernet switch can trade-off buffer space for packet processing functionality or lookup table sizes, and a Fabric Element may trade-off cell buffering for area. A Fabric Element based on the design choices in [19] will require just 45% of the area of an RMT switch⁹. Table 10 does represent, however, existing devices and not a theoretical case.

D Cost Estimation

Cost estimation is conducted using list prices collected from online resellers on September 12, 2018. The prices are listed in Table 3. The websites are authorized resellers of the listed components. Prices are used to calculate cost ratio, rather than as indicative costs. Our calculations do not take into account other real-world considerations that affect cost, such as the pricing for high quantities or a buyer’s bargaining power.

E Resilience

Recovering from a link failure follows a local detection of a link failure, and the propagation of this information to all other devices. Local detection of a failure is done on a nanosecond scale, as a result of a loss of signal or high bit error rate (BER). The time it takes to propagate the information using reachability messages is configurable. This ap-

Item	Price [USD]	Source
Edgecore AS7816-64X 64-Port 100GbE	\$16200	Colfax
Edgecore Wedge 100BF-65X 65-Port 100GbE	\$16200	Colfax
Passive Copper Cable 100GbE, 2 meters	\$84	Colfax
100G QSFP28 Optical Module Short Range	\$435	Colfax
50G QSFP28 Optical Module Short Range	\$280	Estimated
25G SFP28 Optical Module Short Range	\$125	Colfax
Fiber, 10m	\$8	FS
Fiber, 100m	\$62	FS

Table 3: Indicative component costs - List prices.

pendix formulates this time, and Table 4 lists the parameters used below.

Let f be the core frequency of a device, and c the configurable number of clock cycles between messages, per link. The time t' between every pair of messages will be:

$$t' = \frac{c}{f}$$

This means a message is generated every c/f seconds. Assume that each reachability message is of B bytes and includes a bitmap of b bits indicating the reachability of b Fabric Adapter devices. Assume that each Fabric Adapter connects to h hosts. The number of messages M required to propagate the reachability table of a DCN connecting N hosts will be:

$$M = \left\lceil \frac{N}{h \times b} \right\rceil$$

If a network has n tiers, then at the worst case, the link that failed was between a Fabric Adapter and a Fabric Element, which means that the information needs to propagate through the entire network ($2 \times n$ hops), minus one hop (the failed link). The time it would take to propagate the updated reachability table through the entire network would be:

$$t = t' \times M \times (2 \times n - 1) = \\ \frac{c}{f} \times \left\lceil \frac{N}{h \times b} \right\rceil \times (2 \times n - 1)$$

⁹ [19] has insufficient data to compare to the Fabric Adapter.

Parameter	Description	Example Value
f	Core frequency	1GHZ
c	Number of cycles between messages	10,000
t'	Time between reachability messages	10μs
b	Reachability bitmap size	128
B	Reachability message size	24B
h	Number of hosts per Fabric Adapter	40
N	Number of hosts connected to the DCN	32,000
n	Number of tiers	2
M	Number of messages propagating a full reachability table	7
th	Reachability status change threshold	3
pd _i	Propagation delay through link i	50ns, 500ns
s	link speed	50Gbps

Table 4: The parameters used to describe the propagation of reachability updates.

To avoid momentary effects and account for a potential loss of a reachability message during the updates, one would usually opt for multiple updates of a link’s value (th) before updating its state, which means the time it takes to react to a link failure would be $t \times th$.

A slightly more accurate calculation will also take into account the propagation delay on each link (pd_i):

$$t \times th = \sum_{i=1}^{2 \times n-1} \left((t' + pd_i) \times M \times th \right) = \\ \sum_{i=1}^{2 \times n-1} \left(\frac{c}{f} + pd_i \right) \times \left\lceil \frac{N}{h \times b} \right\rceil \times th$$

The value of c is determined such that the effect of reachability messages on the available bandwidth will be minimal. Assume that the link speed is s . The relative bandwidth consumed by reachability messages would be:

$$\frac{B \times 8}{t' \times s} = \frac{B \times 8 \times f}{c \times s}$$

Using the example values indicated in Table 4, the overhead of reachability updates is 0.04% of the bandwidth, and the time it takes to recover from a failed link, which is the time it takes to propagate an update to the farthest Fabric Adapter, is 652μs. Note that the difference from the illustrative example in §5.9 is the propagation delay on the links.

F Push Fabric vs Pull Fabric

§5.2 made the case for Pull Fabric vs Push Fabric, with no traffic classes. Here we illustrate the case of traffic classes on an Ethernet based network fabric, compared with Stardust. The scenario is depicted in Fig. 12. On a single device there are two 100GE ports, A and B. From one input device, 100Gbps of a *high* traffic class are injected toward A (marked red), and additional 100Gbps of a *low* traffic class are injected toward B (marked green). Additional 100Gbps of a *high* traffic class are injected towards A from a second

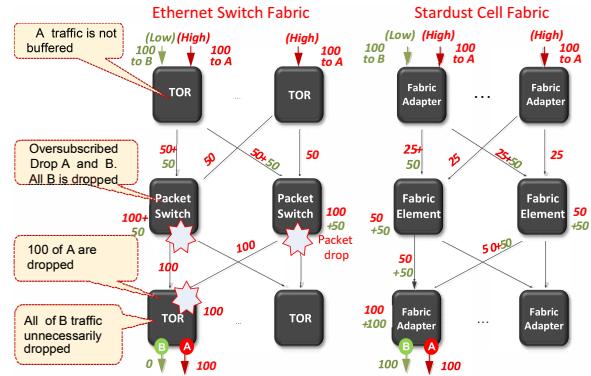


Figure 12: Packet drops in Ethernet switch fabric vs. Stardust scheduled fabric with traffic classes

device (marked red). As a result, B’s traffic will be pushed to the fabric, where all of it will be dropped at the packet switch. The throughput of the destination ToR will be 100Gbps, all toward port A.

In Stardust, credits are sent separately from port A and from port B to each ingress Fabric Adapter. Each input device receives credits sufficient for just 50Gbps toward port A, and the first device received credits sufficient to send 100Gbps toward port B. As cells are distributed across all links, the traffic toward A and B is evenly divided between the links, meaning that no link is over-subscribed and no traffic is dropped within the network. The eventual throughput from Stardust is twice the throughput using the standard Ethernet switch.

G Performance Simulations

The simulator of choice for performance comparison is *htsim* [72]. Htsim is used as it provides, under an open-source license, implementations of several data center protocols (TCP, MPTCP, DCTCP, DCQCN and NDP), along with scripts allowing the experiments of [41] to be reproduced. The simulator used in §6.2 is unsuitable for this purpose, as it is an architecture-specific, low-level simulator, that would take days to run experiments that are conducted in htsim within minutes.

The setup used in §6.3 reproduces the setup used in [41], using a 432-node Fat-tree. All links in the system are of 10Gbps, and the same number of tiers is used across protocols and the experiments. Although the htsim simulator is fast, it still takes many hours to simulate a permutation matrix of Stardust in a 432 node Fat-tree.

Our simulations reproduce as-is the experiments from [41] using DCTCP with ECN, DCQCN and MPTCP. We implement a model of Stardust into this environment. We use an unmodified TCP (New Reno) on top of Stardust, which is the least favorable scenario. While the Fabric Adapter supports sending congestion notifications to the host, the presented results do not take advantage of such features, presenting ‘raw’ Stardust performance. DCTCP and MPTCP use 100 packet output queues, and DCQCN uses 100 buffers per port, shared between interfaces. Stardust uses similar buffering resources, providing an apple-to-apple comparison.

The Stardust configuration used in the experiment uses 512B cell size, 4KB credit size and 3% credit speed up. The use of 512B cell size and 4KB credit size is intended to reduce simulation time. A simulation using 256B cells and 2KB credits takes 12 hours per permutation on a Xeon E5-2660 v4 server. Both settings are realistic, though a smaller credit size will improve fairness in the given setup. The scheduler at the egress Fabric Adapter uses a simple round robin between all flows, intended to show fairness. Other scheduling schemes are also supported by the Fabric Adapter (§4).

The performance simulations use different workloads.

The throughput simulation (Fig. 10(a)) uses continuous flows and 9000B packets, which favors existing data center protocols. The FCT simulation uses the Facebook Web workload [74] flow size distribution, which was implemented by [41]. Background traffic is similar to the throughput simulation. The Incast experiment, sending traffic from an increasing number of sources to a single destination, uses 450KB flow size and 9000B packets. In all the simulations, the packets are chopped into cells by Stardust.

The simulation results presented in this paper do not capture all the minutiae of the Stardust design, for example, avoiding synchronization issues when distributing cells across links. Consequently, the actual performance of Stardust, e.g., as measured in §6.1, is better than simulated in §6.3. Most of the aforementioned features are not implemented due to the level of abstraction provided by the simulator. Htsim treats the network as a collection of pipes and queues, and is oblivious to hardware limitations, such as described in §2.

DCQCN is not included in the incast experiment depicted in Fig. 10(b) as it is missing from the reproducibility package of [40], and due to insufficient resources to accurately re-implement it. We opt to omit NDP from our performance comparison as we find that it is very sensitive to the experimental setup and configuration. For example, a very realistic scenario of using 1500B packets rather than 9000B reduces its mean utilization by 14%. In a different case, changing the simulation’s seed resulted in performance collapse. We refer the reader to [83] for a detailed analysis.

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach*, Edgar Costa Molero*, Maria Apostolaki*
Alberto Dainotti†, Stefano Vissicchio‡, Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: (i) select which ones to track; (ii) reliably and quickly detect major traffic disruptions; and (iii) recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: (i) achieves sub-second rerouting for large fractions of Internet traffic; and (ii) prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.*, Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon remote failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix

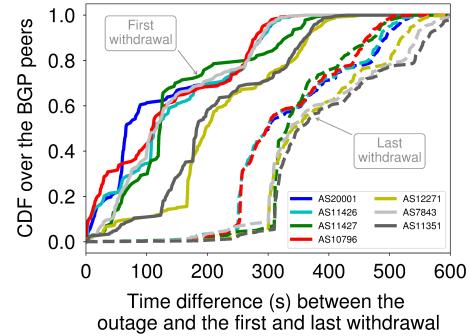


Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.*, they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(\text{minutes})$ for the *first* BGP update to propagate after the corresponding data-plane failure.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0 , the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

In short, a fundamental question is still open: *Is it possible to build a fast-reroute framework for ISPs that can converge in $O(\text{seconds})$ for both local and remote failures?*

Blink: fast, data-driven convergence upon remote failures. We answer this question affirmatively by developing Blink, a *data-driven* fast-reroute framework built on top of programmable data planes. Blink key insight is to reroute based on data-plane signals rather than control-plane ones. Quickly after a failure, data-plane traffic indeed exhibits a predictable behavior: all the TCP endpoints start retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. With Blink, we show that this signal can be efficiently tracked at line rate and enables sub-second convergence after most remote failures.

Key challenges. Tracking failure signals in the data plane is challenging for at least three reasons. First, monitoring all flows is impossible because of memory constraints. At the same time, randomly sampling flows often results in tracking useless flows, *e.g.*, ones that seldom transmit. We address this problem by developing a *flow selector* which automatically evicts inactive flows and replaces them with active ones.

Second, packet loss routinely happens in the Internet, *e.g.*, due to temporary congestion. Rerouting upon any retransmission would result in huge, and counterproductive traffic shifts. We address this challenge by: (i) focusing on timeout-induced retransmissions, which are infrequent (as we confirmed analyzing real traces); and (ii) leveraging the fact that failures affect many flows simultaneously.

Third, data-plane signals provide no information about the root cause of the problem. Worse, uncoordinated rerouting decisions can lead to forwarding issues such as blackholes, forwarding loops, and oscillations. In Blink, we solve these problems by also making the backup selection data-driven, *i.e.*, by tracking if flows resume after rerouting them.

We fully implemented Blink in P4₁₆. Our evaluation, which includes experiments on a real Barefoot Tofino switch, shows that Blink retrieves connectivity within 1 s for the vast majority of the considered failure cases.

Main contributions. Our main contributions are:

- A new approach for quickly recovering connectivity upon remote failures based on data-plane signals (§2).
- The Blink pipeline, which enables programmable data planes to track failure signals at line rate and to automatically retrieve connectivity (§3 and §4).
- An implementation¹ of Blink in Python and P4₁₆ (§5).
- An evaluation of Blink using synthetic and real packet traces, emulations, and hardware experiments (§6).
- A discussion on how to deploy Blink, along with how to protect it from malicious and crafted traffic (§7).

¹Our source code is available at: <https://blink.ethz.ch>

2 Key Principles and Challenges

In this section, we first show that TCP traffic exhibits a characteristic pattern upon failures (§2.1). We then discuss the key challenges and requirements to detect such a pattern, and recover connectivity by rerouting the affected prefixes, while operating entirely in the data plane, at line rate (§2.2).

2.1 Data-plane signals upon failures

Consider an Internet path (A, B, C, D) carrying tens of thousands of TCP flows, destined to thousand prefixes, in which the link (B, C) suddenly fails. We are interested in monitoring the data-plane “failure signal” perceived at A , with the goal of enabling A to detect it and to also recover connectivity by rerouting traffic through a different path (if any). Observe that A is not adjacent to the failure, *i.e.*, the failure is remote.

As the link (B, C) fails, the TCP endpoints stop receiving acknowledgements (ACKs), and each of them will timeout after its retransmission timeout (RTO) expires, which will cause it to reset its congestion window to one segment and start retransmitting the first unacknowledged segment. Since the RTO is computed according to the RTT observed, each TCP endpoint will retransmit at a different time. Specifically, each TCP endpoint adjusts its RTO using the following relation: $\text{RTO} = \text{sRTT} + 4 * \text{RTTVar}$ (see [31]), where sRTT corresponds to the smoothed RTT, and RTTVar corresponds to the RTT variation. After each retransmission, each TCP endpoint further doubles its RTO (exponential backoff).

We illustrate the behavior of a TCP flow experiencing a failure in Figure 2. We assume that the TCP endpoint has an estimated RTO of 200 ms and that its congestion window can hold 4 packets. We denote by t the time at which the TCP endpoint transmits the first packet following the failure. The TCP endpoint experiences consecutive RTO expirations and retransmits the packet with sequence number 1000 at time $t + 200\text{ms}$, $t + 600\text{ms}$, $t + 1400\text{ms}$, etc. We experimentally verified that this behavior is similar across all TCP flavors implemented in the latest Linux kernel.

When multiple flows experience the same failure, the signal obtained by counting the overall retransmissions consists of “retransmission waves”. Since this behavior is systematic, pronounced, and quick, we leverage it in Blink to perform failure detection in the data plane. This suggests Blink does not depend on specific TCP implementation details and would keep working effectively with future congestion control algorithms as long as they exhibit a similar behavior upon failures.

Note however the shape of these retransmission waves, *i.e.*, their amplitude and width, depends on the distribution of the estimated RTTs. As an illustration, Figure 3 shows the retransmission count for a trace that we generated with the ns-3 simulator [3] after simulating a link failure (according to the methodology in §6.1). In the left diagram, we used the distribution of the average RTTs of the TCP flows from an actual traffic trace (#8 in Table 3 in §E). In the right diagram, we increased the RTTs of this distribution by 1.5 to obtain

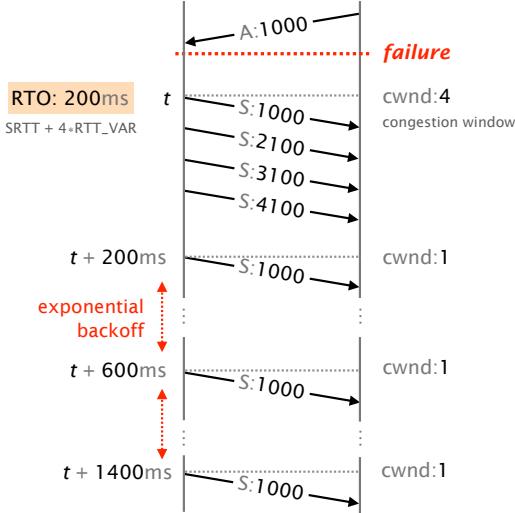


Figure 2: After the failure, a TCP flow keeps retransmitting the last unacknowledged segment according to an exponential backoff. The exact timing of the retransmissions depends on the estimated RTO before the failure (here 200ms).

a larger standard deviation. We can clearly see the waves of retransmissions appearing within a second after each failure. RTT distributions with small variance make the flows more synchronized they will be when retransmitting. This translates into narrow peaks of retransmissions with a high amplitude. Conversely, if the flows have very different RTTs (*i.e.*, the variance is high), the peaks will have a smaller amplitude and will spread over a longer time. We elaborate on the challenges deriving from these observations hereafter.

2.2 Key challenges and requirements when fast rerouting using data-plane signals

We now highlight four key challenges and requirements that must be addressed to: (i) efficiently capture the failure signal we just described; and (ii) recover connectivity. We describe in §3 how does Blink address them entirely in the data plane.

Dealing with noisy signal. To discover its fair share of bandwidth, a TCP endpoint keeps increasing its transmission rate until a packet loss is detected, triggering a retransmission. TCP retransmissions therefore occur naturally, even without network failures. Likewise, minor temporary congestion events can also lead to bursts of packet drops, which will trigger subsequent bursts of retransmissions, again, without necessarily implying a failure.

Requirement 1: A data-plane-driven fast-reroute system should only react to major disruptive events while being immune to noise and ordinary protocol behavior.

Dealing with fading signals. As shown in Figure 3, the amplitude of the signal (*i.e.*, the count of TCP retransmissions)

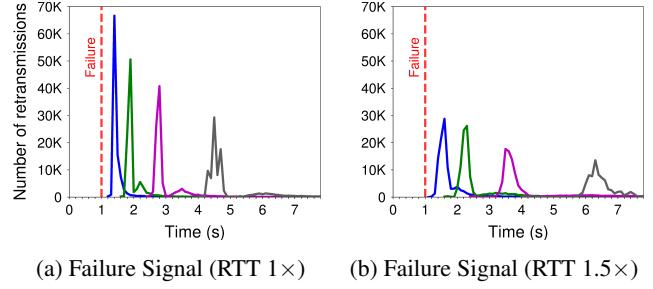


Figure 3: The signal generated by TCP flows experiencing a connectivity problem is characteristic and composed of subsequent waves of retransmissions (in different colors). The waves have decreasing amplitude and increasing width.

quickly fades with the backoff round as the compounded signal spreads over longer and longer periods.

Requirement 2: A data-plane-driven fast-reroute system should catch the failure signal within the first retransmission rounds.

Mitigating the effect of sampling. As tracking retransmissions in real-time requires state, monitoring *all* flows is not possible. As such, a fast-reroute system will necessarily have to track and detect failures using a subset of the flows. Yet, not all flows are equally useful when it comes to failure reaction: intuitively, highly active flows will retransmit almost immediately, while long-lived flows might not retransmit at all (*if* no packet was sent recently). From a fast-reroute viewpoint, tracking non-active flows is useless.

Requirement 3: A data-plane-driven fast-reroute system should select the flows it tracks according to their activity.

Ensuring forwarding correctness without control plane. While data-plane signals are faster to propagate than control-plane ones, they carry no information about the cause of the failure and how to avoid it. As such, simply rerouting to a backup next-hop upon detecting a problem might not work, as it might also be affected by the failure. Worse, the problem can even be at the destination itself, in which case no alternative next-hop will actually work. Given this lack of precise information, a data-plane-driven fast-reroute system has no other choice but trying and observing.

Requirement 4: A data-plane-driven fast-reroute system should select its backup next-hops in a data-driven manner, verifying that traffic resumes.

3 Overview

In this section, we provide a high-level description of Blink. We first focus on its data-plane implementation (§3.1) and how it: (i) selects flows to track; (ii) detects failures; and (iii) reroutes traffic. We then describe how Blink can be deployed at the network level (§3.2).

3.1 Blink, at the node level

Figure 4 describes the overall workflow of a Blink data-plane pipeline. The pipeline is essentially composed of three consecutive stages: (i) a *selection* stage which efficiently identifies active flows to monitor; (ii) a *detection* stage which analyzes RTO-induced retransmissions across the monitored flows and looks for any significant increase; and (iii) a *rerouting* stage which is in charge of retrieving connectivity by probing alternative next-hops upon failure. We now briefly describe the key ingredients behind each stage and provide details in §4.

Selecting flows to track. For efficiency and scalability, a Blink node cannot track all possible 700k+ IP prefixes or even all the flows destined to some prefixes. An initial design choice thus concerns which prefixes to track, and which specific flows to track for the selected prefixes.

Any approach based on data-plane signals is able to effectively monitor only the prefixes carrying a certain amount of packets. Blink is no exception, and therefore focuses on the most popular destination prefixes. While this might seem a limitation, it is actually a feature: the Internet traffic is typically skewed [32], and a very limited fraction of prefixes carries most of the traffic, while the rest of the prefixes see little to none. Blink can thus reroute the vast majority of the traffic by tracking a limited number of prefixes. We designed Blink to accommodate at least 10 k prefixes in current programmable switches (§5).

Regarding which flows to track for a prefix, Blink adopts a simple but effective strategy. For each monitored prefix, the Flow Selector tracks a very small subset (64, by default) of *active flows*—*i.e.*, flows that send at least one packet within a moving time window (2 s by default). Tracked flows are replaced as soon as they become inactive, or after a given timeout (8.5 min by default) even if they remain active. We did not reuse heavy hitter detection algorithms such as [36], since they are designed to offer higher accuracy than we need (heaviest flows instead of just active ones) at the expense of additional complexity and resources.

Detecting failures. A central idea of our approach is to infer remote failures, affecting a destination prefix, from the loss of connectivity for a statistically significant number of previously active flows towards that destination. While possible in principle, Blink does not look at the flows progression to detect a failure, as only a subset of the flows may be affected, *e.g.*, because of load-balancing. This enables Blink to detect partial failures (see §D).

The detection stage looks for evidence of connectivity dis-

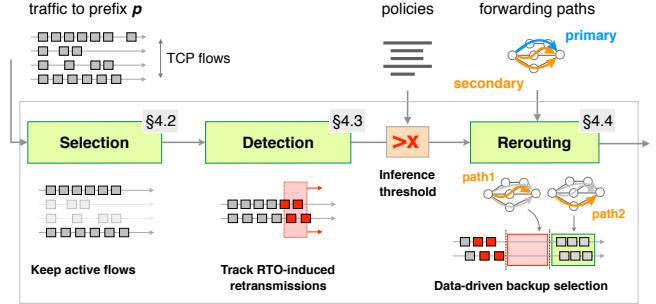


Figure 4: Blink data-plane workflow and key ingredients.

ruption across the flows identified by the Flow Selector. It stores key information on the last seen packet for each flow and determines if a new packet traversing the data-plane pipeline is a duplicate of the last seen one – an hallmark of RTO-induced TCP retransmissions (see Figure 2). Based on this check, for each destination prefix, it monitors the number of flows with at least one recent retransmission over a sliding time window of limited size (800 ms, by default). When the majority of the monitored flows experience at least one retransmission in the same time window, Blink infers a failure.

Rerouting quickly. When Blink detects a failure, the Rerouting Module quickly reroutes traffic by modifying the next-hop to which packets are forwarded, at line rate. In Blink’s current implementation, the decisions of both when to reroute and to which backup next-hop to reroute are configurable by the operator based on their policies, as we believe that operators want to be in charge of this critical, network-specific operation.

When rerouting, the Rerouting Module sends few flows to each backup path to check which one is able to restore connectivity. It then uses the best and working one for all the traffic. The next-hops are configured at runtime by the operator to re-align the data-plane forwarding to the control-plane (*e.g.*, BGP) routes when the control plane has converged.

3.2 Blink, at the network level

The “textbook” deployment of Blink consists in deploying it on all the border routers of the ISP to track all the transit traffic. In this deployment, border routers either reroute traffic locally (if possible) or direct it to another border router (*e.g.*, through an MPLS tunnel). Of course, nothing prevents the deployment of Blink inside the ISP as well. In fact, Blink also works for intra-ISP failures, *e.g.*, local to the Blink node or on the path from the Blink switch and an ISP egress point.

Blink is partially deployable. Deploying Blink on a single node already enables to speed up connectivity recovery for all traffic traversing that particular node. Also, Blink requires no coordination with other devices: each Blink node autonomously extracts data-plane signals from the traversing packets, infers major connectivity disruptions, and fast reroutes accordingly. To avoid forwarding issues, Blink veri-

fies the recovery of connectivity for the rerouted packets by monitoring the data plane (see §4.4.2).

When rerouting, Blink also notifies the control plane, and possibly the ISP operator. This enables coordination with the control plane (*e.g.*, future SDN controllers), such as imposing the next-hop upon control-plane convergence, or discarding routes that are not working in the data plane.

4 Data-plane design

In this section, we describe the data-plane pipeline that runs on a Blink node, its internal algorithms, design choices and parameter values (that we further discuss in §A). Figure 5 depicts the four main components of the Blink data-plane pipeline: the Prefix Filter (§4.1), the Flow Selector (§4.2), the Failure Inference (§4.3), and the Rerouting Module (§4.4).

4.1 Monitoring the most important prefixes

To limit the resources used by Blink, the operator should activate Blink only for a set of important prefixes. A sensible approach would be to activate it for the most popular destination prefixes, as they carry most traffic, although nothing prevents the operator to select other prefixes – as long as there is enough TCP traffic destined to each of them (§6.1.1). To activate Blink for a prefix, the control plane adds an entry in the metadata table at runtime which matches the traffic destined to this prefix using a longest prefix match. Traffic destined to a prefix for which Blink is not active goes directly to the last stage of the data-plane pipeline and is forwarded normally (*i.e.*, find the next-hop and replace the layer 2 header).

The metadata table attaches to the matched packets a distinct ID according to their destination prefix. As memory (*e.g.*, register arrays) is often shared between the prefixes, this ID is used as an index to the memory. Observe that Blink could combine prefixes with common attributes (*e.g.*, origin AS or AS path) and which are likely to fail at the same time by mapping them to the same ID. This would increase the intensity of the signal, and would allow Blink to cover more traffic. Additionally, packets that do not carry useful information, *i.e.*, non-TCP traffic, and certain signaling-only packets such as SYN and ACK packets with no payload are not considered by Blink and are directly sent to the final stage.

4.2 Selecting active flows to monitor

Packets destined to a monitored prefix go to the Flow Selector, which will select a limited number of active flows (64 per prefix), and keep information about each of them.

Limiting the number of selected flows. Each flow for a given prefix is mapped to one of the 64 cells of a per-prefix flow array using a 6-bit hash of the 4-tuple². While we expect many flows to collide in the same cell, only one occupies a cell. This is enforced by storing the `flow_key`, namely a 32-bit hash of the 4-tuple in each cell of the flow array.

Flows colliding in the same cell are possible candidates to substitute the flow currently occupying that cell when it becomes inactive. It can happen that two flows mapped to the same cell have the same `flow_key`, in which case both would end up occupying the same cell, causing Blink to mix packets from two distinct flows and thus preventing it to correctly detect retransmissions for either flows. However, since we use a total of 38 bits (6 bits to identify the cell and 32 bits for the `flow_key`) to identify each flow, such collisions will rarely happen. The probability of collision can be computed from a generalization of the *birthday problem*: given n flows, what is the probability that none of them returns the same 38 bit hash value? This probability is equal to $\frac{2^{38}}{(2^{38}-n)!} * \frac{1}{2^{38^n}} \approx e^{-\frac{n(n-1)}{2^{38}}}$. With $n = 10,000$ flows for a given prefix, the probability to have a collision is only 0.02%.

Replacing inactive flows. The challenge behind selecting active flows is that flows have different packet rates, which also change over time, *e.g.*, an active flow at time t may not be active anymore at time $t + 1s$. A naive *first-seen, first-selected* strategy would clearly not work, because the selected flows might send packets at such a low rate that they would not provide any timely information upon a connectivity disruption – simply because there is no packet to retransmit³.

The Flow Selector monitors the activity of each selected flow by tracking the timestamp of the last packet seen in the register `last_pkt_ts`. As soon as the difference between the current timestamp and `last_pkt_ts` is greater than an eviction timeout, the flow is evicted and immediately replaced by another flow colliding in the same cell. A TCP FIN packet also causes immediate eviction. Intuitively, flow eviction makes the Flow Selector work very well for prefixes which have many high-rate flows at any moment in time, or a decent fraction of long-living ones – which we expect to be often the case for traffic towards popular destinations. Our evaluation on real traffic patterns (see §6) confirms that this simple strategy is sufficient to quickly infer major connectivity disruptions.

Calibrating the eviction timeout. A remaining question for this component of the pipeline is how to dimension the eviction timeout. On one hand, we would like to evict flows as soon as their current packet rate is not amongst the highest for that prefix. On the other hand though, Blink needs to keep track of the flows long enough to see the first few packet retransmissions induced by a RTO expiration upon connectivity interruptions. Indeed, an eviction timeout of few hundred milliseconds is likely to be too low in many cases, since a flow takes *at least* 200ms to issue the first pair of duplicate packets⁴ (see §2). By default, the eviction timeout is set to 2 s, which ensures to detect up to two pairs of consecutive duplicates for typical TCP implementations.

³Our experimental evaluation in §6 confirms this intuition

⁴200ms only happens if there is no new packet between the first unacknowledged packet and the first retransmission. Also, remember that Blink considers only consecutive duplicates as packet retransmissions.

²The 4-tuple includes source and destination IP and the port numbers.

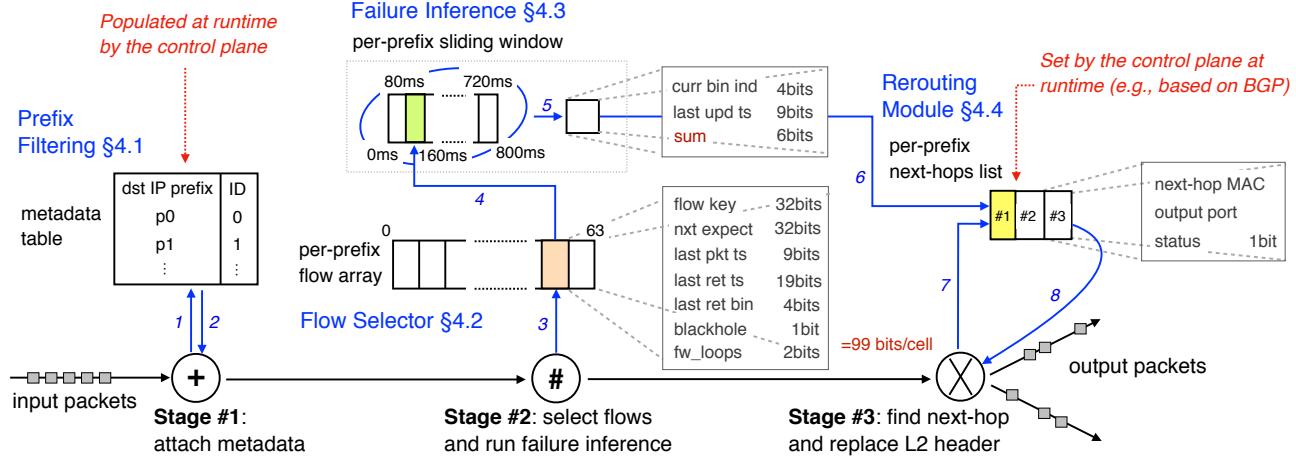


Figure 5: Blink data-plane pipeline and its data structures. The blue arrows indicate all the steps a packet goes through.

4.3 Detecting failures

We now describe how Blink detects RTO-induced retransmissions on the set of selected flows, and uses this information to accurately infer failures.

Detecting RTO-induced retransmissions. A partial or full retransmission of payload of the TCP packet can be detected by comparing, the sum of its sequence number and payload length to the corresponding sum of the previous packet of the same flow. For example, in Figure 2 when the packet S:1000 (packet with sequence number 1000) arrives Blink will store 2100 (sum of sequence number and payload). If the next arriving packet triggers storing 2100 as well, a retransmission is detected. Observe that we store the expected sequence number per flow instead of the current to account for cases where a packet is only partially acked.

Our design targets consecutive retransmissions for two reasons. First, RTO-induced retransmissions are consecutive (see Figure 2), whereas congestion-induced retransmissions (*i.e.*, noise) are likely to be interleaved by non-retransmissions, and hence will (correctly) not be detected. Second, this detection mechanism requires a fixed number of memory per flow, regardless the flow’s packet rate.

Counting the number of flows experiencing retransmissions over time. Figure 3 shows that the TCP signal upon a failure is short and fading over time. To quickly and accurately detect the compounded signal across multiple flows, we use a per-prefix sliding window. To implement a sliding window of size k seconds in P4, we divide it in 10 consecutive bins of 6-bit each, each storing the number of selected flows experiencing retransmissions during $k/10$ seconds. As a result, instead of sliding for every packet received, the sliding window moves every $k/10$ seconds period. More bins can improve the precision but would require more memory. This design enables us to implement the sliding window in P4 using only three information per prefix: (i) `current_index`, the

index of the bin focusing on the current period of time, (ii) `sum`, the sum of all the 10 bins, and (iii) `last_ts_sliding`, the timestamp in millisecond precision⁵ of the last time the window slid. The additional 19-bit and 4-bit per-flow information `last_ret_ts` and `last_ret_bin` are also required to ensure that a flow is counted maximum one time during a time window. We provide more details about the implementation in §B.

Calibrating the sliding window. The duration of the sliding window affects the failure detection mechanism. A long time window (*e.g.*, spanning several seconds) has more chance to include unrelated retransmissions (*e.g.*, caused by congestions), whereas a short time window (*e.g.*, 100ms) may miss a large portion of the retransmissions induced by the same failure because of the different RTO timers. We set the duration of the sliding window to 800ms, with 10 bins of 80ms. First, because the minimum RTO is 200ms, a 800ms sliding window ensures to include all the retransmissions induced by the failure within the first second after the failure. Second, because under realistic conditions (in terms of RTT [5, 18, 34] and RTT variation [5]), flows would often send their first two retransmissions within the first second after the failure.

Infering failures. A naive strategy consisting in inferring a failure when *all* the selected flows experience retransmissions would result in a high number of false negatives due to the fact that some flows may not send traffic during the failure, or simply because some flows have a very high RTT (*e.g.*, >1 s). On the other hand, inferring a failure when only few flows experience retransmissions may result in many false positives because of the noise. As a result, by default Blink infers a failure for a prefix if the majority of the monitored flows (*i.e.*, 32) destined to that prefix experience retransmissions.

⁵We explain in §B how Blink can obtain millisecond precision

4.4 Rerouting at line rate

As soon as Blink detects a failure for a prefix, it immediately reroutes the traffic destined to it, at line rate. We first show in §4.4.1 how Blink maintains the per-prefix next-hops list used for (re)routing traffic. Then, we show in §4.4.2 how Blink avoids forwarding issues when it reroutes traffic.

4.4.1 Maintaining the per-prefix next-hops list

To reroute at line rate, Blink relies on pre-computed per-prefix backup next-hops. The control plane computes the next-hops consistently with BGP routes and specific policies defined by the operator. For each prefix, Blink maintains a list of next-hops, which are sorted according to their preference (see Figure 2). Each next-hop has a status bit. To reroute at line rate, Blink deactivates the primary next-hop by setting its status bit to 1 (*i.e.*, not working). Per-prefix next-hops are stored in register arrays and are updated at runtime by the controller, *e.g.*, whenever a new BGP route is learned or withdrawn. If a next-hop is not directly connected to the Blink node, Blink can translate it into a forwarding next-hop using IGP (or MPLS) information, as a normal router would do.

Falling back to the primary next-hop after rerouting. After an outage, BGP eventually converges and Blink updates the primary next-hop and use it for routing traffic. However, Blink cannot know when BGP has fully converged. Our current implementation waits for a fixed time (*e.g.*, few minutes, so that BGP is likely to have converged) after rerouting before falling back to the new primary next-hop. We acknowledge that this approach might not be optimal (*e.g.*, it potentially sacrifices path optimality), but it guarantees packet delivery by using policy-compatible routes and avoids possible disruptions caused by BGP path exploration [28]. Investigating a better interaction with the control plane is left for future work.

4.4.2 Avoiding forwarding issues

Since Blink runs entirely in the data plane, it likely reroutes traffic before receiving any control-plane information possibly triggered by the disruption. In addition, even when carefully selecting backup next-hops (*e.g.*, by taking the most disjoint AS path with respect to the primary path), we fundamentally cannot have a-priori information about where the root cause of a future disruption is, or where the backup next-hop sends the rerouted traffic after the disruption. As a result, Blink *fundamentally cannot prevent* forwarding issues such as blackholes (*i.e.*, when the next-hop is not able to deliver traffic to the destination) or forwarding loops to happen. The good news, though, is that Blink includes mechanisms to *quickly react to forwarding issues* that may inevitably occur upon rerouting.

Probing the backup next-hops to detect anomalies. When rerouting, Blink reacts to forwarding anomalies by probing each backup next-hop with a fraction of the selected flows in order to assess whether they are working or not. For example, with 2 backup next-hops, one half of the selected flows is

rerouted to each of them. The non-selected flows destined to this prefix are rerouted to the preferred and working backup next-hop. Blink does this in the data plane using the per-prefix next-hops list.

When a backup next-hop is assessed as not working, Blink updates its status bit. After a fixed period of time since rerouting (1s, by default), Blink stops probing the backup paths and uses the preferred and working one for all the traffic, including the selected flows. If all the backup next-hops are assessed as not working, Blink reroutes to the primary next-hop and falls back to waiting for the control plane to converge.

Avoiding blackholes. Blink detects blackholes by looking at the proportion of restarted flows. After rerouting, Blink tags a flow as restarted by switching its blackhole bit to 1 as soon as it sees a packet for this flow which is not a retransmission. When the probing period is over, Blink assesses a backup next-hop as not working if less than half of the flows routed to that next-hop have restarted. The duration of the probing period (1s) is motivated by our goal of restoring connectivity at a second-level time scale, while also providing retransmissions with a reasonable time for reaching the destination through the backup next-hop and triggering the restart of the flows. For example, if Blink reroutes 778 ms after a failure (the median case, see §6.1.1) and assuming a reasonable RTT (*e.g.*, the median case in [5, 18, 34]), it is likely that the rerouted flows will send a retransmission and receive the acknowledgment (if the next-hop is working) within the following 1 s period.

Breaking forwarding loops. Blink detects forwarding loops by counting the number of duplicate packets for each flow. The key intuition is that forwarding loops have a quite strong signature: the same packets are seen over and over again by the same devices. This signature is very similar to the TCP signature upon a failure, where TCP traffic sources start resending duplicate copies of the same packets for every affected flows, at increasingly spaced epochs. As a result, the algorithm used by Blink to detect retransmissions also detects looping packets. To differentiate between normal retransmissions and looping packets, Blink relies on the delay between each duplicate packet. TCP can send for a flow up to 2 retransmissions in 1 s because of the exponential backoff (see §2.1), whereas a packet trapped in a forwarding loop can be seen many more times by the Blink node. Hence, Blink counts the number of duplicate packets it detects for each flow after the rerouting using the information fw_loops stored in each cell of the flow array, and tags a backup next-hop as not working by switching its status bit as soon as it detects more than 3 duplicate packets for a flow rerouted to this backup next-hop.

Observe that this mechanism reacts very quickly to the most dangerous loops, *i.e.*, the ones that recirculate packets very fast and hence are most likely to overload network links and devices. Longer and slower loops are mitigated in at most 1 s, as Blink assumes the respective next-hop cannot deliver packets to the destination (*i.e.*, there is a blackhole).

5 Implementation

We have fully implemented the data-plane pipeline of Blink as described in §4 in ≈ 900 lines of P4₁₆ [37] code and in Python. We have also developed a P4_{Tofino} implementation of Blink that runs on a Barefoot Tofino switch [8]. Our P4_{Tofino} implementation currently only supports two next-hops, one primary and one backup. Unlike our P4₁₆ implementation, our P4_{Tofino} implementation uses the resubmission primitive⁶ in two cases: whenever the Flow Selector evicts a flow, or if two retransmissions from same flow are reported within 800ms, *i.e.*, the duration of the sliding window. Note that these two cases only occur for the set of selected flows (*i.e.*, only 64 flows per prefix, see §4.2).

Our implementations of Blink only require one entry in the metadata table, as well 6418 bits of memory (*i.e.*, registers) for each prefix monitored. This number is fixed, *i.e.*, only this amount of memory is required regardless of the amount of traffic. This is an important feature for a system such as Blink, which is intended to run on hardware with strong limitations. On current programmable switches, we expect Blink to support at least 10k prefixes. We explain in §C how we precisely derive the resources required by Blink.

6 Evaluation

We evaluate Blink’s accuracy, speed, and effectiveness in selecting a working next-hop based on simulations and synthetic data (§ 6.1, § 6.2). We then evaluate Blink using real traces and actual hardware (§ 6.3).

6.1 Blink’s failure detection algorithm

Packet traces of real Internet traffic are hard to gather, and for the few traces publicly available [9, 12], there is no ground truth about possible remote failures on which Blink should reroute. Still, it makes little sense to evaluate Blink on traffic with non-realistic characteristics, or without knowing if Blink is correctly or incorrectly rerouting packets. We therefore adopt the following evaluation methodology.

Methodology. We consider 15 publicly available traces [9, 12] (listed in §E), accounting for a total of 15.8 hrs of traffic and 1.5 TB of data. For each prefix, we extract the distributions of flow size, duration, average packet size, and RTT.⁷ We then run simulations with ns-3 [3] on a dumbbell topology similar to [38], where traffic sources generate flows exhibiting the same distribution of parameters than the one extracted from the real traces.

In some of our simulations, we introduce a failure after 10 seconds on the single link connecting the sources with the destinations, thus affecting all flows. We refer the reader to §D for an evaluation on partial failures. In other experiments, we

⁶A resubmitted packet goes twice in the ingress to take more actions while being forwarded by the switch.

⁷To measure the RTT of the flows, we use the time difference between the SYN and ACK packets sent by the initiator of a connection as described in [22, 34].

introduce random packet drops and no failure at all. We collect traffic traces for all simulations, feed them to our Python-based implementation of Blink one by one, and check if and when our system would fast reroute traffic.

Baselines. Since we are not aware of any previous work on real-time failures detection on the basis of TCP-generated signals, we compare Blink against two baseline strategies. The first strategy, *All flows*, consists in monitoring up to 10k flows for each prefix, and rerouting if any 32 of them sees retransmissions within the same time window. This strategy provides an upper bound on Blink’s ability to reroute upon actual failures while ignoring memory constraints. The second strategy, ∞ *Timeout*, is a variant of Blink where flows are only evicted when they terminate (with a FIN packet), and never because of the eviction timeout. This strategy assesses the effectiveness of Blink’s flow eviction policy.

6.1.1 Blink often detects actual failures, quickly

We first evaluate Blink’s ability to detect connectivity disruptions. For each real trace in our dataset, we randomly consider 30 prefixes which see a large number of flows (> 1000 flows in the trace), and we generate 5 synthetic traces per prefix, each with a different number of flows starting every second (from 100 to 500 flows generated *per second*) and each containing a failure at a preconfigured moment in time.

We then compute the True Positive Rate (TPR) of Blink on these traces. For each synthetic trace, we check whether Blink detects the failure (True Positive or TP) or not (False Negative or FN). The TPR is computed over all the tested synthetic traces, and is equal to $TP/(TP + FN)$.

Figure 6a shows the TPR of Blink and our baseline strategies as a function of the real trace used to generate the synthetic ones. As expected, the *All flows* strategy exhibits the best TPR among the three considered strategies at the cost of impractical memory usage. We see that Blink has a TPR which is very close to (*i.e.*, less than 10% lower than) the *All flows* strategy—while tracking only 64 flows. Overall, Blink correctly reroutes more than 80% of the times for 13 traces out of 15, with a minimum at 65% and a peak at 94%. At the other extreme, the TPR of the ∞ *Timeout* strategy is much lower than Blink, below 50% for most traces, highlighting the importance of Blink’s flow eviction policy.

As the RTT of the flows affects the failure signal used by Blink to detect failures (see §2.1), we also look at the TPR as a function of the RTT. On the synthetic traces with a median RTT below 50 ms (resp. above 300 ms), Blink has a TPR of 90.6% (resp. 76.0%). This shows that Blink is useful even when flows have a high RTT.

As a follow-up, we then analyzed how much Blink’s TPR varies with the number of flows active upon the failure (an important factor for Blink’s performance). Figure 6b shows that Blink’s TPR unsurprisingly increases if there are more flows active during the failure. With very few active flows, Blink cannot perform well, since the data-plane signal is too

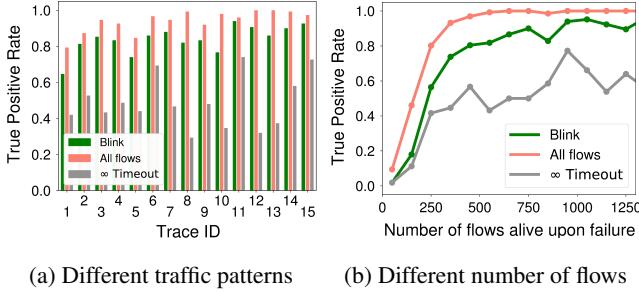


Figure 6: Blink has high TPR when relatively few flows (*e.g.*, more than 350) are active upon the failure.

weak. However, Blink’s TPR is already around 74% when about 350 flows are active, and reaches high values (more than 90%) with about 750 active flows. Again, Blink is much closer to the *All flows* strategy than to the ∞ *Timeout* one, although the *All flows* strategy reaches higher levels of TPR for lower number of active flows. These results suggest that Blink is likely to have a high TPR in a real deployment since we expect to see $\gg 750$ active flows for popular destinations.

Not only does Blink detect failures in most cases, but it also recovers connectivity quickly upon failure detection. Figure 7 shows the time needed for Blink to restore connectivity for each of the real traces used to generate the synthetic ones, restricting on the cases where Blink detects the failure. Each box shows the inter-quartile range of Blink’s reaction time. The line in each box depicts the median value; the whiskers show the 5th and 95th percentile. Blink retrieves connectivity in less than 778 ms for 50% of the traces, and within 1 s for 69% of the traces. The *All flows* strategy restores connectivity within 365 ms in the median case, whereas the ∞ *Timeout* strategy needs 1.07 s (median). Naturally, Blink is faster when the RTT of the flows is low. On the synthetic traces with a median RTT below 50 ms, Blink reroutes within 625 ms in the median case. Yet, when the median RTT is above 300 ms, Blink is still fast and reroutes within 1.2 s in the median case.

6.1.2 Blink distinguishes failures from noise

One may wonder if Blink’s ability to detect failures may not be due to it overestimating disruptions. By design, Blink cannot detect a failure without TCP retransmissions. Hence, the question is if Blink tends to overreact to relatively few, unrelated retransmissions, *e.g.*, induced by random packet loss.

To verify this, we generate synthetic traces with no failure but with an increasing level of random packet loss (from 1% to 9%) for all traffic. The trace synthesis follows our methodology of mimicking characteristics of real traffic for one prefix. For each real trace and loss percentage, we repeat the trace generation for 10 randomly extracted prefixes which see a large number of flows. For this experiment, we generate traces from 1-minute simulations where many (*i.e.*, 500) new flows start every second to ensure that Blink’s Flow Selector is filled with flows, all potentially sending retransmissions.

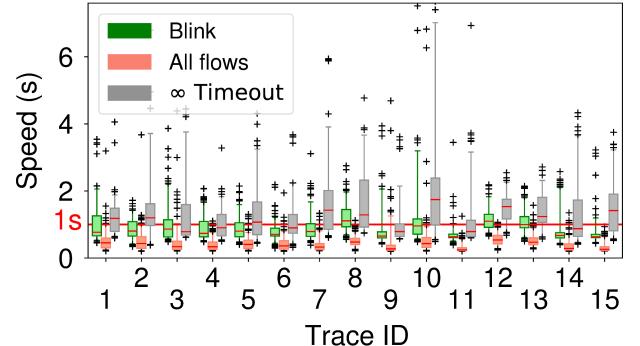


Figure 7: Blink is fast, for all traffic patterns.

For all these synthetic traces, we check whether Blink detects a failure (FP) or not (TN) and compute the False Positive Rate (FPR) as $FP/(FP + TN)$. Contrary to what happens for the TPR in §6.1.1, we expect *All flows* to be a worst case scenario as it sees all the retransmissions across all flows. On the other hand, ∞ *Timeout* should perform better than Blink because inactive flows (which are not evicted) do not contribute to the number of observed retransmissions.

Table 1 shows the FPR as a function of packet loss. Below 4% packet loss, Blink never detects failures. Between 4% and 7%, Blink incorrectly detected a failure for *one* synthetic trace out of the 150 generated. This indicates that Blink would work well under realistic traffic (we confirm this in §6.3.1), where the packet loss is often below these values. As a reference, the *All flows* strategy has an extremely high FPR, around 60% (resp. 85%) for traces with 1% (resp. 2%) packet loss. The ∞ *Timeout* strategy has only one false positive when the packet loss is between 5% and 10%, which, rather than a feature, is an artifact of tracking non-active flows.

Summary. Our results show that Blink strikes a good balance between detection of actual failures and robustness to noise (*i.e.*, TCP retransmissions not originated from a prefix-wide connectivity disruption). Blink’s tradeoff is much better than the naive strategies: not evicting flows would significantly lower Blink’s ability to correctly reroute upon failures, while monitoring all crossing flows comes with high sensitivity to noise (in addition to a likely impractical memory cost). Blink also recovers connectivity quickly (often within 1 s in our experiments) when it detects a disruption.

6.2 Blink’s rerouting algorithm

We now focus on the Blink’s Rerouting Module. Our design ensures that rerouting is done entirely in the data plane, at line rate —we confirm this by experimenting with a Tofino switch, as described in §6.3. In this section, we therefore evaluate whether Blink is effective in rerouting to a working next-hop.

Methodology. We emulate the network shown in Figure 8 in a virtual machine attached to 12 cores (2.2 GHz). The P4 switch has three possible next-hops to reach the destination, R1 being

packet loss %	1	2	3	4	5	6	7	8	9
	False Positive Rate (%)								
Blink	0	0	0	0.67	0.67	0.67	0.67	1.3	2.0
All flows	59	85	93	94	95	96	97	97	98
∞ timeout	0	0	0	0	0.67	0.67	0.67	0.67	0.67

Table 1: Blink avoids incorrectly inferring failures when packet loss is below 4%.

the primary next-hop, R2 the most preferred backup next-hop and R3 the less preferred backup one. R1 and R3 use R5 as next-hop to reach the destination. R2 uses a different next-hop to reach the destination depending on the experiment, thus we do not depict it in the figure.

We emulate the P4 switch by running our P4₁₆ implementation of Blink in the P4 behavioral model software switch [4]. The P4 switch running Blink is linked to a Mininet network emulating the other switches. The source and the destination are Mininet hosts running TCP cubic.

We start 1000 TCP flows from the source towards the destination. To show the effectiveness of the Flow Selector, 900 flows have a low packet rate (chosen uniformly at random between 0.2 and 1 packet/s) while only 100 have a high packet rate (chosen uniformly at random between 2.5 and 20 packet/s). We use `tc` to control the per-flow RTT (chosen uniformly at random between 10 ms and 300 ms), and to drop 1% of the packets on the link between R5 and the destination in order to add a moderate level of noise. We first start the 900 flows with a low packet rate, so that the Flow Selector first selects them. Right after, we start the 100 remaining flows. Finally, after 20 s to 30 s, we fail the link between R1 and R5.

Blink quickly detects and breaks loops. We configure R2 to use the P4 switch as next-hop to reach the destination so that it creates a forwarding loop (by sending traffic back to the source) when Blink reroutes traffic to R2. Figure 9a shows the traffic captured at R1, R2 and R3 (top) and at the destination (bottom). Prior the failure, the traffic goes through R1, the primary next-hop. Upon the failure, Blink probes if any of the available next-hops can recover connectivity: it sends half of the flows in the Flow Selector to R2 and the other half to R3. All the remaining flows go to R2 (preferred over R3). Blink detects the forwarding loop induced by R2 very quickly (only 8 packets were captured on R2) and immediately deactivates this next-hop to reroute all the traffic to R3, restoring connectivity within a total of 800 ms.

Blink quickly detects and routes around blackholes. In a separate experiment, we configure R2 to use R5 as next-hop, and we fail the link between R2 and R5 in addition to the one between R1 and R5. Figure 9b shows the traffic captured at R1, R2 and R3 (top) and at the destination (bottom). Upon the failure, Blink reroutes to R3 half of the selected flows, and to R2 the other half of the selected flows plus all the non-selected

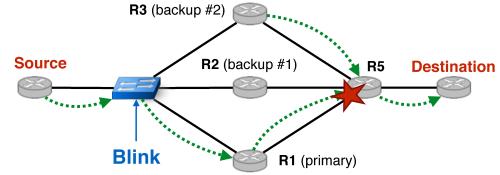


Figure 8: The network used to evaluate Blink’s rerouting. Arrows indicate forwarding next-hops (R2 uses different next-hops depending on the experiment).

ones (since R2 is preferred over R3). However, because the link between R2 and R5 is down, the packets sent to R2 are just dropped by R2. After 1 s, Blink detects the blackhole and reroutes all the traffic to R3, restoring connectivity. The total downtime induced by the failure is 1.7 s.

6.3 Blink in the real world

So far, we have evaluated Blink with simulations and emulations. We now report on experiments that we run on real traffic traces and on a Barefoot Tofino switch.

6.3.1 Running Blink on real traces

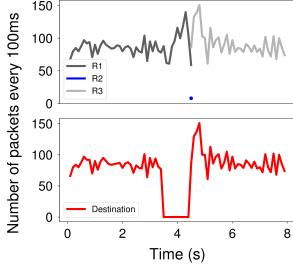
In §6.1, we use simulated (but realistic) traffic traces to gain some confidence on Blink’s accuracy in detecting connectivity disruptions. An objection might be that our synthetic traces are not fully realistic. We therefore run Blink on the original real traces listed in §E and tracked when it detects a failure⁸. Observe that unlike in §6.1, here we do not simulate failures. Since we do not have ground truth about actual failures in real traces, we manually checked each case for which Blink detected a failure so as to confirm the connectivity disruption.

Over the 15.8 hrs of real traces, Blink detected 6 failures. In these 6 cases, the retransmitting flows represent 42%, 57%, 71%, 82%, 82% and 85% of all the flows active at that time and destined to the affected prefix. These numbers confirms that Blink is *not* sensitive to normal congestion events, and only reroutes in cases where a large fraction of flows experience retransmissions at the same time.

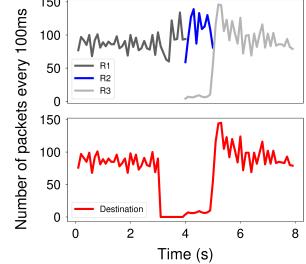
6.3.2 Deploying Blink on Barefoot Tofino switches

We finally evaluate our P4-Tofino implementation of Blink on a Barefoot Tofino Wedge 100BF-32X. To do so, we generate TCP traffic between two servers connected via our Tofino switch. The server receiving the traffic has a primary and a backup physical link with the Tofino switch. We generate 1000 flows, 900 of which have a low packet rate and 100 a high one (similarly to §6.2). To show the influence of the RTTs on Blink when running on Tofino, we run two experiments, one with sub-1ms RTT, and another one in which we use `tc` to simulate for each flow an RTT chosen uniformly at random between 10 ms and 300 ms. After 30 s, we simulate

⁸We omitted failures detected for 73 prefixes (out of 2.28M) which constantly showed high-level of retransmissions (>20% of the flows retransmitting >50% of the time). Blink could detect such outliers at runtime.



(a) Blink quickly breaks loops



(b) Blink reacts to blackholes

Figure 9: Traffic measurements quantifying Blink’s speed in reacting to forwarding anomalies upon rerouting.

a failure on the primary path, and measure the time Blink takes to retrieve connectivity via the backup link.

Blink-Tofino managed to restore connectivity in <1s. Blink retrieves connectivity in only 460ms with sub-1ms RTT, and in about 1.1 s when the RTT of the flows is between 10 ms and 300 ms. We obtain comparable results (470 ms of downtime with sub-1 ms RTT) when running the same experiments with 3,000 flows among which 300 has a high packet rate.

7 Deployment considerations

We now discuss three possible operational concerns when deploying Blink in a real ISP network: security, adaptability and interaction with deployments of Blink in other ISPs.

Security. As potentially any Internet user can generate data-plane traffic, security could be a concern for running Blink in operational networks. The main threat is that malicious users could manipulate Blink to reroute traffic by sending fake retransmissions for flows towards a victim destination.

Blink’s design itself significantly mitigates security risks. For any given destination, Blink reroutes traffic if most of the 64 monitored flows retransmit nearly at the same time. The monitored flows are selected among all active flows for the given destination, and flows sending more packets are intuitively privileged by the Flow Selector substitution policy. Hence, a brute-force attack would have to generate an amount of traffic comparable to the legitimate traffic destined to the attacked prefix in order to have a reasonable success chance. The fact that Blink focuses on popular destinations, typically attracting large traffic volumes and many flows, implies that the attacker would have to generate lots of traffic to trick Blink—a condition under which the attack would be quite visible, could be monitored and potentially mitigated at runtime.

A Blink-savvy attacker could instead produce few flows with a high and constant packet rate, and which never terminate, so that when one of them is selected by the Flow Selector, it remains selected forever. Blink will eventually monitor 32 of them, making the attacker ready to operate. However, Blink is built to evict a flow, *even if active*, after a fixed time (8.5 min by default, see §C). This implies that the attacker only has

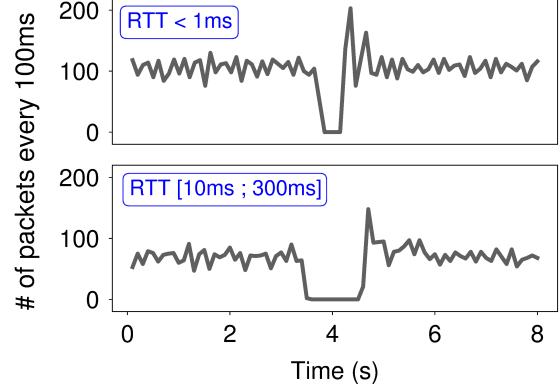


Figure 10: A Tofino switch running Blink retrieves connectivity within 460 ms if the RTT of the affected flows is below 1 ms (top figure); or within 1.1 s if the per-flow RTT ranges between 10 ms and 300 ms (bottom figure).

a short time window during which she can perform the attack. Evicting active flows more frequently (*e.g.*, every few seconds instead of 8.5 min) would better prevent such attacks, but would affect the failure detection mechanism of Blink because some retransmissions can be missed.

While the above mechanisms do not make Blink bullet-proof, we believe that they make our system’s attack surface reasonably small. We plan to perform a deeper analysis of Blink’s security concerns in future work.

Adaptability. Clearly, a challenge in a real deployment of Blink is how to set its parameters correctly (see §A). This is hard because operators have different requirements, and traffic can exhibit varying characteristics. In §6, we show that it is definitely possible to set the parameter values so that Blink works well in real situations. Yet, in a real deployment, we envision that Blink could first be run in “learning mode”, where it sends notifications to the controller instead of rerouting traffic. The controller then evaluates the accuracy of the system, for instance using control-plane data, and turns Blink on if the accuracy is good, or tune some parameters otherwise.

Internet scale deployment. So far, we have described Blink’s deployment in a single network (see §3). Of course, all ISPs have the same incentives to deploy Blink (*i.e.*, for fast connectivity recovery), so we envision that multiple, possibly all, ISPs might deploy Blink. Multi-AS deployment of Blink makes rerouting trickier. For example, if an Internet path traverses multiple Blink switches, it is not clear which ones will reroute, and whether the resulting backup path will be optimal. Blink switches can also interfere with each other. For example, if a Blink switch reroutes traffic to a backup path, a downstream Blink switch in the original path may lose part of the data-plane signal, preventing it to detect the failure. Finally, Blink’s rerouting can also increase the likelihood of creating inter-domain loops, since Blink selects backup next-

hops based on BGP information, which might not be truthful if the downstream switches also run Blink.

While a full characterization of Blink’s behavior in an Internet-scale deployment is outside the scope of this paper, Blink’s design already guarantees some basic correctness properties. Blink already monitors for possible forwarding loops, and quickly breaks them by using additional backup next-hops (see §4.4.2). After having explored all possible backup next-hops, Blink also falls back to the primary next-hop indicated by the control-plane, even if not working: this would prevent oscillations where two or more Blink switches keep changing their respective next-hops in the attempt to restore connectivity. Finally, Blink switches do use BGP next-hops after BGP convergence.

Path optimality is much harder to guarantee within a system like Blink, where network nodes independently reroute traffic, without any coordination. However, we believe that path optimality can be transiently sacrificed⁹ in the interest of restoring Internet connectivity as quickly as possible.

8 Related Work

We now discuss related work beyond fast reroute frameworks for local failures (such as [14, 29, 35]), which we already discussed in the introductory section.

Recent research explores how to quickly localize remote inter-domain failures and reroute upon them, assuming no data-plane programmability. Prominently, ARROW [30] uses tunnels between endpoints and ISPs in combination with special control-plane packets. Gummadi et al. [17] infer failures from the data plane, and attempt to recover connectivity by routing indirectly through a small set of intermediaries. Several other approaches (*e.g.*, [13, 19, 21]) infer connectivity problems from BGP messages. Some of them also diagnose these problems and fast-reroute upon their detection. By inferring failures from data-plane packets, Blink is fundamentally faster than control-plane based solutions (by minutes, *e.g.*, in Figure 1). Also, unlike [17, 30], Blink is deployable on a single node, and does not require interaction with other devices.

Data-Driven Connectivity [25] (DDC) ensures connectivity *within a network* via data-plane mechanisms. Chiesa et al. [11] consider generalizations of the DDC approach, and study the relationship between the resilience achieved through data-plane primitives and network connectivity. The work from Sedar et al. [33] shows how to program a hardware switch so that it automatically reroutes the traffic to a working path upon failure of directly connected links. In contrast to the above line of work, Blink fast recovers upon failures occurring in other networks, and uses data-plane programmability to detect connectivity disruptions rather than to dynamically configure post-failure paths.

Data-plane traffic has also been widely used in the past for

⁹Even with an Internet-wide deployment of Blink, path optimality will be restored when the control plane converges to post-failure paths, or operators will manually solve connectivity disruptions after Blink’s notifications.

ex-post measurement analyses. For example, WIND [20] infers network performance problems, including outages, from traffic traces by leveraging (among others) structural characteristics of TCP flows. In Blink, we perform online packet analysis, at line rate, but only to infer major connectivity disruptions – with simple yet effective algorithms that fit the limited resources of real switches.

Few approaches monitor traffic using a programmable data plane. DAPPER is an in-network solution using TCP-based signals to identify the cause of misbehaving flows (whether the problem is in the network or not) [16]. Blink does not aim at identifying the cause of a particular flow failing but rather that many flows (for the same prefix) fail at the same time. In addition, unlike Blink, DAPPER requires symmetric routing for its analysis, which is often not the case in ISP environments. Sivaraman et al. [36] propose a heavy-hitter detection mechanism running entirely in the data plane. As Blink, it stores flows in an array and relies on flow eviction to keep track of the heaviest ones. Unlike [36], Blink looks for active flows instead of the heaviest ones, on a per-prefix basis.

Recent work from Molero et al. [26] shows that key control-plane tasks such as failure detection, notifications and new path computations can be offloaded to the data plane. Such systems could directly benefit from Blink, *e.g.*, by leveraging it to detect remote failures and trigger network-wide convergence accordingly.

9 Conclusions

Blink is the first data-driven fast-reroute framework targeting remote connectivity failures. By operating entirely in the data plane, at line rate, Blink restores connectivity in $O(s)$. We evaluate Blink with realistic traces, taking into account different traffic conditions as well as noise due to significant packet loss. Our results show that Blink enables sub-second connectivity retrieval in the majority of the scenarios, while preventing unnecessary traffic shifts in the presence of noise. By deploying Blink on a Barefoot Tofino switch, we also confirm that it can run in commercial programmable devices.

10 Acknowledgments

We are grateful to the NSDI anonymous reviewers and our shepherd, Harsha Madhyastha, for their insightful comments. We also thank the members of the Networked Systems Group at ETH Zurich for their valuable feedback. This work was supported by a Swiss National Science Foundation Grant (Data-Driven Internet Routing, #200021-175525). This research was also supported by the U.S. Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) via contract number 70RSAT18CB0000015.

References

- [1] CNN - Time Warner Cable comes back from nationwide Internet outage, 2014. <https://money.cnn.com/2014/08/27/media/time-warner-cable-outage/index.html>.
- [2] RIPE RIS Raw Data, 2016. <https://www.ripe.net/data-tools/stats/ris/>.
- [3] Network Simulator 3., 2018. <https://www.nsnam.org/>.
- [4] P4 behavioral model., 2018. <https://github.com/p4lang/behavioral-model>.
- [5] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in tcp round-trip times. In *ACM IMC'03*, New York, NY, USA.
- [6] Alia K. Atlas and Gagan Choudhury and David Ward. IP Fast Reroute Overview and Things we are struggling to solve. https://bgp.nu/~dward/IPFRR/IPFRR_overview_NANOG.pdf.
- [7] A. Atlas and A. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 5286, September 2008.
- [8] Barefoot. Barefoot Tofino, World's fastest P4-programmable Ethernet switch ASICs. <https://barefoottofino.com/products/brief-tofino/>.
- [9] CAIDA. The CAIDA UCSD Anonymized 2013/2014/2015/2016/2018 Internet Traces. http://www.caida.org/data/passive/passive_2013_dataset.xml.
- [10] CAIDA. The UCSD Network Telescope. https://www.caida.org/projects/network_telescope/.
- [11] Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrovic, Andrei Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. On the re-siliency of static forwarding tables. *IEEE/ACM Transactions on Networking*, 2017.
- [12] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the wide project. In *USENIX ATEC'00*.
- [13] Anja Feldmann, Olaf Maennel, Z Morley Mao, Arthur Berger, and Bruce Maggs. Locating Internet routing instabilities. *ACM SIGCOMM CCR*, 2004.
- [14] Clarence Filsfils. BGP Convergence in much less than a second, 2007. Presentation NANOG 23.
- [15] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. *ACM SIGCOMM CCR*, 2005.
- [16] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *ACM SOSR'17*, 2017.
- [17] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. Improving the reliability of internet paths with one-hop source routing. In *USENIX OSDI'04*.
- [18] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. Measuring latency variation in the internet. In *ACM CoNEXT '16*.
- [19] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. SWIFT: Predictive Fast Reroute. In *ACM SIGCOMM'17*.
- [20] Polly Huang, Anja Feldmann, and Walter Willinger. A non-intrusive, wavelet-based approach to detecting network performance problems. In *ACM SIGCOMM Workshop on Internet Measurement*, 2001.
- [21] Umar Javed, Italo Cunha, David Choffnes, Ethan Katz-Bassett, Thomas Anderson, and Arvind Krishnamurthy. PoiRoot: Investigating the Root Cause of Interdomain Path Changes. In *ACM SIGCOMM*, 2013.
- [22] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *ACM SIGCOMM CCR*, 2002.
- [23] D. Katz and D. Ward. Bidirectional Forwarding Detection. RFC 5880, 2010.
- [24] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. *ACM SIGCOMM CCR*, 2000.
- [25] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *USENIX NSDI'13*.
- [26] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Hardware-accelerated network control planes. In *ACM HotNets '18*.
- [27] University of Oregon. Route Views Project, 2016. www.routeviews.org.
- [28] Ricardo Oliveira, Beichuan Zhang, Dan Pei, Rafit Izak-Ratzin, and Lixia Zhang. Quantifying path exploration in the internet. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 269–282. ACM, 2006.
- [29] P. Pan, G. Swallow, and A. Atlas. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090, May 2005.
- [30] Simon Peter, Umar Javed, Qiao Zhang, Doug Woos, Thomas Anderson, and Arvind Krishnamurthy. One tunnel is (often) enough. *ACM SIGCOMM CCR*, 2014.
- [31] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.
- [32] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM CCR*, 2012.
- [33] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. Supporting emerging applications with low-latency failover in p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, NEAT '18. ACM, 2018.
- [34] S. Shakkottai, N. Brownlee, A. Broido, and k. claffy. The RTT distribution of TCP flows on the Internet and its impact on TCP based flow control. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), Mar 2004.
- [35] M. Shand and S. Bryant. IP Fast Reroute Framework. RFC 5714, January 2010.
- [36] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR'17*.
- [37] The P4 Language Consortium. P4 16 Language Specification. <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>.
- [38] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. *ACM SIGCOMM CCR*, 2006.

Appendix

A Blink parameters

In this section, we list in Table 2 the main parameters used by Blink, and show how each of them can affect the performance of the system. We denote as TPR the True Positive Rate over all the synthetic traces generated from the 15 real traces listed in Table 3 and used in §6.1.1. The TPR with Blink’s default values over all the synthetic traces is 83.9%. FPR denotes the False Positive Rate and is computed similarly as in §6.1.2.

B Implementation of a sliding window in P4₁₆

In this appendix section, we show how we implemented a sliding window in the P4₁₆ language.

Blink uses one sliding window per prefix to count the number of flows experiencing retransmissions over time among the selected flows (§4.3). Besides the 10 bins, Blink needs three other meta-information: (i) `current_index`, the index of the bin focusing on the current period of time, (ii) `sum`, the sum of all the 10 bins and (iii) `last_ts_sliding`, the timestamp in millisecond precision of the last time the window slid. When Blink detects a retransmission, it increments by one both the value associated with the bin at the index `current_index` and the `sum`. Upon reception of a packet at timestamp t , and assuming the window covers a period of k millisecond, if $t - \text{last_ts_sliding} > k/10$, the window slides by one doing the following operations. To find the index of the bin that has expired, Blink computes $(\text{current_index} + 1) \bmod 10^{10}$. Then, it subtracts to sum the value stored in the expired bin, and then resets it. Then, Blink makes `current_index` point to the expired bin and finally updates `last_ts_sliding` to `last_ts_sliding + k`. As a result, the counter `sum` always returns the number of flows experiencing retransmissions during the last $9/10k$ to k seconds.

It can happen that a flow sends several retransmissions within a time window. To avoid summing several retransmissions from same flow within the same time window, Blink uses two additional per-flow metadata called `last_ret_ts` and `last_ret_bin`. The former stores the timestamp of the last retransmission reported for the corresponding flow. The latter stores the bin index corresponding to this timestamp. Consider that a retransmission for a flow is reported at time t , then if $t - \text{last_ret_ts} < k$, Blink decrements by one the value in the bin at the index `last_ret_bin`, and increments by one the value associated to the current bin. The sum remains the same, and `last_ret_ts` is set to the current timestamp and `last_ret_bin` is set to the current bin index.

C Hardware Resource Usage

Blink is intended to run on programmable switches with limited resources. As a result, we designed Blink to scale based

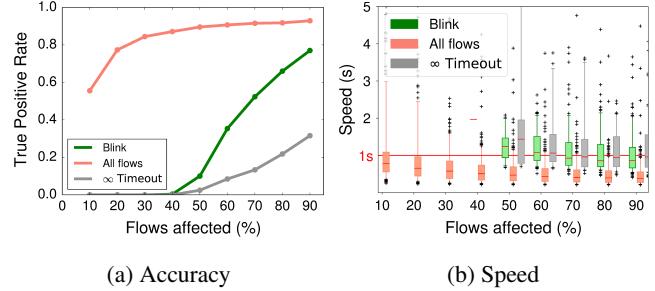


Figure 11: Blink is accurate and fast even for partial failures affecting 70% or more of the flows.

on the number prefixes it monitors, and not on the actual amount of traffic destined to those prefixes. In this section, we derive the resources required by Blink to work for one prefix, and show that it can easily scale to thousands of prefixes.

First, for every prefix, Blink needs one entry in the meta-data table. Then, for each selected flows, the Flow Selector needs 99 bits (see Figure 5). As Blink monitors 64 flows per prefix, a total of $64 * 99 = 6336$ bits are required for one prefix. To save memory, Blink does not store the timestamps (*e.g.*, `last_pkt_ts` and `last_ret_ts`) in 48 bits (the original size of the metadata), but instead approximates them using only 9 bits for seconds and 19 bits for milliseconds. To obtain the second (resp. millisecond) approximation of the current timestamp, Blink shifts the original 48-bit timestamp to the right. To fill in 9 (resp. 19) bits, Blink also resets the timestamps every 512 s (≈ 8.5 min) by subtracting to the original 48-bit timestamp a `reference_timestamp`. The `reference_timestamp` is simply a copy of the original 48-bit timestamp (stored in a register shared by all the prefixes) that is updated only every 512 s. Note that the Flow Selector evicts a flow if the current timestamp is lower than the timestamp of the last packet seen for that flow, which happens whenever the timestamps are reset (*i.e.*, every ≈ 8.5 min). This is actually good for security, as we explain in §7.

The sliding window requires 10 bins of 6 bits each, as well as $4 + 9 + 6 = 19$ bits to store additional information (see Figure 5), making a total of 79 bits. For the rerouting, Blink only requires 1 status bit per next-hop. With three next-hops, 3 extra bits are required. In total, for one prefix, Blink requires $6336 + 79 + 3 = 6418$ bits. As current programmable switches have few megabytes of memory, we expect Blink to support up to 10k prefixes, possibly even more.

D Evaluating Blink on partial failures

In this section, we evaluate the performance of Blink upon partial failures, and compare it to our two baselines, the *All flows* and ∞ *Timeout* strategies. We consider partial failures to be those affecting only a portion of the traffic (*e.g.*, due to load-balancing).

For this evaluation, we randomly picked 10 prefixes from

¹⁰As the modulo operator is not available in P4₁₆, we implement this with if-else conditions.

Component	Name	Default value	Tradeoff
Flow Selector §4.2	Eviction timeout	2s	With a short eviction time (<i>e.g.</i> , 0.5s) flows can be evicted while they are retransmitting, reducing the TPR to 66.3%. With a longer eviction time (<i>e.g.</i> , 3s) inactive flows take more time to be substituted by active ones, reducing the TPR to 77.7%
	Number of cells per prefix	64	Monitoring a small fraction of flows may result in a FPR increase. For example, with only 16 cells, the FPR is 2% for only a 3% of packet loss. However, the bigger the number of cells the smaller the amount of prefixes we can monitor due to memory constraints. With 64 cells (=64 flows monitored per prefix) Blink can support at least 10k prefixes.
Failure Inference §4.3	Sliding window duration	800ms	A long time window (<i>e.g.</i> , 1.6s) is more likely to report all the retransmitting flows, increasing the TPR to 89.8%, but also reports more unrelated retransmissions, increasing the FPR (0.67% for 3% of packet loss). A shorter time window (<i>e.g.</i> , 400ms) limits the FPR (0% for 9% of packet loss) but decreases the TPR to 49.4%.
	Sliding window number of bins	10	More bins increases the precision, at the price of using slightly more of memory. 10 bins give a precision > 90%.
	Inference threshold	50%	A lower threshold, such as 25% (<i>i.e.</i> , 16 flows retransmitting when using 64 cells) gives a better TPR (94.8%), but increases the FPR to 7.3% for 4% of packet loss.
Rerouting Module §4.4	Backup next-hop probing time	1s	A longer probing period better prevents wrongly assessing a next-hop as not working, at the price of waiting more time to reroute.

Table 2: Parameters used by Blink, with their default values, and how they can affect the performance of the system.

each real trace listed in Table 3, and generated 1 synthetic trace for each of them, following the guidelines described in §6.1. For each trace, we simulated partial failures with 9 different intensities (from 10% to 90% of the flows being affected). For these synthetic traces, 1223 flows (resp. 264) were active upon the failures in the median case (resp. 10th percentile).

Blink works in the majority of the cases for failures affecting 70% or more of the flows. Figure 11a shows the TPR of Blink as a function of the percentage of flows affected by the failure. Unsurprisingly, because Blink needs to detect at least 32 flows experiencing retransmissions to detect the failure, the TPR is close to 0% if the failure affects less than 50% of the flows. For failures affecting 70% of the flows (resp. 90%), Blink works for 53% (resp. 77%) of the failures. The *All flows* strategy performs well even for small failures affecting 20% of traffic, whereas the ∞ *Timeout* performs badly even for

failures affecting 90% of the traffic.

Blink restores connectivity within one second in the median case for failures affecting at least 70% of the flows. Figure 11b shows the time needed for Blink to restore connectivity upon a partial failure. Logically, as we decrease the amount of affected flows, the detection speed of Blink increases. Yet, Blink is able to restore connectivity within 1 s for the majority of the cases if a failure affects 70% of the flows or more.

E Real traces used in the evaluation

In order to evaluate Blink with different traffic patterns (see §6), we use 15 real traces from different years, captured on different links and provided by different organizations, namely CAIDA [9] and MAWI [12]. Table 3 lists them and some of their characteristics.

Trace ID	Name	Date	Duration	Bit Rate	Trace Size	RTT median
1	caida-equinix-chicago.dirA	29-05-2013	3719 s	1631 Mbps	67 GB	200.22 ms
2	caida-equinix-chicago.dirB	29-05-2013	3719 s	2119 Mbps	73 GB	155.65 ms
3	caida-equinix-sanjose.dirA	21-03-2013	3719 s	2920 Mbps	122 GB	104.94 ms
4	caida-equinix-sanjose.dirB	21-03-2013	3719 s	1618 Mbps	82 GB	191.11 ms
5	caida-equinix-chicago.dirA	19-06-2014	3719 s	1629 Mbps	60 GB	209.72 ms
6	caida-equinix-chicago.dirB	19-06-2014	3719 s	6271 Mbps	163 GB	160.91 ms
7	caida-equinix-sanjose.dirA	19-06-2014	3719 s	3722 Mbps	144 GB	169.71 ms
8	caida-equinix-chicago.dirA	17-12-2015	3776 s	2540 Mbps	111 GB	240.18 ms
9	caida-equinix-chicago.dirB	17-12-2015	3776 s	3151 Mbps	99 GB	68.30 ms
10	caida-equinix-chicago.dirA	21-01-2016	3819 s	2250 Mbps	126 GB	224.09 ms
11	caida-equinix-chicago.dirB	21-01-2016	3819 s	4959 Mbps	143 GB	69.57 ms
12	caida-equinix-nyc.dirA	15-03-2018	3719 s	3027 Mbps	94 GB	306.76 ms
13	caida-equinix-nyc.dirA	19-04-2018	3719 s	3893 Mbps	125 GB	283.82 ms
14	mawi-samplepoint-F	12-04-2017	7199 s	878Mbps	74 GB	124.14 ms
15	mawi-samplepoint-F	07-05-2018	900 s	1098 Mbps	10 GB	156.20 ms
Total			15.8 h		1.5 TB	

Table 3: List of 15 real traces that we use to evaluate Blink. Altogether, they cover a total of 15.8 hrs of traffic.

Hydra: a federated resource manager for data-center scale analytics

Carlo Curino Subru Krishnan Konstantinos Karanasos Sriram Rao*
Giovanni M. Fumarola Botong Huang Kishore Chaliparambil Arun Suresh
Young Chen Solom Heddaya Roni Burd Sarvesh Sakalanaga Chris Douglas
Bill Ramsey Raghu Ramakrishnan

Microsoft

Abstract

Microsoft’s internal data lake processes exabytes of data over millions of cores daily on behalf of thousands of tenants. Scheduling this workload requires 10× to 100× more decisions per second than existing, general-purpose resource management frameworks are known to handle. In 2013, we were faced with a growing demand for workload diversity and richer sharing policies that our legacy system could not meet. In this paper, we present *Hydra*, the resource management infrastructure we built to meet these requirements.

Hydra leverages a federated architecture, in which a cluster is comprised of multiple, loosely coordinating sub-clusters. This allows us to scale by delegating placement of tasks on machines to each sub-cluster, while centrally coordinating only to ensure that tenants receive the right share of resources. To adapt to changing workload and cluster conditions promptly, *Hydra*’s design features a control plane that can push scheduling policies across tens of thousands of nodes within seconds. This feature combined with the federated design allows for great agility in developing, evaluating, and rolling out new system behaviors.

We built *Hydra* by leveraging, extending, and contributing our code to Apache Hadoop YARN. *Hydra* is currently the primary big-data resource manager at Microsoft. Over the last few years, *Hydra* has scheduled nearly one trillion tasks that manipulated close to a Zettabyte of production data.

1 Introduction

As organizations amass and analyze unprecedented amounts of data, dedicated data silos are being abandoned in favor of more cost-effective, shared data environments, such as private or public clouds. Sharing a unified infrastructure across all analytics frameworks and across tenants avoids the resource fragmentation associated with operating multiple smaller clusters [37] and lowers data access barriers. This

is the vision of the *data lake*: *empower every data scientist to leverage all available hardware resources to process any dataset using any framework seamlessly* [26]. To realize this vision, cloud vendors and large enterprises are building and operating data-center scale clusters [7, 15, 37].

At Microsoft, we operate one of the biggest data lakes, whose underlying compute capacity comprises hundreds of thousands of machines [7, 26]. Until recently, our clusters were dedicated to a single application framework, namely Scope [44], and were managed by our custom distributed scheduler, Apollo [7]. This architecture scaled to clusters¹ of more than 50k nodes, supported many thousands of scheduling decisions per second, and achieved state-of-the-art resource utilization. New requirements to share the same physical infrastructure across diverse application frameworks (both internal and popular open-source ones) clashed with the core assumption of our legacy architecture that all jobs had homogeneous scheduling patterns. Further, teams wanted more control over how idle capacity was shared, and system operators needed more flexibility while maintaining the fleet. This motivated us to build *Hydra*, a resource management framework that today powers the Microsoft-wide data lake. *Hydra* is the scheduling counterpart of the storage layer presented in [26].

Hydra matches the scalability and utilization of our legacy system, while supporting diverse workloads, stricter sharing policies, and testing of scheduling policies at scale (§2). This is achieved by means of a new *federated* architecture, in which a collection of loosely coupled sub-clusters coordinates to provide the illusion of a single massive cluster (§3). This design allows us to scale the two underlying problems of *placement* and *share-determination* separately. *Placement* of tasks on physical nodes can be scaled by running it independently at each sub-cluster, with only local visibility. On the other hand, *share-determination* (i.e., choosing how many resources each tenant should get) requires global vis-

*The work was done while the author was at Microsoft; currently employed by Facebook.

¹By *cluster* we refer to a logical collection of servers that is used for quota management and security purposes. A cluster can span data centers, but each job has to fit into a cluster’s boundaries.

ibility to respect sharing policies without pinning tenants to sub-clusters. We scale share-determination by operating on an aggregate view of the cluster state.

At the heart of *Hydra* lie *scheduling policies* that determine the behavior of the system’s core components. Given our diverse workloads and rapidly changing cluster conditions, we designed *Hydra*’s control plane to allow us to dynamically “push” policies. Cluster operators and automated systems can change *Hydra*’s scheduling behaviors of a 50k node cluster within seconds, without redeploying our platform. This agility allowed us to experiment with policies and to cope with outages swiftly. We discuss several policies, and show experimentally some of their trade-offs, in §4.

This federated architecture, combined with flexible policies, also means that we can tune each sub-cluster differently, e.g., to optimize interactive query latencies, scale to many nodes, operate on virtualized resources, or A/B test new scheduling behaviors. *Hydra* makes this transparent to users and applications, which perceive the resources as a continuum, and allows operators to mix or segregate tenants and behaviors in a dynamic, lightweight fashion. The architecture also enables several additional scenarios by allowing individual jobs to span sub-clusters: owned by different organizations, equipped with specialized hardware (e.g., GPUs or FPGAs), or located in separate data centers or regions [8]. In addition to the flexibility offered to users who submit jobs, these capabilities are invaluable for operators of the data lake, enabling them to manage complex workloads during system upgrades, capacity changes, or outages.

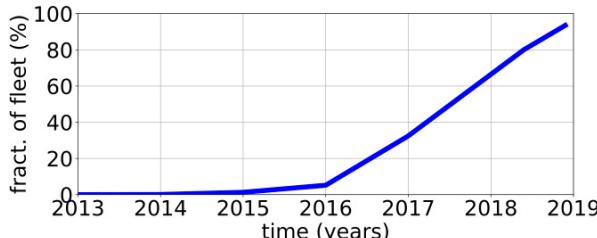


Figure 1: *Hydra* deployment in our production fleet (hundreds of thousands of nodes) over time.

An additional contribution of this paper is an open-source implementation of our production-hardened system (§5), as well as a summary of lessons learned during a large-scale migration from our legacy system. The migration consisted of a carefully choreographed in-place migration process of a massive production environment (§2), while the entirety of Microsoft depended on it. This journey was not without challenges, as we describe in §6. Fig. 1 shows the deployment of *Hydra* across our fleet over time. Since we started deploying it, *Hydra* has scheduled and managed nearly *one trillion tasks that processed close to a Zettabyte of data*. We report on our production deployments in §7, explicitly comparing its performance with our legacy system [7].

Apart from the new material presented in this paper, *Hydra* draws from several existing research efforts [7, 9, 11, 17, 19, 18, 27, 36]. In §8, we put *Hydra* in context with its related work, mostly focusing on production-ready resource managers [7, 15, 20, 36, 37].

2 Background and Requirements

At Microsoft we operate a massive data infrastructure, powering both our public cloud and our internal offerings. Next, we discuss the peculiarities of our internal clusters and workload environments (§2.1), as well as how they affect our requirements for resource management (§2.2) and our design choices in building *Hydra* (§2.3).

2.1 Background on our Environment

Cluster environment. Tab. 1 summarizes various dimensions of our big-data fleet.

Dimension	Description	Size
Daily Data I/O	Total bytes processed daily	>1EB
Fleet Size	Number of servers in the fleet	>250k
Cluster Size	Number of servers per cluster	>50k
# Deployments	Platform deployments monthly	1-10

Table 1: Microsoft cluster environments.

Our target cluster environments are very large in scale and heterogeneous, including several generations of machines and specialized hardware (e.g., GPU/FPGA). Our system must also be compatible with multiple hardware management and deployment platforms [16, 5]. Thus, we make minimal assumptions on the underlying infrastructure and develop a control-plane to push configurations and policies.

We observe up to 5% machine unavailability in our clusters due to various events, such as hardware failures, OS upgrades, and security patches. Our resource management substrate should remain highly available despite high hardware/software churn.

Sharing across tenants. As shown in Tab. 2, our clusters are shared across thousands of users. Users have access to hierarchical queues, which are logical constructs to define storage and compute quotas. The queue hierarchy loosely follows organizational and project boundaries.

Dimension	Description	Size
# Users	Number of users	>10k
# Queues	Number of (hierarchical) queues	>5k
Hierarchy depth	Levels in the queue hierarchy	5-12
Priority levels	Number of priority levels (avg/max)	10/1000

Table 2: Tenant details in Microsoft clusters.

In our setting, tenants pay for guaranteed compute capacity (quota) as a means to achieve predictable execution [17]. Tenants typically provision their production quotas for their

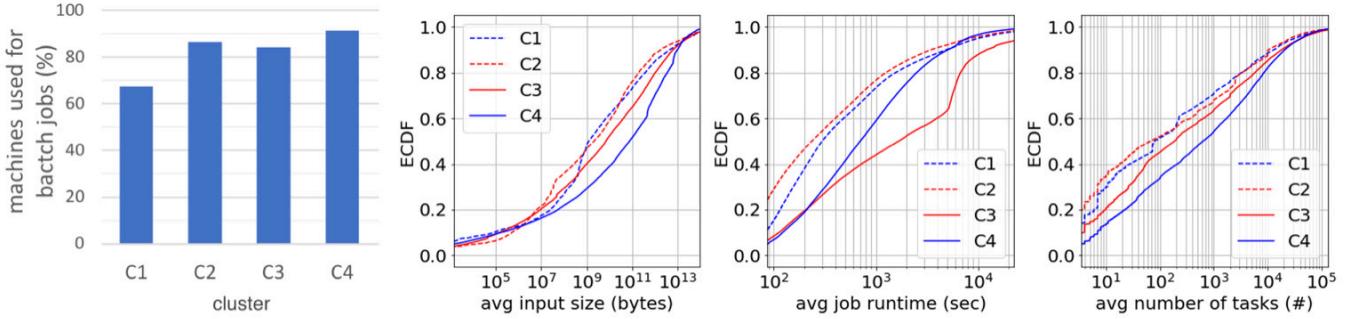


Figure 2: Number of machines dedicated to batch jobs in clusters C1–C4 (leftmost figure). Empirical CDF (ECDF) of various job metrics—each point is the average over a month for recurring runs of a periodic job, grouped by cluster (remaining figures).

peak demand, which would result in significantly underutilized resources. To increase cluster utilization, it is desirable to allow tenants to borrow unused capacity. Our customers demand for this to be done “fairly”, proportionally to a tenant’s guaranteed quota [12].

Workload. The bulk of our workload today is batch analytical computations, with streaming and interactive applications growing at a fast pace. The leftmost figure in Fig. 2 shows that 65–90% of machines in four of our clusters are dedicated to batch jobs. Each of these clusters has more than 50K machines; this is not the entirety of our fleet, but it is representative. Note that in our legacy infrastructure, the machines used for non-batch jobs had to be statically determined, which led to resource under-utilization, hot-spots, and long container placement latencies. Tab. 3 reports the key dimensions of our workloads. The overall scale of individual large jobs and the number of jobs that run on common datasets drove us to build large shared clusters. Beside large cluster sizes, the scheduling rate is the most challenging dimension of our scalability.

Dimension	Description	Size
# Frameworks	Number of application frameworks	>5
# Jobs	Number of daily jobs	>500k
# Tasks	Number of daily tasks	Billions
Scheduling rate	Scheduling decisions per second	>40k
Data processed	Bytes processed by individual jobs	KBs-PBs

Table 3: Microsoft workload characteristics.

We quantify more metrics of our batch workload in Fig. 2. The empirical CDFs in the figures capture properties of the four of the aforementioned large clusters. Each point in the CDFs represents a *recurring analytical job*, and its average behavior over one month. We group jobs by cluster and plot one line for each cluster. Jobs have very diverse behaviors: from KBs to PBs of input sizes, from seconds to days of runtime, from one to millions of tasks.

Legacy system. Prior to *Hydra*, our cluster resources were managed by our legacy system, Apollo [7]. Apollo’s distributed scheduling architecture allowed us to scale to our target cluster sizes and scheduling rates, while achieving

good resource utilization. However, it only supported a single application framework and offered limited control over sharing policies, which are among our core requirements, as described below. An overview of Apollo is provided in §A.2.

2.2 Requirements

We summarize our requirements as follows:

- R1 **Workload size and diversity:** Our workloads range from very small and fast jobs to very large ones (e.g., millions of tasks spanning tens of thousands of servers), and from batch to streaming and interactive jobs. They include both open-source and Microsoft’s proprietary frameworks. Many jobs access popular datasets. The resource management framework must support this wide workload spectrum and large-scale data sharing.
- R2 **Utilization:** High utilization is paramount to achieve good Return On Investment (ROI) for our hardware.
- R3 **Seamless migration:** backward compatibility with our existing applications and transparent, in place replacement—to preserve existing investments in user codebase, tooling, and hardware infrastructure.
- R4 **Sharing policies:** Customers are demanding better control over sharing policies (e.g., fairness, priorities, time-based SLOs).
- R5 **Operational flexibility:** Diverse and fast-evolving workloads and deployment environments require operators to change core system behaviors quickly (e.g., within minutes).²
- R6 **Testing and innovation:** The architecture must support partial and dynamic rolling upgrades, to support experimentation and adoption of internal or open-source innovations, while serving mission critical workloads.

2.3 Design Philosophy

From the above we derive the following design choices.

²Redeployments at a scale of tens of thousands of nodes may take days, so it is not a viable option.

Large shared clusters. Requirements R1/R2 push us to share large clusters to avoid fragmentation and to support our largest jobs. Scaling the resource manager becomes key.

General-purpose resource management. R1/R3/R4 force us to invest in a general-purpose resource management layer, arbitrating access from multiple frameworks, including the legacy one, as first class citizens. R3 is at odds with the frameworks-specific nature of our previous distributed scheduling solution [7].

Agile infrastructure behavior. R5/R6 rule out custom, highly scalable, centralized approaches, as integrating open-source innovations and adapting to different conditions would become more delicate and require higher engineering costs. This pushed us towards a federated solution building upon the community innovation at each sub-cluster.

Aligning with open-source. We chose to implement *Hydra* by re-architecting and extending Apache Hadoop YARN [36, 19]. This allows us to leverage YARN’s wide adoption in companies such as Yahoo!, LinkedIn, Twitter, Uber, Alibaba, Ebay, and its compatibility with popular frameworks such as Spark, Hive, Tez, Flink, Cascading, HBase, TensorFlow.

3 Overview of *Hydra*

We now describe the user model (§3.1) and federated architecture of *Hydra* (§3.2), and then present the life-cycle of a job in our system (§3.3).

3.1 Model of User Interaction

The overall cluster capacity is logically organized into *queues* of a given *guaranteed* capacity, i.e., a configured amount of physical cluster resources that is dedicated to each queue.³ Each user has access to one or more queues and can submit jobs to them. Queues support job priorities, optional gang semantics (i.e., minimum required parallelism for a job to launch), as well as many other quota mechanisms, which we omit for brevity. An important value proposition of *Hydra* is to provide users with the illusion of a single large cluster; the details of how this is realized must not be exposed.

Jobs are collections of *tasks* (i.e., processes). Each task runs within a *container*, which is a bundle of physical resources (e.g., <RAM, CPU, IOs, ...>) on a single worker node. Containers can be isolated by means of virtual machines, OS containers, or simple process boundaries. Jobs may dynamically negotiate access to different amounts of resources, which are taken from the guaranteed capacity of the queue they are submitted to. This negotiation is done by a special container: the job’s *Application Master* (AM).

³Servers are *not* partitioned among queues but shared dynamically.

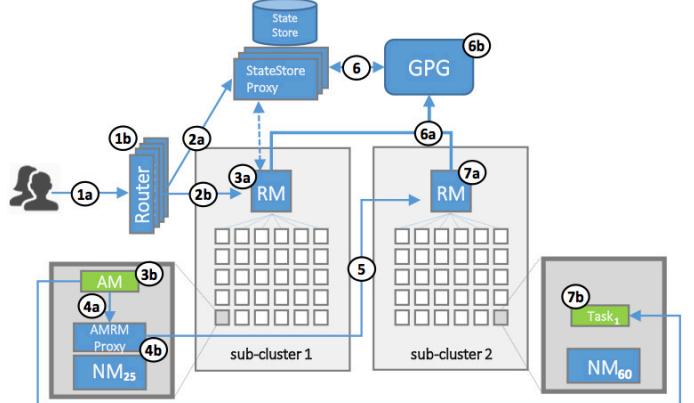


Figure 3: Architecture of *Hydra*.

3.2 System Architecture

In order to meet our goals of: supporting diverse workloads compatible with Apache Hadoop, enforcing stricter sharing invariants, and allowing flexible scheduling policies, we arrived at a federated architecture, depicted in Fig. 3. *Hydra* splits the cluster into a set of sub-clusters that coordinate with each other to present to the user the illusion of a single cluster. Below, we discuss the various system components used to implement this architecture.

Federated architecture. *Hydra* divides the servers of a cluster into logical *sub-clusters*, shown in light-gray boxes in Fig. 3. Each sub-cluster operates as an independent Apache Hadoop YARN [36, 19] deployment (see §A.1 for an overview of YARN’s architecture). In each sub-cluster, a centralized, highly available *Resource Manager* (RM) governs the life-cycle of jobs and tasks, which execute inside containers on the worker nodes. The RM receives resource availability information from the worker nodes through heartbeats, and determines container placement. Each worker runs an instance of *NodeManager* (NM) which is in charge of the life-cycle of the tasks running on that node. The NM is the node-local enforcer for the RM decisions.

Note that each sub-cluster is comprised of a few thousand machines. We discuss in §6 how we set our sub-cluster size, based on YARN’s scalability limitations in the number of worker nodes and scheduling decisions per second (SDPS).

Sub-cluster coordination. Each *Hydra* cluster includes a *Global Policy Generator* (GPG), a component that oversees the entire federation. The GPG periodically obtains an aggregate view of each sub-cluster’s state through RM-GPG heartbeats, and dynamically determines the scheduling policy that should be used by each sub-cluster. Then, the RMs act as the sub-cluster-local enforcers of the scheduling policies dictated by the GPG. The GPG is never in the critical path of individual container scheduling decisions, but rather influence the behavior of other components—this avoids it becoming a scalability and availability bottleneck.

The *StateStore* is a highly available, centralized store⁴ that contains the authoritative copy of all system configurations and policies, and allows us to change the system behavior dynamically. Each component in the system periodically reports liveness to the StateStore through heartbeats, and gets informed of new policies. The *StateStore Proxy* is a caching layer that improves the read-scalability.

Single-cluster illusion. In order to provide users with the illusion of a single large cluster, we need to hide the presence of multiple sub-clusters when (i) a job is submitted, and (ii) an AM requests new resources (containers). To this end, we introduce layers of indirection through two components, namely the *Router* and the *AM–RM Proxy*. Each Router provides the users with a single entry point to the overall platform. Routers mediate all interactions between users and the RMs, dynamically determining the sub-cluster that a job should be launched in. This will be the “home” sub-cluster, where the job’s AM will be running. The *AM–RM Proxy* is a service that runs at every worker node and translates AM container requests to asks to one or more RMs. *This allows individual jobs to span sub-clusters.* Similar to what the Router does for external users, the *AM–RM Proxy* hides the plurality of RMs and shields the applications from the system’s internal complexity.

Along with providing a single-cluster illusion, the Router and *AM–RM Proxy* components allow us to: (i) mask availability issues of RMs (an RM failing is masked by rerouting resource demands to other RMs), (ii) protect RMs from an excessive number of requests (e.g., coming from malicious AMs) that could lead to denial-of-service issues, (iii) balance load among sub-clusters, and (iv) flexibly handle maintenance and flighting operations.

Policy-driven design. All scheduling decisions in *Hydra* are policy-driven and depend on the policies that the GPG installs in the various system components. The choice of policies can significantly change the system’s behavior. In particular, Routers determine where a job is started (thus affecting job queuing time), *AM–RM Proxy* shapes the load that each sub-cluster receives, and RMs determine how requests from multiple queues are fulfilled. Finally, GPG determines the share determination, i.e., deciding how many resources each tenant will get.

Importantly, *the GPG is not on the critical path of container allocation decisions*, but asynchronously updates the RMs’ scheduling policy. The RMs operate independently, in accordance with the most recently received policy. Even if the GPG is not reachable, the RM can continue performing allocations, which ensures that the cluster remains highly available and highly utilized. We provide more details about our policies in §4.

⁴We provide multiple implementations, including SQL Server, HBase, and ZooKeeper, depending on the deployment setting.

3.3 Life of a Job

We illustrate the federated architecture of *Hydra* through the life-cycle of a job (the corresponding steps are also marked in Fig. 3).

- (1) Job j is submitted to queue q via the Router (1a), which determines through a policy the sub-cluster that should be the job’s home, e.g., sub-cluster 1 in Fig. 3 (1b).
- (2) The Router records its decision in the StateStore (2a) and forwards the job submission request to the *RM* of the chosen sub-cluster (2b).
- (3) The *RM* performs admission control and determines when and where to starts the job’s AM (3a). The AM is launched on a node, e.g., NM_{25} (3b).
- (4) The AM begins requesting containers on nodes (e.g., NM_{60}) via the *AM–RM Proxy* (4a), which consults a policy to determine the RMs to forward the request and how to split/merge/modify the requests if needed (4b).
- (5) The *AM–RM Proxy* impersonates the job and contacts all the RMs required to fulfill the demand, e.g., RM_2 . Each job spans as many sub-clusters as needed.
- (6) Each RM summarizes its state (usage and demands) and forwards it to the GPG every few seconds through a separate heartbeat channel (6a). The GPG follows policies to determine share-determination in aggregate, and provides guidance back to the RMs on how to grant access to resources for different tenants (6b).
- (7) The RM uses the most recent policy suggestion from GPG to allocate tasks on behalf of the AM, e.g., $Task_1$ on NM_{60} (7a).⁵ The task is launched, and can begin its computation and to communicate directly with the AM (7b).

4 Scheduling Policies

We now describe the policies governing *Hydra*. Our main goal is to scale placement (§4.1) and share-determination (§4.2) to large numbers of nodes and scheduling decisions per second.

4.1 Placement

Placement of containers on machines affects locality (e.g., between computations and their input data or their hardware preferences like GPU or FPGA), load balancing, and fault tolerance of jobs in *Hydra*. We scale placement by parallelizing decision-making—each sub-cluster performs placement decisions independently. The following policies, associated with different system components (see §3), determine the routing of requests and allocation of containers.

Router policies assign jobs’ Application Masters (AMs) to sub-clusters. Since AMs consume limited resources and typ-

⁵Note that 7a could happen before 6a-6b, if the previous round of allocation policies from GPG allow the RM to allocate more for queue q .

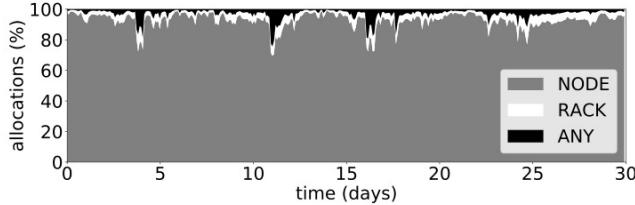


Figure 4: Placement quality over a period of one month: >97% of requests on average are placed on the preferred node or rack.

ically have no locality requirements, Router policies focus on fault tolerance by spreading AMs across sub-clusters. To this end, the Router retrieves the current sub-cluster state (e.g., active RM’s URL, current utilization, outstanding demand) from the StateStore (cached for performance). The Router itself is fully stateless and thus is horizontally scalable. We achieve fault tolerance by deploying multiple Router instances behind a load balancer. Internally the Router adopts an interceptor design pattern, which allows us to dynamically change and configure its policies.

AM–RM Proxy policies affect the placement of all non-AM containers. We experimented with various policies in our production environments, trying to strike a balance between locality,⁶ load balancing, and fault tolerance. The policy we settled for tries to optimize for locality preferences by forwarding requests to the RM that owns the node specified in the request. Our policy extends the notion of delay-scheduling [42] to operate across sub-clusters, falling back to less loaded RMs after a configurable time-out.⁷ Requests that do not specify locality preferences are spread across sub-clusters to improve load balancing. This is done by leveraging YARN’s existing notion of headroom [2], that is, an RM estimated amount of resources that a specific user/application should expect to obtain based on the current state of the sharing invariants. In case of an RM failure, the AM–RM Proxy will automatically discover (via the StateStore) the new master RM and send all outstanding requests to it. If an RM becomes unavailable for a longer duration (due to overload conditions or network partitions), the AM–RM Proxy re-distributes pending requests among other RMs inversely proportionally to their load, thus increasing our resiliency to failures. The AM–RM Proxy is stateful and utilizes the existing NM mechanisms for failure recovery [3], in a way transparent to both the AM and RM.

RMs employ complex algorithms [36] that handle locality within a sub-cluster via delay-scheduling [42]. The RM first

⁶Batch applications typically express soft preferences to be co-located with their input data—matching these preferences at least at the sub-cluster level is important, as it minimizes cross-datacenter data transfers, given that sub-clusters do not cross datacenter boundaries while clusters often do.

⁷We are experimenting with variants of this relaxation mechanism based on load, configured weights, or simple randomization.

tries to place a task at the requested node, then it falls back to the respective rack, and then to any cluster node. *Hydra* also supports the notion of node labels [19] (tags used to logically group machines) and the more sophisticated placement constraints described in Medea [11, 24]. While our software stack supports all these, the more advanced constraints are not yet employed by most of our production users.

NM policies govern the local use of resources among containers of different types. We leverage the notion of guaranteed and opportunistic containers from [18, 7] to ensure high cluster utilization. Opportunistic containers are queued locally to the NM (similar to [18, 27]) and run when there are unused resources, thus hiding feedback latencies that would lead to underutilized machines when running short-lived tasks or jobs that request part of their containers in gangs (both very common in our workloads; see §2).

4.1.1 Quality of Placement

Adapting Scope applications [44] to run on *Hydra*, we reuse Apollo’s logic to request the node on which a task should run. We then quantify the quality of placement in *Hydra* by measuring the fraction of container requests that got placed exactly on the node the AM requested, or on the corresponding rack. Fig. 4 shows the achieved placement quality over one month across our fleet. *Hydra* achieves on average 92% node-local allocations and 97% rack-local allocations. These results demonstrate that we are able to meet the practical needs of our clusters’ users and are comparable to what we observed when operating non-federated YARN clusters.

4.2 Share-Determination

Share-determination requires a global point of view, but we observe that an aggregated view of the cluster state and workload demands is sufficient, given that placement decisions are performed locally at each sub-cluster. The GPG receives a summarized view (i.e., a snapshot of the current utilization and outstanding demand) of each sub-cluster on heartbeats (6a of Fig. 3), and performs share-determination decisions, which are then pushed down to each sub-cluster to affect allocations. We discuss alternative policies below.

Gang admission control (GAC) is a policy that performs admission control of jobs based on gang scheduling semantics. In particular, the GPG maintains a queue hierarchy with statically defined quotas, which are pushed to the Routers. When a job with a declared parallelism of k containers gets submitted to a queue, it waits until k containers become available from the queue’s quota. GAC is the share-determination policy currently used in production, both for its simplicity and because it matches the behavior of the legacy system [7] that our customers are used to. Moreover, GAC has been hardened in production in our legacy system for several years.

As we introduce more application frameworks in our clusters,⁸ we can leverage *Hydra*'s flexible control plane to perform share-determination dynamically at the container request level, rather than once at job submission time. *Global instantaneous fairness (GIF)* is one such promising policy that we are currently experimenting with in our test clusters.

GIF copes with scale by aggregating the cluster state and workload demands at the granularity of queues. It logically reassigned the entire cluster resources according to work-preserving fair-sharing [12, 6], while taking into account the sub-cluster that the resource demand is coming from, i.e., allocating to each sub-cluster at most the resources demanded there. The output of GIF is the instantaneous resource capacity that a queue can receive at each sub-cluster. For example, consider a cluster with two equally sized sub-clusters S_A, S_B and a queue q_1 that is assigned 20% of the overall cluster's capacity. By default, q_1 will get the static 20% of resources at each sub-cluster. Instead, assuming q_1 has higher demand on S_A , GIF can allow q_1 to get, say, 30% of resources in S_A and 10% in S_B . This results in the same overall capacity for q_1 , but can improve resource utilization.

To validate the feasibility and significance of such flexible policies, we compare GIF with the following approaches: (i) an idealized single centralized YARN scheduler that overlooks the entire cluster accounting for node locality (*Centr.*); (ii) one scheduler for every sub-cluster, where each queue is replicated in each sub-cluster, with a quota uniformly proportional to the sub-cluster size (*Uniform Distr.*); (iii) similar to (ii) but with queue quotas dynamically mapped to sub-clusters based on the workload demand for resources on each sub-cluster (*Load-Based Distr.*). We use a few hundreds of different cluster setups, each with a few tens of sub-clusters and with queue hierarchies of varying sizes and complexities, created by a generator that we built (and plan to open-source) for this purpose. These comparisons are based on simulations, given that (i) cannot support our cluster and workload sizes.

[Fig. 5](#) shows our results in terms of utilization and fairness. To measure fairness we compute the mean absolute error between each candidate solution and a reference ideal implementation consisting of a centralized scheduler, similar to (i), that is *also* allowed to arbitrarily relax node locality—this represents an oracle for fairness as other constraints are relaxed. Mean absolute error is measured as a percentage error from the centralized solution. As an intuition to read this graph, a value of 0.1% corresponds to just a few hundred unfairly allocated containers across our 50k-node clusters. GIF performs better (much higher utilization and comparable fairness error) than the sub-cluster-local approaches (ii) and (iii), and closely tracks the behaviors of the (impractical)

⁸Note that GAC can already support such jobs, but relies heavily on opportunistic execution (as in [18]) to achieve high cluster utilization, making it harder to maintain fairness across users.

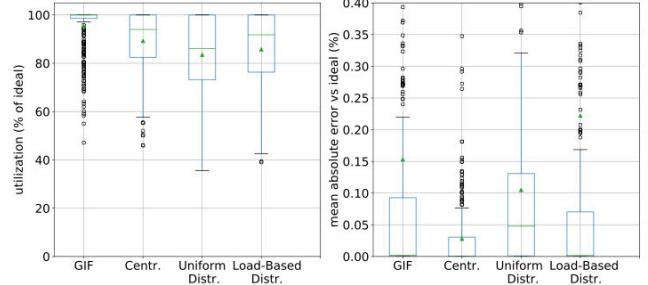


Figure 5: Comparing policies for share-determination with respect to utilization (left) and fairness (right). The boxes indicate the 25th and 75th percentile; the green line within each box is the median; the small triangle is the average; the whiskers represent the 5th and 95th percentiles.

centralized solutions. GIF's runtime is also acceptable, with <300ms runtime when simulating our largest clusters.

5 Implementation and Open-Sourcing

In this section, we provide details on the implementation and open-sourcing of *Hydra*. Given Microsoft's commitment to supporting open-source workloads and our team's expertise on Apache Hadoop, YARN [36] was a natural choice as a base to build *Hydra*, as we also discuss in §8. Over the past few years, Microsoft has contributed over 200k lines of code to Apache Hadoop related to *Hydra*.

To enable *Hydra*'s federated architecture, we extended YARN with the various components described in §3, including the GPG, Router, AM–RM Proxy, and StateStore. Each component runs as a separate microservice in YARN. More implementation details are available at [41, 13].

Apart from the main federated architecture described in this paper, we also leverage several other efforts that have been contributed to YARN by Microsoft over the past years. In fact, the federated architecture constitutes the last piece of the puzzle in operating YARN at our scale. [Tab. 4](#) summarizes the main Microsoft contributions to YARN that are exploited by *Hydra*. For example, opportunistic execution is key to achieve high resource utilization, given the scale of our scheduling decisions and the way our workloads were tuned to run on our legacy system.

Additionally, while deploying *Hydra* in production, we had to perform several other improvements to YARN. Here we briefly mention a few representative ones. First, to support our workload's large-scale parallelism and requirements for large numerous resource files, we enhanced the open-source version of resource localization at each worker node. In particular, we added finer control of simultaneous resource downloads (accounting for available bandwidth I/O, and guaranteed vs. opportunistic containers), and leveraged

Feature	JIRAs	Hadoop version	Publications
Federated architecture	[41]	2.9	this paper
Scheduling policies	[13]	ongoing	this paper
Opportunistic execution	[22, 39]	2.9	[18, 27]
Reservation planning	[29, 28]	2.9	[9, 17]
Placement constraints	[24]	3.1	[11]
Container preemption	[25]	2.1	[36]

Table 4: Key Microsoft contributions to YARN, which are relevant to *Hydra*. For each feature, we provide the main JIRA numbers (where the open-source effort can be tracked), the version of Apache Hadoop that first included it, and related publications.

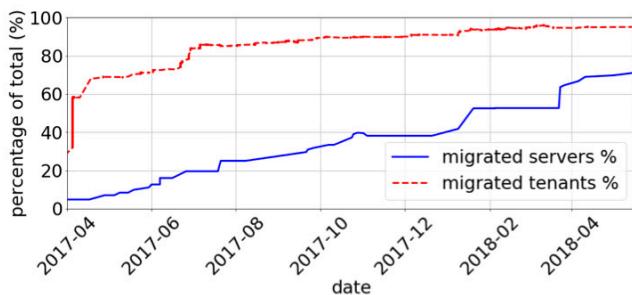


Figure 6: Capacity roll-out over time as a fraction of our overall fleet.

an existing peer-to-peer file segmentation, distribution and caching service. Further, we made improvements related to the scheduling latency at each sub-cluster, by adding time-based relaxation of locality. This was crucial to support our scheduling decisions rate and meet locality preferences of requests. Finally, we enhanced the default logging and metrics collection in YARN.

6 Deployment and Lessons Learned

In this section, we discuss our experience in deploying *Hydra* in our production clusters and migrating a live service to it (§6.1), as well as an approach to evolutionary design of *Hydra*'s architecture (§6.2).

6.1 Deployment Over Time

Given the size of our fleet and the mission-critical nature of our workloads, the migration of our service to *Hydra* was by itself a highly complex project. *Hydra* had to be deployed “in-place”, with almost no spare resources and with stringent SLOs for at least part of our workload.

In-place progressive migration. Our migration proceeded one tenant at a time, by migrating their workload to *Hydra*. At the same time, we were reassigning machines from the legacy system to *Hydra*. We grouped tenants into 4 tiers, from the smaller (in workload size and resource demands)

and least mission-critical of tier-4 to the larger and SLO-sensitive tenants of tier-1. Migration proceeded from tier-4 upwards to tier-1. This progression can be observed in Fig. 6, where a rather linear machine migration is paired with a non-linear tenant migration. Tenant migration was fast at first (small tier-4 and tier-3 customers), and slowed down as more of the customers from tier-2 and tier-1 were onboarded and large capacity is assigned with numerically few tenants.

Throughout the deployment, we paused migration for operational and general engineering cadence. The rollout was not without surprises, as every set of tenants exercised our system in different and challenging ways. We constantly measured job performance and reliability, using predictable, recurring jobs as canaries. Whenever regression was detected by our team or by customers, we temporarily rolled back part of our workload to the legacy system, investigated, addressed the issues, and resumed migration. In the process, we made many performance and reliability improvements to YARN, many of which have been committed back to the Apache Hadoop project (see also §5). One large incident was around November 2017, when thousands of nodes were rolled back, as seen in Fig. 6.

Mismatched load/hardware. A surprising challenge consisted in migrating the right number of machines for each portion of the workload we switched over. At first, we naively started by migrating the nominally purchased capacity for each tenant. We quickly learned how heavily most tenants relied on scavenged capacity to support important application scenarios. In hindsight, stronger workload models could have predicted some of these problems. We subsequently built tooling to prevent similar regressions.

Locality during migration. Jobs cannot span the legacy system and *Hydra*. By migrating machines between the two systems, we limited the chances of co-location with data (which is kept in place and is accessible through a distributed file system akin to HDFS [26]). Our hypothesis, confirmed during migration, was that locality matters less with modern networks. Nonetheless, to ameliorate possible negative effects, we migrated machines by striping across racks, i.e., first a few machines in each rack were migrated to *Hydra* and then more machines were added. This strategy fundamentally worked, with a few hiccups in the early phases when the number of *Hydra* machines was so limited that many racks had zero or just one machine (leading to frequent cross network-spine reads, which impacted job performance).⁹

Sub-cluster size. Given *Hydra*'s ability to support jobs that span sub-clusters, we have the freedom to pick the number and size of each sub-cluster. We proceeded by trial and error, picking the largest sub-cluster size for which the sub-cluster's RM could comfortably handle our workload. In our

⁹We did not observe this issue at the opposite end of the spectrum (when the migration was close to completion), as the queues being moved were very large, thus avoiding some of these border effects.

deployments, the deciding factor for this limit is the scheduling rate and not the number of nodes. In its current implementation, the RM cannot support more than 1k scheduling decisions per second (SDPS), while recent, extensive efforts to improve RM’s scalability have raised this bar to 5k SDPS under ideal conditions [40]. This is still about an order magnitude lower than our target SDPS, and has not yet been tested in production at scale. *Hydra*’s ability to tweak each sub-cluster’s size allowed us to experiment with different sizes while impacting only part of our capacity (e.g., setting one sub-cluster to be larger/smaller than others). In our current deployments, we operate clusters with 15–20 sub-clusters, each with 2k–3k nodes. We are also working towards merging some of these clusters, creating even larger and cross-DC *Hydra* deployments. So far, we have not encountered any obvious upper bound to *Hydra*’s scalability.

6.2 Evolutionary Design

Through careful analysis and design, we were reasonably successful at designing the “static” aspects of the system, such as core architectural choices and basic mechanisms. Conversely, predicting all system “dynamics” and the effects of multiple interacting algorithmic policies was exceedingly hard. Moreover, we knew that the workload and the hardware environment *Hydra* had to target in its lifetime were certainly going to change over time.

Therefore, we invested in faithful simulation infrastructure [30], which proved invaluable (similar to what was reported by the Borg team [37]), and focused on designing mechanisms that allowed us to test and deploy policies very quickly. In particular, *Hydra* can propagate new policy behaviors across many thousands of nodes within seconds (bypassing the slower mechanisms provided by the underlying deployment infrastructure [16]). This agility allowed us to experiment with policies and respond to outages quickly. This flexibility enabled us to leverage *Hydra* to support sub-cluster level A/B testing and specialized tuning for alternative environments. *Hydra* is, in fact, now being considered for smaller/specialized clusters, where federation is not used to achieve scale, but rather operational flexibility. The ability to test policies on parts of our clusters and quickly revert them, made it possible to experiment with advanced policies, as the ones discussed in §4. This enables faster innovation, which we believe is a generally under-appreciated aspect of real-world systems. Similar considerations were discussed by the TensorFlow team [4].

7 Production Validation

In this section we provide data from our production environments running *Hydra*, comparing (whenever possible) with our legacy system [7]. As already discussed (see §1, §2), our

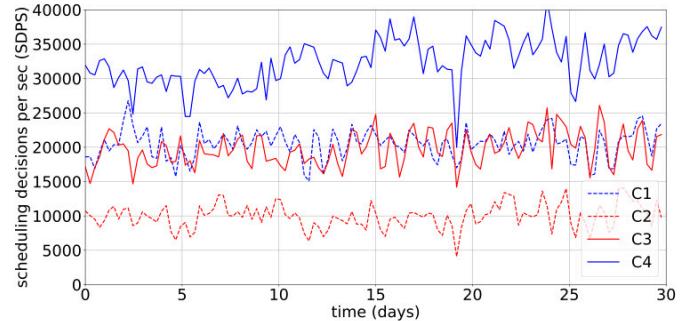


Figure 7: Scheduling decisions per second for 5 large clusters (average behavior over 6-hour windows).

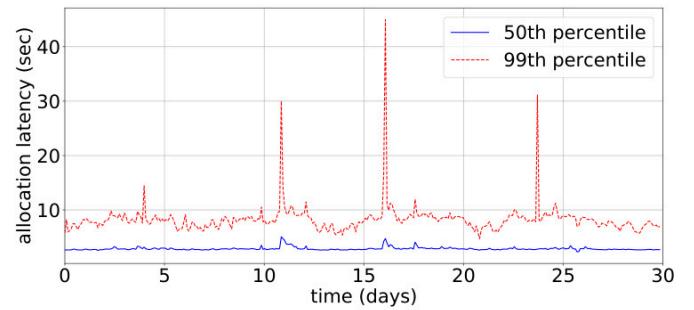


Figure 8: Allocation latency over a period of one month for one of our busiest clusters.

key challenge in building *Hydra* was to match the scalability and utilization of the mature, highly tuned, application-specific legacy system it replaces [7], which has the advantage of relying on a fully distributed scheduler. We show numerically that this has been achieved. At the same time, *Hydra* provides the benefits of a centralized scheduler, namely the ability to support arbitrary application frameworks, stricter scheduling invariants, and better operational controls, as discussed qualitatively and anecdotally in §6.

7.1 Scheduling Rates and Latency

Fig. 7 shows the number of scheduling decisions per second for four of our largest clusters, each comprised of 15–20 sub-clusters. We observe that while the cluster sizes are comparable, the workloads differ substantially. By way of example, cluster C2 requires around 10k scheduling decisions per second (SDPS), while cluster C4 requires sustained rates of 30k to 40k SDPS. Note that each task might require multiple scheduling decisions, because tasks are often promoted/demoted between guaranteed and opportunistic execution throughout their lifetime [18]. Each request for an allocation, promotion, demotion, or termination corresponds to a scheduling decision.

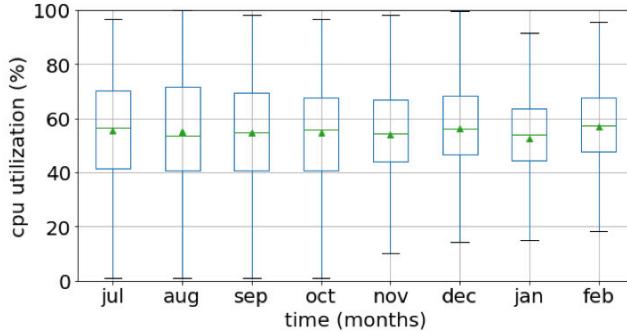


Figure 9: CPU utilization across our fleet over a period of 8 months, during which we were deploying *Hydra* in an increasing number of machines.

[Fig. 8](#) reports the median and the 99th percentile of allocation latency across all allocations performed by *Hydra* in a 30-day period in one of our busiest clusters. The median allocation latency is 2-3 seconds. The 99th percentile is typically no more than 10 seconds with a few exceptions. This is measured at the client side and focuses on the most expensive type of allocations (guaranteed containers per [18] terminology). In comparison, Apollo had no centralized task scheduling delays, but it could incur high queuing delays when tasks were dispatched to each node. *Hydra*'s centralized approach is allowing us to experiment with centrally coordinated locality relaxation, which is reducing scheduling latencies further in our test clusters. Recall that, as we showed in §4.1 ([Fig. 4](#)), *Hydra* also delivers high quality placement decisions, managing to allocate 70-99% of the tasks on their preferred node (and 75-99.8% within rack).

7.2 Utilization

[Fig. 9](#) shows the CPU utilization (i.e., actual busy CPU cycles) across a large fraction of our fleet. During a period of eight months, most of these machines were migrated to *Hydra* (see [Fig. 6](#)). The plot shows that there was no resource utilization regression. In fact, we observe a slight increase in average CPU utilization of about 1-2% (although this may be due to increased workload pressure). Importantly, we observe that the load distribution is more even across our fleet, as seen in [Fig. 9](#) where the gap between high and low percentiles narrows as *Hydra* is rolled out. This indicates that *Hydra*'s placement decisions lead to better load balancing.

To more clearly highlight where the CPU utilization improvements come from, we zoom into one of our largest clusters. [Fig. 10](#) shows for each machine in the cluster the number of tasks run on each node and their average input size. Different shades of black represent different server configurations with varying number of cores, memory, disks, and network sub-systems—the darker the color the more powerful the machine, thus we expect lighter colors to receive less

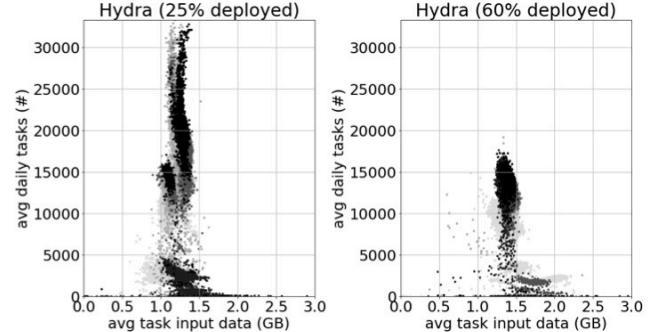


Figure 10: Scatter plot of task input data vs. count of tasks running daily on each server (one month average for legacy and *Hydra*). Each dot represents one server, and darker color indicates more powerful hardware configuration.

load. Comparing *Hydra* (on the right-hand side of the figure) to our legacy system (on the left-hand side), it is evident that *Hydra* distributes tasks (roughly the same total number) more evenly across servers. This explains the tighter distribution of CPU utilizations we observed in [Fig. 9](#).

7.3 Task Runtime and Efficiency

We now study the task runtime before and after we deployed *Hydra* across our fleet. The empirical CDFs of [Fig. 11](#) capture the behavior of 135B tasks in terms of input data, runtime, and processing efficiency (calculated by dividing the runtime by the input size). Each figure includes two CDFs: the “*Hydra* (25% deployed)”, which corresponds to an early phase in our deployment (with most tasks running the legacy system and around 25% of nodes migrated to *Hydra*); the “*Hydra* (60% deployed)”, which corresponds to a one month window when over 60% of the nodes were migrated. From the CDFs we notice that users have increased the input size of our smallest tasks, which lead to an increase in task runtime. This is not due to *Hydra* deployment, but rather to exogenous causes (e.g., coinciding changes in the tuning of the application frameworks, growth in job input). When it comes to task throughput, *Hydra* is comparable to the legacy system, which was our primary goal. In particular, the overall throughput increased in the lower end (less efficient tasks improved by a large amount), while slightly decreased for very efficient tasks. This is consistent with the better cluster load balancing we observed, as fewer tasks are run in under- or over-loaded machines.

7.4 Job Runtime

We conclude with the most crucial end-to-end metrics that our customers care about: job runtime. For a precise comparison, we focus on periodic jobs, i.e., recurring jobs that run the same computation on new data every week, day, or

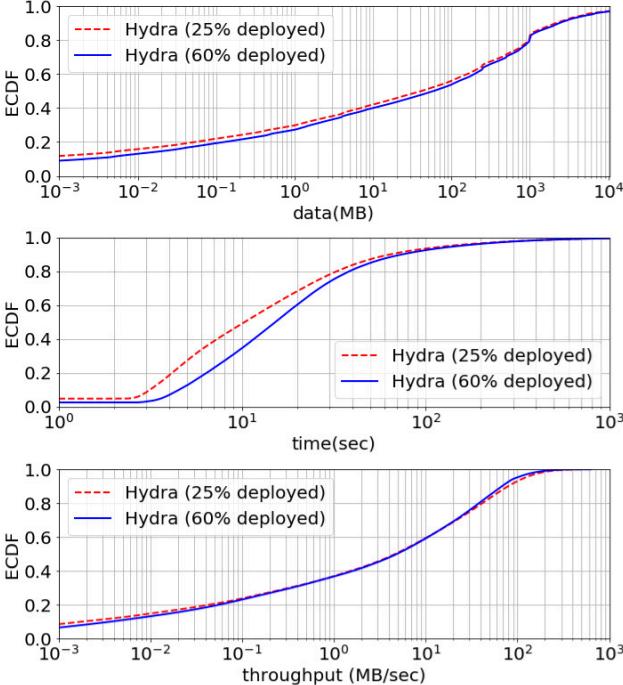


Figure 11: Task behavior before and after migration.

hour. Fig. 12 reports runtimes of periodic jobs and resource utilization (in task-hours) before and after migration to *Hydra*. To control for substantial job changes, we focus on periodic jobs whose input size changed by no more than 2X (increase or decrease). The figure presents results for 18k periodic job templates, corresponding to 1.25M job instances, which consist of 10B tasks and 110M task-hours (tasks are multi-threaded, each assigned 2 cores, but potentially using more). The graph is plotted as an empirical CDF of the ratio of job runtime ($\frac{\text{after}}{\text{before}}$) and total task-hours. We observe that, while the size of jobs grew substantially (the median increased by 18%) during this period of time (more tasks and longer runtimes), the job runtime has not substantially changed (with median runtime decreasing by 4%). By inspecting many example jobs, we conclude that the “tails” of this CDF are mostly due to increase/decrease of the overall job input data. While the journey to a fully deployed *Hydra* was not without challenges, our customers are broadly satisfied with job performance at this point.

7.5 Discussion

The production data presented in this section show that *Hydra* matches the utilization and scalability of our legacy system, and delivers consistent (or slightly improved) job runtimes and improved load distribution across machines. At the same time, *Hydra* supports multiple application frameworks, and advances our ability to quickly experiment and innovate in the space of scheduling policies.

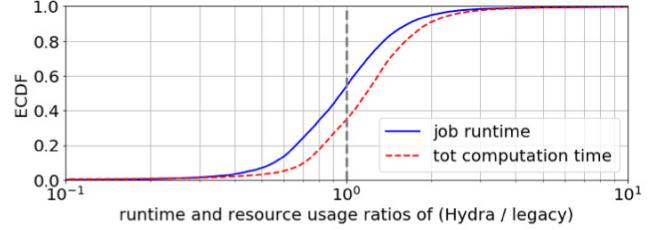


Figure 12: Empirical CDF for 18k recurring job templates (1.25M job instances). We plot the ratio (*Hydra*/legacy) for the job runtimes and total task-hours. All values are averages over one month.

8 Related Work

We focus our related work comparison on production-ready, scalable, big-data resource management frameworks, as these were the only relevant contenders given our goals. Among open-source systems, we will discuss Apache Hadoop YARN [36, 19], Apache Mesos [15], and Kubernetes [20], and among proprietary ones we consider Borg [37] (not accessible to us) and Apollo [7]. None of these systems, in their current form, could handle all our requirements (scale of machines and scheduling rates, arbitrary application frameworks, global sharing invariants).

Apollo, [7] our legacy system (see also §A.2), scaled by carefully choreographing a homogeneous collection of batch jobs [44]. Our requirement to support a wide variety of proprietary and open-source ecosystems forced us to evolve beyond Apollo’s fully distributed scheduling. We retained Apollo’s distributed job-level placement algorithms, run in the job AM, but *Hydra*’s resource management infrastructure has the final say on if and where a job can run. Controlling this decision in *Hydra* allows us to share our clusters between arbitrary application frameworks and improve load balancing. In our previous work [18, 27], we added support for opportunistic containers to YARN, a notion inspired by Apollo, which we leverage in *Hydra* to ensure high cluster utilization—a must at our scale.

YARN. By building upon YARN [36], *Hydra* inherits its capabilities and strengths [19]. First and foremost, *Hydra* is wire-compatible with unmodified YARN applications, thanks to *Hydra*’s Routers and AM–RM Proxy that hide the federated nature of our infrastructure. This allows us to co-locate Microsoft’s native frameworks [44] with open-source applications, such as Spark [43], TensorFlow [4], Hive [33], REEF [38], and Heron [14].

The choice to federate multiple YARN sub-clusters instead of scaling up a single RM was driven by the scale in machine number and scheduling rate (§6.1) and our operational requirements (e.g., rolling upgrades, A/B testing, deploying new policies). On the non-technical front, we also considered long-term cost of ownership. Modifying YARN to scale up would either impede future innovations by the

Hadoop community (forcing every new feature only if it did not affect scalability to tens of thousands of nodes) or lead to isolation of Microsoft from the community—leading to high porting/stabilization of every new feature, defeating the purpose of an open-source system. By federating smaller sub-clusters, we rely on some of the stabilization work done by vendors (Hortonworks, Cloudera) and other big companies (Yahoo!, Twitter, Uber, LinkedIn). Examples of innovations relevant to *Hydra* include our work in opportunistic containers [18, 27], time-based SLOs [9, 17], placement constraints [11], advanced preemption [25], as well as community-driven efforts, such as node labels [21] and support for GPUs [19]. Finally, the naive alternative of running an independent RM per sub-cluster would lead to unacceptable resource fragmentation—this was studied by the authors of [37] and consistent with our experiments (§4.2).

Mesos [15] is best understood as a highly scalable cluster manager, focusing on deploying long running services. At Microsoft, we support three similar frameworks, AutoPilot [16], Azure Service Fabric [5], and Kubernetes [20]. *Hydra* can be deployed using any of these cluster managers, but focuses on supporting high-rate batch workloads, as well as interactive and streaming applications. The rate of scheduling decisions per second that *Hydra* can achieve is substantially higher than what Mesos targets. Additionally, *Hydra* supports resource sharing across application frameworks.

Kubernetes [20] is a more recent effort that occupies a similar space as Mesos. At the time of this writing, the Kubernetes scheduler is very simple and provides little support for short-lived containers and batch applications. Kubernetes’ current scalability (5k nodes at the time of this writing) is also short of our target. Nonetheless, Kubernetes has an elegantly modular architecture and very substantial momentum in open-source, so any of the limitations we discuss here are likely to be addressed sooner or later.

Borg [37] is Google’s proprietary resource manager. Unsurprisingly, this system comes closest to matching our requirements (though it is not open-source, so our understanding is limited to what is discussed in [37]). Borg’s notion of cells is akin to a sub-cluster from a deployment/maintenance perspective, but Borg jobs cannot span cells, and so the typical configuration has one large production cell and some smaller test cells with the median cell size being 10k nodes. Borg cells do not span data centers, contrary to what we do in some of our *Hydra* deployments. The reported scheduling rate for Borg is 10k decisions per min (or 166 decisions per second). Our production deployment routinely reaches 40k decisions per second in our busiest clusters. While a centralized approach is likely to eventually top out, we believe the current scheduling rate for Borg is due to natural co-evolution of the application framework and the scheduling infrastructure, rather than an intrinsic limitation. The Borg microservice architecture elegantly decouples admis-

sion control (quotas) from (sharded) state management and scheduling. This design is close in spirit to *Hydra*. However, *Hydra*’s ambition to support dynamic enforcement of strict scheduling invariants is not compatible with checking quotas only during admission control (which is, to the best of our understanding, the only mechanism supported by Borg). Finally, *Hydra* inherits YARN’s ability to support soft/hard data locality preferences on a per-container basis, which is not supported in Borg.

HPC systems. Prior to the current generation of large-scale resource managers, much effort has been devoted to scheduling in the context of High Performance Clusters (HPC). Prominent systems included Condor [32], Torque [31], Moab [10], and Maui [1]. As discussed in [36], these systems’ original focus on MPI applications leads to a limited support for elastic scheduling and data-locality—instrumental for batch big data applications. Furthermore, their limited compatibility with the current open-source big data ecosystems made them incompatible with some of our fundamental requirements.

Influences. In building *Hydra*, we were influenced by a body of research that extends well beyond the production systems discussed above. These include dominant resource fairness [12, 6], delay scheduling [42], distributed scheduling [23], constraint- and utility-based scheduling [34, 35], and the container reuse/executor model of [38, 43].

9 Conclusion

This paper summarizes our journey in building *Hydra*, a general-purpose resource management framework, capable of supporting a broad set of open-source and Microsoft’s proprietary application frameworks. *Hydra* scales to clusters of 50k+ nodes, performing tens of thousands of scheduling decisions per second. Our design pivots around the following core ideas: (i) a federation-based architecture, (ii) de-coupling of share-determination from placement of tasks to machines, and (iii) a flexible control-plane that can modify the system’s scheduling behavior within seconds.

Hydra is the main resource manager at Microsoft, and it is deployed across hundreds of thousands of servers. It has so far scheduled a trillion tasks that processed close to a Zettabyte of data. The utilization and scalability that *Hydra* matches our state-of-the-art, distributed scheduler, while allowing us to support arbitrary open-source and internal analytics frameworks. We contributed *Hydra*’s implementation to Apache Hadoop (~ 200 k lines of code), and are committed to continuously open-source future advancements.

Acknowledgments. We would like to thank our shepherd, John Wilkes, and the anonymous reviewers for their insightful feedback. We also thank the Microsoft BigData team for their help in improving and deploying *Hydra* in production. Finally, we thank the Apache Hadoop community for the discussions and code-reviewing during *Hydra*’s open-sourcing.

References

- [1] Maui Scheduler Open Cluster Software. <http://mauischeduler.sourceforge.net/>, 2005.
- [2] Hadoop: Writing yarn applications. <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/WritingYarnApplications.html>, 2012.
- [3] Work-preserving NodeManager restart. <https://issues.apache.org/jira/browse/YARN-1336>, 2014.
- [4] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P. A., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI* (2016).
- [5] BAI, H. *Programming Microsoft Azure Service Fabric*. Microsoft Press, 2018.
- [6] BHATTACHARYA, ARKA, C., DAVID CULLER, F., ERIC FRIEDMAN, G., ALI, S., SCOTT, A. S., AND STOICA, I. Hierarchical Scheduling for Diverse Datacenter Workloads. In *SOCC* (2013).
- [7] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI* (2014).
- [8] CANO, I., WEIMER, M., MAHAJAN, D., CURINO, C., FUMAROLA, G. M., AND KRISHNAMURTHY, A. Towards geo-distributed machine learning. *IEEE Data Eng. Bull.* (2017).
- [9] CURINO, C., DIFALLAH, D. E., DOUGLAS, C., KRISHNAN, S., RAMAKRISHNAN, R., AND RAO, S. Reservation-based Scheduling: If You're Late Don't Blame Us! In *SOCC* (2014).
- [10] EMENEKER, W., JACKSON, D., BUTIKOFER, J., AND STANZIONE, D. Dynamic virtual clustering with Xen and Moab. In *ISPA* (2006).
- [11] GAREFALAKIS, P., KARANASOS, K., PIETZUCH, P. R., SURESH, A., AND RAO, S. Medea: scheduling of long running applications in shared production clusters. In *EuroSys* (2018).
- [12] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI* (2011).
- [13] Federation v2: Global optimizations. <https://issues.apache.org/jira/browse/YARN-7402>, 2018.
- [14] Heron. <http://apache.github.io/incubator-heron>, 2018.
- [15] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI* (2011).
- [16] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review* (2007).
- [17] JYOTHI, S. A., CURINO, C., MENACHE, I., NARAYANAMURTHY, S. M., TUMANOV, A., YANIV, J., MAVLYUTOV, R., GOIRI, I., KRISHNAN, S., KULKARNI, J., AND RAO, S. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI* (2016).
- [18] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *USENIX ATC* (2015).
- [19] KARANASOS, K., SURESH, A., AND DOUGLAS, C. Advancements in YARN Resource Manager. *Encyclopedia of Big Data Technologies* (February 2018).
- [20] Kubernetes. <http://kubernetes.io>, 2018.
- [21] Node labels: Allow for (admin) labels on nodes and resource-requests. <https://issues.apache.org/jira/browse/YARN-796>, 2014.
- [22] Scheduling of opportunistic containers through YARN RM. <https://issues.apache.org/jira/browse/YARN-5220>, 2017.
- [23] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *SOSP* (2013).
- [24] Rich placement constraints in YARN. <https://issues.apache.org/jira/browse/YARN-6592>, 2018.
- [25] Preemption: Scheduler feedback to AM to release containers. <https://issues.apache.org/jira/browse/YARN-45>, 2013.
- [26] RAMAKRISHNAN, R., SRIDHARAN, B., DOUCEUR, J. R., KASTURI, P., KRISHNAMACHARI-SAMPATH, B., KRISHNAMOORTHY, K., LI, P., MANU, M., MICHAYLOV, S., RAMOS, R., SHARMAN, N., XU, Z., BARAKAT, Y., DOUGLAS, C., DRAVES, R., NAIDU, S. S., SHAstry, S., SIKARIA, A., SUN, S., AND VENKATESAN, R. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *SIGMOD* (2017).
- [27] RASLEY, J., KARANASOS, K., KANDULA, S., FONSECA, R., VOJNOVIC, M., AND RAO, S. Efficient Queue Management for Cluster Scheduling. In *EuroSys* (2016).
- [28] Support for recurring reservations in the YARN ReservationSystem. <https://issues.apache.org/jira/browse/YARN-5326>, 2018.
- [29] Yarn admission control/planner: enhancing the resource allocation model with time. <https://issues.apache.org/jira/browse/YARN-1051>, 2018.
- [30] Scheduler Load Simulator for Apache Hadoop YARN. <https://issues.apache.org/jira/browse/YARN-5065>, 2017.
- [31] STAPLES, G. TORQUE resource manager. In *IEEE SC* (2006).
- [32] TANNENBAUM, T., WRIGHT, D., MILLER, K., AND LIVNY, M. Condor: A Distributed Job Scheduler. In *Beowulf Cluster Computing with Linux* (2001).
- [33] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTHONY, S., LIU, H., AND MURTHY, R. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE* (2010).
- [34] TUMANOV, A., CIPAR, J., GANGER, G. R., AND KOZUCH, M. A. Alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds. In *SOCC* (2012).
- [35] TUMANOV, A., ZHU, T., PARK, J. W., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys* (2016).
- [36] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC* (2013).
- [37] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *EuroSys* (2015).
- [38] WEIMER, M., CHEN, Y., CHUN, B.-G., CONDIE, T., CURINO, C., DOUGLAS, C., LEE, Y., MAJESTRO, T., MALKHI, D., MATUSEVYCH, S., MYERS, B., NARAYANAMURTHY, S., RAMAKRISHNAN, R., RAO, S., SEARS, R., SEZGIN, B., AND WANG, J. REEF: Retainable evaluator execution framework. In *SIGMOD* (2015).
- [39] Extend YARN to support distributed scheduling. <https://issues.apache.org/jira/browse/YARN-2877>, 2017.
- [40] Move YARN scheduler towards global scheduler. <https://issues.apache.org/jira/browse/YARN-5139>, 2018.
- [41] Enable YARN RM scale out via federation using multiple RM's. <https://issues.apache.org/jira/browse/YARN-2915>, 2017.
- [42] ZAHARIA, M., BORTHKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys* (2010).

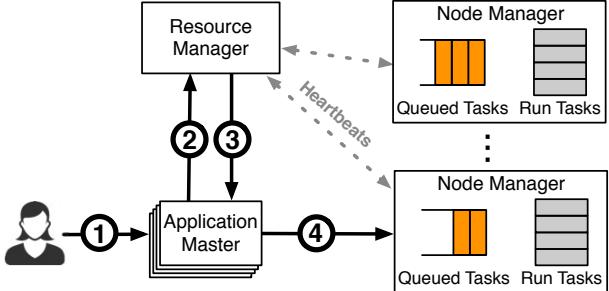


Figure 13: (Non-federated) YARN architecture.

- [43] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *HotCloud* (2010).
- [44] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-Å., CHAIKEN, R., AND SHAKIB, D. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636.

A Overview of YARN and Apollo

As discussed in Sections 2.3 and 3, we built *Hydra* by extending and re-architecting Apache Hadoop YARN [36] to substitute our legacy system Apollo from which we also drew ideas [7]. For convenience to the reader, below we provide a brief overview of YARN and Apollo, given their close ties to *Hydra*.¹⁰

A.1 YARN

YARN follows a centralized architecture (depicted in Fig. 13), in which a single logical component, the Resource Manager (RM), allocates resources to jobs submitted to the cluster. The resource requests handled by the RM are intentionally generic, while specific scheduling logic required by each application is encapsulated in the Application Master that any framework can implement. This allows YARN to support a wide range of applications using the same RM component. Below we describe its main components.

Node Manager (NM). The NM is a daemon running at each of the cluster’s worker nodes. NMs are responsible for monitoring resource availability at the host node, reporting faults, and managing containers’ life-cycle (e.g., start, monitoring, pause, queuing, and killing of containers).

Resource Manager (RM). The RM runs on a dedicated machine, arbitrating resources among various competing applications. Multiple RMs can be used for high availability, with one of them being the master. The NMs periodically inform the RM of their status through a heartbeat mechanism for scalability. The RM also maintains the resource requests of all applications. Given its global view of the cluster, and

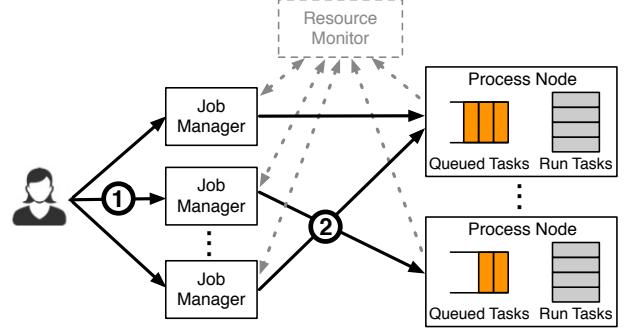


Figure 14: Apollo architecture.

based on application demand, resource availability, scheduling priorities, and sharing policies (e.g., fairness), the scheduler of the RM performs the matchmaking between application requests and machines, and hands leases on *containers* to applications. A container is a logical resource bundle (e.g., 2GB RAM, 1 CPU) bound to a specific node.

YARN includes two widely adopted scheduler implementations, namely the Fair and Capacity Schedulers. The former imposes fairness between applications, while the latter dedicates a share of the cluster resources to groups of users. When jobs are submitted to the RM, they go through an admission control phase, during which security credentials are validated and various operational and administrative checks are performed.

Application Master (AM). The AM is the job orchestrator (one AM is instantiated per submitted job), managing all its life-cycle aspects, including dynamically increasing and decreasing resource consumption, managing the execution flow (e.g., running reducers against the output of mappers), and handling faults. The AM can run arbitrary user code, written in any programming language. By delegating all these functions to AMs, YARN’s architecture achieves significant scalability, programming model flexibility, and improved upgrading/testing.

An AM will typically need to harness resources from multiple nodes to complete a job. To obtain containers, the AM issues resource requests to the RM via heartbeats. When the scheduler assigns a resource to the AM, the RM generates a lease for that resource. The AM is then notified and presents the container lease to the NM for launching the container at that node. The NM checks the authenticity of the lease and then initiates the container execution.

A.2 Apollo

Apollo is our legacy system that was used before *Hydra* to manage the resources of our big-data clusters. Unlike YARN, Apollo adopts a distributed scheduling architecture, in which the scheduling of each job is performed independently. As discussed in §2 and §6, this architecture enables

¹⁰Excerpts of the text below are borrowed from previous works [19, 27, 7].

Apollo to meet the scalability demands of our production workload, but does not allow for arbitrary applications to share the cluster nor for tight control over sharing policies. Apollo’s main components are depicted in Fig. 14 and are summarized below.

Process Node (PN). A PN process running on each worker node is responsible for managing the local resources on that node and performing local scheduling, similar to YARN’s NM described above.

Resource Monitor (RMon). The RMon periodically aggregates load information from PNs across the cluster, creating a global view of the cluster status. While treated as a single logical entity, the RMon is implemented physically in a master-slave configuration for scalability and high availability purposes. In contrast to YARN’s RM, Apollo’s RMon is not at the critical path of scheduling decisions and might contain stale information about the PN load.

Job Manager (JM). A Job Manager (JM), also called a scheduler, is instantiated for each job to manage the job’s

life-cycle. The JM relies on the global cluster load information provided by the RMon in order to perform informed scheduling decisions.

To better predict resource utilization in the near future and to optimize scheduling quality, each PN maintains a local queue of tasks assigned to the node and advertises its future resource availability to the RMon in the form of a wait-time matrix. Apollo thereby adopts an estimation-based approach to making task scheduling decisions. Each scheduler consults the cluster status from the RMon, together with the individual characteristics of tasks to be scheduled, such as the data locality. However, cluster dynamics pose many challenges in practice. RMon’s information might be stale, estimates might be suboptimal, and the cluster environment might be unpredictable. Apollo therefore incorporates correction mechanisms for robustness and dynamically adjusts scheduling decisions at runtime. Finally, it employs opportunistic scheduling to increase resource utilization while providing guaranteed resources to jobs that need it.

Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure

Qifan Pu (UC Berkeley), Shivaram Venkataraman (UW Madison), Ion Stoica (UC Berkeley)

Abstract

Serverless computing is poised to fulfill the long-held promise of transparent elasticity and millisecond-level pricing. To achieve this goal, service providers impose a fine-grained computational model where every function has a maximum duration, a fixed amount of memory and no persistent local storage. We observe that the fine-grained elasticity of serverless is key to achieve high utilization for general computations such as analytics workloads, but that resource limits make it challenging to implement such applications as they need to move large amounts of data between functions that don't overlap in time. In this paper, we present Locus, a serverless analytics system that judiciously combines (1) cheap but slow storage with (2) fast but expensive storage, to achieve good performance while remaining cost-efficient. Locus applies a performance model to guide users in selecting the type and the amount of storage to achieve the desired cost-performance trade-off. We evaluate Locus on a number of analytics applications including TPC-DS, CloudSort, Big Data Benchmark and show that Locus can navigate the cost-performance trade-off, leading to $4\times$ - $500\times$ performance improvements over slow storage-only baseline and reducing resource usage by up to 59% while achieving comparable performance with running Apache Spark on a cluster of virtual machines, and within $2\times$ slower compared to Redshift.

1 Introduction

The past decade has seen the widespread adoption of cloud computing infrastructure where users launch virtual machines on demand to deploy services on a provisioned cluster. As cloud computing continues to evolve towards more elasticity, there is a shift to using *serverless* computing, where storage and compute is separated for both resource provisioning and billing. This trend was started by services like Google BigQuery [9], and AWS Glue [22] that provide cluster-free data warehouse analytics, followed by services like Amazon Athena[5] that allow users to perform interactive queries against a remote object storage without provisioning a compute cluster. While the aforementioned services mostly focus on providing SQL-like analytics, to meet the growing demand, all major cloud providers now offer “general” serverless computing platforms, such as AWS Lambda, Google Cloud Functions, Azure Functions and IBM OpenWhisk. In these platforms short-lived user-defined functions are scheduled and executed in the cloud. Compared to virtual machines, this model provides more fine-grained elasticity with sub-second start-up times, so that

workload requirements can be dynamically matched with continuous scaling.

Fine-grained elasticity in serverless platforms is naturally useful for on-demand applications like creating image thumbnails [18] or processing streaming events [26]. However, we observe such elasticity also plays an important role for data analytics workloads. Consider for example an ad-hoc data analysis job exemplified by say TPC-DS query 95 [34] (See section 5 for more details). This query consists of eight stages and the amount of input data at each stage varies from 0.8MB to 66GB. With a cluster of virtual machines users would need to size the cluster to handle the largest stage leaving resources idle during other stages. Using a serverless platform can improve resource utilization as resources can be immediately released after use.

However, directly using a serverless platform for data analytics workloads could lead to extremely inefficient execution. For example we find that running the CloudSort benchmark [40] with 100TB of data on AWS Lambda, can be up to $500\times$ slower (Section 2.3) when compared to running on a cluster of VMs. By breaking down the overheads we find that the main reason for the slowdown comes from slow data shuffle between asynchronous function invocations. As the ephemeral, stateless compute units lack any local storage, and as direct transfers between functions is not always feasible¹, intermediate data between stages needs to be persisted on shared storage systems like Amazon S3. The characteristics of the storage medium can have a significant impact on performance and cost. For example, a shuffle from 1000 map tasks to 1000 reduce tasks leads to 1M data blocks being created on the storage system. Therefore, throughput limits of object stores like Amazon S3 can lead to significant slow downs (Section 2.3).

Our key observation is that in addition to using elastic compute and object storage systems we can also provision *fast* memory-based resources in the cloud, such as in-memory Redis or Memcached clusters. While naively putting all data in fast storage is cost prohibitive, we can appropriately combine fast, but expensive storage with slower but cheaper storage, similar to the memory and disk hierarchy on a local machine, to achieve the best of both worlds: approach the performance of a pure in-memory execution at a significantly lower cost. However, achieving such a sweet spot is not trivial as it depends on a variety of configuration parameters, including storage type and size, degree of task parallelism, and the memory size of each serverless func-

¹Cloud providers typically provide no guarantees on concurrent execution of workers.

tion. This is further exacerbated by the various performance limits imposed in a serverless environment (Section 2.4).

In this paper we propose Locus, a serverless analytics system that combines multiple storage types to achieve better performance and resource efficiency. In Locus, we build a performance model to aid users in selecting the appropriate storage mechanism, as well as the amount of fast storage and parallelism to use for map-reduce like jobs in serverless environments. Our model captures the performance and cost metrics of various cloud storage systems and we show how we can combine different storage systems to construct hybrid shuffle methods. Using simple micro-benchmarks, we model the performance variations of storage systems as other variables like serverless function memory and parallelism change.

We evaluate Locus on a number of analytics applications including TPC-DS, Daytona CloudSort and the Big Data Benchmark. We show that using fine-grained elasticity, Locus can reduce cluster time in terms of total core-seconds by up to 59% while being close to or beating Spark’s query completion time by up to 2 \times . We also show that with a small amount of fast storage, for example, with fast storage just large enough to hold 5% of total shuffle data, Locus matches Apache Spark in running time on CloudSort benchmark and is within 13% of the cost of the winning entry in 2016. While we find Locus to be 2 \times slower when compared to Amazon Redshift, Locus is still a preferable choice to Redshift since it requires no provisioning time (vs. minutes to setup a Redshift cluster) or knowing an optimal cluster size beforehand. Finally, we also show that our model is able to accurately predict shuffle performance and cost with an average error of 15.9% and 14.8%, respectively, which allows Locus to choose the most appropriate shuffle implementation and other configuration variables.

In summary, the main contributions of this paper are:

- We study the problem of executing general purpose data analytics on serverless platforms to exploit fine-grained elasticity and identify the need for efficient shuffles.
- We show how using a small amount of memory-based fast storage can lead to significant benefits in performance while remaining cost effective.
- To aid users in selecting the appropriate storage mechanism, We propose Locus, a performance model that captures the performance and cost metrics of shuffle operations.
- Using extensive evaluation on TPC-DS, CloudSort and Big Data Benchmark we show that our performance model is accurate and can lead to 4 \times -500 \times performance improvements over baseline and up to 59% cost reduction compared to traditional VM deployments, and within 2 \times slower compared to Redshift.

2 Background

We first present a brief overview of serverless computing and compare it with the traditional VM-based instances. Next we discuss how analytics queries are implemented on serverless infrastructure and present some of the challenges in executing large scale shuffles.

2.1 Serverless Computing: What fits?

Recently, cloud providers and open source projects [25, 32] have proposed services that execute *functions* in the cloud or providing Functions-as-a-Service. As of now, these functions are subject to stringent resource limits. For example, AWS Lambda currently imposes a 5 minute limit on function duration and 3GB memory limit. Functions are also assumed to be *stateless* and are only allocated 512MB of ephemeral storage. Similar limits are applied by other providers such as Google Cloud Functions and Azure Functions. Regardless of such limitations, these offerings are popular among users for two main reasons: ease of deployment and flexible resource allocation. When deploying a cluster of virtual machines, users need to choose the instance type, number of instances, and make sure these instances are shutdown when the computation finishes. In contrast, serverless offerings have a much simpler deployment model where the functions are automatically triggered based on events, e.g., arrival of new data.

Furthermore, due to their lightweight nature, containers used for serverless deployment can often be launched within seconds and thus are easier to scale up or scale down when compared to VMs. The benefits of elasticity are especially pronounced for workloads where the number of cores required varies across time. While this naturally happens for event-driven workloads for example where say users upload a photo to a service that needs to be compressed and stored, we find that elasticity is also important for *data analytics* workloads. In particular, user-facing ad-hoc queries or exploratory analytics workloads are often unpredictable yet have more stringent responsiveness requirements, making it more difficult to provision a traditional cluster compared to recurring production workloads.

We present two common scenarios that highlight the importance of elasticity. First, consider a stage of tasks being run as a part of an analytics workload. As most frameworks use a BSP model [15, 44] the stage completes only when the last task completes. As the same VMs are used across stages, the cores where tasks have finished are idle while the slowest tasks or stragglers complete [3]. In comparison, with a serverless model, the cores are immediately relinquished when a task completes. This shows the importance of elasticity *within* a stage. Second, elasticity is also important *across* stages: if we consider say consider TPC-DS query 95 (details in 5), the query consists of 8 stages with input data per stage

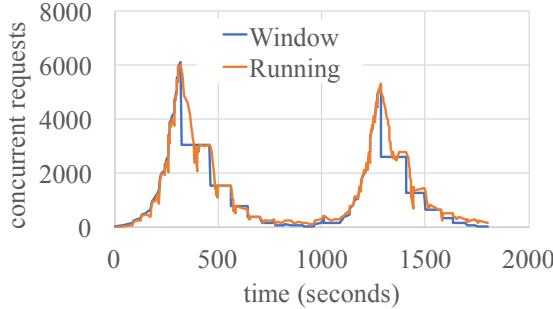


Figure 1: S3 rate limiting in action. We use a TCP-like additive-increase/multiplicative-decrease (AIMD) algorithm to probe the number of concurrent requests S3 can support for reading 10KB objects. We see that S3 not only enforces a rate ceiling, but also continues to fail requests after the rate is reduced for a period of time. The specific rate ceiling can change over time due to S3’s automatic data-partition scaling.

varying from 0.8Mb to 66Gb. With such a large variance in data size, being able to adjust the number of cores used at every stage leads to better utilization compared to traditional VM model.

2.2 Analytics on serverless: Challenges

To execute analytics queries on a serverless infrastructure we assume the following system model. A driver process, running on user’s machine, “compiles” the query into a multi-stage DAG, and then submits each task to the cloud service provider. A task is executed as one function invocation by the serverless infrastructure. Tasks in consecutive stages exchange data via a variety of communication primitives, such as shuffle and broadcast [11]. Each task typically consists of three phases: read, compute, and write [33]. We next discuss why the communication between stages i.e., the shuffle stage presents the biggest challenge.

Input, Output: Similar to existing frameworks, each task running as a function on a serverless infrastructure reads the input from a shared storage system, such as S3. However, unlike existing frameworks, functions are not co-located with the storage, hence there is no data *locality* in this model. Fortunately, as prior work has shown, the bandwidth available between functions and the shared storage system is comparable to the disk bandwidths [1], and thus we typically do not see any significant performance degradation in this step.

Compute: With serverless computing platforms, each function invocation is put on a new container with a virtualized compute core. Regardless of the hardware heterogeneity, recent works have shown that the almost linear scaling of serverless compute is ideal for supporting embarrassingly parallel workloads [16, 18].

Shuffle: The most commonly used communication pattern to transfer data across stages is the shuffle operation. The map stage partitions data according to the number of reducers and each reducer reads the corresponding data partitions from the all the mappers. Given M mappers and R reduc-

ers we will have $M * R$ intermediate data partitions. Unfortunately, the time and resource limitations imposed by the serverless infrastructures make the implementation of the shuffle operation highly challenging.

A direct approach to implementing shuffles would be to open connections between serverless workers [18] and transfer data directly between them. However, there are two limitations that prevent this approach. First cloud providers do not provide any guarantees on when functions are executed and hence the sender and receiver workers might not be executing at the same time. Second, even if the sender and receiver overlap, given the execution time limit, there might not be enough time to transfer all the necessary data.

A natural approach to transferring data between ephemeral workers is to store intermediate data in a persistent storage system. We illustrate challenges for this approach with a distributed sorting example.

2.3 Scaling Shuffle: CloudSort Example

The main challenge in executing shuffles in a serverless environment is handling the large number of intermediate files being generated. As discussed before, functions have stringent resource limitations and this effectively limits the amount of data a function can process in one task. For example to sort 100TB, we will need to create a large number of map partitions, as well as a large number of reduce partitions, such that the inputs to the tasks can be less than the memory footprint of a function. Assuming 1GB partitions, we have 10^5 partitions on both the map side and the reduce side. For implementing a hash-based shuffle one intermediate file is created for each (mapper, reducer) pair. In this case we will have a total of 10^{10} , or 10 billion intermediate files! Even with traditional cluster-based deployment, shuffling 10 billion files is quite challenging, as it requires careful optimization to achieve high network utilization [31]. Unfortunately, none of the storage systems offered by existing cloud providers meets the performance requirements, while also being cost-effective. We next survey two widely available storage systems classes and discuss their characteristics.

2.4 Cloud Storage Systems Comparison

To support the diverse set of cloud applications, cloud providers offer a number of storage systems each with different characteristics in terms of latency, throughput, storage capacity and elasticity. Just as within a single machine, where we have a storage hierarchy of cache, memory and disk, each with different performance and cost points, we observe that a similar hierarchy can be applied to cloud storage systems. We next categorize two major storage system classes.

Slow Storage: All the popular cloud providers offer support for scalable and elastic blob storage. Examples of such

systems include Amazon S3, Google Cloud Storage, Azure Blob Store. However, these storage systems are not designed to support high throughput on reading and writing small files. In fact, all major public cloud providers impose a global transaction limit on shared object stores [37, 7, 20]. This should come as no surprise, as starting with the Google File System [21], the majority of large scale storage systems have been optimized for reading and writing large chunks of data, rather than for high-throughput fine-grained operations.

We investigated the maximum throughput that one can achieve on Amazon S3 and found that though the throughput can be improved as the number of buckets increases, the cloud provider throttles requests when the aggregate throughput reaches a few thousands of requests/sec (see Figure 1). Assuming a throughput of 10K operations per second, this means that reading and writing all the files generated by our CloudSort example could take around 2M seconds, or $500\times$ slower than the current record [42]. Not only is the performance very low, but the cost is prohibitive as well. While the cost per write request is as low as \$0.005 per 1,000 requests for all three aforementioned cloud providers, shuffling 10^{10} files would cost \$5,000 alone for write requests. Thus, supporting large shuffles requires a more efficient and economic solution for storing intermediate data.

Fast Storage: One approach to overcome the performance limitations of the slow storage systems is to use much faster storage, if available. Examples of faster storage are in-memory storage systems backed by Memcached or Redis. Such storage systems support much higher request rates (more than 100,000 requests/sec per shard), and efficiently handle objects as small as a few tens of bytes. On the flip side, these systems are typically much more expensive than large-scale blob storage systems. For example to store 1GB of data for an hour, it costs 0.00319 cents in AWS S3 while it costs 2.344 cents if we use a managed Redis service such as AWS ElastiCache, which makes it $733\times$ more expensive!²

Given the cost-performance trade-off between slow (e.g., S3) and fast (e.g., ElastiCache) storage, in the following sections we show that by judiciously combining these two types of storage systems, we can achieve a cost-performance sweet spot in a serverless deployment that is comparable, and sometimes superior to cluster-based deployments.

3 Design

In this section we outline a performance model that can be used to guide the design of an efficient and cost-effective shuffle operations. We start with outlining our system model,

²We note that ElastiCache is not “serverless”, and there is no serverless cache service yet as of writing this paper and users need to provision cache instances. However, we envision that similar to existing storage and compute, fast storage as a resource (possibly backed by memory) will also become elastic in the future. There are already several proposals to provide disaggregated memory across datacenters [19] to support this.

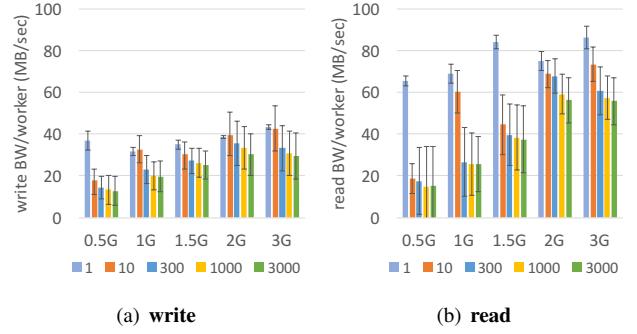


Figure 2: S3 bandwidth per worker with varying concurrency (1 to 3000) and Lambda worker size (0.5G to 3G).

Table 1: Measured throughput (requests/sec) limit for a single S3 bucket and a single Redis shard.

object size	10KB	100KB	1M	10M	100M
S3	5986	4400	3210	1729	1105
Redis	116181	11923	1201	120	12

and then discuss how different variables like worker memory size, degree of parallelism, and the type of storage system affect the performance characteristics of the shuffle operation.

3.1 System Model

We first develop a high level system model that can be used to compare different approaches to shuffle and abstract away details specific to cloud providers. We denote the function-as-a-service module as *compute cores* or *workers* for tasks. Each function invocation, or a worker, is denoted to run with a single core and w bytes of memory (or the worker memory size). The degree of the parallelism represents the number of function invocations or workers that execute in parallel, which we denote by p . The total amount of data being shuffled is S bytes. Thus, the number of workers required in the map and reduce phase is at least $\frac{S}{w}$ leading to a total of $(\frac{S}{w})^2$ requests for a full shuffle.

We next denote the bandwidth available to access a storage service by an individual worker as b bytes/sec. We assume that the bandwidth provided by the elastic storage services scale as we add more workers (we discuss how to handle cases where this is not true below). Finally, we assume each storage service limits the aggregate number of requests/sec: we denote by q_s and q_f for the slow and the fast storage systems, respectively.

To measure the cost of each approach we denote the cost of a worker function as c_l \$/sec/byte, the cost of fast storage as c_f \$/sec/byte. The cost of slow storage has two parts, one for storage as c_s \$/sec/byte, and one for access, denoted as c_a \$/op. We assume that both the inputs and the outputs of the shuffle are stored on the slow storage. In most cases in practice, c_s is negligible during execution of a job. We find the

Table 2: Cloud storage cost from major providers (Feb 2019).

	Service	\$/Mo/GB	\$/million writes
Slow	AWS S3	0.023	5
	GCS	0.026	5
	Azure Blob	0.023	6.25
Fast	ElastiCache	7.9	-
	Memorystore	16.5	-
	Azure Cache	11.6	-

above cost characteristics apply to all major cloud platforms (AWS, Google Cloud and Azure), as shown in Table 2.

Among the above, we assume the shuffle size (S) is given as an input to the model, while the worker memory size (w), the degree of parallelism (p), and the amount of fast storage (r) are the model knobs we vary. To determine the characteristics of the storage systems (e.g., b , q_s , q_f), we use offline benchmarking. We first discuss how these storage performance characteristics vary as a function of our variables.

3.2 Storage Characteristics

The main storage characteristics that affect performance are unsurprisingly the read and write **throughput** (in terms of requests/sec, or often referred as IOPS) and **bandwidth** (in terms of bytes/sec). However, we find that these values are not stable as we change the degree of parallelism and worker memory size. In Figure 2 we measure how a function’s bandwidth (b) to a large-scale store (i.e., Amazon S3, the slow storage service in our case) varies as we change the degree of parallelism (p) and the worker memory size (w). From the figure we can see that as we increase the parallelism both read and write bandwidths could vary by 2-3 \times . Further we see that as we increase the worker memory size the bandwidth available increases but that the increase is sub-linear. For example with 60 workers each having 0.5G of memory, the write bandwidth is around 18 MB/s per worker or 1080 MB/s in aggregate. If we instead use 10 workers each having 3GB of memory, the write bandwidth is only around 40 MB/s per worker leading to 400 MB/s in aggregate.

Using a large number of small workers is not always ideal as it could lead to an increase in the number of small I/O requests. Table 1 shows the throughput we get as we vary the object size. As expected, we see that using smaller object sizes means that we get a lower aggregate bandwidth (multiplying object size by transaction throughput). Thus, jointly managing worker memory size and parallelism poses a challenging trade-off.

For fast storage systems we typically find that throughput is not a bottleneck for object sizes > 10 KB and that we saturate the storage bandwidth. Hence, as shown in Table 1 the operation throughput decreases linearly as the object size increases. While we can estimate the bandwidth available for fast storage systems using an approach similar to the one used for slow storage systems, the current deployment

Table 3: Comparison of time taken by different shuffle methods. S refers to the shuffle data size, w to the worker memory size, p the number of workers, q_s the throughput to slow storage, q_f throughput to fast storage b network bandwidth from each worker.

storage type	shuffle time
slow	$2 \times \max\left(\frac{S^2}{w^2 \times q_s}, \frac{S}{b \times p}\right)$
fast	$2 \times \max\left(\frac{S^2}{w^2 \times q_f}, \frac{S}{b_{eff}}\right)$, where $b_{eff} = \min(b_f, b \times p)$
hybrid	$\frac{S}{r} T_{rnd} + T_{mrg}$, where $T_{rnd} = 2 \times \max(T_{fb}, T_{sb}, T_{sq})$ $T_{mrg} = 2 \times \max\left(\left(\frac{Sw}{r}\right)^2 T_{sq}, \frac{S}{r} T_{sb}\right)$ $T_{fb} = \frac{r}{b_{eff}}$, $T_{sb} = \frac{r}{b \times p}$ $T_{sq} = \frac{r^2}{w^2 \times q_s}$

method where we are allocating servers for running Memcached / Redis allows us to ensure they are not a bottleneck.

3.3 Shuffle Cost Models

We next outline performance models for three shuffle scenarios: using (1) slow storage only, (2) fast storage only, and (3) a combination of fast and slow storage.

Slow storage based shuffle. The first model we develop is using slow storage only to perform the shuffle operation. As we discussed in the previous section there are two limits that the slow storage systems impose: an operation throughput limit (q_s) and a bandwidth limit (b). Given that we need to perform $(\frac{S}{w})^2$ requests with an overall operation throughput of q_s , we can derive T_q , the time it takes to complete these requests is $T_q = \frac{S^2}{w^2 \times q_s}$, assuming q_s is the bottleneck. Similarly, given the per-worker bandwidth limit to storage, b , the time to complete all requests assuming b is bottleneck is $T_b = \frac{S}{b \times p}$. Considering both potential bottlenecks, the time it takes to write/read all the data to/from intermediate storage is thus $\max(T_q, T_b)$. Note that this time already includes reading data from input storage or writing data to output storage, since they can be pipelined with reading/writing to intermediate storage. Finally, the shuffle needs to first write data to storage and then read it back. Hence the total shuffle time is $T_{shuf} = 2 \times \max(T_q, T_b)$.

Table 4 shows our estimated running time and cost as we vary the worker memory and data size.

Fast storage based-shuffle. Here we develop a simple performance model for fast storage that incorporates the throughput and bandwidth limits. In practice we need to make one modification to factor in today’s deployment model for fast storage systems. Since services like ElastiCache are deployed by choosing a fixed number of instances, each having some fixed amount of memory, the aggregate bandwidth of the fast storage system could be a significant bottleneck, if we are not careful. For example, if we had

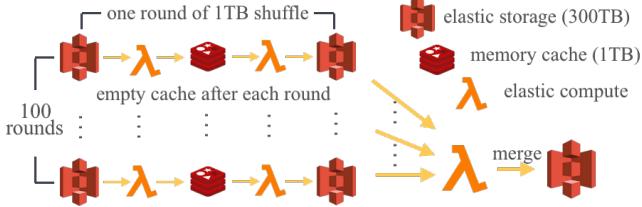


Figure 3: Illustration for hybrid shuffle.

Table 4: Projected sort time and cost with varying worker memory size. Smaller worker memory results in higher parallelism, but also a larger number of files to shuffle.

worker mem(GB)	0.25	0.5	1	1.25	1.5
20GB time(s)	36	45	50	63	72
20GB cost(\$)	0.02	0.03	0.03	0.04	0.05
200GB time(s)	305	92	50	63	75
200GB cost(\$)	0.24	0.30	0.33	0.42	0.51

just one ElastiCache instance with 10Gbps NIC and 50G of memory, the aggregate bandwidth is trivially limited to 10Gbps. In order to model this aspect, we extend our formulation to include b_f , which is the server-side bandwidth limit for fast storage. We calculate the effective bandwidth as $b_{eff} = \min(b \times p, b_f)$.

Using the above effective bandwidth we can derive the time taken due to throughput and bandwidth limits as $T_q = \frac{s^2}{w^2 \times q_f}$ and $T_b = \frac{s}{b_{eff}}$, respectively. Similar to the previous scenario, the total shuffle time is then $T_{shuf} = 2 \times \max(T_q, T_b)$.

One interesting scenario in this case is that as long as the fast storage bandwidth is a bottleneck (i.e. $b_f < b \times p$), using more fast memory improves not only the performance, but also reduces the cost! Assume the amount of fast storage is r . This translates to a cost of $p * c_l * T_{shuf} + r * c_f * T_{shuf}$, with slow storage request cost excluded. Now, assume we double the memory capacity to $2 \times r$, which will also result in doubling the bandwidth, i.e., $2 \times b_f$. Assuming that operation throughput is not the bottleneck, the shuffle operations takes now $\frac{s}{2b_f} = \frac{T_{shuf}}{2}$, while the cost becomes $p * c_l * \frac{T_{shuf}}{2} + 2 * r * c_f * \frac{T_{shuf}}{2}$. This does not include reduction in request cost for slow storage. Thus, while the cost for fast storage (second term) remains constant, the cost for compute cores drops by a factor of 2. In other words, the overall running time has improved by a factor of 2 while the cost has decreased.

However, as the amount of shuffle data grows, the cost of storing all the intermediate data in fast storage becomes prohibitive. We next look at the design of a hybrid shuffle method that can scale to much larger data sizes.

3.4 Hybrid Shuffle

We propose a hybrid shuffle method that combines the inexpensive slow storage with the high throughput of fast storage

to reach a better cost-performance trade-off. We find that even with a small fast storage, e.g., less than $\frac{1}{20}$ th of total shuffle data, our hybrid shuffle can outperform slow storage based shuffle by orders of magnitude.

To do that, we introduce a multi-round shuffle that uses fast storage for intermediate data within a round, and uses slow storage to merge intermediate data across rounds. In each round we range-partition the data into a number of buckets in fast storage and then combine the partitioned ranges using the slow storage. We reuse the same range partitioner across rounds. In this way, we can use a merge stage at the end to combine results across all rounds, as illustrated in Figure 3. For example, a 100 TB sort can be broken down to 100 rounds of 1TB sort, or 10 rounds of 10TB sort.

Correspondingly the cost model for the hybrid shuffle can be broken down into two parts: the cost per round and the cost for the merge. The size of each round is fixed at r , the amount of space available on fast storage. In each round we perform two stages of computation, partition and combine. In the partition stage, we read input data from the slow storage and write to the fast storage, while in the combine stage we read from the fast storage and write to the slow storage. The time taken by one stage is then the maximum between the corresponding durations of the stage when the bottleneck is driven either by (1) the fast storage bandwidth $T_{fb} = \frac{r}{b_{eff}}$, (2) the slow storage bandwidth $T_{sb} = r/(b * p)$, or (3) the slow storage operation throughput $T_{sq} = \frac{r^2}{w^2 \times q_s}$ ³. Thus, the time per-round is $T_{rnd} = 2 * \max(T_{fb}, T_{sb}, T_{sq})$.

The overall shuffle consists of $\frac{S}{r}$ such rounds and a final merge phase where we read data from the slow storage, merge it, and write it back to the slow storage. The time of the merge phase can be similarly broken down into throughput limit $T_{mq} = (\frac{Sw}{r})^2 * T_{sq}$ and bandwidth limit $T_{mb} = \frac{S}{r} * T_{sb}$, where T_{sb} and T_{sq} follows from the definitions from previous paragraph. Thus, $T_{mrg} = 2 * \max(T_{mq}, T_{mb})$, and the total shuffle time is $\frac{S}{r} * T_{rnd} + T_{mrg}$.

How to pick the right fast storage size? Selecting the appropriate fast storage/memory size is crucial to obtaining good performance with the hybrid shuffle. Our performance model aims to determine the optimal memory size by using two limits to guide the search. First, provisioning fast storage does not help when slow storage bandwidth becomes bottleneck, which provides an upper bound on fast storage size. Second, since the final stage needs to read outputs from all prior rounds to perform the merge, the operation throughput of the slow storage provides an upper bound on the number of rounds, thus a lower bound of the fast storage size.

Pipelining across stages An additional optimization we perform to speed up round execution and reduce cost is to pipeline across partition stage and combine stage. As shown in Figure 3, for each round, we launch partition tasks to read

³We ignore the fast storage throughput, as we rarely find it to be bottleneck. We could easily include it in our model, if needed.

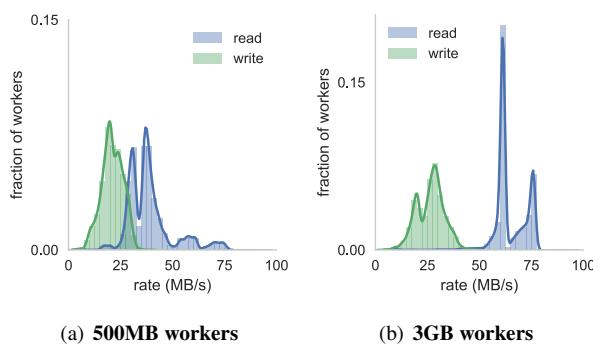


Figure 4: **Lambda to S3 bandwidth distribution exhibits high variance. A major source of stragglers.**

input data, partition them and write out intermediate files to the fast storage. Next, we launch combine tasks that read files from the fast storage. After each round, the fast storage can be cleared to be used for next round.

With pipelining, we can have partition tasks and combine tasks running in parallel. While the partition tasks are writing to fast storage via `append()`, the merge tasks read out files periodically and perform atomic delete-after-read operations to free space. Most modern key-value stores, e.g., Redis, support operations such as append and atomic delete-after-read. Pipelining gives two benefits: (1) it overlaps the execution of the two phases thus speeding up the in-round sort, and (2) it allows a larger round size without needing to store the entire round in memory. Pipelining does have a drawback. Since we now remove synchronization boundary between rounds, and use `append()` instead of setting a new key for each intermediate data, we cannot apply speculative execution to mitigate stragglers, nor can we obtain task-level fault tolerance. Therefore, pipelining is more suitable for smaller shuffles.

3.5 Modeling Stragglers

The prior sections provided several basic models to estimate the time taken by a shuffle operation in a serverless environment. However, these basic models assume all tasks have uniform performance, thus failing to account for the presence of stragglers.

The main source of stragglers for the shuffle tasks we consider in this paper are network stragglers, that are caused by slow I/O to object store. Network stragglers are inherent given the aggressive storage sharing implied by the serverless architecture. While some containers (workers) might get better bandwidth than running reserved instances, some containers get between 4-8× lower bandwidth, as shown in Figure 4. To model the straggler mitigation scheme described above we initialize our model with the network bandwidth CDFs as shown in Figure 4. To determine running time of each stage we then use an execution simulator [33] and sample network bandwidths for each container from the CDFs.

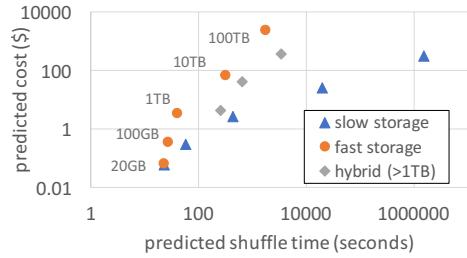


Figure 5: **Predicted time and cost for different sort implementations and sizes.**

Furthermore, our modeling is done for each worker memory size, since bandwidth CDFs vary across worker sizes.

There are many previous works on straggler mitigation [45, 4, 36, 2]. We use a simple online method where we always launch speculative copies after $x\%$ of tasks finish in the last wave. Having short-lived tasks in the serverless model is more advantageous here. The natural elasticity of serverless infrastructure makes it possible to be aggressive in launching speculative copies.

3.6 Performance Model Case Study

We next apply our performance model described above to the CloudSort benchmark and study the cost-performance trade-off for the three approaches described above. Our predictions for data sizes ranging from 20GB to 100TB are shown in Figure 5 (we use experimental results of a real prototype to validate these predictions in Section 5). When the data shuffle size is small (e.g., 20GB or smaller), both the slow and fast storage only solutions take roughly the same time, with the slow storage being slightly cheaper. As the data size increases to around 100GB, using fast storage is around 2× faster for the same cost. This speed up from fast storage is more pronounced as data size grows. For very large shuffles (≥ 10 TB), hybrid shuffle can provide significant cost savings. For example, at 100TB, the hybrid shuffle is around 6x cheaper than the fast storage only shuffle, but only 2x slower.

Note that since the hybrid shuffle performs a merge phase in addition to writing all the data to the fast storage, it is always slower than the fast storage only shuffle. In summary, this example shows how our performance model can be used to understand the cost-performance trade-off from using different shuffle implementations. We implement this performance modeling framework in Locus to perform automatic shuffle optimization. We next describe the implementation of Locus and discuss some extensions to our model.

4 Implementation

We implement Locus by extending PyWren [16], a Python-based data analytics engine developed for serverless environments. PyWren allows users to implement custom functions that perform data shuffles with other cloud services, but it lacks an actual shuffle operator. We augment PyWren

with support for shuffle operations and implement the performance modeling framework described before to automatically configure the shuffle variables. For our implementation we use AWS Lambda as our compute engine and use S3 as the slow, elastic storage system. For fast storage we provision Redis nodes on Amazon ElastiCache.

To execute SQL queries on Locus, we devise physical query plan from Apache Spark and then use Pandas to implement structured data operations. One downside with Pandas is that we cannot do “fine-grained pipelining” between data operations inside a task. Whereas in Apache Spark or Redshift, a task can process records as they are read in or written out. Note this fine-grained pipelining is different from pipelining across stages, which we discuss in Section 3.4.

4.1 Model extensions

We next discuss a number of extensions to augment the performance model described in the previous section

Non-uniform data access: The shuffle scenario we considered in the previous section was the most general all-to-all shuffle scenario where every mapper contributes data to every reducer. However, a number of big data workloads have more skewed data access patterns. For example, machine learning workloads typically perform AllReduce or broadcast operations that are implemented using a tree-based communication topology. When a binary tree is used to do AllReduce, each mapper only produces data for one reducer and correspondingly each reducer only reads two partitions. Similarly while executing a broadcast join, the smaller table will be accessed by every reducer while the larger table is hash partitioned. Thus, in these scenarios storing the more frequently accessed partition on fast storage will improve performance. To handle these scenarios we introduce an access counter for each shuffle partition and correspondingly update the performance model. We only support this currently for cases like AllReduce and broadcast join where the access pattern is known beforehand.

Storage benchmark updates: Finally one of the key factors that make our performance models accurate is the storage benchmarks that measure throughput (operations per sec) and network bandwidth (bytes per second) of each storage system. We envision that we will execute these benchmarks the first time a user installs Locus and that the benchmark values are reused across a number of queries. However, since the benchmarks are capturing the behavior of cloud storage systems, the performance characteristics could change over time. Such limits change will require Locus to rerun the profiling. We plan to investigate techniques where we can profile query execution to infer whether our benchmarks are still accurate over extended periods of time.

5 Evaluation

We evaluate Locus with a number of analytics workloads, and compare Locus with Apache Spark running on a cluster of VMs and AWS Redshift/Redshift Spectrum⁴. Our evaluation shows that:

- Locus’s serverless model can reduce cluster time by up to 59%, and at the same time being close to or beating Spark’s query completion time by up to 2×. Even with a small amount of fast storage, Locus can greatly improve performance. For example, with just 5% memory, we match Spark in running time on CloudSort benchmark and are within 13% of the cost of the winning entry in 2016.
- When comparing with actual experiment results, our model in Section 3 is able to predict shuffle performance and cost accurately, with an average error of 15.9% for performance and 14.8% for cost. This allows Locus to choose the best cost-effective shuffle implementation and configuration.
- When running data intensive queries on the same number of cores, Locus is within $1.61 \times$ slower compared to Spark, and within $2 \times$ slower compared to Redshift, regardless of the baselines’ more expensive unit-time pricing. Compared to shuffling only through slow storage, Locus can be up to $4 \times$ - $500 \times$ faster.

The section is organized as follows, we first show utilization and end-to-end performance with Locus on TPC-DS [34] queries (5.1) and Daytona CloudSort benchmark (5.2). We then discuss how fast storage shifts resource balance to affect the cost-performance trade-off in Section 5.3. Using the sort benchmark, we also check whether our shuffle formulation in Section 3 can accurately predict cost and performance(5.4). Finally we evaluate Locus’s performance on joins with Big Data Benchmark [8](5.5).

Setup: We run our experiments on AWS Lambda and use Amazon S3 for slow storage. For fast storage, we use a cluster of r4.2xlarge instances (61GB memory, up to 10Gbps network) and run Redis. For our comparisons against Spark, we use the latest version of Apache Spark (2.3.1). For comparison against Redshift, we use the latest version as of 2018 September and ds2.8xlarge instances. To calculate cost for VM-based experiments we pro-rate the hourly cost to a second granularity.⁵ For Redshift, the cost is two parts using AWS pricing model, calculated by the uptime cost of cluster VMs, plus \$5 per TB data scanned.

⁴When reading data of S3, AWS Redshift automatically uses a shared, serverless pool of resource called the Spectrum layer for S3 I/O, ETL and partial aggregation.

⁵This is presenting a lower cost than the minute-granularity used for billing by cloud providers like Amazon, Google.

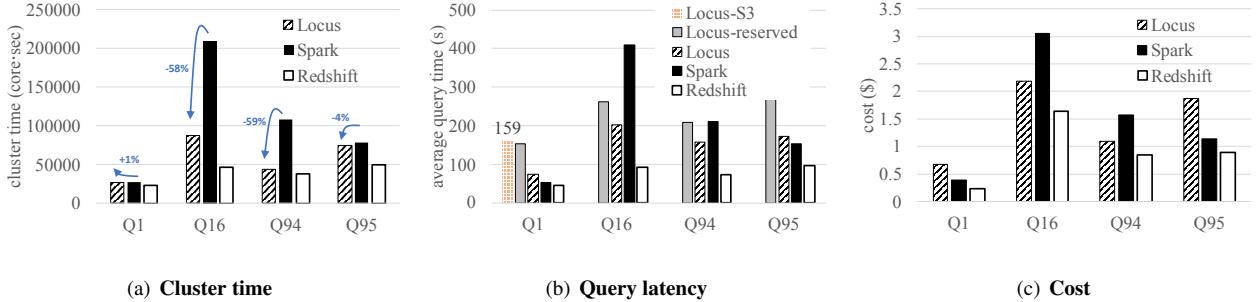


Figure 6: **TPC-DS results for Locus, Apache Spark and Redshift under different configurations.** Locus-S3 runs the benchmark with only S3 and doesn’t complete for many queries; Locus-reserved runs Locus on a cluster of VMs.

5.1 TPC-DS Queries

The TPC-DS benchmark has a set of standard decision support queries based on those used by retail product suppliers. The queries vary in terms of compute and network I/O loads. We evaluate Locus on TPC-DS with scale factor of 1000, which has a total input size of 1TB data for various tables. Among all queries, we pick four of them that represent different performance characteristics and have a varying input data size from 33GB to 312GB. Our baselines are Spark SQL deployed on a EC2 cluster with c3.8xlarge instances and Redshift with ds2.8xlarge instances, both with 512 cores. For Locus, we obtain workers dynamically across different stages of a query, but make sure that we never use more core·secs of Spark execution.

Figure 6(b) shows the query completion time for running TPC-DS queries on Apache Spark, Redshift and Locus under different configurations and Figure 6(a) shows the the total core·secs spent on running those queries. We see that Locus can save cluster time up to 59%, while being close to Spark’s query completion time to also beating it by 2×. Locus loses to Spark on Q1 by 20s. As a result, even for now AWS Lambda’s unit time cost per core is $1.92 \times$ more expensive than the EC2 c3.8xlarge instances, Locus enjoys a lower cost for Q1 and Q4 as we only allocate as many Lambdas as needed. Compared to Redshift, Locus is $1.56 \times$ to $1.99 \times$ slower. There are several causes that might contribute to the cost-performance gap: 1) Redshift has a more efficient execution workflow than that of Locus, which is implemented in Python and has no fine-grained pipelining; 2) ds2.8xlarge are special instances that have 25Gbps aggregate network bandwidths; 3) When processing S3 data, AWS Redshift pools extra resource, referred as the serverless Spectrum layer, to process S3 I/O, ETL and partial aggregation. To validate these hypotheses, we perform two what-if analyses. We first take Locus’s TPC-DS execution trace and replay them to numerically simulate an pipelined execution by overlapping I/O and compute within a task. We find that with pipelining, query latencies can be reduced by 23% to 37%, being much closer to the Redshift numbers. Similarly, using our cost-performance model, we also find that if Lo-

cus’s Redis nodes have 25Gbps links, the cost can be further reduced by 19%, due to a smaller number of nodes needed. Performance will not improve due to 25Gbps links, as network bottleneck on Lambda-side remains. Understanding remaining performance gap would require further breakdown, i.e., porting Locus to a lower-level programming language.

Even with the performance gap, an user may still prefer Locus over a data warehousing service like Redshift since the latter requires on-demand provisioning of a cluster. Currently with Amazon Redshift, provisioning a cluster takes minutes to finish, which is longer than these TPC-DS query latencies. Picking an optimal cluster size for a query is also difficult without knowledge of underlying data.

We also see in Figure 6(b) that Locus provides better performance than running on a cluster of 512-core VMs (Locus-reserved). This demonstrates the power of elasticity in executing analytics queries. Finally, using the fast storage based shuffle in Locus also results in successful execution of 3 queries that could not be executed with slow storage based shuffle, as the case for Locus-S3 or PyWren.

To understand where time is spent, we breakdown execution time into different stages and resources for Q94, as shown in Figure 7. We see that performing compute and network I/O takes up most of the query time. One way to improve overall performance given this breakdown is to do “fine-grained pipelining” of compute and network inside a task. Though nothing fundamental, it is unfortunately difficult to implement with the constraints of Pandas API at the time of writing. Compute time can also be improved if Locus is prototyped using a lower-level language such as C++.

Finally, for shuffle intensive stages such as stage 3 of Q94, we see that linearly scaling up fast storage does linearly improve shuffle performance (Figure 8).

5.2 CloudSort Benchmark

We run the Daytona CloudSort benchmark to compare Locus against both Spark and Redshift on reserved VMs.

The winner entry of CloudSort benchmark which ranks the cost for sorting 100TB data on public cloud is currently held by Apache Spark [42]. The record for sorting 100TB

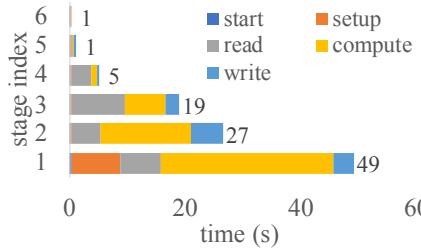


Figure 7: Time breakdown for Q94. Each stage has a different profile and, compute and network time dominate.

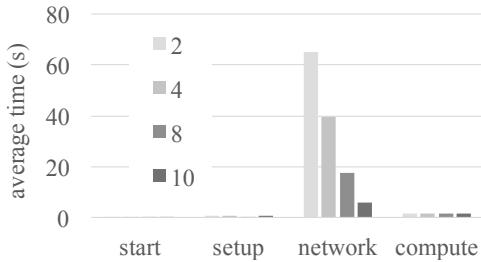


Figure 8: Runtime for stage 3 of Q94 when varying the number of Redis nodes (2, 4, 8, 10).

was achieved in 2983.33s using a cluster of 395 VMs, each with 4 vCPU cores and 8GB memory. The cost of running this was reported as \$144.22. To obtain Spark numbers for 1TB and 10TB sort sizes, we varied the number of i2.8xlarge instances until the sort times matched those obtained by Locus. This allows a fair comparison on the cost. As discussed in Section 3, Locus automatically picks the best shuffle implementation for each input size.

Table 5 shows the result cost and performance comparing Locus against Spark. We see that regardless of the fact Locus’s sort runs on memory-constrained compute infrastructure and communicates through remote storage, we are within 13% of the cost for 100TB record, and achieve the same performance. Locus is even cheaper for 10TB (by 15%) but is 73% more expensive for 1TB. This is due to using fast storage based-shuffle which yields a more costly trade-off point. We discuss more trade-offs in Section 5.3.

Table 6 shows the result of sorting 1TB of random string input. Since Redshift does not support querying against random binary data, we instead generate random string records as the sort input as an approximation to the Daytona Cloud-Sort benchmark. For fair comparison, we also run other systems with the same string dataset. We see that Locus is an order of magnitude faster than Spark and Redshift and is comparable to Spark when input is stored on local disk.

We also run the same Locus code on EC2 VMs, in order to see the cost vs. performance difference of only changing hardware infrastructure while using the same programming language (Python in Locus). Figure 9 shows the results for running 100GB sort. We run Locus on AWS Lambda with various worker memory sizes. Similar to previous section, we then run Locus on a cluster and vary the number

Table 5: CloudSort results vs. Apache Spark

Sort size	1TB	10TB	100TB
Spark nodes	21	60	395[31]
Spark time (s)	40	394	2983
Locus time (s)	39	379	2945
Spark cost (\$)	1.5	34	144
Locus cost (\$)	2.6	29	163

Table 6: 1TB string sort w/ various configurations

	time	cost(\$)
Redshift-S3	6m8s	20.2
Spark RDD-S3	4m27s	15.7
Spark-HDFS (\$)	35s	2.1
Locus (\$)	39s	2.6

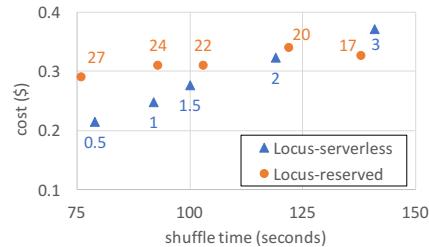


Figure 9: Running 100GB sort with Locus on a serverless infrastructure vs. running the same code on reserved VMs. Labels for serverless series represents the configured memory size of each Lambda worker. Labels for reserved series represents the number of c1.xlarge instances deployed.

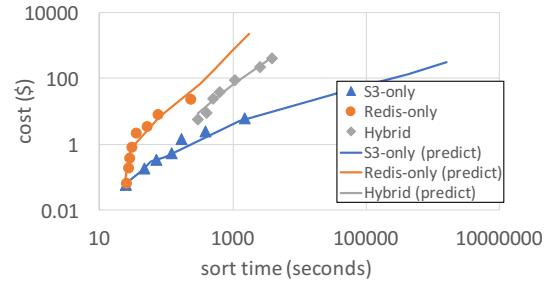


Figure 10: Comparing the cost and performance predicted by Locus against actual measurements. The lines indicate predicted values and the dots indicate measurements.

of c1.xlarge instances to match the performance and compare the cost. We see that both cost and performance improves for Locus-serverless when we pick a smaller memory size. The performance improvement is due to increase in parallelism that results in more aggregate network bandwidth. The cost reduction comes from both shorter run-time and lower cost for small memory sizes. For Locus-reserved, performance improves with more instances while the cost remains relatively constant, as the reduction in run-time compensates for the increased allocation.

We see that even though AWS Lambda is considered to be more expensive in terms of \$ per CPU cycle, it can be cheaper in terms of \$ per Gbps compared to reserved instances. Thus, serverless environments can reach a better

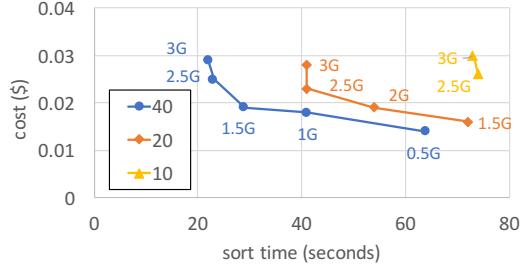


Figure 11: 10GB slow storage-only sort, with varying parallelism (lines) and worker memory size (dots).

cost performance point for network-intensive workloads.

5.3 How much fast storage is needed?

One key insight in formulating the shuffle performance in Locus is that adding more resources does not necessarily increase total cost, e.g., increasing parallelism can result in a better configuration. Another key insight is that using fast storage or memory, sometimes even a small amount, can significantly shift resource balance and improve performance.

We highlight the first effect with an example of increasing parallelism and hence over allocating worker memory compared to the data size being processed. Consider the case where we do a slow storage-only sort for 10GB. Here, we can further increase parallelism by using smaller data partitions than the worker memory size. We find that by say using a parallelism of 40 with 2.5G worker memory size can result in $3.21 \times$ performance improvement and lower cost over using parallelism of 10 with 2.5G worker memory (Figure 11).

However, such performance increase does require that we add resources in a balanced manner as one could also end up incurring more cost while not improving performance. For example, with a 100GB sort (Figure 12), increasing parallelism from 200 to 400 with 2.5G worker memory size (Figure 12) makes performance $2.5 \times$ worse, as now the bottleneck shifts to object store throughput and each worker will run slower due to a even smaller share. Compared to the 10GB sort, this also shows that the same action that helps in one configuration can be harmful in another configuration.

Another way of balancing resources here is to increase parallelism while adding fast storage. We see this in Figure 12, where increasing parallelism to 400 becomes beneficial with fast storage as the storage system can now absorb the increased number of requests. These results provide an example of the kinds of decisions automated by the performance modeling framework in Locus.

The second insight is particularly highlighted for running 100TB hybrid sort. For 100TB sort, we vary the fast storage used from 2% to 5%, and choose parallelism for each setting based on the hybrid shuffle algorithm. As shown in Table 7, we see that even with 2% of memory, the 100TB sort becomes attainable in 2 hours. Increasing memory from 2% to 5%, there is an almost linear reduction in terms of end-

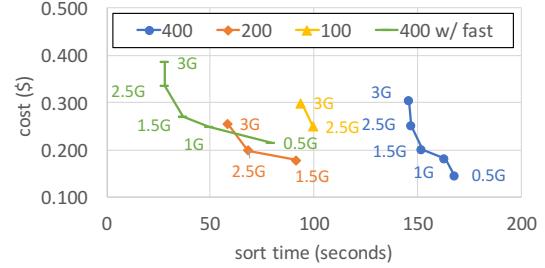


Figure 12: 100GB slow storage-only sort with varying parallelism (different lines) and worker memory size (dots on same line). We include one configuration with fast-storage sort.

Table 7: 100TB Sort with different cache size.

cache	5%	3.3%	2.5%	2%
time (s)	2945	4132	5684	6850
total cost (\$)	163	171	186	179

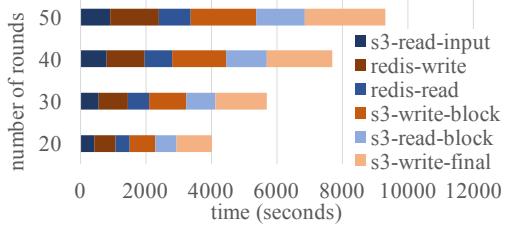


Figure 13: Runtime breakdown for 100TB sort.

to-end sort time when we use larger cache size. This matches the projection in our design discussion. Further broken down in Figure 13, we see that the increase of cost per time unit is compensated by reduction in end-to-end run time.

5.4 Model Accuracy

To automatically choose a cost-effective shuffle implementation, Locus relies on a predictive performance model that can output accurate run-time and cost for any sort size and configuration. To validate our model, we ran an exhaustive experiment with varying sort sizes for all three shuffle implementations and compared the results with the predicted values as shown in Figure 10.

We find that Locus’s model predicts performance and cost trends pretty well, with an average error of 16.9% for run-time and 14.8% for cost. Among different sort implementations, predicting Redis-only is most accurate with an accuracy of 9.6%, then Hybrid-sort of 18.2%, and S3-only sort of 21.5%. This might due to the relatively lesser variance we see in network bandwidth to our dedicated Redis cluster as opposed to S3 which is a globally shared resource. We also notice that our prediction on average under-estimates run-time by 11%. This can be attributed to the fact that we don’t model a number of other overheads such as variance in CPU time, scheduling delay etc. Overall, similar to database query optimizers, we believe that this accuracy is good enough to make coarse grained decisions about shuffle methods to use.

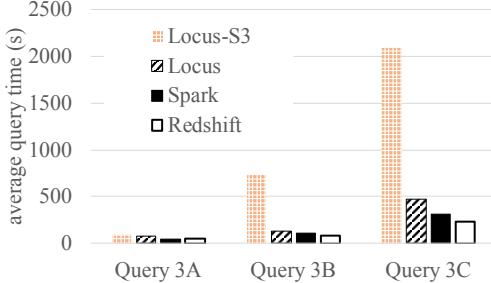


Figure 14: Big Data Benchmark

5.5 Big Data Benchmark

The Big Data Benchmark contains a query suite derived from production databases. We consider Query 3, which is a join query template that reads in 123GB of input and then performs joins of various sizes. We evaluate Locus to see how it performs as join size changes. We configure Locus to use 160 workers, Spark to use 5 c3.xlarge, and Redshift to use 5ds2.8xlarge, all totalling 160 cores. Figure 14 shows that even without the benefit of elasticity, Locus performance is within $1.75 \times$ to Apache Spark and $2.02 \times$ to Redshift across all join sizes. The gap is similar to what we observe in Section 5.1. We also see that using a default slow-storage only configuration can be up to $4 \times$ slower.

6 Related Work

Shuffle Optimizations: As a critical component in almost all data analytics system, shuffle has always been a venue for performance optimization. This is exemplified by Google providing a separate service just for shuffle [23]. While most of its technical details are unknown, the Google Cloud Shuffle service shares the same idea as Locus in that it uses elastic compute resources to perform shuffle externally. Modern analytics systems like Hadoop [39] or Spark [43] often provide multiple communication primitives and sort implementations. Unfortunately, they do not perform well in a serverless setting, as shown previously. There are many conventional wisdom on how to optimize cache performance [24], we explore a similar problem in the cloud context. Our hybrid sort extends on the classic idea of mergesort (see survey [17]) and cache-sensitive external sort [30, 38] to do joint optimization on the cache size and sort algorithm. There are also orthogonal works that focus on the network layer. For example, CoFlow [12] and Varys [13] proposed coordinated flow scheduling algorithms to achieve better last flow completion time. For join operations in databases, Locus relies on existing query compilers to generate shuffle plans. Compiling the optimal join algorithm for a query is an extensively studied area in databases [14], and we plan to integrate our shuffle characteristics with database optimizers in the future.

Serverless Frameworks: The accelerated shift to serverless has brought innovations to SQL processing [9, 5, 22,

35], general computing platforms (OpenLambda [25], AWS Lambda, Google Cloud Functions, Azure Functions, etc.), as well as emerging general computation frameworks [6, 18] in the last two years. These frameworks are architected in different ways: AWS-Lambda [6] provides a schema to compose MapReduce queries with existing AWS services; ExCamera [18] implemented a state machine in serverless tasks to achieve fine-grained control; Prior work [16] has also looked at exploiting the usability aspects to provide a seamless interface for scaling unmodified Python code.

Database Cost Modeling: There has been extensive study in the database literature on building cost-models for systems with multi-tier storage hierarchy [28, 27] and on targeting systems that are bottlenecked on memory access [10]. Our cost modeling shares a similar framework but examines costs in a cloud setting. The idea of dynamically allocating virtual storage resource, especially fast cache for performance improvement can also be found in database literature [41]. Finally, our work builds on existing techniques that estimate workload statistics such as partition size, cardinality, and data skew [29].

7 Conclusion

With the shift to serverless computing, there have been a number of proposals to develop general computing frameworks on serverless infrastructure. However, due to resource limits and performance variations that are inherent to the serverless model, it is challenging to efficiently execute complex workloads that involve communication across functions. In this paper, we show that using a mixture of slow but cheap storage with fast but expensive storage is necessary to achieve a good cost-performance trade-off. We present Locus, an analytics system that uses performance modeling for shuffle operations executed on serverless architectures. Our evaluation shows that the model used in Locus is accurate and that it can achieve comparable performance to running Apache Spark on a provisioned cluster, and within $2 \times$ slower compared to Redshift. We believe the performance gap can be improved in the future, and meanwhile Locus can be preferred as it requires no provisioning of clusters.

Acknowledgement

We want to thank the anonymous reviewers and our shepherd Jana Giceva for their insightful comments. We also thank Alexey Tumanov, Ben Zhang, Kaifei Chen, Peter Gao, Ionel Gog, and members of PyWren Team and RISELab for reading earlier drafts of the paper. This research is supported by NSF CISE Expeditions Award CCF-1730628, and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

- [1] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Disk-locality in datacenter computing considered irrelevant. In *HotOS* (2011).
- [2] ANANTHANARAYANAN, G., HUNG, M. C.-C., REN, X., STOICA, I., WIERMAN, A., AND YU, M. Grass: Trimming stragglers in approximation analytics. In *NSDI* (2014).
- [3] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI* (2010).
- [4] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in map-reduce clusters using mantri. In *OSDI* (2010).
- [5] Amazon Athena. <http://aws.amazon.com/athena/>.
- [6] Serverless Reference Architecture: MapReduce. <https://github.com/awslabs/lambda-refarch-mapreduce>.
- [7] Azure Blob Storage Request Limits. <https://cloud.google.com/storage/docs/request-rate>.
- [8] Big Data Benchmark. <https://amplab.cs.berkeley.edu/benchmark/>.
- [9] Google BigQuery. <https://cloud.google.com/bigquery/>.
- [10] BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases* (1999).
- [11] CHOWDHURY, M., AND STOICA, I. Coflow: A Networking Abstraction for Cluster Applications. In *Proc. HotNets* (2012), pp. 31–36.
- [12] CHOWDHURY, M., AND STOICA, I. Coflow: A networking abstraction for cluster applications. In *HotNets* (2012).
- [13] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *SIGCOMM* (2014).
- [14] CHU, S., BALAZINSKA, M., AND SUCIU, D. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD* (2015).
- [15] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI* (2004).
- [16] ERIC JONAS, QIFAN PU, SHIVARAM VENKATARAMAN, ION STOICA, BENJAMIN RECHT. Occupy the Cloud: Distributed Computing for the 99%. In *SoCC* (2017).
- [17] ESTIVILL-CASTRO, V., AND WOOD, D. A survey of adaptive sorting algorithms. *ACM Comput. Surv.* (1992).
- [18] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI* (2017).
- [19] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network requirements for resource disaggregation. In *OSDI* (2016).
- [20] Google Cloud Storage Request Limits. <https://docs.microsoft.com/en-us/azure/storage/common/storage-scalability-targets>.
- [21] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google File System. In *Proc. SOSP* (2003), pp. 29–43.
- [22] Amazon Glue. <https://aws.amazon.com/glue/>.
- [23] Google Cloud Dataflow Shuffle. <https://cloud.google.com/dataflow/>.
- [24] GRAY, J., AND GRAEFE, G. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.* (1997).
- [25] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with OpenLambda. In *HotCloud* (2016).
- [26] Using AWS Lambda with Kinesis. <http://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html>.
- [27] LISTGARTEN, S., AND NEIMAT, M.-A. Modelling costs for a mm-dbms. In *RTDB* (1996).
- [28] MANEGOLD, S., BONCZ, P., AND KERSTEN, M. L. Generic database cost models for hierarchical memory systems. In *VLDB* (2002).
- [29] MANNINO, M. V., CHU, P., AND SAGER, T. Statistical profile estimation in database systems. *ACM Comput. Surv.* (1988).

- [30] NYBERG, C., BARCLAY, T., CVETANOVIC, Z., GRAY, J., AND LOMET, D. Alphasort: A cache-sensitive parallel external sort. *The VLDB Journal* (1995).
- [31] O'MALLEY, O. TeraByte Sort on Apache Hadoop. <http://sortbenchmark.org/YahooHadoop.pdf>.
- [32] OpenWhisk. <https://developer.ibm.com/openwhisk/>.
- [33] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *NSDI* (2015), pp. 293–307.
- [34] POESS, M., SMITH, B., KOLLAR, L., AND LARSON, P. Tpc-ds, taking decision support benchmarking to the next level. In *SIGMOD* (2002).
- [35] Amazon Redshift Spectrum. <https://aws.amazon.com/redshift/spectrum/>.
- [36] REN, X., ANANTHANARAYANAN, G., WIERMAN, A., AND YU, M. Hopper: Decentralized speculation-aware cluster scheduling at scale. *SIGCOMM* (2015).
- [37] S3 Request Limits. <https://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html>.
- [38] SALZBERG, B., TSUKERMAN, A., GRAY, J., STUEWART, M., UREN, S., AND VAUGHAN, B. Fastsort: A distributed single-input single-output external sort. In *SIGMOD* (1990).
- [39] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Mass storage systems and technologies (MSST)* (2010).
- [40] Sort Benchmark. <http://sortbenchmark.org>.
- [41] SOUNDARARAJAN, G., LUPEI, D., GHANBARI, S., POPESCU, A. D., CHEN, J., AND AMZA, C. Dynamic resource allocation for database servers running on virtual storage. In *FAST* (2009).
- [42] WANG, Q., GU, R., HUANG, Y., XIN, R., WU, W., SONG, J., AND XIA, J. NADSort. <http://sortbenchmark.org/NADSort2016.pdf>.
- [43] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCUALEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI* (2011).
- [44] ZAHARIA, M., DAS, T., LI, H., SHENKER, S., AND STOICA, I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing* (2012), USENIX Association.
- [45] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *OSDI* (2008).

dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces

Da Yu[†], Yibo Zhu[§], Behnaz Arzani[§], Rodrigo Fonseca[†], Tianrong Zhang[§], Karl Deng[§], Lihua Yuan[§]

[†]*Brown University*

[§]*Microsoft*

Abstract

Distributed, in-network packet capture is still the last resort for diagnosing network problems. Despite recent advances in collecting packet traces scalably, effectively utilizing pervasive packet captures still poses important challenges. Arbitrary combinations of middleboxes which transform packet headers make it challenging to even identify the same packet across multiple hops; packet drops in the collection system create ambiguities that must be handled; the large volume of captures, and their distributed nature, make it hard to do even simple processing; and the one-off and urgent nature of problems tends to generate ad-hoc solutions that are not reusable and do not scale. In this paper we propose dShark to address these challenges. dShark allows intuitive groupings of packets across multiple traces that are robust to header transformations and capture noise, offering simple streaming data abstractions for network operators. Using dShark on production packet captures from a major cloud provider, we show that dShark makes it easy to write concise and reusable queries against distributed packet traces that solve many common problems in diagnosing complex networks. Our evaluation shows that dShark can analyze production traces with more than 10 Mpps throughput on a commodity server, and has near-linear speedup when scaling out on multiple servers.

1 Introduction

Network reliability is critical for modern distributed systems and applications. For example, an ISP outage can cause millions of users to disconnect from the Internet [45], and a small downtime in the cloud network can lead to millions of lost revenue. Despite the advances in network verification and testing schemes [18, 26, 27, 34, 44], unfortunately, network failures are still common and are unlikely to be eliminated given the scale and complexity of today’s networks.

As such, diagnosis is an integral part of a network operator’s job to ensure high service availability. Once a fault that cannot be automatically mitigated happens, operators must quickly analyze the root cause so that they can correct the fault. Many tools have been developed to ease this process.

We can group existing solutions into host-based [40, 56, 57], and in-network tools [44, 68]. While able to diagnose several problems, host-based systems are fundamentally limited in visibility, especially in cases where the problem causes packets not to arrive at the edge. On the other hand, most in-network systems are based on aggregates [32], or on strong assumptions about the topology [56]. Switch hardware im-

provements have also been proposed [21, 28, 42, 56]. However, it is unlikely the commodity switches will quickly adopt these features and replace all the existing switches soon.

Because of these limitations, in today’s production networks, operators have *in-network packet captures* as the last resort [50, 68]. They provide a capture of a packet at each hop, allowing for gathering a full view of packets’ paths through the network. Analyzing such “distributed” traces allows one to understand how a packet, a flow, or even a group of flows were affected as they traversed each switch along their path. More importantly, most, if not all, commodity switches support various packet mirroring functionalities.

In this paper, we focus on making the analysis of in-network packet captures practical. Despite the diagnosing potential, this presents many unsolved challenges. As a major cloud provider, although our developers have implemented a basic analysis pipeline similar to [68], which generates some statistics, it falls short as our networks and fault scenarios get more complicated. Multi-hop captures, middleboxes, the (sometimes) daunting volume of captures, and the inevitable loss in the capture pipeline itself make it hard for operators to identify the root problem.

The packets usually go through a combination of header transformations (VXLAN, VLAN, GRE, and others) applied repeatedly and in different orders, making it hard to even parse and count packets correctly. In addition, the packet captures, which are usually generated via switch mirroring and collector capturing, are noisy in practice. This is because the mirrored packets are normally put in the lowest priority to avoid competing with actual traffic and do not have any retransmission mechanisms. It is pretty common for the mirrored packet drop rate to be close to the real traffic drop rate being diagnosed. This calls for some customized logic that can filter out false drops due to noise.

These challenges often force our operators to abandon the statistics generated by the basic pipeline and develop ad-hoc programs to handle specific faults. This is done in haste, with little consideration for correctness guarantees, performance, or reusability, and increasing the mean time to resolution.

To address these challenges, we design dShark, a scalable packet analyzer that allows for the analysis of in-network packet traces in near real-time and at scale. dShark provides a streaming abstraction with flexible and robust grouping of packets: all instances of a single packet at one or multiple hops, and all packets of an aggregate (*e.g.*, flow) at one or multiple hops. dShark is robust to, and hides the details of, compositions of packet transformations (encapsulation,

tunneling, or NAT), and noise in the capture pipeline. dShark offers flexible and programmable parsing of packets to define packets and aggregates. Finally, a query (*e.g.*, is the last hop of a packet the same as expected?) can be made against these groups of packets in a completely parallel manner.

The design of dShark is inspired by an observation that a general programming model can describe all the typical types of analysis performed by our operators or summarized in prior work [56]. Programming dShark has two parts: a declarative part, in JSON, that specifies how packets are parsed, summarized, and grouped, and an imperative part in C++ to process groups of packets. dShark programs are concise, expressive, and in languages operators are familiar with. While the execution model is essentially a windowed streaming map-reduce computation, the specification of programs is at a higher level, with the ‘map’ phase being highly specialized to this context: dShark’s parsing is designed to make it easy to handle multiple levels of header transformations, and the grouping is flexible to enable many different types of queries. As shown in §4, a typical analysis can be described in only tens of lines of code. dShark compiles this code, links it to dShark’s scalable and high-performance engine and handles the execution. With dShark, the time it takes for operators to start a specific analysis can be shortened from hours to minutes.

dShark’s programming model also enables us to heavily optimize the engine performance and ensures that the optimization benefits all analyses. Not using a standard runtime, such as Spark, allows dShark to integrate closely with the trace collection infrastructure, pushing filters and parsers very close to the trace source. We evaluate dShark on packet captures of production traffic, and show that on a set of commodity servers, with four cores per server, dShark can execute typical analyses in real time, even if all servers are capturing 1500B packets at 40Gbps line rate. When digesting faster capturing or offline trace files, the throughput can be further scaled up nearly linearly with more computing resources.

We summarize our contributions as follows: 1) dShark is the first general and scalable software framework for analyzing distributed packet captures. Operators can quickly express their intended analysis logic without worrying about the details of implementation and scaling. 2) We show that dShark can handle header transformations, different aggregations, and capture noise through a concise, yet expressive declarative interface for parsing, filtering, and grouping packets. 3) We show how dShark can express 18 diverse monitoring tasks, both novel and from previous work. We implement and demonstrate dShark at scale with real traces, achieving real-time analysis throughput.

2 Motivation

dShark provides a scalable analyzer of distributed packet traces. In this section, we describe why such a system is needed to aid operators of today’s networks.

2.1 Analysis of In-network Packet Traces

Prior work has shown the value of in-network packet traces for diagnosis [50, 68]. In-network packet captures are widely supported, even in production environments which contain heterogeneous and legacy switches. These traces can be described as the most detailed “logs” of a packet’s journey through the network as they contain per-packet/per-switch information of what happened.

It is true that such traces can be heavyweight in practice. For this reason, researchers and practitioners have continuously searched for replacements to packet captures diagnosis, like flow records [13, 14], or tools that allow switches to “digest” traces earlier [21, 42, 56]. However, the former necessarily lose precision, for being aggregates, while the latter requires special hardware support which in many networks is not yet available. Alternatively, a number of tools [5, 20, 53] have tackled diagnosis of specific problems, such as packet drops. However, these also fail at diagnosing the more general cases that occur in practice (§3), which means that the need for traces has yet to be eliminated.

Consequently, many production networks continue to employ in-network packet capturing systems [59, 68] and enable them on-demand for diagnosis. In theory, the operators, using packet traces, can reconstruct what happened in the network. However, we found that this is not simple in practice. Next, we illustrate this using a real example.

2.2 A Motivating Example

In 2017, a customer on our cloud platform reported an unexpected TCP performance degradation on transfers to/from another cloud provider. The customer is in the business of providing real-time video surveillance and analytics service, which relies on stable network throughput. However, every few hours, the measured throughput would drop from a few Gbps to a few Kbps, which would last for several minutes, and recover by itself. The interval of the throughput drops was non-deterministic. The customer did a basic diagnosis on their end hosts (VMs) and identified that the throughput drops were caused by packet drops.

This example is representative – it is very common for network traffic to go through multiple different components beyond a single data center, and for packets to be transformed multiple times on the way. Often times our operators do not control both ends of the connections.

In this specific case (Figure 1), the customer traffic leaves the other cloud provider, X’s network, goes through the ISP and reaches one of our switches that peers with the ISP (①). To provide a private network with the customer, the traffic is first tagged with a customer-specific 802.1Q label (②). Then, it is forwarded in our backbone/WAN in a VXLAN tunnel (③). Once the traffic arrives at the destination datacenter border (④), it goes through a load balancer (SLB), which uses IP-in-IP encapsulation (⑤,⑥), and is redirected to a VPN gateway, which uses GRE encapsulation (⑦, ⑧), before

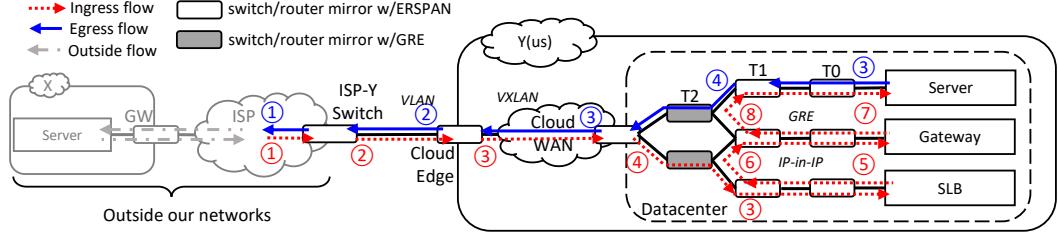


Figure 1: The example scenario. We collect per-hop traces in our network (Y and ISP-Y-switch) and do not have the traces outside our network except the ingress and egress of ISP-Y-switch. The packet format of each numbered network segment is listed in Table 1.

Number	Header Format									
	Headers Added after Mirroring			Mirrored Headers						
①	ETHERNET	IPV4	ERSPAN	ETHERNET						IPV4 TCP
②	ETHERNET	IPV4	ERSPAN	ETHERNET						802.1Q IPV4 TCP
③	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
④	ETHERNET	IPV4	GRE	ETHERNET	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑤	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4 TCP
⑥	ETHERNET	IPV4	GRE	ETHERNET	IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4 TCP
⑦	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4			GRE	ETHERNET	IPV4 TCP
⑧	ETHERNET	IPV4	GRE		IPV4			GRE	ETHERNET	IPV4 TCP

Table 1: The packet formats in the example scenario. Different switch models may add different headers before sending out the mirrored packets, which further complicates the captured packet formats.

reaching the destination server. Table 1 lists the corresponding captured packet formats. Note that beyond the differences in the encapsulation formats, different switches add different headers when mirroring packets (*e.g.*, ERSPAN vs GRE). On the return path, the traffic from the VMs on our servers is encapsulated with VXLAN, forwarded to the datacenter border, and routed back to X.

When our network operators are called up for help, they must answer two questions in a timely manner: 1) are the packets dropped in our network? If not, can they provide any pieces of evidence? 2) if yes, where do they drop? While packet drops seem to be an issue with many proposed solutions, the operators still find the diagnosis surprisingly hard in practice.

Problem 1: many existing tools fail because of their specific assumptions and limitations. As explained in §2.1, existing tools usually require 1) full access to the network including end hosts [5, 20]; 2) specific topology, like the Clos [53], or 3) special hardware features [21, 32, 42, 56]. In addition, operators often need evidence for “the problem is not because of” a certain part of the network (in this example, our network but not ISP or the other cloud network), for pruning the potential root causes. However, most of those tools are not designed to solve this challenge.

Since all these tools offer little help in this scenario, network operators have no choice but to enable in-network capturing and analyze the packet traces. Fortunately, we already deployed Everflow [68], and are able to capture per-hop traces of a portion of flows.

Problem 2: the basic trace analysis tools fall short for the complicated problems in practice. Even if network operators have complete per-hop traces, recovering what happened in the network is still a challenge. Records for the same packets spread across multiple distributed captures, and none of the well-known trace analyzers such as Wireshark [2] has the ability to join traces from multiple vantage points. Grouping them, even for the instances of a single packet across multiple hops, is surprisingly difficult, because each packet may be modified or encapsulated by middleboxes multiple times, in arbitrary combinations.

Packet capturing noise further complicates analysis. Mirrored packets can get dropped on their way to collectors or dropped by the collectors. If one just counts the packet occurrence on each hop, the real packet drops may be buried in mirrored packet drops and remain unidentified. Again, it is unclear how to address this with existing packet analyzers.

Because of these reasons, network operators resort to developing ad-hoc tools to handle specific cases, while still suffering from the capturing noise.

Problem 3: the ad-hoc solutions are inefficient and usually cannot be reused. It is clear that the above ad-hoc tools have limitations. First, because they are designed for specific cases, the header parsing and analysis logic will likely be specific. Second, since the design and implementation have to be swift (cloud customers are anxiously waiting for mitigation!), reusability, performance, and scalability will likely not be priorities. In this example, the tool developed was single threaded and thus had low throughput. Consequently,

operators would capture several minutes worth of traffic and have to spend multiples of that to analyze it.

After observing these problems in a debugging session in production environment, we believe that a general, easy-to-program, scalable and high-performance in-network packet trace analyzer can bring significant benefits to network operators. It can help them understand, analyze and diagnose their network more efficiently.

3 Design Goals

Motivated by many real-life examples like the one in §2.2, we derive three design goals that we must address in order to facilitate in-network trace analysis.

3.1 Broadly Applicable for Trace Analysis

In-network packet traces are often used by operators to identify where network properties and invariants have been violated. To do so, operators typically search for abnormal behavior in the large volume of traces. For different diagnosis tasks, the logic is different.

Unfortunately, operators today rely on manual processing or ad-hoc scripts for most of the tasks. Operators must first parse the packet headers, *e.g.*, using Wireshark. After parsing, operators usually look for a few key fields, *e.g.*, 5-tuples, depending on the specific diagnosis tasks. Then they apply filters and aggregations on the key fields for deeper analysis. For example, if they want to check all the hops of a certain packet, they may filter based on the 5-tuple plus the IP id field. To check more instances and identify a consistent behavior, operators may apply similar filters many times with slightly different values, looking for abnormal behavior in each case. It is also hard to join instances of the same packet captured in different points of the network.

Except for the initial parsing, all the remaining steps vary from case to case. We find that there are four types of aggregations used by the operators. Depending on the scenario, operators may want to analyze 1) each single packet on a specific hop; 2) analyze the multi-hop trajectory of each single packet; 3) verify some packet distributions on a single switch or middlebox; or 4) analyze complicated tasks by correlating multiple packets on multiple hops. Table 2 lists diagnosis applications that are commonly used and supported by existing tools. We classify them into above four categories.

dShark must be broadly applicable for all these tasks – not only these four aggregation modes, but also support different analysis logic after grouping, *e.g.*, verifying routing properties or localizing packet drops.

3.2 Robust in the Wild

dShark must be robust to practical artifacts in the wild, especially header transformations and packet capturing noise.

Packet header transformations. As shown in §2.2, these are very common in networks, due to the deployment of various middleboxes [49]. They become one of the main

obstacles for existing tools [43, 56, 69] to perform all of the diagnosis logic (listed in Table 2) in one shot. As we can see from the table, some applications need to be robust to header transformations. Therefore, dShark must correctly group the packets as if there is no header transformation. While parsing the packet is not hard (indeed, tools like Wireshark can already do that), it is unclear how operators may specify the grouping logic across different header formats. In particular, today’s filtering languages are often ambiguous. For example, the “ip.src == X” statement in Wireshark display filter may match different IP layers in a VXLAN-in-IP-in-IP packet and leads to incorrect grouping results. dShark addresses this by explicitly indexing multiple occurrences of the same header type (*e.g.*, IP-in-IP), and by adding support to address the innermost ([−1]), outermost ([0]), and all ([:]) occurrences of a header type.

Packet capturing noise. We find that it is challenging to localize packet drops when there is significant packet capturing noise. We define noise here as drops of mirrored packets in the network or in the collection pipeline. Naïvely, one may just look at all copies of a packet captured on all hops, check whether the packet appears on each hop as expected. However, 1% or even higher loss in the packet captures is quite common in reality, as explained in §2.2 as well as in [61]. With the naïve approach, every hop in the network will have 1% false positive drop rate in the trace. This makes localizing any real drop rate that is comparable or less than 1% challenging because of the high false positive rate.

Therefore, for dShark, we must design a programming interface that is flexible for handling arbitrary header transformations, yet can be made robust to packet capturing noise.

3.3 Fast and Scalable

The volume of in-network trace is usually very large. dShark must be fast and scalable to analyze the trace. Below we list two performance goals for dShark.

Support real-time analysis when collocating on collectors. Recent efforts such as [68] and [50] have demonstrated that packets can be mirrored from the switches and forwarded to trace collectors. These collectors are usually commodity servers, connected via 10Gbps or 40Gbps links. Assuming each mirrored packet is 1500 bytes large, this means up to 3.33M packets per second (PPS). With high-performance network stacks [1, 52, 61], one CPU core is sufficient to capture at this rate. Ideally, dShark should co-locate with the collecting process, reuse the remaining CPU cores and be able to keep up with packet captures in real-time. Thus, we set this as the first performance goal – with a common CPU on a commodity server, dShark must be able to analyze *at least* 3.33 Mpps.

Be scalable. There are multiple scenarios that require higher performance from dShark: 1) there are smaller packets even though 1500 bytes is the most typical packet size in our production network. Given 40Gbps capturing rate, this means higher PPS; 2) there can be multiple trace collectors [68] and

Group pattern	Application	Analysis logic	In-nw ck. only	Header transf.	Query LOC
One packet on one hop	Loop-free detection [21] <i>Detect forwarding loop</i>	<i>Group:</i> same packet(ipv4[0].ipid, tcp[0].seq) on one hop <i>Query:</i> does the same packet appear multiple times on the same hop	No	No	8
	Overloop-free detection [69] <i>Detect forwarding loop involving tunnels</i>	<i>Group:</i> same packet(ipv4[0].ipid, tcp[0].seq) on tunnel endpoints <i>Query:</i> does the same packet appear multiple times on the same endpoint	Yes	Yes	8
One packet on multiple hops	Route detour checker <i>Check packet's route in failure case</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches <i>Query:</i> is valid detour in the recovered path(ipv4[:].ttl)	No	Yes*	49
	Route error <i>Detect wrong packet forwarding</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches <i>Query:</i> get last correct hop in the recovered path(ipv4[:].ttl)	No*	Yes*	49
	Netsight [21] <i>Log packet's in-network lifecycle</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches <i>Query:</i> recover path(ipv4[:].ttl)	No*	Yes*	47
	Hop counter [21] <i>Count packet's hop</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches <i>Query:</i> count record	No*	Yes*	6
Multiple packets on one hop	Traffic isolation checker [21] <i>Check whether hosts are allowed to talk</i>	<i>Group:</i> all packets at dst ToR(SWITCH=dst.ToR) <i>Query:</i> have prohibited host(ipv4[0].src)	No	No	11
	Middlebox(SLB, GW, etc) profiler <i>Check correctness/performance of middleboxes</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post middlebox <i>Query:</i> is middlebox correct(related fields)	Yes	Yes	18 [†]
	Packet drops on middleboxes <i>Check packet drops in middleboxes</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post middlebox <i>Query:</i> exist ingress and egress trace	Yes	Yes	8
	Protocol bugs checker(BGP, RDMA, etc) [69] <i>Identify wrong implementation of protocols</i>	<i>Group:</i> all BGP packets at target switch(SWITCH=tar_SW) <i>Query:</i> correctness(related fields) of BGP(FLTR: tcp[-1].src dst=179)	Yes	Yes*	23 [‡]
	Incorrect packet modification [21] <i>Check packets' header modification</i>	<i>Group:</i> same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post the modifier <i>Query:</i> is modification correct (related fields)	Yes	Yes*	4 [◦]
	Waypoint routing checker [21, 43] <i>Make sure packets (not) pass a waypoint</i>	<i>Group:</i> all packets at waypoint switch(SWITCH=waypoint) <i>Query:</i> contain flow(ipv4[-1].src+dst, tcp[-1].src+dst) should (not) pass	Yes	No	11
	DDoS diagnosis [43] <i>Localize DDoS attack based on statistics</i>	<i>Group:</i> all packets at victim's ToR(SWITCH=vic_ToR) <i>Query:</i> statistics of flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No	Yes*	18
	Congested link digestion [43] <i>Find flows using congested links</i>	<i>Group:</i> all packets(ipv4[-1].ipid, tcp[-1].seq) pass congested link <i>Query:</i> list of flows(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	Yes*	14
	Silent black hole localizer [43, 69] <i>Localize switches that drop all packets</i>	<i>Group:</i> packets with duplicate TCP(ipv4[-1].ipid, tcp[-1].seq) <i>Query:</i> get dropped hop in the recovered path(ipv4[:].ttl)	No*	Yes*	52
	Silent packet drop localizer [69] <i>Localize random packet drops</i>	<i>Group:</i> packets with duplicate TCP(ipv4[-1].ipid, tcp[-1].seq) <i>Query:</i> get dropped hop in the recovered path(ipv4[:].ttl)	No*	Yes*	52
Multiple packets on multiple hops	ECMP profiler [69] <i>Profile flow distribution on ECMP paths</i>	<i>Group:</i> all packets at ECMP ingress switches(SWITCH in ECMP) <i>Query:</i> statistics of flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	No	18
	Traffic matrix [43] <i>Traffic volume between given switch pairs</i>	<i>Group:</i> all packets at given two switches(SWITCH in tar_SW) <i>Query:</i> total volume of overlapped flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	Yes*	21

Table 2: We implemented 18 typical diagnosis applications in dShark. “No*” in column “in-network checking only” means this application can also be done with end-host checking with some assumptions. “Yes*” in column “header transformation” needs to be robust to header transformation in our network, but, in other environments, it might not. “ $\text{ipv4}[:].\text{ttl}$ ” in the analysis logic means dShark concatenates all ipv4 ’s TTLs in the header. It preserves order information even with header transformation. Sorting it makes path recovery possible. [†]We profiled SLB. [‡]We focused on BGP route filter. [◦]We focused on packet encapsulation.

3) for offline analysis, we hope that dShark can run faster than the packet timestamps. Therefore, dShark must horizontally scale up within one server, or scale out across multiple servers. Ideally, dShark should have near-linear speed up with more computing resources.

4 dShark Design

dShark is designed to allow for the analysis of distributed packet traces in near real time. Our goal in its design has been to allow for scalability, ease of use, generality, and robustness. In this section, we outline dShark’s design and how it allows us to achieve these goals. At a high level, dShark provides a domain-specific language for expressing distributed network monitoring tasks, which runs atop a map-reduce-like infrastructure that is tightly coupled, for efficiency, with a packet capture infrastructure. The DSL primitives are designed to

enable flexible filtering and grouping of packets across the network, while being robust to header transformations and capture noise that we observe in practice.

4.1 A Concrete Example

To diagnose a problem with dShark, an operator has to write two related pieces: a declarative set of trace specifications indicating relevant fields for grouping and summarizing packets; and an imperative callback function to process groups of packet summaries.

Here we show a basic example – detecting forwarding loops in the network with dShark. This means dShark must check whether or not any packets appear more than once at any switch. First, network operators can write the trace specifications as follows, in JSON:

```

1 {
2   Summary: {

```

```

3     Key: [SWITCH, ipId, seqNum],
4     Additional: []
5   },
6   Name: {
7     ipId: ipv4[0].id,
8     seqNum: tcp[0].seq
9   },
10  Filter: [
11    [eth, ipv4, ipv4, tcp]: {      // IP-in-IP
12      ipv4[0].srcIp: 10.0.0.1
13    }
14  ]
15 }

```

The first part, “Summary”, specifies that the query will use three fields, *SWITCH*, *ipId* and *seqNum*. dShark builds a *packet summary* for each packet, using the variables specified in “Summary”. dShark groups packets that have the same “key” fields, and shuffles them such that each group is processed by the same processor.

SWITCH, one of the only two predefined variables in dShark,¹ is the switch where the packet is captured. Transparent to operators, dShark extracts this information from the additional header/metadata (as shown in Table 1) added by packet capturing pipelines [59, 68].

Any other variable must be specified in the “Name” part, so that dShark knows how to extract the values. Note the explicit index “[0]” – this is the key for making dShark robust to header transformations. We will elaborate this in §4.3.

In addition, operators can constrain certain fields to a given value/range. In this example, we specify that if the packet is an IP-in-IP packet, we will ignore it unless its outermost source IP address is 10.0.0.1.

In our network, we assume that *ipId* and *seqNum* can identify a unique TCP packet without specifying any of the 5-tuple fields.² Operators can choose to specify additional fields. However, we recommend using only necessary fields for better system efficiency and being more robust to middleboxes. For example, by avoiding using 5-tuple fields, the query is robust to any NAT that does not alter *ipId*.

The other piece is a query function, in C++:

```

1 map<string, int> query(const vector<Group>& groups) {
2   map<string, int> r = {"loop", 0}, {"normal", 0};
3   for (const Group& group : groups) {
4     group.size() > 1 ?
5       (r["loop"]++) : (r["normal"]++);
6   }
7   return r;
8 }

```

The query function is written as a callback function, taking an array of groups and returning an arbitrary type: in this case, a map of string keys to integer values. This is flexible for operators – they can define custom counters like in this example, get probability distribution by counting in predefined bins, or pick out abnormal packets by adding entries into the dictionary. In the end, dShark will merge these key-value pairs from all query processor instances by unionizing all

¹The other predefined variable is *TIME*, the timestamp of packet capture.

²In our network and common implementation, IP ID is chosen independently from TCP sequence number. This may not always be true [58].

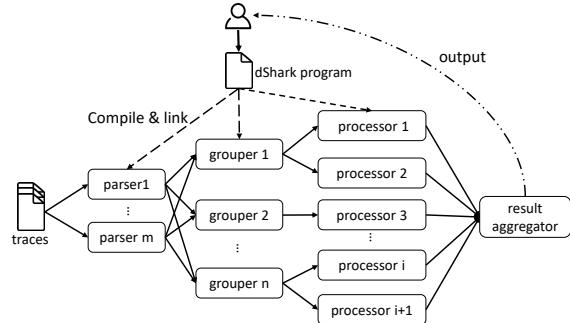


Figure 2: dShark architecture.

keys and summing the values of the same keys. Operators will get a human-readable output of the final key-value pairs.

In this example, the query logic is simple. Since each packet group contains all copies of a packet captured/mirrored by the same switch, if there exist two packet summaries in one group, a loop exists in the network. The query can optionally refer to any field defined in the summary format. We also implemented 18 typical queries from the literature and based on our experience in production networks. As shown in Table 2, even the most complicated one is only 52 lines long. For similar diagnosis tasks, operators can directly reuse or extend these query functions.

4.2 Architecture

The architecture of dShark is inspired by both how operators manually process the traces as explained in 3.1, and distributed computing engines like MapReduce [15]. Under that light, dShark can be seen as a streaming data flow system specialized for processing distributed network traces. We provide a general and easy-to-use programming model so that operators only need to focus on analysis logic without worrying about implementation or scaling.

dShark’s runtime consists of three main steps: *parse*, *group* and *query* (Figure 2). Three system components handle each of the three steps above, respectively. Namely,

- *Parser*: dShark consumes network packet traces and extracts user-defined key header fields based on different user-defined header formats. Parsers send these key fields as packet summaries to groupers. The dShark parsers include recursive parsers for common network protocols, and custom ones can be easily defined.
- *Grouper*: dShark groups packet summaries that have the same values in user-defined fields. Groupers receive summaries from all parsers and create batches per group based on time windows. The resulting packet groups are then passed to the query processors.
- *Query processor*: dShark executes the query provided by users and outputs the result for final aggregation.

dShark pipeline works with two cascading MapReduce-like stages: 1) first, packet traces are (mapped to be) parsed in parallel and shuffled (or reduced) into groups; 2) query processors run analysis logic for each group (map) and finally aggregate the results (reduce). In particular, the *parser* must

handle header transformations as described in §3.2, and the *grouper* must support all possible packet groupings (§3.1). All three components are optimized for high performance and can run in a highly parallel manner.

Input and output to the dShark pipeline. dShark ingests packet traces and outputs aggregated analysis results to operators. dShark assumes that there is a system in place to collect traces from the network, similar to [68]. It can work with live traces when collocating with trace collectors, or run anywhere with pre-recorded traces. When trace files are used, a simple coordinator (§5.4) monitors the progress and feeds the traces to the parser in chunks based on packet timestamps. The final aggregator generates human-readable outputs as the query processors work. It creates a union of the key-value pairs and sums up values output by the processors (§5).

Programming with dShark. Operators describe their analysis logic with the programming interface provided by dShark, as explained below (§4.3). dShark compiles operators’ programs into a dynamic-linked library. All parsers, groupers and query processors load it when they start, though they link to different symbols in the library. dShark chooses this architecture over script-based implementation (*e.g.*, Python or Perl) for better CPU efficiency.

4.3 dShark Programming Model

As shown in the above example, the dShark programming interface consists of two parts: 1) declarative packet trace specifications in JSON, and 2) imperative query functions (in C++). We design the specifications to be declarative to make common operations like *select*, *filter* and *group* fields in the packet headers straightforward to the operators. On the other hand, we make the query functions imperative to offer enough degrees of freedom for the operators to *define* different diagnosis logic. This approach is similar to the traditional approach in databases of embedding imperative user-defined functions in declarative SQL queries. Below we elaborate on our design rationale and on details not shown in the example above.

“Summary” in specifications. A packet summary is a byte array containing only a few key fields of a packet. We introduce packet summary for two main goals: 1) to let dShark compress the packets right after parsing while retaining the necessary information for query functions. This greatly benefits dShark’s efficiency by reducing the shuffling overhead and memory usage; 2) to let groupers know which fields should be used for grouping. Thus, the description of a packet summary format consists of two lists. The first contains the fields that will be used for grouping and the second of header fields that are not used as grouping keys but are required by the query functions. The variables in both lists must be defined in the “Name” section, specifying where they are in the headers.

“Name” in specifications. Different from existing languages like Wireshark filter or BPF, dShark requires an explicit index when referencing a header, *e.g.*, “*ipv4[0]*” instead

of simply “*ipv4*”. This means the first IPv4 header in the packet. This is for avoiding ambiguity, since in practice a packet can have multiple layers of the same header type due to tunneling. We also adopt the Python syntax, *i.e.*, “[*ipv4[-1]*]” to mean the last (or innermost) IPv4 header, “[*ipv4[-2]*]” to mean the last but one IPv4 header, *etc.*

With such header indexes, the specifications are both robust to header transformations and explicit enough. Since the headers are essentially a stack (LIFO), using negative indexes would allow operators to focus on the end-to-end path of a packet or a specific tunnel regardless of any additional header transformation. Since network switches operate based on outer headers, using 0 or positive indexes (especially 0) allows operators to analyze switch behaviors, like routing.

“Filter” in specifications. Filters allow operators to prune the traces. This can largely improve the system efficiency if used properly. We design dShark language to support adding constraints for different types of packets. This is inspired by our observation in real life cases that operators often want to diagnose packets that are towards/from a specific middlebox. For instance, when diagnosing a specific IP-in-IP tunnel endpoint, *e.g.*, 10.0.0.1, we only care IP-in-IP packets whose source IP is 10.0.0.1 (packets after encapsulation), and common IP packets whose destination IP is 10.0.0.1 (packets before encapsulation). For convenience, dShark supports “*” as a wildcard to match any headers.

Query functions. An operator can write the query functions as a callback function that defines the analysis logic to be performed against a batch of groups. To be generally applicable for various analysis tasks, we choose to prefer language flexibility over high-level descriptive languages. Therefore, we allow operators to program any logic using the native C++ language, having as input an array of packet groups, and as output an arbitrary type. The query function is invoked at the end of time windows, with the guarantee that all packets with the same key will be processed by the same processor (the same semantics of a shuffle in MapReduce).

In the query functions, each *Group* is a vector containing a number of summaries. Within each summary, operators can directly refer the values of fields in the packet summary, *e.g.*, *summary.ipId* is *ipId* specified in JSON. In addition, since it is in C++, operators can easily query our internal service REST APIs and get control plane metadata to help analysis, *e.g.*, getting the topology of a certain network. Of course, this should only be done per a large batch of batches to avoid a performance hit. This is a reason why we design query functions to take a *batch* of groups as input.

4.4 Support For Various Groupings

To show that our programming model is general and easy to use, we demonstrate how operators can easily specify the four different aggregation types, which we extend to *grouping* in dShark, listed in §3.1.

Single-packet single-hop grouping. This is the most basic

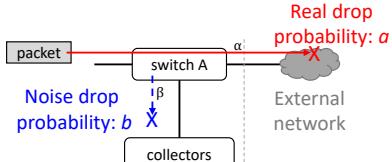


Figure 3: Packet capturing noise may interfere with the drop localization analysis.

Case	Probability	w/o E2E info	w/ E2E info
No drop	$(1-a)(1-b)$	Correct	Correct
Real drop	$a(1-b)$	Correct	Correct
Noise drop	$(1-a)b$	Incorrect	Correct
Real + Noise drop	ab	Incorrect	Incorrect

Table 3: The correctness of localizing packet drops. The two types of drops are independent because the paths are disjoint after A.

grouping, which is used in the example (§4.1). In packet summary format, operators simply specify the “key” as a set of fields that can uniquely identify a packet, and from which switch (*SWITCH*) the packet is collected.

Multi-packet single-hop grouping. This grouping is helpful for diagnosing middlebox behaviors. For example, in our data center, most software-based middleboxes are running on a server under a ToR switch. All packets which go into and out of the middleboxes must pass through that ToR. In this case, operators can specify the “key” as *SWITCH* and some middlebox/flow identifying fields (instead of identifying each packet in the single-packet grouping) like 5-tuple. We give more details in §6.1.

Single-packet multi-hop grouping. This can show the full path of each packet in the network. This is particularly useful for misrouting analysis, *e.g.*, does the traffic with a private destination IP range that is supposed to stay within data centers leak to WAN? For this, operators can just set packet identifying fields as the key, without *SWITCH*, and use the [-1] indexing for the innermost IP/TCP header fields. dShark will group all hops of each packet so that the query function checks whether each packet violates routing policies. The query function may have access to extra information, such as the topology, to properly verify path invariants.

Multi-packet multi-hop grouping. As explained in §3.2, loss of capture packets may impact the results of localizing packet drops, by introducing false positives. In such scenarios dShark *should* be used with multi-packet multi-hop groupings, which uses the 5-tuple and the sequence numbers as the grouping keys, without *ipId*. This has the effect of grouping together transport-level retransmissions. We next explain the rationale for this requirement.

4.5 Addressing Packet Capture Noise

To localize where packets are dropped, in theory, one could just group all hops of each packet, and then check where in the network the packet disappears from the packet captures on the way to its destination. In practice, however, we find that the noise caused by data loss in the captures themselves, *e.g.*,

drops on the collectors and/or drops in the network on the way to the collector, will impact the validity of such analysis.

We elaborate this problem using the example in Figure 3 and Table 3. For ease of explanation we will refer the to paths of the mirrored packets from each switch to the collector as β type paths and the normal path of the packet as α type paths. Assume switch A is at the border of our network and the ground truth is that drop happens after A. As operators, we want to identify whether the drop happens within our network. Unfortunately, due to the noise drop, we will find A is dropping packets with probability b in the trace. If the real drop probability a is less than b , we will misblame A. This problem, however, can be avoided if we correlate individual packets across different hops in the network as opposed to relying on simple packet counts.

Specifically, we propose two mechanisms to help dShark avoid miss-detecting where the packet was dropped:

Verifying using the next hop(s). If the β type path dropping packets is that from a switch in the middle of the α path, assuming that the probability that *the same* packet’s mirror is dropped on two β paths is small, one can find the packet traces from the next hop(s) to verify whether A is really the point of packet drop or not. However, this mechanism would fail in the “last hop” case, where there is no next hop in the trace. The “last hop” case is either 1) the specific switch is indeed the last on the α path, however, the packets may be dropped by the receiver host, or 2) the specific switch is the last hop before the packet goes to external networks that do not capture packet traces. Figure 3 is such a case.

Leveraging information in end-to-end transport. To address the “last hop” issue, we leverage the information provided by end-to-end transport protocols. For example, for TCP flows, we can verify a packet was dropped by counting the number of retransmissions seen for each TCP sequence number. In dShark, we can just group all packets with the same TCP sequence number across all hops together. If there is indeed a drop after A, the original packet and retransmitted TCP packets (captured at all hops in the internal network) will show up in the group as packets with different IP IDs, which eliminates the possibility that the duplicate sequence number is due to a routing loop. Otherwise, it is a noise drop on the β path.

This process could have false positives as the packet could be dropped both on the β and α path. This occurs with probability of only $a \times b$ – in the “last hop” cases like Figure 3, the drops on β and α path are likely to be independent since the two paths are disjoint after A. In practice, the capture noise b is $\ll 100\%$. Thus any a can be detected robustly.

Above, we focused on describing the process for TCP traffic as TCP is the most prominent protocol used in data center networks [6]. However, the same approach can be applied to any other reliable protocols as well. For example, QUIC [31] also adds its own sequence number in the packet header. For general UDP traffic, dShark’s language also

allows the operators to specify similar identifiers (if exist) based on byte offset from the start of the payload.

5 dShark Components and Implementation

We implemented dShark, including parsers, groupers and query processors, in >4K lines of C++ code. We have designed each instance of them to run in a single thread, and can easily scale out by adding more instances.

5.1 Parser

Parsers recursively identify the header stack and, if the header stack matches any in the Filter section, check the constraints on header fields. If there is no constraint found or all constraints are met, the fields in the Summary and Name sections are extracted and serialized in the form of a byte array. To reduce I/O overhead, the packet summaries are sent to the groupers in batches.

Shuffling between multiple parsers and groupers: When working with multiple groupers, to ensure grouping correctness, all parsers will have to send packet summaries that belong to the same groups to the same grouper. Therefore, parsers and groupers *shuffle* packet summaries using a consistent hashing of the “key” fields. This may result in increased network usage when the parsers and groupers are deployed across different machines. Fortunately, the amount of bandwidth required is typically very small – as shown in Table 2, common summaries are only around 10B, more than 100× smaller than an original 1500B packet.

For analyzing live captures, we closely integrate parsers with trace collectors. The raw packets are handed over to parsers via memory pointers without additional copying.

5.2 Grouper

dShark then groups summaries that have the same keys. Since the grouper does not know in advance whether or not it is safe to close its current group (groupings might be very long-lived or even perpetual), we adopt a tumbling window approach. Sizing the window presents trade-offs. For query correctness, we would like to have all the relevant summaries in the same window. However, too large of a window increases the memory requirements.

dShark uses a 3-second window – once three seconds (in packet timestamps) passed since the creation of a group, this group can be wrapped up. This is because, in our network, packets that may be grouped are typically captured within three seconds.³ In practice, to be robust to the noise in packet capture timestamps, we use the number of packets arriving thereafter as the window size. Within three seconds, a parser with 40Gbps connection receives no more than 240M packets even if all packets are as small as 64B. Assuming that the number of groupers is the same as or more than parsers, we can use a window of 240M (or slightly more) packet

³The time for finishing TCP retransmission plus the propagation delay should still fall in three seconds.

summaries. This only requires several GB of memory given that most packet summaries are around 10B large (Table 2).

5.3 Query Processor

The summary groups are then sent to the query processors in large batches.

Collocating groupers and query processors: To minimize the communication overhead between groupers and query processors, in our implementation processors and groupers are threads in the same process, and the summary groups are passed via memory pointers.

This is feasible because the programming model of dShark guarantees that each summary group can be processed independently, *i.e.*, the query functions can be executed completely in parallel. In our implementation, query processors are child threads spawned by groupers whenever groupers have a large enough batch of summary groups. This mitigates thread spawning overhead, compared with processing one group at one time. The analysis results of this batch of packet groups are in the form of a key-value dictionary and are sent to the result aggregator via a TCP socket. Finally, the query process thread terminates itself.

5.4 Supporting Components in Practice

Below, we elaborate some implementation details that are important for running dShark in practice.

dShark compiler. Before initiating its runtime, dShark compiles the user program. dShark generates C++ meta code from the JSON specification. Specifically, a definition of *struct Summary* will be generated based on the fields in the summary format, so that the query function has access to the value of a field by referring to *Summary.variable_name*. The template of a callback function that extracts fields will be populated using the Name section. The function will be called after the parsers identify the header stack and the pointers to the beginning of each header. The Filter section is compiled similarly. Finally, this piece of C++ code and the query function code will compile together by a standard C++ compiler and generate a dynamic link library. dShark pushes this library to all parsers, groupers and query processors.

Result aggregator. A result aggregator gathers the output from the query processors. It receives the key-value dictionaries sent by query processors and combines them by unionizing the keys and summing the values of the same keys. It then generates human-readable output for operators.

Coordinate parsers. dShark parsers consume partitioned network packet traces in parallel. In practice, this brings a synchronization problem when they process *offline* traces. If a fast parser processes packets of a few seconds ahead of a slower parser (in terms of when the packets are captured), the packets from the slower parser may fall out of grouper moving window (§5.2), leading to incorrect grouping.

To address this, we implemented a coordinator to simulate live capturing. The coordinator periodically tells all parsers

until which timestamp they should continue processing packets. The parsers will report their progress once they reach the target timestamp and wait for the next instruction. Once all parsers report completion, the coordinator sends out the next target timestamp. This guarantees that the progress of different parsers will never differ too much. To avoid stragglers, the coordinator may drop parsers that are consistently slower.

Over-provision the number of instances. Although it may be hard to accurately estimate the minimum number of instances needed (see §6) due to the different CPU overhead of various packet headers and queries, we use conservative estimation and over-provision instances. It only wastes negligible CPU cycles because we implement all components to spend CPU cycles only on demand.

6 dShark Evaluation

We used dShark for analyzing the in-network traces collected from our production networks⁴. In this section, we first present a few examples where we use dShark to check some typical network properties and invariants. Then, we evaluate the performance of dShark.

6.1 Case Study

We implement 18 typical analysis tasks using dShark (Table 2). We explain three of them in detail below.

Loop detection. To show the correctness of dShark, we perform a controlled experiment using loop detection analysis as an example. We first collected in-network packet traces (more than 10M packets) from one of our networks and verified that there is no looping packet in the trace. Then, we developed a script to inject looping packets by repeating some of the original packets with different TTLs. The script can inject with different probabilities.

We use the same code as in §4.1. Figure 4 illustrates the number of looping packets that are injected and the number of packets caught by dShark. dShark has zero false negative or false positive in this controlled experiment.

Profiling load balancers. In our data center, layer-4 software load balancers (SLB) are widely deployed under ToR switches. They receive packets with a virtual IP (VIP) as the destination and forward them to different servers (called DIP) using IP-in-IP encapsulation, based on flow-level hashing. Traffic distribution analysis of SLBs is handy for network operators to check whether the traffic is indeed balanced.

To demonstrate that dShark can easily provide this, we randomly picked a ToR switch that has an SLB under it. We deployed a rule on that switch that mirrors *all* packets that go towards a specific VIP and come out. In one hour, our collectors captured more than 30M packets in total.⁵

Our query function generates both flow counters and packet

⁴All the traces we use in evaluation are from clusters running internal services. We do not analyze our cloud customers traffic without permission.

⁵An SLB is responsible for multiple VIPs. The traffic volume can vary a lot across different VIPs.

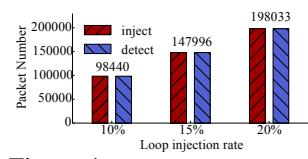


Figure 4: Injected loops are all detected.

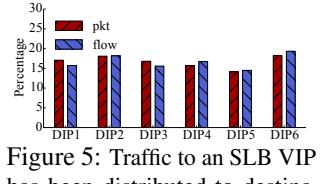


Figure 5: Traffic to an SLB VIP has been distributed to destination IPs.

counters of each DIP. Figure 5 shows the result – among the total six DIPs, DIP5 receives the least packets whereas DIP6 gets the most. Flow-level counters show a similar distribution. After discussing with operators, we conclude that for this VIP, load imbalance does exist due to imbalanced hashing, while it is still in an acceptable range.

Packet drop localizer. Noise can affect the packet drop localizer. Here we briefly evaluate the effectiveness of using transport-level retransmission information to reduce false positives (§4.5). We implemented the packet drop localizer as shown in Table 2, and used the noise mitigation mechanism described in §4.5. In a production data center, we deployed a mirroring rule on *all* switches to mirror *all* packets that originate from or go towards all servers, and fed the captured packets to dShark. We first compare our approach, which takes into account gaps in the sequence of switches, and uses retransmissions as evidence of actual drops, with a naïve approach, that just looks at the whether the last captured hop is the expected hop. Since the naïve approach does not work for drops at the last switch (including ToR and the data center boundary Tier-2 spine switches), for this comparison we only considered packets whose last recorded switch were leaf (Tier-1) switches. The naïve approach reports 5,599 suspected drops while dShark detects 7. The reason for the difference is drops of mirrored packets, which we estimated in our log to be approximately 2.2%. The drops detected by dShark are real, because they generated retransmissions with the same TCP sequence number.

Looking at all packets (and not only the ones whose traces terminate at the Tier-1 switches), we replayed the trace while randomly dropping capture packets with increasing probabilities. dShark reported 5,802, 5,801, 5,801 and 5,784 packet drops under 0%, 1%, 2% and 5% probabilities respectively. There is still a possibility that we miss the retransmitted packet, but, from the result, it is very low (0.3%).

6.2 dShark Component Performance

Next, we evaluate the performance of dShark components individually. For stress tests, we feed offline traces to dShark as fast as possible. To represent commodity servers, we use eight VMs from our public cloud platform, each has a Xeon 16-core 2.4GHz vCPU, 56GB memory and 10Gbps virtual network. Each experiment is repeated for at least five times and we report the average. We verify the speed difference between the fastest run and slowest run is within 5%.

Parser. The overhead of the parser varies based on the layers

of headers in the packets: the more layers, the longer it takes to identify the whole header stack. The number of fields being extracted and filter constraints do not matter as much.

To get the throughput of a parser, we designed a controlled evaluation. Based on the packet formats in Table 1, we generated random packet traces and fed them to parsers. Each trace has 80M packets of a given number of header layers. Common TCP packets have the fewest header layers (three – Ethernet, IPv4, and TCP). The most complicated one has eight headers, *i.e.*, ⑤ in Table 1.

Figure 6 shows that in the best case (parsing a common TCP packet), the parser can reach nearly 3.5 Mpps. The throughput decreases when the packets have more header layers. However, even in the most complicated case, a single-thread parser still achieves 2.6 Mpps throughput.

Grouper. For groupers, we find that the average number of summaries in each group is the most impacting factor to grouper performance. To show this, we test different traces in which each group will have one, two, four, or eight packets, respectively. Each trace has 80M packets.

Figure 7 shows that the grouper throughput increases when each group has more packets. This is because the grouper uses a hash table to store the groups in the moving window (§5.2). The more packets in each group, the less group entry inserts and hash collisions. In the worst case (each packet is a group by itself), the throughput of one grouper thread can still reach more than 1.2 Mpps.

Query processor. The query processor performs the query function written by network operators against each summary group. Of course, the query overhead can vary significantly depending on the operators’ needs. We evaluate four typical queries that represent two main types of analysis: 1) loop detection and SLB profiler only check the size of each group (§4.1); 2) the misrouting analysis and drop localization must examine every packet in a group.

Figure 8 shows that the query throughput of the first type can reach 17 or 23 Mpps. The second type is significantly slower – the processor runs at 1.5 Mpps per thread.

6.3 End-to-End Performance

We evaluate the end-to-end performance of dShark by using a real trace with more than 640M packets collected from production networks. Unless otherwise specified, we run the loop detection example shown in §4.1.

Our first target is the throughput requirement in §3: 3.33 Mpps per server. Based on the component throughput, we start two parser instances and three grouper instances on one VM. Groupers spawn query processor threads on demand. Figure 9 shows dShark achieves 3.5 Mpps throughput. This is around three times a grouper performance (Figure 7), which means groupers run in parallel nicely. The CPU overhead is merely four CPU cores. Among them, three cores are used by groupers and query processors, while the remaining core is used by parsers. The total memory usage is around 15 GB.

On the same setup, the drop localizer query gets 3.6 Mpps with similar CPU overhead. This is because, though the query function for drop localizer is heavier, its grouping has more packets per group, leading to lighter overhead (Figure 7).

We further push the limit of dShark on a single 16-core server. We start 6 parsers and 9 groupers, and achieve 10.6 Mpps throughput with 12 out of 16 CPU cores fully occupied. This means that even if the captured traffic is comprised of 70% 64B small packets and 30% 1500B packets, dShark can still keep up with 40Gbps live capturing.

Finally, dShark must scale out across different servers. Compared to running on a single server, the additional overhead is that the shuffling phase between parsers and groupers will involve networking I/O. We find that this overhead has little impact on the performance – Figure 9 shows that when running two parsers and three groupers on each server, dShark achieves 13.2 Mpps on four servers and 26.4 Mpps on eight servers. This is close to the numbers of perfectly linear speedup 14 Mpps and 28 Mpps, respectively. On a network with full bisection bandwidth, where traffic is limited by the host access links, this is explained because we add parsers and groupers in the same proportion, and the hashing in the shuffle achieves an even distribution of traffic among them.

7 Discussion and Limitations

Complicated mappings in multi-hop packet traces. In multi-hop analysis, dShark assumes that at any switch or middlebox, there exist 1:1 mappings between input and output packets, if the packet is not dropped. This is true in most parts of our networks. However, some layer 7 middleboxes may violate this assumption. Also, IP fragmentation can also make troubles – some fragments may not carry the TCP header and break analysis that relies on TCP sequence number. Fortunately, IP fragmentation is not common in our networks because most servers use standard 1500B MTU while our switches are configured with larger MTU.

We would like to point out that it is not a unique problem of dShark. Most, if not all, state-of-art packet-based diagnosis tools are impacted by the same problem. Addressing this challenge is an interesting future direction.

Alternative implementation choices. We recognize that there are existing distributed frameworks [12, 15, 64] designed for big data processing and may be used for analyzing packet traces. However, we decided to implement a clean-slate design that is specifically optimized for packet trace analysis. Examples include the zero-copy data passing via pointers between parsers and trace collectors, and between groupers and query processors. Also, existing frameworks are in general heavyweight since they have unnecessary functionalities for us. That said, others may implement dShark language and programming model with less lines of code using existing frameworks, if performance is not the top priority.

Offloading to programmable hardware. Programmable hardware like P4 switches and smart NICs may offload dShark

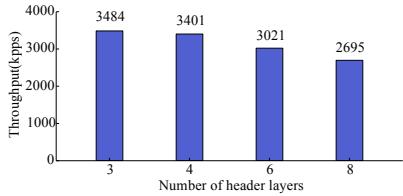


Figure 6: Single parser performance with different packet headers.

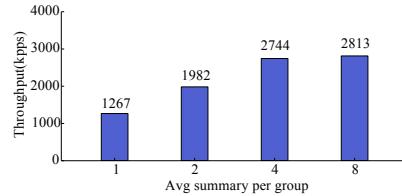


Figure 7: Single grouper performance with different average group sizes.

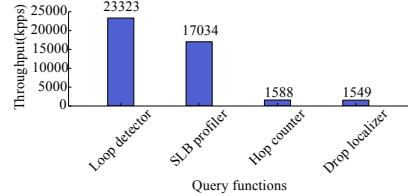


Figure 8: Single query processor performance with different query functions.

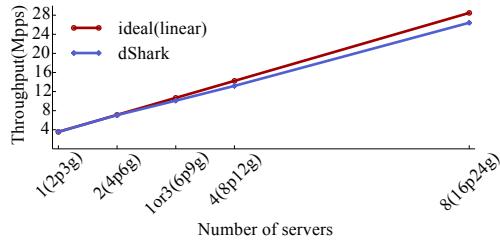


Figure 9: dShark performance scales near linearly.

from CPU for better performance. However, dShark already delivers sufficient throughput for analyzing 40Gbps online packet captures per server (§6) in a practical setting. Meanwhile, dShark, as a pure software solution, is more flexible, has lower hardware cost, and provides operators a programming interface they are familiar with. Thus, we believe that dShark satisfies the current demand of our operators. That said, in an environment that is fully deployed with highly programmable switches,⁶ it is promising to explore hardware-based trace analysis like Marple [42].

8 Related Work

dShark, to the best of our knowledge, is the first framework that allows for the analysis of distributed packet traces in the face of noise, complex packet transformations, and large network traces. Perhaps the closest to dShark are PathDump [56] and SwitchPointer [57]. They diagnose problems by adding metadata to packets at each switch and analyzing them at the destination. However, this requires switch hardware modification that is not widely available in today’s networks. Also, in-band data shares fate with the packets, making it hard to diagnose problems where packets do not reach the destination.

Other related work that has been devoted to detection and diagnosis of network failures includes:

Switch hardware design for telemetry [21, 28, 32, 36, 42]. While effective, these work require infrastructure changes that are challenging or even not possible due to various practical reasons. Therefore, until these capabilities are mainstream, the need to for distributed packet traces remains. Our summaries may resemble NetSight’s postcards [21], but postcards are fixed, while our summaries are flexible, can handle transformations, and are tailored to the queries they serve.

Algorithms based on inference [3, 8, 19, 20, 22, 38, 40, 53, 54, 68]. A number of works use anomaly detection to find the

⁶Unfortunately, this can take some time before happening. In some environments, it may never happen.

source of failures within networks. Some attempt to cover the full topology using periodic probes [20]. However, such probing results in loss of information that often complicates detecting certain types of problems which could be easily detected using packet traces from the network itself. Other such approaches, *e.g.*, [38, 40, 53, 54], either rely on the packet arriving endpoints and thus cannot localize packet drops, or assume specific topology. Work such as EverFlow [68] is complementary to dShark. Specifically, dShark’s goal is to analyze distributed packet captures fed by Everflow. Finally, [7] can only identify the general type of a problem (network, client, server) rather than the responsible device.

Work on detecting packet drops. [11, 16, 17, 23–25, 29, 33, 37, 39, 41, 46, 60, 63, 65–67] While these work are often effective at identifying the cause of packet drops, they cannot identify other types of problems that often arise in practice *e.g.*, load imbalance. Moreover, as they lack full visibility into the network (and the application) they often are unable to identify the cause of problems for specific applications [6].

Failure resilience and prevention [4, 9, 10, 18, 27, 30, 34, 35, 47, 48, 51, 55, 62] target resilience or prevention to failures via new network architectures, protocols, and network verification. dShark is complementary to these works. While they help avoid problematic areas in the network, dShark identifies where these problems occur and their speedy resolution.

9 Conclusion

We present dShark, a general and scalable framework for analyzing in-network packet traces collected from distributed devices. dShark provides a programming model for operators to specify trace analysis logic. With this programming model, dShark can easily address complicated artifacts in real world traces, including header transformations and packet capturing noise. Our experience in implementing 18 typical diagnosis tasks shows that dShark is general and easy to use. dShark can analyze line rate packet captures and scale out to multiple servers with near-linear speedup.

Acknowledgments

We thank our shepherd, Anja Feldmann, and the anonymous reviewers for their insightful comments. Da Yu was partly funded by NSF grant CNS-1320397.

References

- [1] Data plane development kit (DPDK). <http://dpdk.org/>, 2018. Accessed on 2018-01-25.

- [2] Wireshark. <http://www.wireshark.org/>, 2018. Accessed on 2018-01-25.
- [3] ADAIR, K. L., LEVIS, A. P., AND HRUSKA, S. I. Expert network development environment for automating machine fault diagnosis. In *SPIE Applications and Science of Artificial Neural Networks* (1996), pp. 506–515.
- [4] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., MATUS, F., PAN, R., YADAV, N., VARGHESE, G., ET AL. CONGA: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 503–514.
- [5] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., OUTHRED, G., AND LOO, B. T. Closing the network diagnostics gap with vigil. In *Proceedings of the SIGCOMM Posters and Demos* (New York, NY, USA, 2017), SIGCOMM Posters and Demos ’17, ACM, pp. 40–42.
- [6] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., OUTHRED, G., AND LOO, B. T. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association.
- [7] ARZANI, B., CIRACI, S., LOO, B. T., SCHUSTER, A., AND OUTHRED, G. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 440–453.
- [8] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 13–24.
- [9] BODÍK, P., MENACHE, I., CHOWDHURY, M., MANI, P., MALTZ, D. A., AND STOICA, I. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM* (2012), pp. 431–442.
- [10] CHEN, G., LU, Y., MENG, Y., LI, B., TAN, K., PEI, D., CHENG, P., LUO, L. L., XIONG, Y., WANG, X., ET AL. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC* (2016).
- [11] CHEN, Y., BINDEL, D., SONG, H., AND KATZ, R. H. An algebraic approach to practical and scalable overlay network monitoring. *ACM SIGCOMM Computer Communication Review* 34, 4 (2004), 55–66.
- [12] CHOHTIA, Z., LIAGOURIS, J., DIMITROVA, D., AND ROSCOE, T. Online reconstruction of structural information from datacenter logs. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys ’17, ACM, pp. 344–358.
- [13] CLAISE, B., TRAMMELL, B., AND AITKEN, P. RFC7011: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. <https://tools.ietf.org/html/rfc7011>, Sept. 2013.
- [14] CLAISE, B., E. RFC3954: Cisco Systems NetFlow Services Export Version 9. <https://tools.ietf.org/html/rfc3954>, Oct. 2004.
- [15] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.
- [16] DUFFIELD, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory* 52, 12 (2006), 5373–5388.
- [17] DUFFIELD, N. G., ARYA, V., BELLINO, R., FRIEDMAN, T., HOROWITZ, J., TOWSLEY, D., AND TURLETTI, T. Network tomography from aggregate loss reports. *Performance Evaluation* 62, 1 (2005), 147–163.
- [18] FOGL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *NSDI, 12th USENIX Symposium on Networked Systems Design and Implementation* (2015), USENIX.
- [19] GHASEMI, M., BENSON, T., AND REXFORD, J. RINC: Real-time Inference-based Network diagnosis in the Cloud. Tech. rep., Princeton University, 2015. <https://www.cs.princeton.edu/research/techreps/TR-975-14>.
- [20] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM* (2015), pp. 139–152.
- [21] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND McKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 71–85.
- [22] HELLER, B., SCOTT, C., McKEOWN, N., SHENKER, S., WUNDSAM, A., ZENG, H., WHITLOCK, S., JEYAKUMAR, V., HANDIGOL, N., MCCUALEY, J., ET AL. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM HotSDN* (2013), pp. 37–42.
- [23] HERODOTOU, H., DING, B., BALAKRISHNAN, S., OUTHRED, G., AND FITTER, P. Scalable near real-time failure localization of data center networks. In *ACM KDD* (2014), pp. 1689–1698.
- [24] HUANG, Y., FEAMSTER, N., AND TEIXEIRA, R. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 53–58.
- [25] KANDULA, S., KATABI, D., AND VASSEUR, J.-P. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM MineNet* (2005), pp. 173–178.
- [26] KAZEMIAN, P., CHAN, M., ZENG, H., VARGHESE, G., McKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013), pp. 99–111.
- [27] KAZEMIAN, P., VARGHESE, G., AND McKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012), vol. 12, pp. 113–126.
- [28] KIM, C., PARAG BHIDE, E. D., HOLBROOK, H., GHANWANI, A., DALY, D., HIRA, M., AND DAVIE, B. In-band Network Telemetry (INT). <https://p4.org/assets/INT-current-spec.pdf>, June 2016.
- [29] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. IP fault localization via risk modeling. In *USENIX NSDI* (2005), pp. 57–70.
- [30] KUŽNIAR, M., PEREŠINI, P., VASIĆ, N., CANINI, M., AND KOSTIĆ, D. Automatic failure recovery for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 159–160.
- [31] LANGLEY, A., RIDDOCH, A., WILK, A., VICENTE, A., KRASIC, C., ZHANG, D., YANG, F., KOURANOV, F., SWETT, I., IYENGAR, J., BAILEY, J., DORFMAN, J., ROSKIND, J., KULIK, J., WESTIN, P., TENNETI, R., SHADE, R., HAMILTON, R., VASILIEV, V., CHANG, W.-T., AND SHI, Z. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM ’17, ACM, pp. 183–196.
- [32] LI, Y., MIAO, R., KIM, C., AND YU, M. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI* (2016), pp. 311–324.
- [33] LIU, C., HE, T., SWAMI, A., TOWSLEY, D., SALONIDIS, T., AND LEUNG, K. K. Measurement design framework for network tomography using fisher information. *ITA AFM* (2013).
- [34] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 599–613.

- [35] LIU, J., PANDA, A., SINGLA, A., GODFREY, B., SCHAPIRA, M., AND SHENKER, S. Ensuring connectivity via data plane mechanisms. In *USENIX NSDI* (2013), pp. 113–126.
- [36] LIÚ, Y., MIAO, R., KIM, C., AND YUÚ, M. LossRadar: Fast detection of lost packets in data center networks. In *ACM CoNEXT* (2016), pp. 481–495.
- [37] MA, L., HE, T., SWAMI, A., TOWSLEY, D., LEUNG, K. K., AND LOWE, J. Node failure localization via network tomography. In *ACM SIGCOMM IMC* (2014), pp. 195–208.
- [38] MAHAJAN, R., SPRING, N., WETHERALL, D., AND ANDERSON, T. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 106–119.
- [39] MATHIS, M., HEFFNER, J., O’NEIL, P., AND SIEMSEN, P. Pathdiag: Automated TCP diagnosis. In *PAM* (2008), pp. 152–161.
- [40] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM ’16, ACM, pp. 129–143.
- [41] MYSORE, R. N., MAHAJAN, R., VAHDAT, A., AND VARGHESE, G. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC* (2014), pp. 255–267.
- [42] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 85–98.
- [43] NARAYANA, S., TAHHASBI, M., REXFORD, J., AND WALKER, D. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 207–222.
- [44] NELSON, T., YU, D., LI, Y., FONSECA, R., AND KRISHNAMURTHI, S. Simon: Scriptable interactive monitoring for sdns. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (New York, NY, USA, 2015), SOSR ’15, ACM, pp. 19:1–19:7.
- [45] NEWMAN, L. H. How a tiny error shut off the internet for parts of the us. *Wired* (Nov 2017). Accessed Jan 1st, 2018.
- [46] OGINO, N., KITAHARA, T., ARAKAWA, S., HASEGAWA, G., AND MURATA, M. Decentralized boolean network tomography based on network partitioning. In *IEEE/IFIP NOMS* (2016), pp. 162–170.
- [47] PAASCH, C., AND BONAVENTURE, O. Multipath TCP. *Communications of the ACM* 57, 4 (2014), 51–57.
- [48] PORTS, D. R. K., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI* (2015), pp. 43–57.
- [49] RAICIU, C., PAASCH, C., BARRE, S., FORD, A., HONDA, M., DUCHENE, F., BONAVENTURE, O., AND HANDLEY, M. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX Association, pp. 399–412.
- [50] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM ’14, ACM, pp. 407–418.
- [51] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 109–114.
- [52] RIZZO, L. Netmap: a novel framework for fast packet i/o. In *USENIX ATC* (2012).
- [53] ROY, A., BAGGA, J., ZENG, H., AND SNEOREN, A. Passive realtime datacenter fault detection. *ACM NSDI* (2017).
- [54] ROY, A., BAGGA, J., ZENG, H., AND SNEOREN, A. Passive realtime datacenter fault detection. In *ACM NSDI* (2017).
- [55] SCHIFF, L., SCHMID, S., AND CANINI, M. Ground control to major faults: Towards a fault tolerant and adaptive SDN control network. In *IEEE/IFIP DSN* (2016), pp. 90–96.
- [56] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *OSDI* (2016), pp. 233–248.
- [57] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 453–456.
- [58] TOUCH, J. RFC6864: Updated Specification of the IPv4 ID Field. <https://tools.ietf.org/html/rfc6864>, Feb. 2013.
- [59] WANG, M., LI, B. L., AND LI, Z. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In *IEEE ICDCS* (2004), pp. 628–635.
- [60] WIDANAPATHIRANA, C., LI, J., SEKERCOGLU, Y. A., IVANOVICH, M., AND FITZPATRICK, P. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *IEEE UCC* (2011), pp. 261–266.
- [61] WU, W., AND DEMAR, P. Wirecap: A novel packet capture engine for commodity nics in high-speed networks. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC ’14, ACM, pp. 395–406.
- [62] WUNDSAM, A., MEHMOOD, A., FELDMANN, A., AND MAENNEL, O. Network troubleshooting with mirror VNets. In *IEEE GLOBECOM* (2010), pp. 283–287.
- [63] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *USENIX NSDI* (2011).
- [64] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65.
- [65] ZHANG, Y., BRESLAU, L., PAXSON, V., AND SHENKER, S. On the characteristics and origins of internet flow rates. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 309–322.
- [66] ZHANG, Y., ROUGHAN, M., WILLINGER, W., AND QIU, L. Spatio-temporal compressive sensing and internet traffic matrices. *ACM SIGCOMM Computer Communication Review* 39, 4 (2009), 267–278.
- [67] ZHAO, Y., CHEN, Y., AND BINDEL, D. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review* 36, 4 (2006), 219–230.
- [68] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., ET AL. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015), pp. 479–491.
- [69] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., AND ZHENG, H. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM ’15, ACM, pp. 479–491.

Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks

Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, Amin Vahdat
Google, Inc.

Abstract

Clos topologies have been widely adopted for large-scale data center networks (DCNs), but it has been difficult to support incremental expansions for Clos DCNs. Some prior work has claimed that the structure of Clos topologies hinders incremental expansion.

We demonstrate that it is indeed possible to design expandable Clos DCNs, and to expand them while they are carrying live traffic, without incurring packet loss. We use a layer of patch panels between blocks of switches in a Clos DCN, which makes physical rewiring feasible, and we describe how to use integer linear programming (ILP) to minimize the number of patch-panel connections that must be changed, which makes expansions faster and cheaper. We also describe a block-aggregation technique that makes our ILP approach scalable. We tested our “minimal-rewiring” solver on two kinds of fine-grained expansions using 2250 synthetic DCN topologies, and found that the solver can handle 99% of these cases while changing under 25% of the connections. Compared to prior approaches, this solver (on average) reduces the average number of “stages” per expansion from 4 to 1.29, and reduces the number of wires changed by an order of magnitude or more – a significant improvement to our operational costs, and to our exposure (during expansions) to capacity-reducing faults.

1 Introduction

Large-scale Cloud and Internet-application providers are building many data centers, which can contain tens of thousands of machines, consuming tens of MW. These need large-scale high-speed data-center networks. Historically, these networks were built all at once, at the time of cluster commissioning. However, these data centers are filled with servers and storage gradually, often taking 1-2 years to reach capacity. This mismatch leaves substantial capacity idle, waiting for workloads to arrive. Idle capacity not only costs money, but also lengthens the technology-refresh cycle, which can decrease usable compute capacity – the latest servers are hobbled if they must use old network technol-

ogy that lacks modern congestion-control schemes, speed increases, and latency improvements. Hence, we usually start by building a moderate-scale network, and then continually expand the network just ahead of server arrival – *while the network is carrying live traffic*. Live incremental expansion can save millions of dollars in network costs while (more importantly) providing the best possible support for compute and storage infrastructure. However, a naive approach can itself create large, unnecessary costs.

Clos topologies are the *de-facto* standard DCN architecture because they support large-scale DCNs from commodity switches [27, 5, 10, 13]. At Google, our *Jupiter* DCNs are Clos topologies. Different variants of Clos DCN structures, such as Fat Tree [1], VL2 [13], F10 [23], Aspen Tree [30], Rotation Striping [33], etc. have been proposed. However, none of these topologies supports fine-grained incremental expansion. First, some Clos topologies (e.g., Fat Tree) can only be built at certain sizes, which fundamentally prevents incremental expansion. Second, even though some of the Clos topologies (e.g., Rotation Striping) can be constructed at arbitrary sizes, incremental expansion can be expensive, because it requires changing a large fraction of the wiring (see §5.3). In fact, [28] has claimed that the structure of Clos topologies hinders fine-grained incremental expansion; this was an explicit motivation for less-structured topologies that can also exploit commodity switches, such as Jellyfish [28] and Random Folded Clos [4]. In this paper, we show that fine-grained incremental expansion of Clos DCNs is, in fact, feasible, with a novel topology-design methodology.

We want to expand a network *live*: without taking it out of service, which would *strand* compute and storage capacity because those machines would not be usable during expansion, and which would also require us to stop or migrate the applications using that network – a disruptive and expensive process. Live expansion requires maintaining sufficient network throughput during the entire course of a live expansion; to avoid congestion, we must therefore do each expansion in multiple automated stages, each of which only disconnects and adds a limited subset of the network elements. We would

also like to complete each stage as quickly as possible, since rewiring does reduce our spare capacity, and thus exposes us to an increased risk of simultaneous failures.

Over the course of a multi-stage expansion, we may need to rewire many links. If we were to directly connect links between switches, the resulting manual labor for moving long wires would be slow, expensive, and error-prone. Instead, we introduce a patch-panel layer in our Clos DCNs (see Fig. 1). These DCNs are three-tier Clos topologies, with tier-1 top-of-rack (ToR) switches connected to tier-2 server blocks, each of which connects to a set of tier-3 spine blocks. By connecting all the server blocks and all the spine blocks through a group of patch-panels, a DCN topology can thus be created and modified by simply moving fiber jumpers on the back side of the patch panels. Each series of rewiring steps can hence be done in proximity to a single patch panel, although an entire stage may require touching several panels.

Our scale has grown to the point where a simple version of this patch-panel-based expansion technique is too slow to support the rate at which we must execute expansions. Therefore, we needed to minimize the number of rewirings per expansion, while maintaining bandwidth guarantees.

The primary contribution of this paper is a minimal-rewiring solver for Clos DCN topology design. In the literature, most Clos DCN topologies are designed purely to optimize cost and/or performance at a single chosen size [1, 13, 33, 23, 30]. In contrast, our solver explicitly considers the pre-existing topology when designing a larger one. Our solver uses Integer Linear Programming (ILP) to directly minimize the total number of rewirings. By enforcing a number of balance-related constraints, the resulting topology is also guaranteed to have high capacity and high failure resiliency. With minimal rewiring, a DCN expansion can be done in fewer stages, while still maintaining high residual bandwidth during expansions.

Because we build each DCN incrementally over a period of years, we need to incorporate new technologies incrementally via expansions, such as higher-radix switches or faster links. Our ILP-based formulation incorporates various heterogeneities, including different physical structures, switch radices, port speeds, etc., inside a single DCN.

ILP is NP-hard in general, and does not scale well for large-scale DCNs. It may take hundreds of thousands of integer decision variables to formulate a large-scale DCN. Even the most advanced commercial solver, Gurobi [17], might run for days without computing a solution. We tested a simple version of our ILP-based solver on 4500 synthesized DCN configurations, and found that the solver failed to solve 68% of the configurations within a 3-hour limit. (Longer timeouts yield little improvement.)

To make our solver scale, we developed a block-aggregation technique to reduce the number of decision variables in the ILP formulation. Block aggregation exploits various homogeneities in a DCN, and aggregates decision vari-

ables whenever possible. We have a proof that the aggregated decision variables can be decomposed in a later step [32]. Our block-aggregation technique can use different aggregation strategies. With the fastest strategy, all 4500 synthesized DCN configurations can be solved within 10 seconds.

We measure the quality of our solutions in terms of a *rewiring ratio*, the fraction of wires between server blocks and spine blocks in the pre-existing topology that must be disconnected during an expansion. When we use block aggregation, we face a tradeoff: aggregation improves runtime scalability, but sacrifices rewiring optimality. However, we cannot predict the aggregation strategy that will produce the best (lowest) rewiring ratio subject to a chosen deadline. Therefore, our *parallel solver* runs multiple minimal-rewiring solvers with different aggregation strategies at the same time, and picks the solution with the lowest rewiring ratio. This allows us to solve about 99% of the synthesized DCN configurations with a rewiring ratio under 0.25; the median ratio is under 0.05. In turn, these low rewiring ratios allow us to significantly accelerate the entire expansion pipeline. For example, under a constraint that preserves 70% of the pre-expansion bandwidth during expansion, our minimal-rewiring solver reduces the average number of expansion stages required from 4 to 1.29.

2 Prior Work on Expansions

Prior work has described DCN designs that support incremental expansion, and techniques for conducting expansions. Our work focuses on Clos topologies, the de-facto standard for large-scale DCNs; most prior work on expansions has used non-Clos designs.

DCell [15] and BCube [14] are built using iterative structures. As a result, they can only supports expansions at a very coarse granularity, which could lead to substantial stranded DCN capacity after expansion. Similar iteratively-designed DCN structures are also proposed in [16, 21, 22].

JellyFish [28], Space Shuffle [31], Scafida [18], and Expander [29, 9] were designed to support fine-grained incremental expansion using random-graph-based DCN topologies. However, these topologies have not been widely adopted for industrial-scale data centers, possibly due to the increased complexity of cabling and routing and congestion control when deploying large-scale DCNs.

Random Folded Clos [4] is a variant of Clos that supports fine-grained incremental expansion. It maintains a layered structure, but builds inter-layer links based on random graphs. However, Random Folded Clos is only designed for homogeneous DCNs, where all blocks are of the same size and the same port speed. Further, Random Folded Clos is not non-blocking, with reduced capacity when compared to Fat Trees. In contrast, our minimal-rewiring solver can be applied to heterogeneous DCNs, and its topology solution preserves the non-blocking property of Clos topologies.

Similar to our minimal-rewiring DCN topology solver,

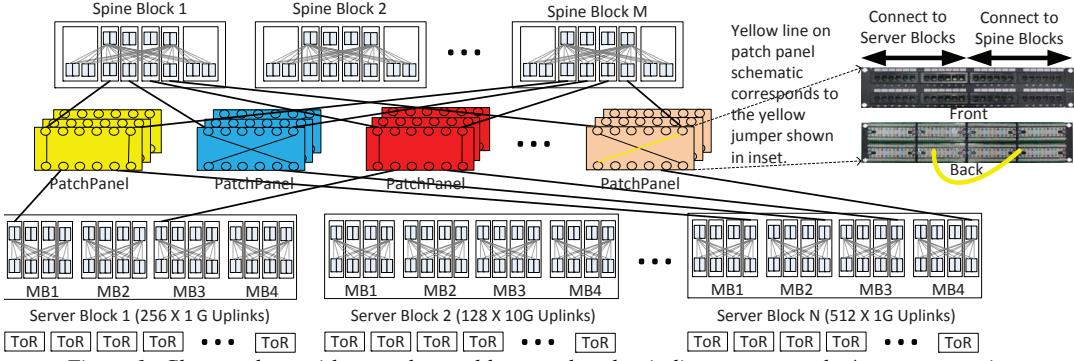


Figure 1: Clos topology with a patch-panel layer, colored to indicate an example 4-stage expansion

optimization-based approaches were also adopted in LEGUP [8] and REWIRE [7]. However, neither paper looked at the topology-design problem in the presence of a patch-panel layer, and could have daunting cabling complexity. Further, LEGUP uses a branch-and-bound algorithm [3], and REWIRE uses simulated annealing [24] to search for the optimal topology. Both algorithms scale poorly, as their convergence time grows exponentially with the problem size.

Condor [25] allowed designers to express goals for DCN topologies via a constraint-based language, and used a constraint solver to generate topologies. Condor addressed the challenge of multi-phase expansions, but their approach was unable to solve expansions for many arbitrary configurations of Clos networks due to computational complexity.

3 Overview of Clos-Based DCN Topology

The Clos topology was initially designed for telephone networks [6]. Years later, it was proposed for DCNs [1] and subsequently became the de-facto standard for large-scale DCNs, at Google [27], Facebook [10], Cisco [5], Microsoft [13], etc.. There are many advantages to Clos networks. First, Clos networks are non-blocking, and thus have high throughput. Second, Clos networks can be built using identical and inexpensive commodity switches, and thus are cost-effective. Third, Clos networks have many of redundant paths for each source-destination pair, and thus are highly failure-resilient.

Clos-based data centers exhibit a layered architecture (see Fig. 1). The lower layer contains a number of *server blocks*¹. The upper layer contains a number of *spine blocks*, used to forward traffic between server blocks. A DCN topology interconnects each server block to each spine block. Because we want to support technology evolution within a single network, each of the server blocks could have a different number of spine-facing uplinks, a different uplink rate, etc².

Connecting server blocks and spine blocks by direct wires³ is highly inefficient. First, a large-scale data center typically has tens of thousands of DCN links. Second, the

server blocks and the spine blocks of a data center may be deployed at different locations on a data center floor, due to space constraints, so some direct links would have to run a long way across the data center. Third, moving, adding, or removing a long link during an expansion requires significant human labor, creates a risk of error, and because the new links might have dramatically different lengths requires a large inventory of cables of various lengths.

To overcome these difficulties, we introduce a patch panel [19, 26] layer between server blocks and spine blocks (Fig. 1). Patch panels are much cheaper than other DCN components (e.g., switches, optical transceivers, long-reach fibers). All the interconnecting ports of the server and spine blocks can be connected to the front side of the patch panels via fiber bundles, and all the connecting links can be established or changed using fiber jumpers on the back side⁴. These patch panels are co-located. As a result, a DCN topology can be wired and modified without walking around the data center floor or requiring the addition or removal of existing fiber. Also, as discussed in [27], deploying fibers in bundles greatly reduces cost and complexity; using patch panels means we can deploy bundles once, without having to change them during an expansion. This patch-panel layer makes it much easier for us to support rapid expansions without excessive capacity reduction.

Patch panels allow us to divide a DCN topology into two layers, *physical* and *logical*. As shown in Fig. 1, each server and spine block is connected to the patch-panel layer; we call this the *physical topology*. When a new block is first deployed, we deploy its corresponding physical topology. Changing physical links is not easy, as it involves moving fiber bundles across different patch panels. Hence, in this paper, readers can view physical links as fixed once deployed.

Logical topology defines how server blocks connect to spine blocks, abstracting out the patch-panel layer. All DCN topologies discussed in literatures refer to the logical topology. Many DCN performance metrics have been defined in terms of logical topologies, including network bandwidth,

¹“Server blocks” are also called “pods” [11], or “edge aggregation blocks” [27].

²Fig. 1 shows three server blocks with different uplink configurations.

³We use the term “wires” to loosely refer to either fiber or copper links.

⁴The inset photo in Fig. 1 depicts how we use these jumpers. The yellow line on the right-most patch panel corresponds to the yellow jumper in the inset, which connects one server-block link to one spine-block link.

failure resiliency, incremental expandability, etc. However, except for Condor [25], no prior work has studied the logical-topology design problem in presence of patch panels. As discussed in §5.2.1, in addition to optimizing the performance metrics listed above, we also need to enforce additional physical constraints, so that the resulting logical topology is compatible with the underlying physical topology.

3.1 Considerations for Clos expansions

During an expansion, the existing physical topology remains unchanged, and we only add new bundles of physical links for the newly added/upgraded server/spine blocks. Note that adding new physical links does not impact any ongoing traffic. In contrast, some of the links in the existing logical topology, which could be carrying significant traffic, will have to change, so we must ensure there is no traffic loss caused by changes to the logical topology.

As shown in Fig. 2 (a), we could add a large amount of capacity to a data center during each expansion, which would allow us to do expansions infrequently. However, this expansion strategy would lead to much more *stranded* network capacity – capacity installed but not usable – as the traffic demand could be far less than the capacity provided. By doing fine-grained expansions, we reduce the mean amount of stranded capacity (see Fig. 2 (b)).

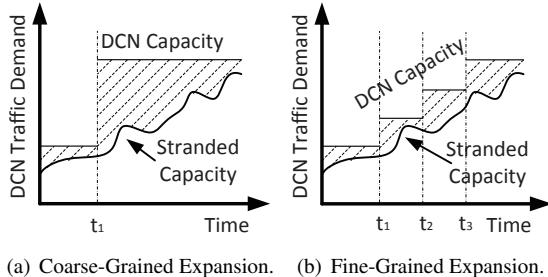


Figure 2: DCN stranded capacity.

4 Patch-Panel Based Expansion Pipeline

We support two types of expansions, at the granularity of a server block (Fig. 3). The first type adds a new server block, to allow more servers to be added to an existing data center. The second type increases the uplink count (“*radix*”) of an existing server block. Typically, the uplinks of a server block are not initially fully populated with optical transceivers, because transceivers are expensive and a new server block has a low bandwidth requirement (as not all of its servers are connected). As a block’s bandwidth requirement increases, we need to populate more uplinks. Note that as we expand the server-block layer, additional spine blocks will also be needed, so expansions generally involve adding server blocks and spine blocks at the same time.

Fig. 4 depicts our pipeline for updating a logical topology during a live expansion. It guarantees that:

- No traffic loss due to routing packets into “black holes.”

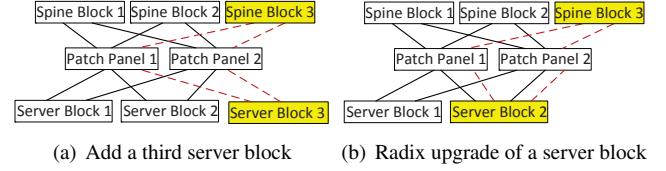


Figure 3: Data center expansion types.

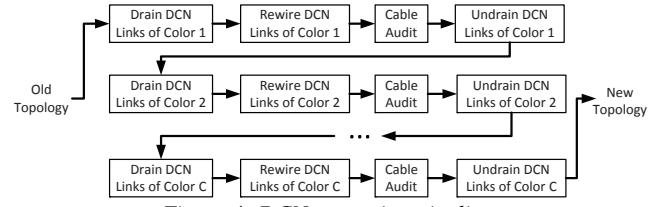


Figure 4: DCN expansion pipeline.

- All wiring changes are made correctly.
- No congestion due to the temporary capacity reduction.

To avoid black holes when changing a set of logical links, we must redirect traffic on these links to other paths. We instruct our SDN-based control plane to “drain” these links. After we verify that there is no longer traffic on the target drained links, we can proceed to rewire the links.

Rewiring links via patch-panel changes is the most labor-intensive and error-prone step. Typically, thousands of links need to be rewired during one expansion, creating the possibility of multiple physical mistakes during the rewiring process. To check for errors, we perform a *cable audit*. In cable audit, we use the Link Layer Discovery Protocol to construct a post-wiring topology, and then cross-validate this against the target logical topology. We also run a bit-error-rate test (BERT) for all the new links, to detect links with issues. This audit results in automated tickets to repair faulty links, followed by a repeat of the audit.

During DCN expansion, we must drain some fraction of the logical links. While Clos networks are resilient to some missing links, draining too many links simultaneously could result in dropping the network’s available capacity below traffic demand. We therefore set a residual-capacity threshold, based on measured demand plus some headroom. We then divide an expansion into stages such that, during each stage, the residual capacity remains above this threshold. In our original approach, we partitioned the set of patch panels into C groups (as illustrated by different colors in Fig. 1), and only rewired links in one group of patch panels per stage. Then, at each stage, we would still have approximately $1 - 1/C$ of the pre-expansion capacity available.

Note that the expansion pipeline migrates the network, in stages, from an existing (old) topology to some new logical topology that connects the new blocks to the existing ones, subject to a set of constraints on the logical connectivity (§5.2.1 formalizes those constraints). However, there are many ways to construct a logical topology that meets our constraints, and our original, simple solution to these

constraints typically required a lot of rewiring for an expansion. This, in turn, forced us to divide expansions into C stages, to ensure that we preserved at least $1 - 1/C$ of the pre-expansion capacity.

As a result, these expansions took a long time, especially if the pre-expansion network was highly utilized. In our experience, each stage takes considerable time, including draining, rewiring, cable auditing, BERTing, and undraining. If we were to expand a network with a requirement to preserve 90% residual capacity, at least 10 stages would be needed⁵.

The root cause of the length of our original expansion pipeline is that it does not attempt to optimize the difference (in terms of wiring changes) between pre-expansion and post-expansion topologies. If we were able to minimize this difference, we could finish an expansion in fewer stages. This is related to the “incremental expansibility” property of a DCN topology. This property is easy to achieve for random-graph topologies such as Jellyfish [28]. However, for Clos networks, none of the existing topology solutions exhibits this property. This motivated us to look for a better approach, which we describe in the rest of this paper.

5 Minimal Rewiring for DCN Expansion

In this section, we describe a new “minimal rewiring” topology-design methodology for Clos networks, which not only achieves high network capacity and failure resiliency, but also minimizes rewiring for expansions. Our approach relies on Integer Linear Programming (ILP) to compute a logical topology. Our results compare well to existing Clos topology solutions [27, 5, 13, 33].

While we initially considered an ILP formulation at the granularity of switch and patch-panel ports, the resulting scale made solving infeasible. Instead, we use multiple phases, first solving a block-level ILP formulation (§5.2) and then performing a straightforward mapping onto a port-level solution (§5.4).

5.1 Definitions and notations

To rigorously formulate the minimal-rewiring topology-design problem, we introduce the following definitions and mathematical notations.

Server Block: We use a server block as a unit of deploying network-switching capacity. On the order of 1000 servers can be connected to a server block via ToR switches. To avoid a single point of failure, we further divide a server block into independent *middle blocks*, typically four⁶ (see Fig. 1); each middle block is controlled by a different routing controller. The uplinks of each ToR switch are evenly spread among the middle blocks in its server block. Even if one middle block is down, servers in the server block are

⁵In fact, since failed links and inexact load balance can cause topology imperfections, more than 10 stages would be required to ensure 90% residual capacity.

⁶Our results hold for any number of middle blocks.

Table 1: Notations used in this paper

E_n, S_m, O_k	Server block n , spine block m , patch panel k
E_n^t	Middle block t in server block n
$G_k(E_n^t)$	Physical link-count for E_n^t via patch panel k
$G_k(S_m)$	Physical link-count for S_m via patch panel k
$b_k(E_n^t, S_m)$	Reference logical-topology link-count between E_n^t and S_m via patch panel O_k
$d_k(E_n^t, S_m)$	Desired logical-topology link-count between E_n^t and S_m via patch panel O_k
$p_{n,m}$	Mean number of links between a server block and a spine block
$q_{n,m}^t$	Mean number of links between a middle block and a spine block
n_g	Server block group index; or a set containing all the server block indices in the n_g -th group
m_g	Spine block group index; or a set containing all the spine block indices in the m_g -th group
k_g	Patch panel group index; or a set containing all the patch panel indices in the k_g -th group
$E_{n_g}, S_{m_g}, O_{k_g}$	Server block group n_g , spine block group m_g , patch panel group k_g
$E_{n_g}^t$	All the t -th middle blocks in E_{n_g}
$b_{k_g}(E_{n_g}^t, S_{m_g})$	Reference logical-topology link-count between $E_{n_g}^t$ and S_{m_g} via patch panel group O_{k_g}
$d_{k_g}(E_{n_g}^t, S_{m_g})$	Desired logical-topology link-count between $E_{n_g}^t$ and S_{m_g} via patch panel group O_{k_g}
x^+	$\max\{0, x\}$

still accessible via its other middle blocks. We assume that a DCN has N server blocks, each of which is represented by $E_n, n = 1, 2, \dots, N$. We denote the middle blocks within server block E_n by $E_n^t, t = 1, 2, 3, 4$.

Spine Block: Spine blocks forward traffic among different server blocks. We use $S_m, m = 1, 2, \dots, M$ to represent a spine block, where M is the total number of spine blocks.

Physical Topology: Assume there are K patch panels, each of which is represented by $O_k, k = 1, 2, \dots, K$. We use $G_k(E_n^t)$ to represent the total number of physical links between the middle block E_n^t and the patch panel O_k . Then, the physical topology of server block E_n can be characterized by $\{G_k(E_n^t), k = 1, \dots, K, t = 1, 2, 3, 4\}$ ⁷.

Similarly, we use $G_k(S_m)$ to represent the total number of physical links between spine block S_m and patch panel O_k . Then, the physical topology of spine block S_m can be characterized by $\{G_k(S_m), k = 1, \dots, K\}$.

Note that our networks are heterogeneous: different server blocks and spine blocks could have different physical topologies (see Table 2 for details).

Reference Topology: Our goal is to minimize rewiring with respect to the old logical topology, called the *reference topology*. We let $b_k(E_n^t, S_m)$ be the total number of reference-topology links between middle block E_n^t and spine block S_m

⁷Here we use the term “topology” somewhat loosely to describe the cardinality of the connectivity between a block and a set of patch panels, rather than to describe either the detailed inter-block topology, or the internal topology within a block composed of multiple commodity switches.

that connect via patch panel O_k . Since the new server and spine blocks do not exist in the reference topology, for these blocks we simply set $b_k(E_n^t, S_m) = 0$.

Logical Topology: Our objective is to compute a new logical topology for the given physical topology. We use $d_k(E_n^t, S_m)$ to represent the total number of logical links between middle block E_n^t and spine block S_m that connect via patch panel O_k . As we will show shortly, as long as $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$ satisfies a set of physical topology constraints, a polynomial-time algorithm can be used to map $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$ to point-to-point configurations in the patch panels. Hence, the objective of the block-level ILP formulation is to compute $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$.

5.2 ILP Formulation of Minimal Rewiring

Our ILP formulation consists of a set of constraints (§5.2.1) and an objective (§5.2.2).

5.2.1 Constraints for Logical Topology

Recall that our objective is to compute $\{d_k(E_n^t, S_m)\}_{k,m,n,t}$. We must impose a set of constraints on the solution, not only to ensure the compatibility of the logical topology with the underlying physical topology, but also to guarantee both high throughput and high failure resiliency.

Physical Topology Constraints: Recall that $d_k(E_n^t, S_m)$ is the total number of logical links between middle block E_n^t and spine block S_m connected via patch panel O_k . Clearly, it must be no larger than the total number of physical links $G_k(E_n^t)$ between O_k and E_n^t , and the total number of physical links $G_k(S_m)$ between O_k and S_m , i.e.,

$$0 \leq d_k(E_n^t, S_m) \leq \min\{G_k(E_n^t), G_k(S_m)\}. \quad (1)$$

To ensure high uplink bandwidth for all the server blocks, we require that each “populated” server block port⁸ must connect to a spine block port. This is guaranteed by

$$\sum_{m=1}^M d_k(E_n^t, S_m) = G_k(E_n^t). \quad (2)$$

Note that constraint (2) requires that the total number of spine block ports must be no smaller than the total number of server block ports on each patch panel. Hence, it is possible that a physically-connected spine block port might not connect to any server block port, which can be expressed as:

$$\sum_{n=1}^N \sum_{t=1}^4 d_k(E_n^t, S_m) \leq G_k(S_m). \quad (3)$$

Capacity Constraints: In order to achieve high DCN capacity, which depends on load balance, we require the uplinks of each server block to be evenly distributed among all spine blocks. Specifically,

$$\lfloor p_{n,m} \rfloor \leq \sum_{k=1}^K \sum_{t=1}^4 d_k(E_n^t, S_m) \leq \lceil p_{n,m} \rceil, \quad (4)$$

⁸Recall that, for cost reasons, we do not initially populate all the ports.

where $p_{n,m}$ is the mean number of links between a server block and a spine block. $p_{n,m} = |E_n| |S_m| / (|S_1| + \dots + |S_M|)$ ⁹, where $|E_n|$ (or $|S_m|$) is the total number of ports in E_n (or S_m), $\lfloor p_{n,m} \rfloor$ is the largest integer that is no larger than $p_{n,m}$, and $\lceil p_{n,m} \rceil$ is the smallest integer that is no smaller than $p_{n,m}$.

Constraint (4) ensures high DCN capacity. In the ideal case where all $p_{n,m}$'s are integers, constraint (4) ensures that traffic between any two server blocks E_{n_1} and E_{n_2} can burst at full rate ($\min\{|E_{n_1}|, |E_{n_2}|\}$). Specifically, E_{n_1} and E_{n_2} can communicate at rate $\min\{p_{n_1,m}, p_{n_2,m}\}$ through the m -th spine block, and thus the total rate would be $\sum_{m=1}^M \min\{p_{n_1,m}, p_{n_2,m}\} = \min\{|E_{n_1}|, |E_{n_2}|\}$.

In the general case where some $p_{n,m}$'s are not integers, there must be some imbalance in the logical topology. Constraint (4) minimizes this imbalance.

Failure-Resiliency Constraints: While commodity switches are highly reliable, an entire middle block can fail as a unit due to software bugs in its routing controller. We also bring down an entire middle block occasionally to upgrade the switch stack softwares. In order for our DCN to be failure-resilient, we need to make sure that throughput remains as high as possible even under middle-block failures. This can be achieved by requiring the middle block links to be evenly distributed among the spine blocks. Specifically,

$$\lfloor q_{n,m}^t \rfloor \leq \sum_{k=1}^K d_k(E_n^t, S_m) \leq \lceil q_{n,m}^t \rceil, \quad (5)$$

where $q_{n,m}^t$ is the mean number of links between a middle block and a spine block¹⁰. $q_{n,m}^t = |E_n^t| |S_m| / (|S_1| + \dots + |S_M|) = p_{n,m}/4$ (assuming 4 middle blocks per server block).

Constraint (5) minimizes the capacity impact under middle block failures. In the ideal case where $q_{n,m}^t$'s are all integers, the throughput impact is exactly 25%. In the general case where some $q_{n,m}^t$'s are not integers, there must be some imbalance in the traffic between the middle blocks and the spine blocks. Constraint (5) minimizes this imbalance. Note that Constraint (5) cannot subsume (4), because all the decision variables are integers.

5.2.2 Minimal-Rewiring ILP Objective

In our ILP-based formulation, it is easy to add a minimal-rewiring objective function. Specifically, our block-level minimal-rewiring solver can be formulated as:

$$\begin{aligned} \min & \sum_{k=1}^K \sum_{n=1}^N \sum_{t=1}^4 \sum_{m=1}^M (b_k(E_n^t, S_m) - d_k(E_n^t, S_m))^+, \\ \text{subject to } & (1) - (5), \end{aligned} \quad (6)$$

where $x^+ = \max\{0, x\}$. This objective function computes the total number of links to be rewired for changing the old (reference) topology to the new topology.

⁹We have assumed, when deriving $p_{n,m}$, that the number of spine block ports is no less than the number of server block ports. In fact, our formulation and the subsequent optimization techniques also apply to the cases where there are fewer spine block ports.

¹⁰Our formulas for p and q require trivial extensions to support heterogeneous link speeds; we omit these for reasons of space.

5.3 Benefits of Minimal Rewiring

We demonstrate the benefit of minimal rewiring using a simple example, comparing our minimal-rewiring approach against the *rotation striping* approach described in [33]. Rotation striping can be used to design Clos topologies for *homogeneous* DCNs of arbitrary size, whereas minimal rewiring works with various block sizes.

Consider a DCN configuration with N server blocks and M spine blocks. Assume that each server block has X ports, and that there is only one patch panel¹¹. Then, rotation striping can be expressed as Algorithm 1.

Input: DCN configuration parameters N, M, X

Output: A DCN topology

- 1 Label all the server block ports with different indices from $1, 2, \dots, NX$. Note that the set of ports $\{(n-1)X+1, (n-1)X+2, \dots, nX\}$ corresponds to the n -th server block.
- 2 Connect port $e \in \{1, 2, \dots, NX\}$ to the $\lceil e/M \rceil$ -th port of the $((e-1)\%M + 1)$ -th spine block.

Algorithm 1: Rotation striping algorithm from [33]

We can quantify the *rewiring ratio* for a solution as the fraction of wires between server blocks and spine blocks in the pre-existing topology that must be disconnected during an expansion procedure.

Consider an expansion in which we add 1 server block and 1 spine block. It is easy to check that with rotation striping, only the first M server-block ports connect to their original peers, and thus the rewiring ratio would be $(NX - M)/NX$.

On the other hand, using minimal-rewiring¹², we can show that only $XN/(M+1)$ links need to be rewired, which corresponds to a rewiring ratio of $1/(M+1)$. This means that even if our expansion is executed in just a single stage, the capacity reduction is just $1/(M+1)$.

To the best of our knowledge, with the exception of Condor [25], none of the prior literature has incorporated patch-panel layer constraints, and Condor’s constraint-satisfaction approach was unable to find solutions in most cases. In practice, we usually need more than one patch panels in order to connect all the server blocks to all the spine blocks, since the number of ports on each patch panel is limited. If one ignores the patch-panel layer, then the resulting DCN topology will be very likely not compatible with the underlying physical topology.

Consider rotation striping in an example with two patch panels (see Fig. 5). Each server block has six links, with three links connecting to the first patch panel and the other three connecting to the second one. There are four spine blocks, with two of them connecting to the first patch panel

¹¹Rotation striping does not consider the patch-panel layer, which is equivalent to setting the number of patch panels to be 1.

¹²Rotation striping can only guarantee the constraints (1)-(4). Hence, for this example, we also only impose those constraints on our minimal rewiring solver.

and the other two connecting to the second one. If we apply rotation striping here, there should be four logical links between the first server block and the first two spine blocks. Note that these four links can only be created through the first patch panel, because the first two spine blocks only connect to the first patch panel. However, this is impossible, as there are only three physical links between the first server block and the first patch panel.

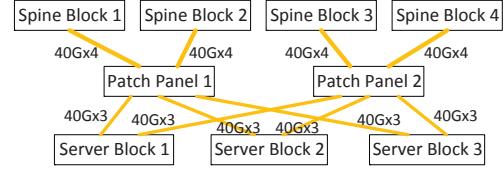


Figure 5: A counterexample for which rotation striping fails.

Our approach incorporates the patch-panel layer via three physical constraints (1)-(3). These three constraints ensure that any solution $\{d_k^*(E_n^t, S_m)\}_{k,m,n,t}$ of (6) can be mapped to port-to-port configurations in the patch panels as we describe below. (Note that (6) may yield multiple solutions.)

5.4 Creating port-to-port mappings

Our ILP formulation tells us the block-level link count between each middle block E_n^t and each spine block S_m in each patch panel O_k , as in (6), but not how individual ports must be connected. We thus developed a straightforward algorithm to compute these port-to-port mappings.

The algorithm’s input consists of the block-level link counts $b_k(E_n^t, S_m)$ and $d_k^*(E_n^t, S_m)$ for the pre-expansion and post-expansion topologies, respectively. We use two passes, both of which iterate over all pairs of middle blocks and spine blocks:

Pass 1: disconnect links as necessary: For each patch panel O_k , note that an expansion changes the link count between middle block E_n^t and spine block S_m from $b_k(E_n^t, S_m)$ to $d_k^*(E_n^t, S_m)$. Therefore, if $b_k(E_n^t, S_m) \leq d_k^*(E_n^t, S_m)$, we simply preserve all pre-existing links; if $b_k(E_n^t, S_m) > d_k^*(E_n^t, S_m)$, we can disconnect any $(b_k(E_n^t, S_m) - d_k^*(E_n^t, S_m))$ of the pre-existing links. This pass disconnects $\sum_{n=1}^N \sum_{t=1}^4 \sum_{m=1}^M (b_k(E_n^t, S_m) - d_k^*(E_n^t, S_m))^+$ links.

Pass 2: connect new links: After the first pass, $\min\{b_k(E_n^t, S_m), d_k^*(E_n^t, S_m)\}$ links remain between E_n^t and S_m . For any block pair E_n^t and S_m with less than $d_k^*(E_n^t, S_m)$ links, we can arbitrarily pick $d_k^*(E_n^t, S_m) - b_k(E_n^t, S_m)$ non-connected ports from E_n^t and S_m respectively, and interconnect them. Feasibility is guaranteed by the physical topology constraints (1)-(3).

5.5 Challenge: Solver Scalability

We use integer linear programming to formulate our minimal-rewiring solver, because our topology-design problem is NP-Complete. Specifically, the decision variables $d_k(E_n^t, S_m)$ contain three dimensions (middle-block dimension E_n^t , spine-block dimension S_m , and patch-panel di-

mension k), and the constraints (2)(3)(5) are essentially for the 2-marginal sums $\sum_k d_k(E_n^t, S_m)$, $\sum_{E_n^t} d_k(E_n^t, S_m)$ and $\sum_{S_m} d_k(E_n^t, S_m)$. In the literature, this is called the Three-Dimensional Contingency Table (3DCT) problem, and has been proven to be NP-Complete [20]. Having failed to find a polynomial-time algorithm for our problem, we decided to use ILP, as there are many readily-available commercial ILP solvers, e.g., Gurobi [17], Google Optimization Tools [12].

However, our problem size is so large that none of the existing commercial solvers scales well. For example, one DCN configuration we evaluate (see §9.7.1) contains 77 server blocks of three kinds, 68 spine blocks of two kinds, and 256 patch panels. Without any optimization, this leads to about 400000 decision variables, and the ILP solver ran for a day without generating a solution. As shown in §9.3, for the 4500 trials we ran without optimization, we could only solve 32% within a 3-hour deadline. (Longer timeouts yield little improvement.)

6 Block Aggregation

To improve the scalability of our minimal-rewiring solver, we developed a block aggregation technique. Block aggregation significantly reduces the total number of decision variables in (6), and thus greatly improves solver scalability.

6.1 ILP Formulation with Block Aggregation

The idea behind block aggregation is to group patch panels, server blocks, spine blocks, and then aggregate decision variables within each group.

Patch-Panel Group: Two patch panels k_1, k_2 belong to the same group if and only if they have the same number of physical links to each middle block and each spine block, i.e., $G_{k_1}(E_n^t) = G_{k_2}(E_n^t)$, $G_{k_1}(S_m) = G_{k_2}(S_m)$ for any n, t, m .

Server-Block Group: Two server blocks n_1, n_2 belong to the same group if and only if they have the same physical topology, i.e., $G_k(E_{n_1}^t) = G_k(E_{n_2}^t)$ for any k, t .

Spine-Block Group: Two spine blocks m_1, m_2 belong to the same group if and only if they have the same physical topology, i.e., $G_k(S_{m_1}) = G_k(S_{m_2})$ for any k .

Assume that these definitions yield K_g patch-panel groups, N_g server-block groups, and M_g spine-block groups. Then, we can define an aggregated decision variable $d_{k_g}(E_n^t, S_m)$ for the k_g -th patch-panel group, the n_g -th server block group, and the m_g -th spine block group as follows:

$$d_{k_g}(E_n^t, S_m) = \sum_{k \in k_g} \sum_{n \in n_g} \sum_{m \in m_g} d_k(E_n^t, S_m). \quad (7)$$

Here, we have abused the notation, and used k_g, n_g , and m_g to represent both the indices and the actual groups.

With the aggregated decision variables $d_{k_g}(E_n^t, S_m)$, the original constraints (1)-(5) also need to be aggregated:

$$0 \leq d_{k_g}(E_n^t, S_m) \leq |k_g| |n_g| |m_g| \min\{G_k(E_n^t), G_k(S_m)\}, \quad (8)$$

$$\sum_{m_g=1}^{M_g} d_{k_g}(E_n^t, S_m) = |k_g| |n_g| G_k(E_n^t), \quad (9)$$

$$\sum_{n_g=1}^{N_g} \sum_{t=1}^4 d_{k_g}(E_n^t, S_m) \leq |k_g| |n_g| G_k(S_m), \quad (10)$$

$$|n_g| |m_g| \lceil p_{n,m} \rceil \leq \sum_{k_g=1}^{K_g} \sum_{t=1}^4 d_{k_g}(E_n^t, S_m) \leq |n_g| |m_g| \lceil p_{n,m} \rceil, \quad (11)$$

$$|n_g| |m_g| \lfloor q_{n,m}^t \rfloor \leq \sum_{k_g=1}^{K_g} d_{k_g}(E_n^t, S_m) \leq |n_g| |m_g| \lceil q_{n,m}^t \rceil. \quad (12)$$

In the aggregated constraints, k, n, m are arbitrary patch panel, server block, and spine block indices, drawn from the k_g -th patch-panel group, the n_g -th server-block group, and the M_g -th spine-block group, respectively. In fact, the values $G_k(E_n^t)$, $G_k(S_m)$, $p_{n,m}$, $q_{n,m}^t$ are all the same, as long as k, n, m are chosen in the same group, respectively. Here, we also view k_g, n_g , and m_g as groups, and have used $|k_g|, |n_g|, |m_g|$ to represent the sizes of these groups.

With block aggregation, we can thus rewrite the optimization problem (6) as follows:

$$\begin{aligned} \min & \sum_{k_g=1}^{K_g} \sum_{n_g=1}^{N_g} \sum_{t=1}^4 \sum_{m_g=1}^{M_g} (b_{k_g}(E_n^t, S_m) - d_{k_g}(E_n^t, S_m))^+, \\ \text{subject to} & (8) - (12), \end{aligned} \quad (13)$$

$$\text{where } b_{k_g}(E_n^t, S_m) = \sum_{k \in k_g} \sum_{n \in n_g} \sum_{m \in m_g} b_k(E_n^t, S_m).$$

Compared to (6), the total number of decision variables in (13) is significantly reduced, from $\Theta(NKM)$ to $\Theta(N_g K_g M_g)$. Thus, the complexity of solving (13) is significantly lower than that of (6).

6.2 Variable Deaggregation

After obtaining a solution $d_{k_g}^*(E_n^t, S_m)$ for (13), we still need to decompose the solution to $d_k^*(E_n^t, S_m)$. Specifically, we need to solve the following problem.

$$\begin{aligned} \min & \sum_{k=1}^K \sum_{n=1}^N \sum_{t=1}^4 \sum_{m=1}^M (b_k(E_n^t, S_m) - d_k(E_n^t, S_m))^+, \\ \text{subject to} & \sum_{k \in k_g} \sum_{n \in n_g} \sum_{m \in m_g} d_k(E_n^t, S_m) = d_{k_g}^*(E_n^t, S_m) \\ & \text{and (1) - (5).} \end{aligned} \quad (14)$$

If there were no constraints (1)-(5) in (14), we could easily compute a solution $d_k^*(E_n^t, S_m)$ in polynomial time using an algorithm similar to the one in §5.4. Because of these constraints, solving (14) becomes much more challenging. In fact, it is not trivial to prove that (14) always has a solution. In the literature, (14) is closely related to the Integer-Decomposition property [2]. In general, the Integer-Decomposition property does not always hold (an example is provided in [32]). Fortunately, thanks to our problem structure, we are able to rigorously prove that (14) indeed has a solution. Specifically, we build an integer decomposition theory specifically for our problem, and prove that a decomposition satisfying (1)-(5) can be found iteratively in polynomial time. The details are available in [32].

Given that (14) has solutions, the next question is to find one that minimizes the objective function. The simplest approach is to directly solve it using integer programming. However, this will destroy the complexity-reduction of block aggregation, as (14) is of exactly the same size as (6).

Our approach is to decompose (14) into $K_g + N_g + M_g$ smaller ILP problems. Specifically, we can decompose patch-panel groups, spine-block groups, and server-block groups by solving three ILPs in separate steps. In each step, different block groups can be completely decoupled, and thus the three ILPs can be further decomposed into K_g , N_g , and M_g ILPs, respectively. These smaller ILP problems are much easier to solve compared to (14). However, as shown in §9.3, computing these smaller ILP problems sequentially can still be slow. To further improve scalability, we map these smaller problems to polynomial-complexity min-cost-flow problems. This min-cost-flow-based decomposition can guarantee a solution that satisfies all constraints, but not rewiring optimality. For details, please refer to [32].

6.3 Impact on Scalability & Optimality

Theoretically, and as confirmed in §9.4, the total number of rewrites would be higher with block aggregation. We are essentially breaking one optimization problem (6) into two problems (13) and (14). Solving (13) and (14) will generate a solution satisfying all constraints in (6), but which might not be optimal wrt. the minimal-rewiring objective (6). Whenever the solver without block aggregation succeeds, it always achieve the smallest (best) rewiring ratio. However, scalability without block aggregation is poor. For a total of 4500 synthesized DCN expansion configurations, only 32% can be solved within a 3-hour limit. (Longer timeouts yield little improvement.)

Breaking (6) into smaller ILPs does not completely solve the solver scalability issue. We may still encounter some intractable ILPs while solving (14) (see §9.3). Our polynomial-time min-cost-flow based variable-deaggregation algorithm solves the scalability issue. However, it may also introduce some sub-optimality in the rewiring ratio (see §9.4 for the detailed comparison). Thus, we face a tradeoff between solver scalability and rewiring optimality, which we address in the next section.

7 Parallel Solving Architecture

While block aggregation makes it feasible to solve most DCN expansion configurations, it creates a tradeoff between solver scalability and rewiring optimality, depending on how one chooses a *strategy* for block aggregation. Block-aggregation strategies define choices for each of several aggregation layers (patch-panel, server-block, or spine-block), and for the decomposition technology (ILP or a min-cost-flow approximate algorithm) applied at each layer.

How can we choose the best strategy among all the options, given that the tradeoff between optimality and solver

run-time is unknown when we start a solver? We observe that since we care more about finding a solution within an elapsed time limit, and less about the total computational resources we use, our best approach is to run, in parallel, a solver instance for each of the options, and then choose the best result, based on a scoring function, among the solvers that complete within a set deadline. We can define scores based simply on rewiring ratio, or on residual bandwidth during expansion, or some combination. Fig. 6 shows the parallel-solver architecture.

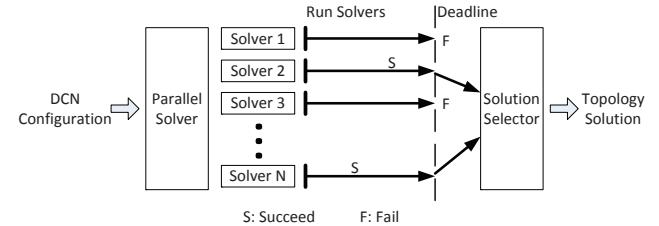


Figure 6: Software architecture of the parallel solver.

8 Changes to the Expansion Pipeline

The introduction of minimal rewiring requires several changes to the DCN expansion pipeline shown in Fig. 4.

Without minimal rewiring, it is fairly easy for an experienced network engineer to determine the number of stages required for an expansion; because almost all logical links need to be rewired, we must use C stages to maintain a residual capacity of $1 - 1/C$. With minimal rewiring, however, one cannot know the number of rewired links before running the solver; based on results in §9.4, the fraction could range from 0 to 30%.

Therefore, we add an automated **expansion planner** step to the front of the pipeline in Fig. 4. The planner first uses the minimal-rewiring solver to generate a post-expansion topology, then iteratively finds the lowest C , starting at $C = 1$, which preserves the desired residual capacity threshold during the expansion (recall from §4 that this threshold is a function of forecasted demand, existing failures, and some headroom). For each C , the planner divides the patch panels into C groups, tentatively generates the C intermediate topologies that would result from draining the affected links within a group, and evaluates those topologies against the threshold. If capacity cannot be preserved for all intermediate topologies, the planner increments C and tries again. Once a suitable C is found, the rest of the pipeline is safe to execute.

Without minimal rewiring, we simply drain all links for the patch panels covered by a stage. Minimal rewiring allows us to preserve more capacity because we only have to drain a subset of the links, rather than an entire patch panel. However, an operator can accidentally rewire the wrong link, or dislodge one inadvertently, a problem that does not occur when we drain entire panels. Therefore, we built a link-status monitor that alerts the operator if an active (undrained) link is disconnected. Since Clos networks by design tolerate some

link failures, this allows the operator enough time to repair the problem without affecting users.

9 Experimental Results

We have been successfully using this minimal-rewiring approach for our own DCNs since early 2017¹³. In order to demonstrate its benefits over a wide range of scales, we evaluated our approach using a set of synthetic DCN topologies (including some similar to our real DCNs), and show the effect of block aggregation on solver run-time and rewiring ratio. We also show that our approach preserves most of the capacity of the original network during expansions, based on several workloads and under some failure scenarios.

9.1 Synthesizing DCN Configurations

To evaluate our approach over a wide range of initial topologies, we synthesized thousands of configurations that were consistent with our past deployment experience. In particular, since we must support heterogeneity among server blocks and spine blocks, we synthesized configurations using three types of server blocks and two types of spine block, in various combinations. Every configuration includes exactly one border block, analogous to a server block but used for external connectivity.

Our synthesized DCN configurations always have 256 patch panels, located in 4 sets, each with 64 patch panels. We use patch panels with 128 server-block-facing ports and 128 spine-block-facing ports, so the entire patch-panel layer supports up to 64 512-port server blocks.

Table 2 lists the physical-topology parameters for the different block types. For example, a Type-1 server block has 512 up links, evenly distributed among all 256 patch panels. Type-2 & Type-3 server blocks are “light” versions of the Type-1 server block, with fewer uplinks; they can be upgraded to Type-1 server blocks. Note that a Type-3 server block only connects to 128 patch panels. We divide the 256 patch panels into two partitions, and connect the Type-3 server blocks to the two partitions by rotation. We also connect the Type-1 & Type-2 spine blocks by rotation.

Table 2: Server Block and Spine Block Types.

Block Type	Uplinks	Connected Patch Panels
Type-1 Server Block	512	64 Per Set×4 Sets
Type-2 Server Block	256	64 Per Set×4 Sets
Type-3 Server Block	256	32 Per Set×4 Sets
Border Block	1024	64 Per Set×4 Sets
Type-1 Spine Block	128	64 Per Set×1 Set
Type-2 Spine Block	512	64 Per Set×1 Set

We generate a total of 2250 initial configurations (pre-expansion “reference topologies” as defined in §5.1) from all possible combinations of $\{3, 6, \dots, 30\}$ Type-1 server blocks, $\{2, 4, \dots, 20\}$ Type-2 server blocks, $\{3, 6, \dots, 30\}$

¹³Even though we have a formal proof that deaggregation always works, it does not guarantee optimality; our experience shows that our approach does work in practice.

Type-3 server blocks, and $\{8, 16\}$ Type-1 spine blocks, with the remainder of the necessary spine-block ports as Type-2 spine blocks (the total number of server-block and spine-block ports must match). We omit any configuration that would require more patch-panel ports than we have available.

These pre-expansion topologies can be generated using our minimal rewiring solver, by simply ignoring the objective function. We run all the configurations in parallel, and allocate 4 CPUs and 16G RAM for each configuration. With block aggregation enabled, all 2250 topologies can be computed within 10 seconds.

9.2 Expansion Benchmarks

As shown in Fig. 3, we support two types of DCN expansions. We construct two benchmarks for each of our 2250 reference topologies:

Benchmark Suite 1: We upgrade two Type-2 server blocks in the reference topology to Type-1 server blocks.

Benchmark Suite 2: We expand the reference topology by one Type-3 server block.

We end up with $2250 \times 2 = 4500$ total target topologies. For each of these, we ran the minimal-rewiring solver with three aggregation strategies:

1. No aggregation.
2. Block aggregation, decomposing the spine-block and server-block layers using ILP, while decomposing the patch-panel layer using MIN_COST_FLOW (See technical report [32] for details).
3. Block aggregation, decomposing all three layers using MIN_COST_FLOW.

9.3 Solver scalability

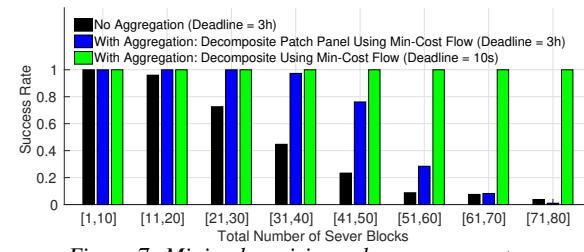


Figure 7: Minimal-rewiring solver success rate.

Fig. 7 plots success rate for our minimal-rewiring solver with different aggregation strategies, grouped by the total number of server blocks; the bars show the fraction of topologies solved for each group. With the third aggregation strategy, we can solve *all* test cases within 10 seconds; this strategy only needs to solve one ILP for the aggregated problem, while using polynomial algorithms for all decompositions. The first strategy scales poorly, and can only solve 32% of the test cases even if we increase the deadline to 3 hours; even for small DCNs (11–20 server blocks), it sometimes times out. The second strategy can solve 67% of the test cases, but we start seeing timeouts for DCNs with 31–

40 server blocks. For all strategies, setting timeouts above 3 hours yields little improvement.

The differences between the second and third strategies shows that variable deaggregation using ILP could take significant run time. (If we also use ILP to deaggregate the patch-panel layer with the second strategy, only 3% of the test cases can be solved.) However, deaggregation via ILP is still useful, because as we discuss next, it can generate more-optimal rewiring solutions.

9.4 Rewiring Ratio

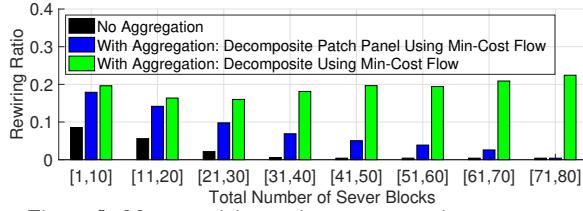


Figure 8: Mean rewiring ratio vs. aggregation strategy.

For the same set of test cases, Fig. 8 plots the rewiring ratios, again grouped by the number of server blocks; here, the bars show the mean value for each group. The third aggregation strategy, while it has the best run time, also leads to the highest (worst) rewiring ratio, often close to 20%; this motivates our use of ILP for decomposition whenever we can tolerate the run time.

9.5 Effectiveness of the parallel solver

§7 described how we use a parallel solver to strike a balance between scalability and rewiring optimality. For each of our benchmarks, we ran, in parallel, solvers with the three different aggregation strategies, with a 3-hour timeout. Because the third strategy works quickly for all instances, this parallel approach always succeeds. It also achieves the best rewiring ratio available within a 3-hour budget. Fig. 9 plots the CDF of the parallel solvers rewiring ratio. For about 82% of the DCN configurations in the first benchmark suite, and about 93% in the second suite, we get solutions with ratios under 20%. (The first suite tends to yield a higher ratio, because the total number of newly added server-block physical links in the first suite is twice that in the second suite.)

Note that the rewiring ratio for our prior approach was always 1.0 – we always replaced all patch-panel jumpers. Fig. 9 shows that minimal rewiring saves us a lot of cost and time; the median rewiring ratio is about 22× better for the first suite and about 38× better for the second suite.

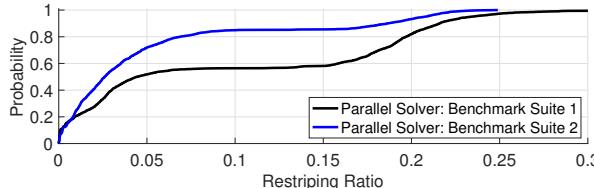


Figure 9: CDFs of rewiring ratios for the parallel solver.

9.6 Topology Capacity Analysis

In addition to solver scalability and rewiring optimality, we also must preserve sufficient capacity of the intermediate topologies during expansion. This requires us to quantify “capacity.” While our DCNs experience a variety of traffic patterns, we specifically evaluate the maximum achievable capacity under one-to-all and one-to-one traffic, which we rigorously define as:

One-to-all capacity: $T_{1\text{-all}}$. Assume that a source server block E_n is sending traffic to all other server blocks, with its demand proportional to the ingress bandwidth of the destination blocks. We increase the traffic until some DCN links are fully utilized. Let $T_{1\text{-all}}^n$ be the ratio between the egress traffic under these assumptions and the best-case egress DCN bandwidth of the server block E_n . $T_{1\text{-all}}^n$ characterizes the one-to-all DCN capacity for server block E_n . We can then define $T_{1\text{-all}} = \min_{n=1}^N T_{1\text{-all}}^n$ as the one-to-all capacity of the entire DCN.

One-to-one capacity: $T_{1\text{-1}}$. Assume that a server block E_{n_1} is sending traffic to another server block E_{n_2} . We increase the traffic until some DCN links are fully utilized. Let $T_{1\text{-1}}^{n_1, n_2}$ be the ratio between the traffic sent and the minimum, over the two server blocks E_{n_1} and E_{n_2} , of their ingress capacity. $T_{1\text{-1}}^{n_1, n_2}$ characterizes the one-to-one DCN capacity for the server blocks E_{n_1} and E_{n_2} . We can then define $T_{1\text{-1}} = \min_{n_1 \neq n_2} T_{1\text{-1}}^{n_1, n_2}$ as the one-to-one capacity of the entire DCN.

We evaluate $T_{1\text{-all}}$ and $T_{1\text{-1}}$ under two different scenarios:

Steady-state no-failures capacity: Recall that we imposed constraint (4) when computing a topology, which ensures high capacity in the non-expansion steady state, without any failures. Ideally, if the $p_{n,m}$'s in (4) were all integers, we would have $T_{1\text{-all}} = 1$ and $T_{1\text{-1}} = 1$. (Neither $T_{1\text{-all}}$ or $T_{1\text{-1}}$ can be larger than 1). Fig. 10 plots CDFs of $T_{1\text{-all}}$ and $T_{1\text{-1}}$ based on the 4500 post-expansion topologies (benchmark suite 1 + benchmark suite 2). Note that both $T_{1\text{-all}}$ and $T_{1\text{-1}}$ are fairly close to 1.

Steady-state capacity under middle block failure: Recall that we imposed constraint (5), to ensure the highest possible capacity if one middle block fails. Ideally, for our typical case of 4 middle blocks, if the $q_{n,m}^l$'s in (5) were all integers, we would have $T_{1\text{-all}} = 0.75$ and $T_{1\text{-1}} = 0.75$, no matter which middle block fails. (Given one failure, neither $T_{1\text{-all}}$ or $T_{1\text{-1}}$ can be larger than 0.75. Fig. 10 also plots the CDFs of $T_{1\text{-all}}$ and $T_{1\text{-1}}$ under middle block failures. Note that both $T_{1\text{-all}}$ and $T_{1\text{-1}}$ are fairly close to 0.75.

9.6.1 Residual Capacity during Expansion

During an expansion, we must disconnect some links, which reduces DCN capacity. We are interested in the residual capacity of the DCN in this state which clearly depends on the rewiring ratio. Fig. 11 shows scatter plots of $T_{1\text{-all}}$ and $T_{1\text{-1}}$ for each (rewiring ratio, residual capacity) pair, based on the two benchmark suites. This figure assumes we do all expansion.

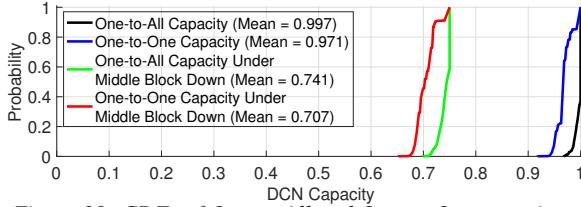


Figure 10: CDFs of One-to-All and One-to-One capacity.

sion in a single stage, even if the residual capacity is lower than we can accept in reality.

The residual capacity decreases approximately linearly with the rewiring ratio. Note that the residual capacity decreases faster than the rewiring ratio increase, because, the rewired DCN links might not be evenly distributed among different server blocks. As a result, some server blocks could suffer more capacity reduction than others.

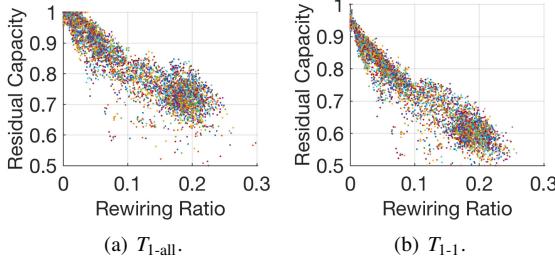


Figure 11: Rewiring ratio vs. residual capacity assuming a 1-stage expansion.

9.7 Number of expansion stages

In practice, we might not be able to do an expansion in just one stage while preserving sufficient residual capacity. §8 described an expansion planner that determines the number of stages required. We evaluated the number of stages saved by using minimal rewiring.

Prior to minimal rewiring, we typically did 4-stage expansions. If the topologies were perfect, with 4 stages we could preserve a residual one-to-all capacity of 0.75, but in practice we cannot achieve perfect balance; we have found it feasible and sufficient to preserve a residual capacity of 0.7.

Table 3 shows how many stages are needed to preserve a residual one-to-all capacity of 0.7, with minimal rewiring, for each of the 4500 benchmarks, based on solver strategy. With strategies 1 and 2, many test cases require four stages, because the solvers time out before finding 1- or 2-stage expansions. With strategy 3, almost all cases require at most 2 stages. The parallel solver finds 1- or 2-stage expansions for all test cases (because the few cases for which strategy 3 requires four stages are handled better by the other strategies), and usually does better than strategy 3 (because if the other strategies succeed within the deadline, they yield better rewiring ratios.) Overall, the parallel solver needs an average of 1.29 stages, vs. 4 stages for our prior approach.

Table 3: Number of expansion stages required.

Number of expansion stages:	1	2	4
Aggregation Strategy (1)	1598	34	2868
Aggregation Strategy (2)	2668	416	1416
Aggregation Strategy (3)	1582	2914	4
Parallel Solver	3176	1324	0

Cells show # of test cases that need given # of stages.

9.7.1 Concrete example

A concrete (arbitrary, but realistic) example demonstrates the benefits of minimal rewiring. Assume a pre-expansion DCN with 30 Type-1, 20 Type-2, 27 Type-3 server blocks, 1 border block, and 16 Type-1, 52 Type-2 spine blocks, which we expand by one Type-3 server block. Without minimal rewiring, we must rewire all the 28056 logical links, in four stages. With minimal rewiring, we need to rewire only 6063 of 28056 links (ratio = 0.216), in two stages. Due to the scale of this example, only the third aggregation strategy succeeds. Table. 4 shows how this maintains mid-expansion capacity (in italics) higher than our prior approach (in bold), and completes in two stages rather than four.

Table 4: Example: minimal rewiring vs. prior approach.

	Stage during expansion timeline					
	Pre	1	2	3	4	Post
Prior approach	0.94	0.70	0.70	0.70	0.70	0.94
Min. Rewiring	0.94	0.78	0.78	0.94	0.94	0.94

Cells show one-to-all capacity $T_{1\text{-all}}$ during expansion.

10 Conclusion

We have demonstrated that it is, in fact, feasible to do fine-grained expansions of heterogeneous Clos DCNs, at large scale, while preserving substantial residual capacity during an expansion, and with a significant reduction in the amount of rewiring (compared to prior approaches). We described how we use a patch-panel layer to reduce the physical complexity of DCN expansions. We then described an ILP formulation that allows us to minimize the amount of rewiring, a block-aggregation approach that allows scaling to large networks, and a parallel solver approach that yields the best tradeoff between elapsed time and rewiring ratio. Our overall approach flexibly handles heterogeneous switch blocks, and enforces “balance constraints that guarantee both high capacity and high failure resiliency. We evaluated our approach on a wide range of DCN configurations, and found on average that it allows us to do expansions in 1.29 stages, vs. 4 stages as previously required.

11 Acknowledgements

We thank Google colleagues for their insights and feedback, including Bob Felderman, David Wetherall, Keqiang He, Yadi Ma, Jad Hachem, Jayaram Mudigonda, and Parthasarathy Ranganathan. We also thank our shepherd Ankit Singla, and the NSDI reviewers.

References

- [1] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM* (August 2008).
- [2] BAUM, S., AND TROTTER, L. E. Integer rounding and polyhedral decomposition for totally unimodular systems. *Optimization and Operations Research* 157 (October 1978), 15–23.
- [3] BOYD, S., GHOSH, A., AND MAGNANI, A. Branch and bound methods. *Notes for EE392o, Stanford University* (November 2003).
- [4] CAMARERO, C., MARTINEZ, C., AND BEIVIDE, R. Random Folded Clos Topologies for Datacenter Networks. In *IEEE International Symposium on High Performance Computer Architecture* (February 2017).
- [5] CISCO. Cisco data center spine-and-leaf architecture: Design overview. *Cisco White Paper* (2016).
- [6] CLOS, C. A study of non-blocking switching networks. *Bell System Technical Journal* 32 (March 1953), 406–424.
- [7] CURTIS, A. R., CARPENTER, T., ELSHEIKH, M., LOPEZ-ORTIZ, A., AND KESHAV, S. REWIRE: An Optimization-based Framework for Data Center Network Design. In *INFOCOM* (March 2012).
- [8] CURTIS, A. R., KESHAV, S., AND LOPEZ-ORTIZ, A. LEGUP: Using Heterogeneity to Reduce the Cost of Data Center Network Upgrades. In *ACM CoNEXT* (November 2010).
- [9] DINITZ, M., SCHAPIRA, M., AND VALADARSKY, A. Explicit Expanding Expanders. *Algorithmica* 78 (August 2017), 1225–1245.
- [10] FARRINGTON, N., AND ANDREYEV, A. Facebook’s data center network architecture. *IEEE Optical Interconnects Conference* (May 2013).
- [11] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM* (August 2010).
- [12] GOOGLE INC. Google Optimization Tools. <https://developers.google.com/optimization/>.
- [13] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: a scalable and flexible data center network. In *SIGCOMM* (August 2009).
- [14] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM* (August 2009).
- [15] GUO, C., WU, H., TAN, K., SHI, L., ZHANG, Y., AND LU, S. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *SIGCOMM* (August 2008).
- [16] GUO, D., CHEN, T., LI, D., LI, M., LIU, Y., AND CHEN, G. Expandable and cost-effective network structures for data centers using dual-port servers. *IEEE Transactions on Computers* 62 (July 2013), 1303–1317.
- [17] GUROBI. www.gurobi.com.
- [18] GYARMATI, L., GULYAS, A., SONKOLY, B., TRINH, T. A., AND BICZOK, G. Free-scaling your data center. *Computer Networks* 57 (June 2013), 1758–1773.
- [19] HIGBIE, C. Point-to-Point Data Center Cable Will Cost You in the Long Run. *Network World* (Feb. 2012). <https://www.networkworld.com/article/2186410/tech-primers/point-to-point-data-center-cable-will-cost-you-in-the-long-run.html>.
- [20] IRVING, R. W., AND JERRUM, M. R. Three-dimensional statistical data security problems. *SIAM Journal on Computing* 23 (February 1994), 170–184.
- [21] LI, D., GUO, C., WU, H., TAN, K., ZHANG, Y., AND LU, S. FiConn: Using Backup Port for Server Interconnection in Data Centers. In *IEEE INFOCOM* (April 2009).
- [22] LI, Z., GUO, Z., AND YANG, Y. Bccc: An expandable network for data centers. *IEEE/ACM Transactions on Networking* 24 (December 2016), 3740–3755.
- [23] LIU, V., HALPERIN, D., KRISHNAMURTHY, A., AND ANDERSON, T. F10: A Fault-Tolerant Engineered Network. In *NSDI* (April 2013).
- [24] RAJASEKARAN, S. On the convergence time of simulated annealing. *Research Report MS-CIS-90-89, University of Pennsylvania, Department of Computer and Information Science* (November 1990).
- [25] SCHLINKER, B., MYSORE, R. N., SMITH, S., MOGUL, J. C., VAHDAT, A., YU, M., KATZ-BASSETT, E., AND RUBIN, M. Condor: Better Topologies Through Declarative Design. In *SIGCOMM* (August 2015).

- [26] SIEMON. Data Center Cabling Considerations: Point-to-Point vs. Structured Cabling. http://www.siemon.com/us/white_papers/09-06-18-data-center-point-to-point-vs-structured-cabling.asp.
- [27] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HLZLE, U., STUART, S., AND VAHDAT, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter. In *SIGCOMM* (September 2015).
- [28] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking Data Centers, Randomly. In *NSDI* (April 2012).
- [29] VALADARSKY, A., SHAHAF, G., DINITZ, M., AND SCHAPIRA, M. Xpander: Towards Optimal-Performance Datacenters. In *CoNEXT* (December 2016).
- [30] WALRAED-SULLIVAN, M., VAHDAT, A., AND MARZULLO, K. Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost. In *CoNEXT* (December 2013).
- [31] YU, Y., AND QIAN, C. Space shuffle: A scalable, flexible, and high-bandwidth data center network. *IEEE Transactions on Parallel and Distributed Systems* 27 (February 2016), 3351–3365.
- [32] ZHAO, S., WANG, R., ZHOU, J., ONG, J., MOGUL, J. C., AND VAHDAT, A. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks: Extended Version. Online: <https://ai.google/research/pubs/pub47492>.
- [33] ZHOU, J., TEWARI, M., ZHU, M., KABBANI, A., POUTIEVSKI, L., SINGH, A., AND VAHDAT, A. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys* (April 2014).

Understanding Lifecycle Management Complexity of Datacenter Topologies

Mingyang Zhang
USC

Radhika Niranjan Mysore
VMWare

Sucha Supittayapornpong
USC

Ramesh Govindan
USC

Abstract

Most recent datacenter topology designs have focused on performance properties such as latency and throughput. In this paper, we explore a new dimension, life cycle management complexity, which attempts to understand the complexity of deploying a topology and expanding it. By analyzing current practice in lifecycle management, we devise complexity metrics for lifecycle management, and show that existing topology classes have low lifecycle management complexity by some measures, but not by others. Motivated by this, we design a new class of topologies, FatClique, that, while being performance-equivalent to existing topologies, is comparable to, or better than them by *all* our lifecycle management complexity metrics.

1 Introduction

Over the past decade, there has been a long line of work on designing datacenter topologies [2, 35, 31, 32, 3, 4, 20, 1]. While most have focused on performance properties such as latency and throughput, and on resilience to link and switch failures, datacenter *lifecycle management* [30, 38] has largely been overlooked. Lifecycle management is the process of building a network, physically deploying it on a data-center floor, and expanding it over several years so that it is available for use by a constantly increasing set of services.

With datacenters living on for years, sometimes up to a decade [31, 12], their lifecycle costs can be high. A data center design that is hard to deploy can stall the rollout of services for months; this can be expensive considering the rate at which network demands have historically increased [31, 23]. A design that is hard to expand can leave the network functioning with degraded capacity impacting the large array of services that depend on it.

It is therefore desirable to commit to a data-center network design only after getting a sense of its lifecycle management cost and complexity over time. Unfortunately, the costs of the large array of components needed for deployment such as switches, transceivers, cables, racks, patch panels¹, and cable trays, are proprietary and change over time, and so are hard to quantify. An alternative approach is to develop *complexity measures* (as opposed to dollar costs) for lifecycle management, but as far as we know, no prior work has addressed this. In part, this is due to the fact that intuitions about lifecycle management are developed over time and with operations experience, and these lessons are not made available universally.

Unfortunately, in our experience, this lack of a clear understanding of lifecycle management complexity often results in costly mistakes in the design of datacenters that are discovered during deployment and therefore cannot be rectified. Our paper is a first step towards useful characterizations of lifecycle management complexity.

Contributions. To this end, our paper makes three contributions. First, we design several *complexity metrics* (§3 and §4) that can be indicative of lifecycle management costs (*i.e.*, capital expenditure, time and manpower required). These metrics include the number of: switches, patch panels, *bundle-types*, expansion steps, and links to be re-wired at a patch panel rack during an expansion step.

We design these metrics by identifying structural elements of network deployments that make their deployment and expansion challenging. For instance, the number of switches in the topology determines how complex the network is in terms of *packaging* – laying out switches into homogeneous racks in a space efficient manner. Wiring complexity can be assessed by the number of cable *bundles* and the patch panels a design requires. As these increase, the complexity of manufacturing and packaging all the different cable bundles efficiently into cable trays, and then routing them from one patch panel to the next can be expected to increase. Finally, because expansion is carried out in steps [38], where the network operates at degraded capacity at each step, the number of expansion steps is a measure of the reduced availability in the network induced by lifecycle management. Wiring patterns also determine the number of links that need to be rewired at a patch panel during each step of expansion, a measure of step complexity [38].

Our second contribution is to use these metrics to compare the lifecycle management costs of two main classes of datacenter topologies recently explored in the research literature (§2), Clos [2] and expander graphs [32, 35]. We find that neither class dominates the other: Clos has relatively lower wiring complexity; its symmetric design leads to more uniform bundling (and fewer cable bundle types); but expander graphs at certain scales can have simpler packaging requirements due to their edge expansion property [32]; they end up using much fewer switches than Clos to achieve the same network capacity. Expander graphs also demonstrate better expansion properties because they have *fat edges* (§4) which permit more links to be rewired in each step.

Finally we design and synthesize a novel and practical class of topologies called FatClique (§5), that has lower overall lifecycle management complexity compared to Clos and expander graphs. We do this by combining favorable design

¹A patch panel or a wiring aggregator is a device that simplifies cable re-wiring.

elements from these two topology classes. By design, FatClique incorporates 3 levels of hierarchy and uses a clique as a building block while ensuring edge expansion. At every level of its hierarchy, FatClique is designed to have fat edges, for easier expansion, while utilizing much fewer patch panels and therefore inter-rack cabling.

Evaluations of these topology classes at three different scales, the largest of which is $16 \times$ the size of Jupiter, shows that FatClique is the best at most scales by *all* our complexity metrics. It uses 50% fewer switches and 33% fewer patch panels than Clos at large scale, and has a 23% lower cabling cost (an estimate we are able to derive from published cable prices). Finally, FatClique can permit fast expansion while degrading network capacity by small amounts (2.5-10%): at these levels, Clos can take $5 \times$ longer to expand the topology.

2 Background

Data center topology families. Data centers are often designed for high throughput, low latency and resilience. Existing data center designs can be broadly classified into the following families: (a) Clos-like tree topologies, *e.g.*, Google’s Jupiter [31], Facebook’s fbfabric [3], Microsoft’s VL2 [13], F10 [22]; (b) Expander graph based topologies, *e.g.*, Jellyfish [32], Xpander [35]; (c) ‘Direct’ topologies built from multi-port servers, *e.g.*, BCube [14], DCCell [15]. (d) Low diameter, strongly-connected topologies that rely on high-radix switches, *e.g.*, Slimfly [4], Dragonfly [20]; (e) Reconfigurable optical topologies like Rotornet and Project-ToR [24, 9, 11, 16, 39].

Of these, Clos and Expander based topologies have been shown to scale using widely deployed merchant silicon. The ecosystem around the hardware used by these two classes, *e.g.*, cabling, cable trays used, rack sizes, is mature and well-understood, allowing us to quantify some of the operational complexity of these topologies.

Direct multi-port server topologies and some reconfigurable optical topologies [24, 11, 16, 39] rely on newer hardware technologies that are not mainstream yet. It is hard to quantify the operational costs of these classes without making significant assumptions about such hardware. Low diameter topologies like Slimfly [4] and Dragonfly [20], can be built with hardware that is available today, but they require strongly connected groups of switches. Their incremental expansion comes at high cost and complexity; high-radix switches either need to be deployed well in advance, or every switch in the topology needs to be upgraded during expansion, to preserve low diameter.

To avoid estimating operational complexity of topologies that rely on new hardware, or on topologies that unacceptably constrain expansion, we focus on the Clos and Expander families.

Clos. A logical Clos topology with N servers can be constructed using switches with radix k connected in $n = \log_{\frac{k}{2}}(\frac{N}{2})$ layers based on a canonical recursive algorithm

in [36]². Fattree [2] and Jupiter [31] are special cases of Clos topology with 3 and 5 layers respectively. Clos construction naturally allows switches to be packaged together to form a chassis [31]. Since there are no known generic Clos packaging algorithm that can help design such a chassis, for a Clos of any scale, we designed one to help our study of its operational complexity. We present this algorithm in §A.1.

Expander graphs. Jellyfish and Xpander benefit from the high *edge expansion* property of expander graph to use a near optimal number of switches, while achieving the same bisection bandwidth as Clos based topologies [35]. Xpander splits N servers among switches by attaching s servers to each switch. With a k port switch, the remaining ports $p = k - s$ are connected to other switches that are organized in p blocks called *metanodes*. Metanodes are a group of switches, containing $l = N/(s \cdot (p + 1))$ switches, which increase as topology scale N increases. There are no connections between the switches of a metanode. Jellyfish is a degree bounded random graph (see [32] for more details).

Takeaway. A topology with high edge expansion [35] can achieve a target capacity with fewer switches, leading to lower overall cost.

3 Deployment Complexity

Deployment is the process of realizing a physical topology in a data center space (*e.g.*, a building), from a given logical topology. Deployment complexity can be reduced by careful *packaging, placement and bundling* strategies [31, 20, 1].

3.1 Packaging, Placement, and Bundling

Packaging of a topology involves careful arrangement of switches into racks, while placement involves arranging these racks into rows on the data center floor. The spatial arrangement of the topology determines the type of cables needed between switches. For instance, if two connected switches are within the same rack, they can use short-range cheaper copper cables, while connections between racks require more expensive optical cables. Optical cable costs are determined by two factors: the cost of transceivers and the length of cables (§3.2). Placement of switches on the datacenter floor can also determine costs: connecting two switches placed at two ends of the data center building might require long range cables and high-end transceivers.

Chassis, racks, and blocks. Packaging connected switches into a single chassis using a backplane completely removes the need for physical connecting cables. At scale, the cost and complexity savings from using a chassis-backplane can be significant. One or more chassis that are interconnected can be packed into racks such that: (a) racks are as homogeneous as possible, *i.e.*, a topology makes use of only a few types of racks to simplify manufacturing and (b) racks are packed as

²This equation for n can be used to build a Clos with 1:1 oversubscription. For a Clos with an over-subscription $x:y$ we would need $n = \log_{\frac{k}{2}}(\frac{y \cdot N/x}{2})$ layers.

densely as possible to reduce space wastage. Some topologies define larger units of co-placement and packaging called *blocks*, which consist of groups of racks. Examples of blocks include pods in Fattree. External cabling from racks within a block are routed to wiring aggregators (*i.e.*, patch panels [25]) to be routed to other blocks. For blocks to result in lower deployment complexity, three properties must be met: (a) the ports on the patch panel that it connects to are not wasted, when the topology is built out to full scale, (b) wiring out of the block should be as uniform as possible, and (c) racks in a block must be placed close to each other to reduce the length and complexity of wiring.

Bundling and cable trays. When multiple fibers from the same set of physically adjoint (or neighboring) racks are destined to another set of neighboring racks, these fibers can be *bundled* together. A fiber *bundle* is a fixed number of identical-length fibers between two clusters of switches or racks. Manufacturing bundles is simpler than manufacturing individual fibers, and handling such bundles significantly simplifies operation complexity. Cable bundling reduces capex and opex by around 40% in Jupiter [31].

Patch panels facilitate bundling since the patch panel represents a convenient aggregation point to create and route bundles from the set of fibers destined to the same patch panel (or the same set of physically proximate patch panels). Figure 1 shows a Clos topology instance (left) and its physical realization using patch panels (right). Each aggregation block in the Clos network connects with one link to each spine block. The figure on the right shows how these links are routed physically. Bundles with two fibers each from two aggregations are routed to two (lower) patch panels. At each patch panel, these fibers are rebundled, by grouping fibers that go to the same spine in new bundles, and routed to two other (upper) patch panels that connect to spines. The bundles from the upper patch panels are then routed to the spines. Figure 1 assumes that patch panels are used as follows: bundles are connected to both the front and back ports on patch panels. For example, bundles from the aggregation layer connect to front ports on patch panels and bundles from spines connect to the back ports of patch panels. This enables bundle aggregation and rebundling and simplifies topology expansion.³

Bundles and fibers are routed through the datacenter on *cable trays*. The cables that aggregate at a patch panel rack must be routed overhead by using over-row and cross-row trays [26]. Trays have capacity constraints [34], which can constrain rack placement, block sizes, and patch panel placement. Today, trays can support at most a few thousand fibers [34].

³[38]’s usage of patch panels is slightly different. All bundles are connected to front ports of patch panels and links are established using jumper cables between the back ports of patch panels. For patch panels of a given port count, both approaches require the same number of patch panels. Our approach enables bundling closer to the aggregation and spine layers; [38] does not describe how bundling is accomplished in their design.

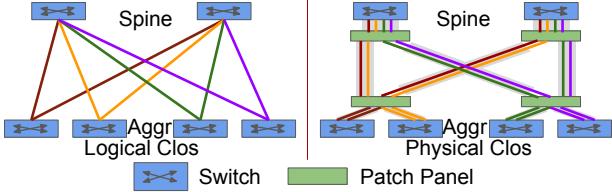


Figure 1: Fiber Re-bundling for Clos at Patch Panels

With current rack and cable tray sizes, a single rack of patch panels can be accommodated by four overhead cable trays, arranged in four directions. In order to avoid aggregating too many links into a single location, it is desirable to space such patch panels apart to accommodate more cable trays. This consideration in turn constrains block sizes; if cables from blocks must be all routed locally, it is desirable that a block only connect to a single rack of patch panel.

3.2 Deployment Complexity Metrics

Based on the previous discussion, we identify several metrics that quantify the complexity of the two aspects of datacenter topology deployment: packaging and placement. In the next subsection, we use these metrics to identify differences between Clos and Expander graph topology classes.

Number of Switches. The total number of switches in the topology determines the capital expenditure for the topology, but it also determines the packaging complexity (switches need to be packed to chassis and racks) and the placement complexity (racks need to be placed on the datacenter floor).

Number of Patch panels. By acting as bundle waypoints, the number of patch panels captures one measure of wiring complexity. The more the number of patch panels, the shorter the cable lengths from switches to the nearest patch panel, but the fewer the bundling opportunities, and vice versa. The number of patch panels needed is a function of topological structure. For instance, in a Clos topology, if an aggregation layer fits into one rack or a neighboring set of racks, a patch panel is not needed between the ToR and the aggregation layer. However, for larger Clos topologies where an aggregation block can span multiple racks, ToR to aggregation links may need to be rebundled through a patch panel. We discuss this in detail in §6.2.

Number of Bundle Types. The number of patch panels alone does not capture wiring complexity. The other measure is the number of distinct *bundle types*. A bundle type is represented by a tuple of (a) the *capacity* of the number of fibers in the bundle, and (b) the *length* of the bundle. If a topology requires only a small number of bundle types, its bundling is more homogeneous; manufacturing and procuring such bundles is significantly simpler, and deploying the topology is also simplified since fewer bundling errors are likely with fewer types.

These complexity measures are *complete*. The number of cable trays, the design of the chassis, and the number of racks can be derived from the number of switches (and the number of servers and the datacenter floor dimensions, which

Topology	4-layer Clos (Medium)	Jellyfish
#servers	131,072	131,072
#switches	28,672	16,384
#bundle types	74	1577
#patch panels	5546	7988

Table 1: Deployment Complexity Comparison

are inputs to the topology design). The number of cables and transceivers can be derived from the number of patch panels.

In some cases, a metric is related to another metric, but not completely subsumed by it. For example, the number of switches determines rack packaging, which only partially determines the number of transceivers per switch. The other determinant of this quantity is the *connectivity in the logical topology* (which switch is connected to which other switch). Similarly, the number of patch panels can influence the number of bundle types, but these are also determined by logical connectivity.

3.3 Comparing Topology Classes

To understand how the two main classes of topologies compare by these metrics, we apply these to a Clos topology and to a Jellyfish topology that support the same number of servers (131,072) and the same bisection bandwidth. This topology corresponds to twice the size of Jupiter. In §6, we perform a more thorough comparison at larger and smaller scales, and we describe the methodology by which these numbers were generated.

Table 1 shows that the two topology classes are qualitatively different by these metrics. Consistent with the finding in [32], Jellyfish only needs a little over half the switches compared to Clos to achieve comparable capacity due to its high edge expansion property. But, by other measures, Clos performs better. It exposes far fewer ports outside the rack (a little over half that of Jellyfish); we say Clos has better *port-hiding*. A pod in this Clos contains 16 aggregation and 16 edge switches⁴. The aggregation switches can be packed into a single rack, so bundles from edge switches to aggregation switches do not need to be rebundled through patch panels, and we only need two layers of patch panels between aggregation and spine layer. However, in Jellyfish, almost all links are inter-rack links, so it requires more patch panels.

Moreover, for Clos, since each pod has the same number of links to each spine, all bundles in Clos have the same capacity (number of fibers). However, the length of bundles can be different, depending on the relative placement of the patch panels between aggregation and spine layers, so Clos has 74 bundle types. However, since Jellyfish is a purely random graph without structure, to enable bundling, we group a fixed amount of neighbor racks as blocks to enable bundling. Since connectivity is random, the number of links between blocks are not uniform, Jellyfish needs almost 20× the number of bundle types. In §6, we show that Xpander also has

qualitatively similar behavior in large scale.

Takeaway. Relative to a structured hierarchical class of topologies like Clos, the expander graph topology has inherently higher deployment complexity in terms of the number of bundle types and cannot support port-hiding well.

4 Topology Expansion

The second important component of topology lifecycle management is *expansion*. Datacenters are rarely deployed to maximal capacity in one shot; rather, they are gradually expanded as network capacity demands increase.

4.1 The Practice of Expansion

In-place Expansion. At a high-level, expanding a topology involves two conceptual phases: (a) procuring new switches, servers, and cables and laying them on the datacenter floor, and (b) re-wiring (or adding) links between switches in the existing topology and the new switches. Phase (b), the *re-wiring phase*, can potentially disrupt traffic; as links are re-wired, network capacity can drop, leading to traffic loss. To avoid traffic loss, providers can either take the existing topology offline (migrate services away, for example, to another datacenter), or can carefully schedule link re-wiring while carrying live traffic, but schedule the re-wiring to maintain a desired target capacity. The first choice can impact service availability significantly.

So, today, datacenters are expanded while carrying live traffic [30, 12, 31, 38]. To do this, expansion is carried out in *steps*, where at each step, the capacity of the topology is guaranteed to be at least a percentage p of the capacity of the existing topology. This fraction is sometimes called the expansion SLO. Today, many providers operate at expansion SLOs of 75% [38]; higher SLOs of 85–90% can impact availability budgets less while allowing providers to carry more traffic during expansion.

The unit of expansion. Since expansion involves procurement, topologies are usually expanded in discrete units called *blocks* to simplify the procurement and layout logistics. In a structured topology, there are natural candidates for blocks. For example, in a Clos, a pod can be a block, while in an Xpander, the metanode can be a block. During expansion, a block is first fully assembled and placed, and links between switches *within* a block are connected (as an aside, an Xpander metanode has no such links). During the re-wiring phase, *only links between existing blocks and new blocks are re-wired*. (This phase does *not* re-wire links between switches within an existing block). Aside from simplifying logistics, expanding at the granularity of a block preserves structure in structured topologies.

4.2 An Expansion Step

What happens during a step. Figure 2 shows an example of Clos expansion. The upper left figure shows a partially-deployed logical Clos, in which each spine and aggregation

⁴we follow the definition of pod in [2].

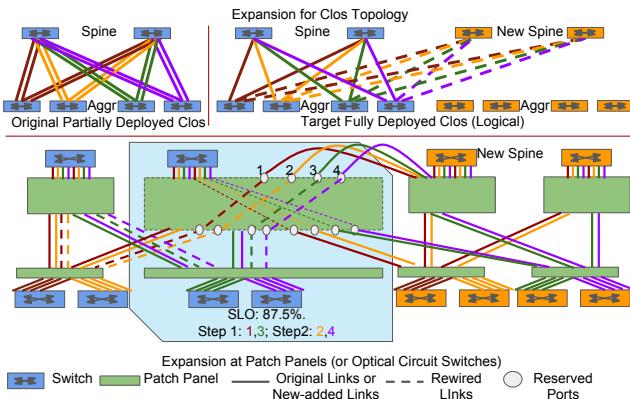


Figure 2: Clos Expansion with Patch Panels

block are connected by two links. The upper right is the target fully-deployed Clos, where each spine and aggregation block are connected by a single link. During expansion, we need to redistribute half of existing links (dashed) to the newly added spines without violating wiring and capacity constraints.

Suppose we want to maintain 87.5% of the capacity of the topology (*i.e.*, the expansion SLO is 0.875), this expansion will require 4 steps in total, where each patch panel is involved in 2 of these steps. In Figure 2, we only show the rewiring process on the second existing patch panels. To maintain 87.5% capacity at each pod, only one link is allowed to be drained. In the first step, the red link from the first existing aggregation block and the green link from the second existing aggregation block are rewired to the first new spine block. In the second step, the orange links from the first existing aggregation block and the purple link from the second existing aggregation block are rewired to the first new spine block. A similar process happens in the first patch panel.

In practice, each step of expansion involves four sub-steps. In the first sub-step, the existing links that are to be re-wired are *drained*. Draining a link involves programming switches at each end of the link to disable the corresponding ports, and may also require reprogramming other switches or ports to route traffic around the disabled link. Second, one or more human operators *physically rewire* the links at a patch panel (explained in more detail below). Third, the newly wired links are *tested* for bit errors by sending test traffic through them. Finally, the new links are *undrained*.

By far the most time consuming part of each step is the second sub-step, which requires human involvement. This sub-step is also the most important from an availability perspective; the longer this sub-step takes, the longer the datacenter operates at reduced capacity, which can impact availability targets [12].

The role of patch panels in re-wiring. The lower figure in Figure 2 depicts the physical realization of the (logical) re-wiring shown in the upper figure. (For simplicity, the figure only shows the re-wiring of links on one patch panel to a new pod). Fibers and bundles originate and terminate at patch panels, so re-wiring requires reconnecting input and output

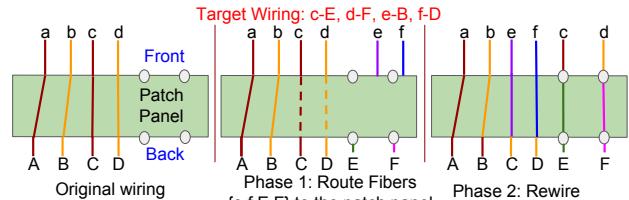
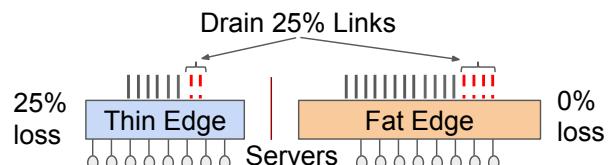


Figure 3: Basic Rewiring Operations at a patch panel



ports at each patch panel. One important constraint in this process is that re-wiring cannot remove fibers that are already part of an existing bundle.

Patch panels help localize rewiring and reuse existing cable bundling during expansions. Figure 3 shows, in more detail the rewiring process at a single patch panel. The leftmost figure shows the original wiring with connections (a, A), (b, B), (c, C), (d, D). To enable expansion, a topology is always deployed such that some ports at the patch panel are reserved for expansion steps. In the figure, we use these reserved ports to connect new fibers e, f, E and F (Phase 1). To get to a target wiring in the expanded network with connections (a, A), (b, B), (e, C), (f, D), (c, E), (d, F), the following steps are taken: (1) Traffic is drained from (c, C), (d, D), (2) Connections (c, C), (d, D) are rewired, with c being connected to E, d being connected to F and so on, and (3) The new links are undrained, allowing traffic to use new capacity.

4.3 Expansion Complexity Metrics

We identify two metrics that quantify expansion complexity and use these metrics to identify differences between Clos and Jellyfish in the next subsection.

Number of Expansion Steps. As mentioned each expansion step requires a series of substeps which cannot be parallelized. Therefore the number of expansion steps determines the total time for expansion.

Average number of rewired links in a patch panel rack per step. With patch panels, manual rewiring dominates the time taken within each expansion step. Within steps, it is possible to parallelize rewiring across racks of patch panels. With such parallelization, the time taken to rewire a single patch panel rack will dominate the time taken for each expansion step.

4.4 Comparing Topology Classes

Table 2 shows the value of these measures for a medium-sized Clos and a comparable Jellyfish topology, when the expansion

Topology	4-layer Clos (Medium)	Jellyfish
Average # links rewired per patch panel rack	832	470
Expansion steps	6	3
North-to-south capacity ratio	1	3

Table 2: Expansion Comparison (SLO = 90%)

SLO is 90%. (§6 has more extensive comparisons for these metrics, and also describes the methodology more carefully). In this setting, the number of links rewired per patch panel can be a factor of two less than Clos. Moreover, Jellyfish requires 3 steps, while Clos twice the number of steps.

To understand why Jellyfish requires fewer steps, we define a metric called the *north-to-south* capacity ratio for a block. This is the ratio of the aggregate capacity of all “northbound” links exiting a block to the aggregate capacity of all “southbound” links to/from the servers within the block. Figure 4 illustrates this ratio: a *thin edge* (left), has an equal number of southbound and northbound links while a *fat edge* (right), has more northbound links than southbound links. A Clos topology has a thin edge, *i.e.*, this ratio is 1, since the block is a pod. Now, consider an expansion SLO of 75%. This means that the southbound aggregate capacity must be at least 75%. That implies that, for Clos, *at most* 25% of the links can be re-wired in a single step. However, Jellyfish has a much higher ratio of 3, *i.e.*, it has a fat edge. This means that *many more links can be rewired in a single step* in Jellyfish than in Clos. This property of Jellyfish is required for reducing the number of expansion steps.

Takeaway. Clos topologies re-wire more links in each patch panel during an expansion step and require many steps because they have a low north-south capacity ratio.

5 Towards Lower Lifecycle Complexity

Our discussions in §3 and §4, together with preliminary results presented in those sections (§6 has more extensive results) suggest the following qualitative comparison between Clos and the expander graph families with respect to lifecycle management costs (Table 3):

- Clos uses fewer bundle types and patch panels.
- Jellyfish has significantly lower switch counts, uses fewer expansion steps, and touches fewer links per patch panel during an expansion step.

In all of these comparisons, we compare topologies with the same number of servers and the same bisection bandwidth.

The question we ask in this paper is: *Is there a family of topologies which are comparable to, or dominate, both Clos and expander graphs by all our lifecycle management metrics?* In this section, we present the design of the FatClique class of topologies and validate in §6 that FatClique answers this question affirmatively.

5.1 FatClique Construction

FatClique (Figure 5) combines the hierarchical structure in Clos with the edge expansion in expander graphs to achieve lower lifecycle management complexity. FatClique has three

	4-layer Clos (Medium)	Jellyfish
switches		✓
bundle types	✓	
patch panels	✓	
re-wired links per patch panel		✓
expansion steps		✓

Table 3: Qualitative comparison of lifecycle management complexity

Auxiliary Variable	Description
$p_s = S_c - 1$	# ports per switch to connect other switches inside a sub-block
$p_b = k - s - p_s - p_c$	# ports per switch to connect other blocks
$R_c = S_c \cdot (p_c + p_b)$	radix of a sub-block
$R_b = S_b \cdot S_c \cdot p_b$	radix of a block
$N_b = N / (S_b \cdot S_c \cdot s)$	#blocks
$L_{cc} = S_c \cdot p_c / (S_b - 1)$	#links between two sub-blocks inside a block
$L_{bb} = R_b / (N_b - 1)$	#links between two blocks

Table 4: FatClique Variables

levels of hierarchy: individual sub-block (top left), interconnected into a block (top right), which are in turn interconnected to form FatClique (bottom). The interconnection used at every level in the hierarchy is a clique, similar to Dragonfly [20]. Additionally, *each level* in the hierarchy is designed to have a fat edge (a north-south capacity ratio greater than 1). The cliques enable high edge expansion, while hierarchy enables lower wiring complexity than random-graph based expanders [32, 35].

FatClique is a class of topologies. To obtain an instance of this class, a topology designer specifies two input parameters: N , the number of servers, and k the chip radix. A *synthesis* algorithm takes these as inputs, and attempts to *instantiate* four *design variables* that completely determine the FatClique instance Table 4. These four design variables are:

- s , the number of ports in a switch that connect to servers
- p_c , the number of ports in each switch that connect to other sub-blocks inside a block
- S_c , the number of switches in a sub-block
- S_b , the number of sub-blocks in a block

The synthesis algorithm searches for the best combination of values for design variables, guided by six constraints, C_1 through C_6 , described below. The algorithm also defines *auxiliary* variables for convenience; these can be derived from the design variables (Table 4). We define these variables in the narrative below.

Sub-block connectivity. In FatClique, the sub-block forms the lowest level of the hierarchy, and contains switches and servers. All sub-blocks have the same structure. Servers are distributed uniformly among all switches of the topology, such that each sub-block has the same number of servers attached. However, because this number of servers may not be an exact multiple of the number of switches, we distribute the remainder across the switches, so that some switches may be connected to one more server than others. The alternative would have been to truncate or round up the number of servers per sub-block to be divisible by the number of switches in

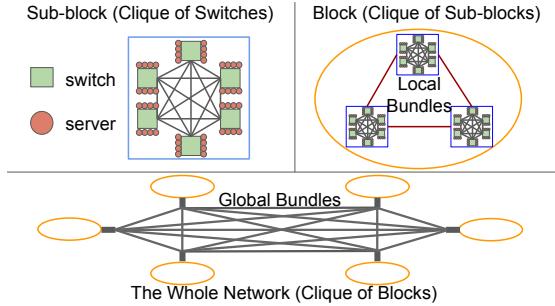


Figure 5: FatClique Topology

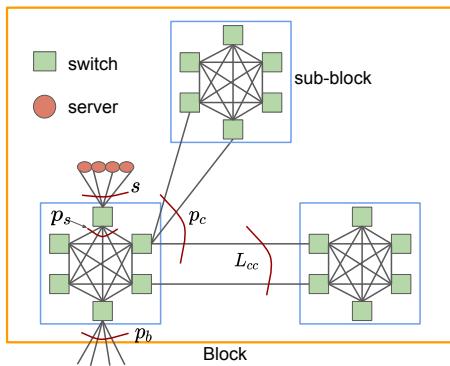


Figure 6: FatClique Block

the sub-block, which could lead to overprovisioning or under-provisioning. Within a sub-block, every switch has a link to every other switch within its sub-block, to form a clique (or complete graph). To ensure a fat edge at the sub-block level, each switch must connect to more switches than servers, captured by the constraint $C_1 : s < r - s$, where r is the switch radix and s is the number of ports on a switch connected to servers.

Block-level connectivity. The next level in the hierarchy is the block. Each sub-block is connected to other sub-blocks within a block using a clique (Figure 5, top-left). In this clique, each sub-block may have *multiple* links to another sub-block; these inter-sub-block links are evenly distributed among all switches in the sub-block such that every pair of switches from different sub-block has at most one link. Ensuring a fat edge at this level requires that a sub-block has more inter-sub-block and inter-block links egressing from the sub-block than the number of servers it connects to. Because sub-blocks contain switches which are homogeneous⁵, this constraint is ensured if the sum of (a) the number ports on each switch connected to other sub-block (p_c) and (b) those connected to other blocks (p_b , an auxiliary variable in Table 4, see also Figure 6) exceeds the number of servers connected to the switch (captured by $C_2 : p_c + p_b > s$).

Inter-block connectivity. The top of the hierarchy is the overall network, in which each block is connected to every

⁵They are nearly homogeneous, since a switch may differ from another by one in the number of servers connected

other block, resulting in a clique. The inter-block links are evenly distributed among all sub-blocks, and, within a sub-block, evenly among all switches. To ensure a fat edge at this level, the number of inter-block links at each switch should be larger than the number of servers it connects to, captured by $C_3 : p_b > s$. Note that C_3 subsumes (is a stronger constraint than) C_2 . Moreover, the constraint that blocks are connected in a clique imposes a constraint on the *block radix* (R_b , a derived variable). The block radix is the total number of links in a block destined to other blocks. R_b should be large enough to reach all other blocks (captured by $C_4 : R_b \geq N_b - 1$) such that the whole topology is a clique.

Incorporating rack space constraints. Beyond connectivity constraints, we need to consider packaging constraints in sub-block design. Ideally, we need to ensure that a sub-block fits completely into one or more racks with *no wasted rack space*. For example, if we use 58RU racks, and each switch is to be connected to 8 1RU servers, we can accommodate 6 switches per sub-block, leaving $58 - (6 \times 8 + 6) = 4$ U in the rack for power supply and other equipment. In contrast, choosing 8 switches per sub-block would be a bad choice because it would need $8 \times 8 + 8 = 72$ U rack space, overflowing into a second rack that would have 44RU un-utilized. We model this packaging fragmentation as a *soft* constraint: our synthesis algorithm generates multiple candidate assignments to the design variables that satisfy our constraints, and of these, we pick the alternative that has the lowest wasted rack space.

Ensuring edge expansion. At each level of the hierarchy, edge expansion is ensured by using a clique. This is necessary for high edge expansion, but not sufficient, since it does not guarantee that every switch connects to as many other switches across the network as possible. One way to ensure this diversity is to make sure that each pair of switches is connected by at most one link. The constraints discussed so far do not ensure this. For instance, consider Figure 6, in which L_{cc} (another auxiliary variable in Table 4) is the number of links from one sub-block to another. If this number is greater than the number of switches S_c in the sub-block, then, some pair of switches might have more than one link to each other. Thus, $C_5 : L_{cc} \leq S_c$ is a condition to ensure that each pair of switches must be connected by a single link. Our topology synthesis algorithm generates assignments to design variables, and a topology generator then assigns links to ensure this property (§5.2).

Incorporating patch panel constraints. The size of the block is also limited by the number of ports in a single patch panel rack (denoted by $PP - Rack_{ports}$). It is desirable to ensure that the inter-block links egressing each block connect to at most $\frac{1}{2}$ the ports in a patch panel rack, so that the rest of the patch panel ports are available for external connections into the block (captured by $C_6 : R_b \leq \frac{1}{2} \cdot PP - Rack_{ports}$).

Topology	Scalability
3-layer Clos (FatTree)	$2 \cdot (k/2)^3$
4-layer Clos	$2 \cdot (k/2)^4$
5-layer Clos (Jupiter)	$2 \cdot (k/2)^5$
FatClique	$O(k^5)$

Table 5: Scalability of Topologies

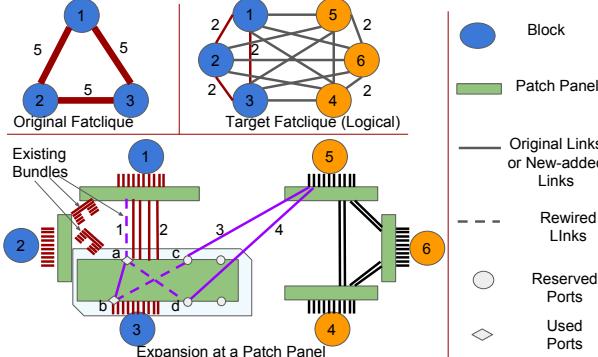


Figure 7: FatClique Expansion example

5.2 FatClique Synthesis Algorithm

Generating candidate assignments. The FatClique synthesis algorithm attempts to assign values to the design variables, subject to constraints C_1 to C_6 . The algorithm enumerates all possible combinations of value assignments for these variables, and filters out each assignment that fails to satisfy all the constraints. For each remaining assignment, it generates the topology specified by the design variable, and determines if the topology satisfies a required capacity Cap^* , which is an input to the algorithm. Each assignment that fails the capacity test is also filtered out, leaving a candidate set of assignments. These steps are described in §A.6.

FatClique placement. For each assignment in this candidate set, the synthesis algorithm generates a *topology placement*. Because FatClique’s design is regular, its topology placement algorithm is conceptually simple. A sub-block may span one or more racks, and these racks are placed adjacent to each other. All sub-blocks within a block are arranged in a rectangular fashion on the datacenter floor. For example, if a block has 25 racks, it is arranged in a 5×5 pattern of racks. Blocks are then arranged in a similar grid-like fashion.

Selecting best candidate. For each placement, the synthesizer computes the cabling cost of the resulting placement (using [7]), and picks the candidate with the lowest cost. This step is not shown in Algorithm 3. This approach implicitly filters out candidates whose sub-block cannot be efficiently packed into racks (§5.1).

5.3 FatClique Expansion

Re-wiring during expansion. Consider a small FatClique topology, shown top left in Figure 7, that has 3 blocks and $L_{bb} = 5$, i.e., five inter-block links. To expand it to a clique with six blocks, we would need to rewire the topology to have $L'_{bb} = 2$ (top right in Figure 7). This means we need to redistribute more than half (6 out of 10) of existing links

(red) at each block to new blocks without violating wiring and capacity constraints.

The expansion process with patch panels is shown in the bottom of Figure 7. Similar to the procedure for Clos described in §4.1, all new blocks (shown in orange) are first deployed and interconnected and links from the new blocks are routed to reserved ports on patch panels associated with existing blocks (shown in blue), before re-wiring begins.

For FatClique, rewiring one existing link requires releasing one patch panel port so that a new link can be added. Since links are already parts of existing bundles and routed through cable trays, we can not rewire them directly, e.g., by rerouting it from one patch panel to another. For example, link 1 (lower half of Figure 7) is originally connected blocks 1 and 3 by connecting ports a and b on the patch panel. Suppose we want to remove that link, and add two links, one from block 1 to block 5 (labeled 3), and another from block 3 to block 5 (labeled 4). The part of the original link (labeled 1) between the two patch panels is already bundled, so we cannot physically reroute it from block 3 to block 5. Instead, we effect re-wiring by releasing port a , connecting link 3 to port a , connecting link 1 to port c . Logically, this is equivalent to connecting ports a and d and b and c on the patch panel shown in lower half of Figure 7. This preserves bundling, while permitting expansion.

If the original topology has N_b blocks, by comparing the old and target topology, the total number of rewired links is computed by $N_b(N_b - 1)(L_{bb} - L'_{bb})/2$. For this example, the total number of links to be rewired is 9.

Iterative Expansion Plan Generation. By design, FatClique has fat edges, which allows draining more and more links at each step of the expansion, as network capacity increases. At each step, we drain links across all blocks uniformly, so that each block loses the same aggregate capacity. However the relationship between overall network capacity, and the number of links drained at every block in FatClique is unclear, because traffic needs to be sent over non-shortest paths to fully utilize the fabric.

Therefore, we use an iterative approach to expansion planning, where, at each step, we search for the maximal ratio of links to be drained that still preserves expansion SLO. (§A.4 discusses the algorithm in more detail). Our evaluation §6 shows that the number of expansion steps computed by this algorithm is much smaller than that for expanding symmetric Clos.

5.4 Discussion

Achieving low complexity. By construction, FatClique achieves low lifecycle management complexity (Table 3), while ensuring full-bisection bandwidth. It ensures high edge expansion, resulting in fewer switches. By packaging clique connections into a sub-block, it exports fewer external ports, an idea we call *port hiding*. By employing hierarchy and a regular (non-random) structure, it permits bundling and re-

quires fewer patch panels. By ensuring fat edges at each level of the hierarchy, it enables fewer re-wired links per patch panel, and fewer expansion steps. We quantify these in §6.

Scalability. Since Xpander and Jellyfish do not incorporate hierarchy, they can be scaled to arbitrarily large sizes. However, because Clos and FatClique are hierarchical, they can only scale to a fixed size for a given chip radix. Table 5 shows the maximum scale of each topology as a function of switch radix k . FatClique scales to the same order of magnitude as a 5-layer Clos. As shown in §6, both of them can scale to 64 times bisection bandwidth of Jupiter.

FatClique and Dragonfly. FatClique is inspired by Dragonfly [20] and they are both hierarchical topologies that use cliques as building blocks, but differ in several respects. First, for a given switch radix, FatClique can scale to larger topologies than Dragonfly because it incorporates one additional layer of hierarchy. Second, the Dragonfly class of topologies is defined by many more degrees of freedom than FatClique, so instantiating an instance of Dragonfly can require an expensive search [33]. In contrast, FatClique’s constraints enable more efficient search for candidate topologies. Finally, since Dragonfly does not explicitly incorporate constraints for expansion, a given instance of Dragonfly may not end up with fat edges.

Routing and Load Balancing on FatClique. Unlike for Clos, ECMP-based forwarding cannot be used achieve high utilization in more recently proposed topologies [20, 35, 32, 19]. FatClique belongs to this latter class, for which a combination of ECMP and Valiant Load Balancing [37] has been shown to achieve performance comparable to Clos [19].

6 Evaluating Lifecycle Complexity

In this section, we compare three classes of topologies, Clos, expander graphs and FatClique by our complexity metrics.

6.1 Methodology

Topology scales. Because the lifecycle complexity of topology classes can be a function of topology scale, we evaluate complexity across three different topology sizes based on the number of servers they support: *small*, *medium*, and *large*. Small topologies support as many servers as a 3-layer clos topology. Medium topologies support as many servers as 4-layer Clos. Large topologies support as many servers as 5-layer Clos topologies⁶. All our experiments in this section are based on comparing topologies at the same scale.

At each scale, we generate one topology for each of Clos, Xpander, Jellyfish, and FatClique. The characteristics of these topologies are listed in Table 6. All these topologies use 32-port switching chips, the most common switch radix available today for all port capacities [5]. To compare topologies

⁶To achieve low wiring complexity, a full 5-layer Clos topology would require patch panel racks with four times as many ports as available today, so we restrict ourselves to the largest Clos that can be constructed with today’s patch panel capacities

fairly, we need to equalize them first. Specifically, at a given scale, each topology has approximately the same bisection bandwidth, computed (following prior work [32, 35]) using METIS [18]. All topologies at the same scale support roughly the same number of servers; small, medium and large scale topologies achieve, respectively, $\frac{1}{4}$, 4, and 16 times capacity of Jupiter. (In A.8, we also compare these topologies using two other metrics).

Table 6 also shows the scale of individual building blocks of these topologies in terms of number of switches. For Clos, we use the algorithm in §A.1 to design building blocks (chassis) and then use them to compose Clos. One interesting aspect of this table is that, at the 3 scales we consider, a FatClique’s sub-block and block designs are *identical*, suggesting lower manufacturing and assembly complexity. We plan to explore this dimension in future work.

For each topology we compute the metrics listed in Table 3: the number of switches, the number of bundle types, the number of patch panels, the average number of re-wired links at a patch panel during each expansion step, and the number of expansion steps. To compute these, we need component parameters, and placement and expansion algorithms for each topology class.

Component Parameters. In keeping with [4, 40], we use optical links for all inter-rack links. We use 96 port 1RU patch panels [10] in our analysis. A 58RU [28] rack with patch panels can aggregate $2 * 96 * 58 = 11,136$ fibers. We call this rack a patch-panel rack. Most datacenter settings, such as rack dimensions, aisle dimensions, cable routing and distance between cable trays follow practices in [26]. We list all parameters used in our paper in §A.7.

Placement Algorithms. For Clos, following Facebook’s fb-fabric [3], spine blocks are placed at the center of the datacenter, which might take multiple rows of racks, and pods are placed at two sides of spine blocks. Each pod is organized into a rectangular area with aggregation blocks placed in the middle to reduce the cable length from ToR to aggregation. FatClique’s placement algorithm is discussed in §5.2. For Xpander, we use the placement algorithm proposed in [19]. We follow the practice that all switches in a metanode are placed closed to each other. However, instead of placing a metanode into a row of racks, we place a metanode into a rectangular area of racks, which reduces cable lengths when metanodes are large. For Jellyfish, we design a random search algorithm to aggressively reduce the cable length (§A.2).

Expansion Algorithms. For Clos, as shown in [38], it is fairly complex to compute the optimal number of rewired links for asymmetric Clos during expansion. However, when the original and target topologies are both symmetric, this number is easy to compute. For this case, we design an optimal algorithm (§A.5) which rewrites the maximum number of links at each step and therefore uses the smallest number of steps to finish expansion. For FatClique, we use the algorithm discussed in §5.3. For Xpander and Jellyfish, we design an

Topology	Clos						FatClique				Xpander			Jellyfish		
	Scale	e	a	sp	pod	cap	svr	sub-block	block	cap	svr	metanode	cap	svr	cap	svr
Small	1	16	1	32	327T	8.2k		6	150	337T	8.1k	41	351T	8.2k	350T	8.2k
Medium	1	16	48	32	5.24P	131k		6	150	5.40P	132k	655	5.56P	131k	5.56P	131k
Large	1	512	48	768	20.96P	524k		6	150	21.36P	523k	2620	22.27P	524k	22.27P	524k

Table 6: Capacities of topologies built with 32 port 40G switches. Small, medium and large scale topologies achieve $\frac{1}{4}$, 4, 16 times capacity of Jupiter. The table also shows sizes of individual building blocks of these topologies in terms of number of switches. Abbreviations: e:edge, a:aggregation, sp:spine, cap:capacity, svr:server.

expansion algorithm based on the intuition from [35, 32] that, to expand a topology by n ports requires breaking $\frac{n}{2}$ existing links. Finally, we have found that for all topologies, the number of expansion steps at a given SLO is *scale invariant*: it does not depend on the size of the original topology as long as the expansion ratio (target-topology-size-to-original-topology-size ratio) is fixed (§A.3).

Presenting results. In order to bring out the relative merits of topologies, and trends of how cost and complexity increase with scale, we present values for metrics we measure for all topologies and scales in the same graph. In most cases, we present the absolute values of these metrics; in some cases though, because our three topologies span a large size range, for some metrics the results across topologies are so far apart that we are unable to do so without loss of information. In these cases, we normalize our results by the most expensive, or complex topology.

6.2 Patch Panel Placement

The placement of patch panels is determined both by the structure of the topology and its scale.

Between edge and aggregation layers in Clos. For small and medium scale Clos, no patch panels are needed between edge and aggregation layers. Each pod at these scales contains 16 aggregation switches, which can be packed into a single rack (we call this an *aggregation-rack*). Given that a pod at this scale is small, all links from the edge can connect to this rack. Since all links connect to one physical location, bundles form naturally. In this case, each bundle from edge racks contains 3×16 fibers⁷. Therefore, no patch panels are needed between edge and aggregation layers.

However, a large Clos needs one layer of patch panels between edge and aggregation layers since a pod at this scale is large. An aggregation block consists of 16 middle blocks⁸, each with 32 switches. The aggregation block by itself occupies a single rack. Based on the logical connectivity, links from any edge need to connect to all middle blocks. Without using patch panels, each bundle could at most contain $3 \times 16/16 = 3$ fibers. In our design, we use patch panels to aggregate local bundles from edges first and then rebundle them on patch panels to form new high capacity bundles from patch panels to aggregation racks. Based on the patch panel

rack capacity constraint, two patch panel racks are enough to form high capacity bundles from edge to aggregation layers. Specifically, in our design 128 edge switches and 8 aggregation racks connect to a single patch panel. In this design, each edge-side bundle contains 48 fibers and each aggregation-side bundle contains 128 fibers.

Between aggregation and spine layers. The topology between aggregation and spine layer in Clos is much larger than that inside a pod. For this reason, to form high capacity bundles, two layers of patch panels are needed. As shown in Figure 1, one layer of patch panels is placed near spine blocks at the center of the data center floor. Each patch panel rack aggregates local bundles from four spine racks in medium and large scale topologies. Similarly, another layer of patch panels are placed near aggregation rack, permitting long bundles between those patch panels.

In expanders and FatClique. For Jellyfish, Xpander and FatClique, patch panels are deployed at the server block side and long bundles form between those patch panels. In FatClique, each block requires one patch panel rack (§5.3). In a large Xpander, since a metanode is too big (Table 6), it is not possible to use one patch panel rack to aggregate all links from a metanode. Therefore, we divide a metanode into homogeneous sections, called sub-metanodes, such that links from a sub-metanode can be aggregated at one patch panel rack. For Jellyfish, we partition the topology into groups, each of which contains the same number of switches as in a block in FatClique, so each group needs one patch panel rack.

6.3 Deployment Complexity

In this section, we evaluate our different topologies by our three measures of deployment complexity (§3.2).

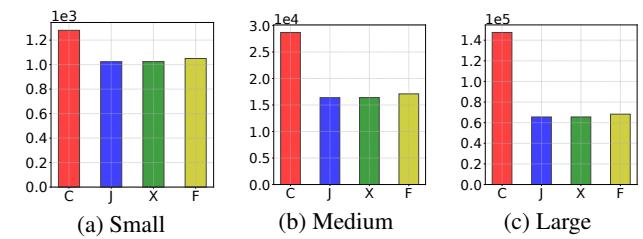


Figure 8: Number of switches. C is Clos, J is Jellyfish, X is Xpander and F is FatClique.

Number of Switches. Figure 8 shows how the different topologies compare in terms of number of switches used at various topology scales. Figure 8(a) shows the total number of

⁷In our setting, each rack with 58RU can accommodate at most 3 switches and 48 associated servers. The total number of links out of this rack is 3×16 .

⁸We follow the terminology in [31]. A middle block is a sub-block in an aggregation block.

switches for the small topologies, Figure 8(b) for the medium, and Figure 8(c) for the large. The y-axes increase in scale by about an order of magnitude from left to right. FatClique has 20% fewer switches than Clos for a small topology, and 50% fewer for the large. The results for Jellyfish and Xpander are similar, consistent with findings in [35, 32]. This benefit comes from the edge expansion property of the non-Clos topologies we consider. This implies that Clos topologies, at large scale, *may require nearly twice the capital expenditures* for switches, racks, and space as the other topologies.

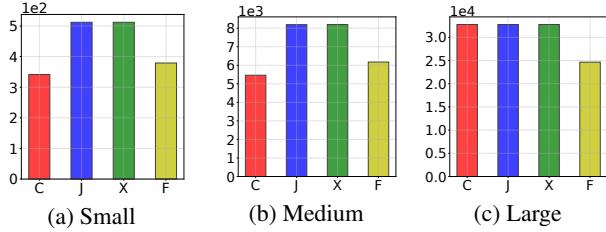


Figure 9: Number of patch panels. **C** is Clos, **J** is Jellyfish, **X** is Xpander and **F** is FatClique.

Number of Patch panels. Figure 9 shows the number of patch panels at different scales. As before, across these graphs, the y-axis scale increases approximately by one order of magnitude from left to right. At small and medium scales, Clos relies on patch panels mainly for connections between aggregation and spine blocks. Of all topologies at these scales, Clos uses the fewest number of patch panels: FatClique uses about 11% more patch panels, and Jellyfish and Xpander use almost 44-50% more. Xpander and Jellyfish rely on patch panels for all northbound links, and therefore in general, as scale increases, the number of patch panels in these networks grows (as seen by the increase in the y-axis scale from left to right).

At large scale, however, Clos needs many more patch panels, comparable to Xpander and Jellyfish. At this scale, Clos aggregation blocks span multiple racks, and patch panels are also needed for connections between ToRs and aggregation blocks. Here, FatClique’s careful packaging strategy becomes more evident, as it needs nearly 25% fewer patch panels than Clos. The majority of patch panels used in FatClique at all scales comes from inter-block links (which increase with scale).

For this metric, Clos and FatClique are comparable at small and medium scales, but FatClique dominates at large scale.

Scale	# Bundle Types			
	Clos	FatClique	Xpander	Jellyfish
Small	8	11	11	28
Medium	74	61	976	1577
Large	322	212	3034	3678

Table 7: Bundle Types (Switch Radix = 32)

Number of Bundle Types. Table 7 shows the number of bundle types used by different topologies at different scales. A bundle type (§3.1) is characterized by (a) the number of

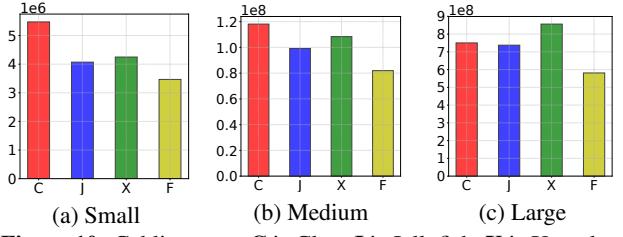


Figure 10: Cabling cost. **C** is Clos, **J** is Jellyfish, **X** is Xpander and **F** is FatClique.

fibers in the bundle, and (b) the length of the bundle. The number of bundle types is a measure of wiring complexity. In this table, if bundles differ by more than 1m in length, they are designated as separate bundle types.

Table 7 shows that Clos and FatClique use the fewest number of bundle types; this is due to the hierarchical structure of the topology, where links between different elements in the hierarchy can be bundled. As the topology size increases, the number of bundle types also increases in these topologies, by a factor of about 40 for Clos to 20 for FatClique when going from small to large topologies.

On the other hand, Xpander and Jellyfish use an order of magnitude more bundle types compared to Clos and FatClique at medium and large scales, but use a comparable number for small scale topologies. Even at the small scale, Jellyfish uses many more bundle types because it uses a random connectivity pattern. At small scales Xpander metanodes use a single patch panel rack and bundles from all metanodes are uniform. With larger scales, Xpander metanodes become too big to connect to a single patch panel rack. We have to divide a metanode into several homogeneous sub-metanodes such that all links from sub-metanodes connect to a patch panel rack. However, because of the randomness in connectivity, this subdivision cannot ensure uniformity of bundles egressing sub-metanode patch panel racks, so we find that Xpander has a large number of bundle types in medium and large topologies.

Thus, by this metric, Clos and FatClique have the lowest complexity across all three scales, while Xpander and Jellyfish have an order of magnitude more complexity. Moreover, across all metrics FatClique has lowest deployment complexity, especially at large scales.

Case Study: Quantifying cabling costs. While not all aspects of lifecycle management complexity can be translated to actual dollar costs, it is possible to estimate one aspect, namely the cost of cables. Cabling cost includes the cost of transceivers and cables, and is reported to be the dominant component of overall datacenter network cost [31, 20]. We can estimate costs because our placement algorithms generate cable or bundle lengths, the topology packaging determines the number of transceivers, and estimates of cable and transceiver costs as a function of cable length are publicly available [7].

Figure 10 quantifies the cabling cost of all topologies,

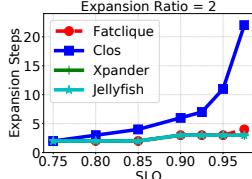


Figure 11: Expansion steps

across different scales. Clos has higher cabling costs at small and medium scales compared to expander graphs, although the relative difference decreases at medium scale. At large scales, the reverse is true. Clos is around 12% cheaper than Xpander in terms of cabling cost since Xpander does not support port-hiding at all and uses more long inter-rack cables. Thus, given that cabling cost is the dominant component of overall cost, it is unclear whether the tradeoff Xpander and Jellyfish makes in terms of number of switches and cabling design pays off in terms of capital expenditure, especially at large scale.

We find that FatClique has the lowest cabling cost of the topologies we study with a cabling cost 23-36% less than Clos. This result came as a surprise to us, because intuitively topologies that require all-to-all clique like connections might use longer length cables (and therefore more expensive transceivers). However on deeper examination, we found that Clos uses a larger number of cables (especially inter-rack cables) compared to other topologies since it has a relatively higher number of switches (Figure 8) to achieve the same bisection bandwidth. Thus, more switches leads to more racks and datacenter floor area, which stretches the cable length. All those factors together explain why Clos cabling costs are higher than FatClique's.

Thus, from an equipment capital expenditure perspective, at large scale a *FatClique* can be at least 23% cheaper than a *Clos*, because it has at least 23% fewer switches, 33% fewer patch panel racks, and 23% lower cabling costs than Clos.

6.4 Expansion Complexity

In this section, we evaluate topologies by our two measures of expansion complexity (§4.3): number of expansion steps required, and number of rewired-links per patch panel rack per step. Since the number of steps is scale-invariant (§6.1), we only present the results from expanding medium size topologies for both metrics⁹. When evaluating Clos, we study the expansion of symmetric Clos topologies; generic Clos expansion is studied in [38]. As discussed in §6.1, for symmetric Clos, we have developed an algorithm with optimal number of rewiring steps.

Number of expansion steps. Figure 11 shows the number of steps (y-axis) required to expand topologies to twice their existing size (*expansion ratio* = 2) at different expansion SLOs (x-axis). We find that at 75% SLO, all topologies require the same number of expansion steps. But the number

⁹We have verified that the relative trend in the number of re-wired links per patch panel holds for small and large topologies

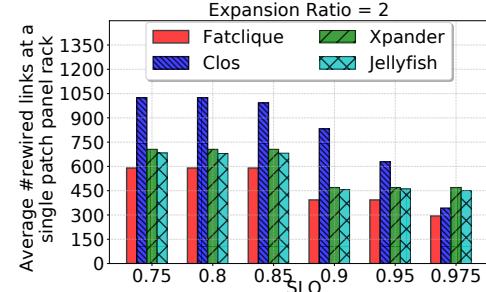


Figure 12: Average Number of Rewired Links at a Single Patch Panel across Steps

of steps required to expand Clos with tighter SLOs steeply increases. This is because the number of links that can be rewired per aggregation block in Clos per step, is limited (due to north-to-south capacity ratio §4.3) by the SLO. The tighter the SLO, fewer the number of links rewired per aggregation block per step, and larger the number of steps required to complete expansion. FatClique, Xpander and Jellyfish require fewer and comparable number of expansion steps due to their fat edge property, allowing many more links to be rewired per block per step. Their curves largely overlap (with FatClique taking one more step as SLO increases beyond 95%).

Number of rewired links per patch panel rack per step. This metric is an indication of the time it takes to finish an expansion step because, today, rewiring each patch panel requires a human operator [38]. A datacenter operator can reduce re-wiring time by employing staff to rewire each patch panel rack in parallel, in which case, the number of links per patch panel rack per step is a good indicator of the complexity of an expansion step. Figure 12 shows the average of the maximum rewired links per patch panel rack, per step (y-axis), when expanding to twice the topology size size at different SLOs (x-axis). Even though the north-to-south capacity ratio restricts the number of links that can be rewired in Clos per step, the number of rewired links per patch panel rack per step in Clos remains consistently higher than other topologies, until we hit 97.5% SLO. The reason is that the links that need to be rewired in Clos are usually concentrated in few patch panel racks by design. As such, it is harder to parallelize rewiring in Clos, than it is in the other topologies. FatClique has the lowest rewiring step complexity across all topologies.

6.5 FatClique Result Summary

We find that FatClique is the best at most scales by *all* our complexity metrics. (The one exception is that at small and medium scales, Clos has slightly fewer patch panels). It uses 50% fewer switches and 33% fewer patch panels than Clos at large scale, and has a 23% lower cabling cost (an estimate we are able to derive from published cable prices). Finally, FatClique can permit fast expansion while degrading network capacity by small amounts (2.5-10%): at these levels, Clos can take 5 × longer to expand the topology, and each step of Clos expansion can take longer than FatClique because the

number of links to be rewired at each step per patch panel can be 30-50% higher.

7 Related Work

Topology Design. Previous topology designs have focused on cost effective, high capacity and low diameter datacenter topologies like [6, 35, 32, 4, 20]. Although they achieve good performance and cost properties, the lifecycle management complexity of these topologies have not been investigated either in the original papers or in subsequent work that has compared topologies [26, 27]. In contrast to these, we explore topology designs that have low lifecycle complexity. Recent work has explored datacenter topologies based on free space optics [24, 11, 9, 16, 39] but because we lack operational experience with them at scale, it is harder to design and evaluate lifecycle complexity metrics for them.

Topology Expansion. Prior work has discussed several aspects of topology expansion [30, 32, 35, 8, 38]. Condor [30] permits synthesis of Clos-based datacenter topologies with declarative constraints some of which can be used to specify expansion properties. A more recent paper [38] attempts to develop a target topology for expansion, given an existing Clos topology, that would require the least number of link rewiring. REWIRE [8] finds target expansion topologies with highest capacity and smallest latency without preserving topological structure. Jellyfish [32] and Xpander [35] study expansion properties of their topology, but do not consider practical details in re-wiring. Unlike these, our work is examines lifecycle management as a whole, across different topology classes, and develops new performance-equivalent topologies with better lifecycle management properties.

8 Conclusions and Future Work

In this paper, we have attempted to characterize the complexity of lifecycle management of datacenter topologies, an unexplored but critically important area of research. Lifecycle management consists of network deployment and expansion, and we devise metrics that capture the complexity of each. We use these to compare topology classes explored in the research literature: Clos and expander graphs. We find that each class has low complexity by some metrics, but high by others. However, our evaluation suggests topological features important for low lifecycle complexity: hierarchy, edge expansion and fat edges. We design a family of topologies called FatClique that incorporates these features, and this class has low complexity by all our metrics at large scale.

As the management complexity of networks increases, the importance of *designing for manageability* will increase in the coming years. Our paper is only a first step in this direction; several future directions remain.

Topology oversubscription. In our comparisons, we have only considered topologies with an over-subscription ratio of 1:1. Jupiter [31] permits over-subscription at the edge of the network, but there is anecdotal evidence that providers also

over-subscribe at higher levels in Clos topologies. To explore the manageability of over-subscribed topologies it will be necessary to design over-subscription techniques in FatClique, Xpander and Jellyfish in a way in which all topologies can be compared on a equal footing.

Topology heterogeneity. In practice, topologies have a long lifetime over which they accrue heterogeneity: new blocks with higher radix switches, patch panels with different port counts *etc.* These complicate lifecycle management. To evaluate these, we need to develop data-driven models for how heterogeneity accrues in topologies over time and adapt our metrics for lifecycle complexity to accommodate heterogeneity.

Other management problems. Our paper focuses on topology lifecycle management, and explicitly does not consider other network management problems like fault isolation or control plane complexity. Designs for manageability must take these into account.

References

- [1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. Hyperx: Topology, routing, and packaging of efficient large-scale networks. In *Proc. SC9*, 2009.
- [2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM*, 2008.
- [3] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. <https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/>.
- [4] M. Besta and T. Hoefer. Slim fly: A cost effective low-diameter network topology. In *Proc. SC14*, 2014.
- [5] Broadcom Inc. Broadcom Tomahawk Switching chips. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series>.
- [6] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953.
- [7] Colfax International. Colfax direct. <http://www.colfaxdirect.com>.
- [8] Andrew R. Curtis, Tommy Carpenter, Mustafa Elsheikh, Alejandro López-Ortiz, and Srinivasan Keshav. Rewire: An optimization-based framework for unstructured data center network design. In *Proc. IEEE INFOCOMM*, 2012.
- [9] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proc. ACM SIGCOMM*, 2010.
- [10] FS.COM. 96 Fibers 12x MTP/MPO-8 to LC/UPC Single Mode 1U 40GB QSFP+ Breakout Patch Panel Flat. <https://www.fs.com/products/43552.html>.
- [11] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper. Projector: Agile reconfigurable data center interconnect. In *Proc. ACM SIGCOMM*, 2016.
- [12] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proc. ACM SIGCOMM*, 2016.
- [13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sen-gupta. Vi2: a scalable and flexible data center network. In *Proc. ACM SIGCOMM*, 2009.
- [14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, 2008.
- [15] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proc. ACM SIGCOMM*, 2008.
- [16] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proc. ACM SIGCOMM*, 2014.
- [17] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc.*, 43(04):439–562, August 2006.
- [18] G. Karypis and V. Kumar. A fast and high quality multi-level scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.
- [19] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proc. ACM SIGCOMM*, 2017.
- [20] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, 2008.
- [21] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. Computers*, C-24(12):1145–1155, Dec 1975.
- [22] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proc. USENIX NSDI*, 2013.
- [23] S. Mandal. Lessons learned from b4, google's sdn wan. https://www.usenix.org/sites/default/files/conference/protected-files/atc15_slides_mandal.pdf.
- [24] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proc. ACM SIGCOMM*, 2017.
- [25] J. Mitchell. What are Patch Panels & When to Use Them? <https://www.lonestarracks.com/news/2016/10/28/patch-panels/>.

- [26] J. Mudigonda, P. Yalagandula, and J. C. Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. In *Proc. USENIX ATC*, 2011.
- [27] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A cost comparison of datacenter network architectures. In *Proceedings of the 6th International Conference, Co-NEXT '10*, 2010.
- [28] RackSolutions. Open Frame Server Racks. <https://www.racksolutions.com/server-racks-cabinets-enclosures.html>.
- [29] Rackspace US, INC. The Rackspace Cloud. www.rackspacecloud.com.
- [30] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *Proc. USENIX NSDI*, 2015.
- [31] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proc. ACM SIGCOMM*, 2015.
- [32] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Proc. USENIX NSDI*, 2012.
- [33] M. Y. Teh, J. J. Wilke, K. Bergman, and S. Rumley. Design space exploration of the dragonfly topology. In *ISC Workshops*, 2017.
- [34] The Siemon Company. Trunk Cable Planning & Installation Guide. https://www.siemon.com/us/white_papers/07-09-24-trunk-cable-planning-installation.asp.
- [35] A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. ACM CoNEXT*, 2016.
- [36] M. R. Zargham. *Computer Architecture: Single and Parallel Systems*. Prentice Hall, 1996.
- [37] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone with valiant load-balancing. In *Proc. IEEE IWQoS*, 2005.
- [38] S. Zhao, R. Wang, J. Zhou, J. Ong, J. Mogul, and A. Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *Proc. USENIX NSDI*, 2019.
- [39] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *Proc. ACM SIGCOMM*, 2012.
- [40] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, and T. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. ACM SIGCOMM*, 2017.

A Appendix

A.1 Clos Generation Algorithm

For Clos topologies, the canonical recursive algorithm in [36] can only generate non-modular topologies as shown in Figure 13. In practice, as shown in Jupiter [31], the topology is composed of heterogenous building blocks (chassis), which are packed into a single rack and therefore enforce port hiding (the idea that as few ports from a rack are exposed outside the rack). Although Jupiter is modular and supports port hiding, it is single instance of a Clos-like topology with a specific set of parameters. We seek an algorithm that can take any valid set of Clos parameters and produce chassis-based topologies automatically. Besides, it would be desirable for this algorithm to generate all possible feasible topologies satisfying the parameters, so we can select the one that is most compactly packed.

Our logical Clos generation algorithm achieves these goals. Specifically, the algorithm uses the following steps:

1. *Compute the total number of layers of homogeneous switching chips needed.* Namely, given N servers and radix k switches, we use $n = \log_k(\frac{N}{2})$ to compute the number of layers of chips n needed.
2. *Determine the total number of layers of chips for edge, aggregation and core layers,* which are represented by e , a and s respectively, such that $e + a + s = n$.
3. *Identify blocks for edge, aggregation and core layer.* Clos networks rely on every edge being able to reach every spine through exactly one path, by fanning out via as many different aggregation blocks as possible (and vice versa). We find that the resulting interconnection is a derivative of the classical perfect shuffle Omega network ([21], e.g., aggregation blocks in Figure 14 and Figure 15). Therefore, we use Omega networks to build both the edge and aggregation blocks, and to define the connections between edge-aggregation and aggregation-spines. The spine block on the other hand needs to be rearrangeably-nonblocking, so it can relay flows from any edge to any other edge with full capacity. Therefore it is built as a smaller Clos topology [6] (e.g., spine blocks in Figure 14).
4. *Compose the whole network using edge, aggregation and core blocks.* The process to compose the whole topology is to link all these blocks and uses the same procedure as Jupiter[31].

We have verified that topologies generated by our construction algorithm, such as the ones in Figure 14 and Figure 15, are isomorphic to a topology generated using the canonical algorithm in Figure 13. By changing different combinations of e , a and s , we can obtain multiple candidate topologies, as shown in Figure 14 and Figure 15.

A.2 Jellyfish Placement Algorithm

For Jellyfish, we use a heuristic random search algorithm to place switches and servers. The algorithm works as follows. At each stage of the algorithm, a node can be in one of two states: placed, or un-placed. A placed node is one which has been positioned in a rack. Each step of the algorithm randomly selects an un-placed node. If the selected node has logical neighbor nodes that have already been placed, we place this node at the centroid of the area formed by its placed logical neighbors. If no placed neighbor exists, the algorithm randomly selects a rack to place the node. We have also tried other heuristics like neighbor-first, which tries to place a switch's logical neighbors as close as possible around it. However, this performs worse than our algorithm.

A.3 Scale-invariance of Expansion

Scale-invariance of Expandability for Symmetric Clos. For a symmetric Clos network, the number of expansion steps is scale-invariant and independent of the degree to which the original topology is partially deployed. Consider a simplified Clos where the original topology has g aggregation blocks. Each aggregation block has p ports for spine-aggregation links, each of which has the unit capacity. Assume the worst-case traffic in which all sources are located in the left half of aggregation blocks and all destinations are in the right half. This network contains $g \cdot p/2$ crossing links between left and right halves. If, during expansion, the network is expected to support a demand of d units capacity per aggregation block, the total demand traversing the cut between the left and right halves in one direction is $d \cdot g/2$. Then, the maximum number of links that can be redistributed in an expansion step is $k = g \cdot p/2 - d \cdot g/2 = g(p-d)/2$, which is linear in the number of aggregation blocks (network size). This linearity between k and g implies scale-invariant expandability, e.g., when an aggregation block is doubled to $2g$, the maximum number of redistributed links per expansion step becomes $2k$.

Scale-invariance of Expandability for Jellyfish, Xpander, and FatClique. A random graph consists of s nodes, which is a first-order approximation for Jellyfish's switch, Xpander's metanode and FatClique's block. Each node has p inter-node ports, so there are $s \cdot p/2$ inter-node links. We can treat the network as a bipartite graph. We assume the worst-case traffic matrix, where all traffic is sent through one part of the bipartite graph to the other. Suppose an expansion SLO requires each source-destination node pair to support d unit demand. Then the total demands from all sources are $d \cdot s/2$. The probability of a link being a cross link is $1/2$, and the expected number of cross links is $s \cdot p/4$. These cross links are expected to be the bottleneck between the source-destinations pairs. Therefore, in the first expansion step, we can redistribute at most $k = s \cdot p/4 - d \cdot s/2 = s(p/4 - d/2)$ links, and the maximum number of redistributed links is linear in the number of nodes (network size), e.g., if the number of

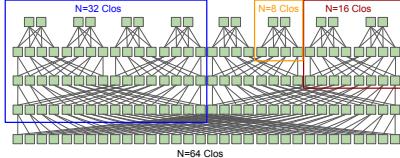


Figure 13: Recursive Construction

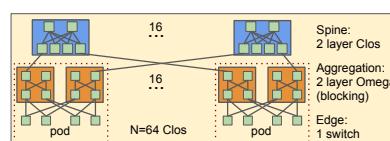


Figure 14: Block-Based Construction 1

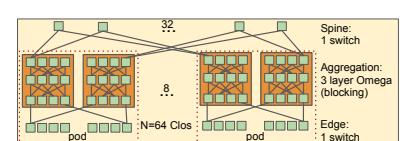


Figure 15: Block-Based Construction 2

nodes is doubled to $2s$, we can redistribute $2k$ links in the first step. It is easy to see that, after each expansion step, the number of links added to the bottleneck is also linear with the number of nodes, so the expandability is scale-invariant.

A.4 FatClique Expansion Algorithm

Algorithm 1 shows the expansion algorithm for FatClique. The input to the algorithm includes original and target topologies T^o and T^n , the link break ratio during an expansion step α , multipliers $\beta < 1$ and $\gamma > 1$, which are used to adjust α based on network capacity. α specifies the fraction of existing links that must be broken for re-wiring. The output of the algorithm is the expansion plan $Plan$.

Our expansion algorithm is an iterative trial-and-error approach (Line 4). Each iteration tries to find the right amount of links to break while satisfying the aggregate capacity constraint (Line 11) and the *edge capacity constraint* (Line 6), which guarantees that the *north-to-south* capacity ratio is always not smaller than 1 during any expansion step. If all constraints are satisfied, we accept this plan and tentatively increase the link break ratio α (Line 16, by multiplying by γ) due to capacity increase. Otherwise, the link break ratio α (Line 12) is decreased (by multiplying by β conservatively.)

```

input :  $T^o, T^n, SLO$ 
output:  $Plan$ 
1 Initialize  $\alpha \in (0, \infty), \beta \in (0, 1), \gamma \in (1, \infty)$ 
2 Find the total set of links to break,  $L$ , based on  $T^o$  and  $T^n$ 
3 Compute original capacity  $c_0$ 
4 while  $|L| > 0$  do
5   Select a subset of links  $L_b$ , from  $L$  uniformly across all
      blocks, where  $|L_b| = \alpha |L|$ .
6   if  $L_b$  does not satisfy edge capacity constraint then
7      $\alpha = \alpha \cdot \beta$ 
8   end
9   Delete  $L_b$  from  $T^o$ 
10   $c = \text{ComputeCapacity}(T^o)$ 
11  if  $c < c_0 \cdot SLO$  then
12     $\alpha = \alpha \cdot \beta$ 
13    add  $L_b$  back to  $T^o$ 
14  else
15     $T^o = \text{AddNewLinks}(L_b, T^o, T^n)$ 
16     $\alpha = \alpha \cdot \gamma$ 
17     $Plan.add(L_b)$ 
18  end
19 end
```

Algorithm 1: FatClique Expansion Plan Generation

A.5 Expansion for Clos

Since the motivation of this work is to compare topologies, we only focus on developing optimal expansion solutions for symmetrical Clos. More general algorithms for Clos' expansion can be found in [38]. Also, similar to [38], we assume the worst case traffic matrices for Clos, *i.e.*, servers under a pod will send traffic using full capacity to servers in other pods.

Target Topology Generation. As mentioned in §4.1, a pod is the unit of expansion in Clos. When we add new pods and associated spines to a Clos topology for expansion, the wiring pattern inside a pod remains unchanged. To make the target topology non-blocking and to ease expansion (*i.e.*, number of to-be-redistributed links on each pod is the same), links from a pod should be distributed across all spines as evenly as possible.

Expansion plan generation. Once a target Clos topology is generated, the next step is to redistribute links to convert the original topology into the target topology. By comparing the original and target topology, it is easy to figure out which new links should be routed to which patch panels to satisfy the wiring constraint. In this section, we mainly focus on how to drain links such that the capacity constraint is satisfied and the number of expansion steps is minimized.

Insight 1: Maximum rewired links at each pod is bounded. At each expansion step, when links are drained, network capacity drops. At the same time, as expansion proceeds, new devices are added incrementally, the overall network capacity increases gradually during the whole expansion process. In general, during expansion, the incrementally added capacity should be leveraged to speed up the expansion process. Due to the thin edges in Clos, no matter what the overall network capacity is, the maximum number links to be drained at each pod is bounded by the number of links on each pod multiplied by $(1 - SLO)$. Figure 16 shows an example. The leftmost figure is a folded Clos, where each pod has 16 links (4 trunks). If the SLO is 75%, the maximum number of links to be drained at a single step is $16 \times (1 - 0.75) = 4$. For our expansion plan generation algorithm, we try to achieve this bound at each pod at every single step.

Insight 2: Drain links at spines uniformly across edges (pods). Given the number of links allowed to be drained at each pod, we need to carefully select which links are to be drained. Figure 16 shows two draining plans. Drain plan 1 will drain links from two spines uniformly across all pods. The residual capacity is 48, satisfying the requirement

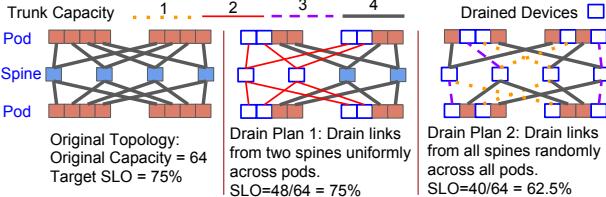


Figure 16: Original topology is a Folded Clos with capacity=64. The required SLO during expansion is 75%, which means capacity should be no smaller than 48. There are 16 links on each pod. Due to the SLO constraint, for all plans, 4 links are allowed to be drained at each pod.

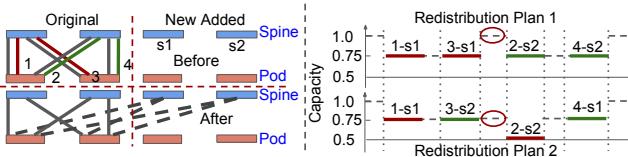


Figure 17: Clos Draining Link Redistribution Scheduling.

SLO=75%. By uniformly, we mean the number of drained links between the spine and all pods are the same. Drain plan 2 also drains 4 links from each pods but not uniformly (for example, more links are drained at the third spine compared to the fourth spine), which violates the SLO requirement since the residual capacity is only 40, smaller than the 48 in Drain plan 1.

Insight 3: Create physical loops by selecting the right target spines. Ideally, drained links with the same index on a pod on the same original spine should be redistributed to the same spine because the traffic sent from the pod to the target spine has a return path to the pod. Otherwise, the traffic will be dropped. Figure 17 illustrates this insight. The right side of the figure shows the performance of two redistribution plans. The y axis shows the normalized capacity of the network at each expansion step. In the first plan, link 1 is first moved to spine s1 (1-s1), followed by link 3 to the same spine s1 (3-s1) which results in 75% capacity loss, since the two pods are connected by three paths instead of four. Once links 1 and 3 are undrained, s1 connects the two pods by a fourth path, and the normalized capacity is restored to 1. This redistribution step now provides leeway for supporting 25% capacity loss in the next step. In this next step, links 2 and 4 are rewired to connect to s2. During the rewiring, capacity again drops to 75%, with three paths between the pods. On undraining links 2 and 4, the capacity is once again restored to 1. In contrast, redistribution plan 2 violates SLO because it does not focus on restoring capacity by establishing paths via the new spine, as suggested by the insight (links 1 and 3 are moved to different spines).

Inspired by these insights, we designed Algorithm 2, which can achieve all our insights simultaneously when both original and target topologies are symmetric. *The algorithm is optimal since at every expansion step, it achieves the upper bound of the links that could be drained. Therefore, our algorithm uses smallest steps to expand Clos.*

The input to the algorithm is the original and new *symmetric* topology T^o and T^n . We use T_{sp}^o and T_{sp}^n to represent the number of links between spine s and pod p in the old and new topology respectively. Initially, $T_{s'p}^o = 0$, where s' is a new spine. The output of the algorithm is the draining plan, $Subplan_i$, for expansion step i . The final expansion plan $Plan = \{Subplan_i\}$ and the number of $Subplan$, $|Plan|$, is the total expansion step.

The algorithm starts by indexing old spines, new spines and links on each pod from left to right respectively (Line 1-2), which are critical for the correctness of the algorithm since the algorithm relies on these indexes to break ties when selecting spines and links to redistribute. Then, based on our Insight 1, Line 3 computes the upper bound on the number of links to be redistributed on each pod, n_p . We show experimentally that our algorithm can always achieve this upper bound in each individual step as long as T^o and T^n are symmetric. Next, the algorithm iterates over all indexed old spines (Line 4) and tries to drain n_p links uniformly across all pods (Line 5) such that Insight 2 is satisfied. Line 6 compares the number of remaining to-be-redistributed links δ_{sp} and n_p and is useful only at the last expansion step. For each pod, the algorithm needs to find spines to redistribute links to (Line 7-14) while satisfying the constraint in Insight 3, i.e., drained links with the same index on a pod on the same original spine are redistributed to the same spine. Due to indexing and symmetric structure of Clos, our algorithm can always satisfy Insight 3. Specifically, when selecting spines, the spine satisfying $\delta_{s'p} = T_{s'p}^n - T_{s'p}^o > 0$ with the smallest index will be considered first (Line 8-Line 10). When selecting links from pod to redistribute, we always select the first n_a links to redistribute (Line 14).

Theorem 1 *Algorithm 2 produces the optimal expansion plan for Clos topology.*

The proof is simple. Since at every expansion step, our algorithm achieves the upper bound of the links that could be drained, our algorithm uses smallest steps to finish the expansion.

A.6 FatClique Topology Synthesis Algorithm

The topology synthesis algorithm for FatClique is shown in Algorithm 3. Essentially, the algorithm is a search algorithm, and leverages the constraints C_1 to C_6 in §5.1 to prune the search space. It works as follows. The outermost loop (Line 2) enumerates the number of racks used for a sub-block. Based on the rack space constraints, sub-block size S_c is determined Line 4. Next, the algorithm iterates over the number of sub-blocks in a block S_b Line 5, whose size is constrained by $MaxBlockSize$. Inside this loop, we leverage constraints C_1 to C_6 and derivations in §5.1 to find the feasible set of p_c , which is represented by P_c (Line 6). Then we construct FatClique based all design variables Line 8 and compute its capacity Line 9. If the capacity matches the target capacity,

```

input :  $T^o, T^n, SLO$ 
output: Subplan
1 Index original and new spines from left to right starting from 1
   respectively
2 Index links at each pod from left to right starting from 1
3  $\forall$  pod  $p, n_p = \text{num\_links\_per\_pod} \cdot (1 - SLO)$  // Insight 1
4 foreach Original Spine  $s$  do
5   foreach pod  $p$  do // Insight 2
6      $\delta_{sp} = T_{sp}^o - T_{sp}^n, n_p = \min(n_p, \delta_{sp})$  // Insight 2
7     while  $n_p > 0$  do
8       foreach New Spine  $s'$  do // Insight 3
9          $\delta_{s'p} = T_{s'p}^n - T_{s'p}^o$ 
10        if  $\delta_{s'p} > 0$  then break
11      end
12       $n_a = \min(\delta_{s'p}, n_p)$ 
13      Find the first  $n_a$  to-be-distributed links,  $L_{sp}$ 
14       $n_p = n_p - n_a$ , update( $T^o$ )
15      Subplan.add( $L_{sp}$ )
16    end
17  end
18 end
19 end

```

Algorithm 2: Single Step Clos Expansion Plan Generation

we add this topology into candidate set (Line 15). If the capacity is larger than required, the algorithm will increase s by 1 which will decrease the number of switches used $n = N/s$ (N is fixed) and therefore reduce the network capacity in next search step (Line 13). If the capacity is smaller than required, the algorithm will decrease s by 1 (Line 11) to increase the number of switches and capacity in next search step.

A.7 Parameter Setting

The cable price with transceivers used in our evaluation is listed in Table 9. We found that a simple linear model does not fit the data. The data is better approximated by a piecewise linear function: cables shorter than 100 meters are fit using one linear model and cables beyond 100 meters are fit using another linear model. The latter has a larger slope because beyond 100 meters, more advanced and expensive transceivers are necessary. In our experiment, since we only know the discrete price for cables and associated transceivers, we do the following: if the length of the cable is X , we use the exact price; if the length is larger than X , we use the first cable price larger than X .

```

input :  $N, r, Cap^*, s_0$ 
output: candidate
1 candidate = []
2 for  $i = 1; i < MaxRackPerSubblock; i++$  do
3    $s = s_0$ 
4    $S_c = i \cdot RackCapacity / (1 + s)$ 
5   for  $S_b = 1; S_b <= MaxBlockSize; S_b++$  do
6      $P_c = \text{CheckConstraints}(S_c, S_b)$ 
7     foreach  $p_c$  in  $P_c$  do
8        $T = \text{ConstructTopology}(S_c, S_b, s, p_c)$ 
9        $Cap = \text{ComputeCapacity}(T)$ 
10      if  $Cap < Cap^*$  then
11         $s = s - 1$ 
12      else if  $Cap > Cap^*$  then
13         $s = s + 1$ 
14      else
15        candidate.append( $T$ )
16      end
17    end
18  end
19 end

```

Algorithm 3: FatClique Topology Synthesis Algorithm

Rack width	24 inches
Rack depth	28.875 inches
Rack height	108 inches
Tray-to-rack distance	24 inches
Dist. Betw. cross-trays	48 inches
Aisle Width	48 inches
Rack units per rack	58 RU [29]
#Ports per patch panel	48 [10]
Patch panel space	1 RU
Cable tray size	24 inches x 4 inches [34]

Table 8: Datacenter settings mostly [26]

Length	3	5	10	15	20	30
Price	303	310	318	334	350	399
Length	50	100	200	300	400	
Price	489	753	1429	2095	2700	

Table 9: 40G QSFP Mellanox cable length in meter (Length) and price with transceivers (Price) [7]

A.8 Other Metrics

In our evaluations, we have tried to topologies with qualitatively similar properties 6. In this section, we quantify other properties of these topologies.

Edge Expansion and Spectral Gap. Since computing edge expansion is computationally hard, we follow the method in [35] using spectral gap [17] to approximate edge expansion. A larger spectral gap implies larger edge expansion. To fairly compare topologies, we equalize their bisection bandwidth first. As shown before, to achieve the same bisection

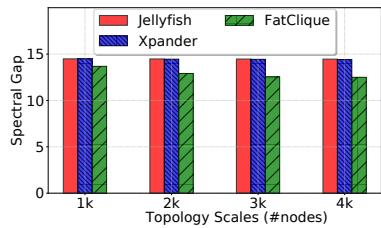


Figure 18: Spectral Gap

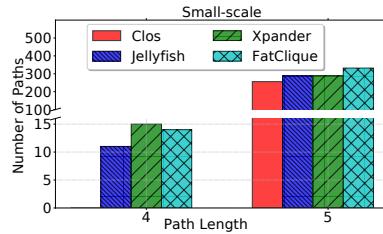


Figure 19: Path Diversity for Small-scale Topologies

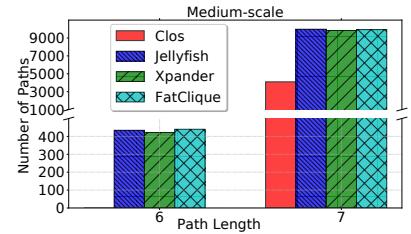


Figure 20: Path Diversity for Medium-scale Topologies

bandwidth, Clos uses many more switches. Also, Clos is not a d -regular graph and do not know of a way to compute the spectral graph for Clos-like topologies. Therefore, we compare the spectral gap only for d -regular graphs, Jellyfish, Xpander and FatClique at different scales (1k-4k nodes). The spectral gap is defined as follows [17]. Let G with node degree d and $A(G)$ denote the d -regular topology and its adjacent matrix. The matrix $A(G)$ has n real eigenvalues which we denote by $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$. Spectral gap $SG = d - \lambda_2$. In our experiments, chip radix is 32 and each node in those topologies connects to 8 servers, $d = 24$. The result is shown in Figure 18. First, we observe that spectral gap stays roughly the same under different scales. Also, the spectral gap of FatClique is slightly lower than that of other topologies, which implies that FatClique has slightly smaller edge expansion compared to Jellyfish and Xpander. This is to be expected, since FatClique adds some hierarchical structure to cliques.

Path Diversity. We compute the path diversity for different topologies. For Clos, we only calculate the number of shortest paths between two ToR switches from different pods. For other topologies, we compute the number of paths which are no longer than the shortest paths in the same-scale Clos. For example, for small-scale Clos, the shortest path length is 5. We will only calculate paths whose length is no larger than 5 in other topologies. This is a rough metric for path diversity. The results are shown in Figure 19 and Figure 20. We found that Jellyfish, Xpander and FatClique have the same level of path diversity, which is higher than that of Clos. Also, those topologies have shorter paths than Clos.

Shoal: A Network Architecture for Disaggregated Racks

Vishal Shrivastav
Cornell University

Asaf Valadarsky
*Hebrew University
of Jerusalem*

Hitesh Ballani
Microsoft Research

Paolo Costa
Microsoft Research

Ki Suh Lee
Waltz Networks

Han Wang
Barefoot Networks

Rachit Agarwal
Cornell University

Hakim Weatherspoon
Cornell University

Abstract

Disaggregated racks comprise dense pools of compute, memory and storage blades, all interconnected through an internal network. However, their density poses a unique challenge for the rack’s network: it needs to connect an order of magnitude more resource nodes than today’s racks without exceeding the rack’s fixed power budget and without compromising on performance. We present Shoal, a power-efficient yet performant intra-rack network fabric built using fast circuit switches. Such switches consume less power as they have no buffers and no packet inspection mechanism, yet can be reconfigured in nanoseconds. Rack nodes transmit according to a static schedule such that there is no in-network contention without requiring a centralized controller. Shoal’s congestion control leverages the physical fabric to achieve fairness and both bounded worst-case network throughput and queuing. We use an FPGA-based prototype, testbed experiments, and simulations to illustrate Shoal’s mechanisms are practical, and can simultaneously achieve high density and high performance: 71% lower power and comparable or higher performance than today’s network designs.

1 Introduction

Traditional datacenter use a server-centric architecture in which a number of racks, each comprising tens of servers connected via a top-of-the-rack (ToR) switch, are interconnected by the datacenter network. However, the end of Dennard’s scaling [18] and the slowdown of Moore’s Law [14] are challenging the long-term sustainability of this architecture [19]. Consequently, a new paradigm has emerged: *rack-scale* architecture, where a server is replaced by a rack as the unit of computation, with each rack hosting a number of System-on-Chip (SoC) [15, 35, 65] microservers, each comprising multi-core CPUs integrated with some local memory, combined with separate pools of non-volatile memory, storage and custom compute (e.g., Google TPUs [82], GPGPUs [74, 78] and FPGAs [43]) blades, all interconnected through an internal network. This enables *resource disaggregation* as compute units are decoupled from memory and storage units. The benefits disaggregation are well understood in the computer architecture

community [5, 41]: it enables fine-grained resource pooling and provisioning, lower power consumption and higher density than traditional server-centric architectures, thus enabling each rack to host hundreds of resource “nodes” (compute/memory/storage blades). Several examples of rack-scale architecture have been proposed both in industry (Intel [72], Facebook [65, 74], Microsoft [43], SeaMicro [79], HPE [28], Google [82]) and academia [5, 6, 15, 27, 35, 41].

Increasing rack density, however, poses new challenges for the rack’s network. Traditional ToR switches can support only around a hundred ports at high speed. Therefore, interconnecting several hundreds or even a thousand nodes requires either a high-port count chassis switch or a number of low-port count switches arranged in a hierarchical topology, e.g., a folded Clos [1]. Such a design, when coupled with state-of-the-art protocols [2, 4, 20, 25], can provide high throughput and low latency that could potentially meet the requirements of disaggregated workloads [19]. Unfortunately, such packet-switched designs are significantly less power and cost efficient as compared to today’s intra-rack networks (§2). Power is a particular concern as the rack’s total power has a hard limit due to cooling [35, 60], so network inefficiency ultimately limits the density of other resources.

The limitations of packet-switched networks have already prompted network designs that leverage circuit switches in datacenters [11, 23, 24, 38, 42, 53]. Such switches can be optical or electrical, and the fact that they operate at the physical layer with no buffers, no arbitration and no packet inspection mechanisms means they can be cheaper and more power efficient than an equivalent packet switch (§5). Adopting these designs for intra-rack connectivity would thus alleviate the power concern. However, achieving low latency would still be challenging as traditional circuit switches have reconfiguration delays of the order of few microseconds to even milliseconds. Such a solution, thus, would either compromise on performance or still have to rely on a separate packet-switched network to handle latency-sensitive traffic. In summary, adapting existing network solutions to high-density racks would either compromise on power (packet-switched) or on performance (purely circuit-switched).

In this paper, we show that it is possible to design a rack-

scale network that operates comfortably within the rack’s power budget while achieving performance comparable to packet-switched networks. Our work is motivated by fast circuit switches that can be reconfigured in a few to tens of nanoseconds while still being power-efficient. These are available commercially [76] as well as research prototypes [10, 16, 17, 30, 35, 36, 48, 52]. Unfortunately, it is not sufficient to simply take existing circuit-switch-based architectures and upgrade their switches as these architectures were designed under the assumption of slow reconfiguration times. In particular, these solutions rely either on a centralized controller to reconfigure the switches [11, 23, 24, 35, 42, 53], which would be infeasible at a nanosecond scale, or on a scheduler-less design with a large congestion control loop [38], which prevents taking advantage of fast reconfiguration speeds.

We present Shoal, a power-efficient yet performant network fabric for disaggregated racks built using fast circuit switches. Shoal reconfigures the fabric using a static schedule that connects each pair of rack node at an equal rate. This avoids the need for a centralized scheduler that can operate at a sub-microsecond granularity. To accommodate dynamic traffic patterns atop a static schedule, traffic from each node is uniformly distributed across all rack nodes which then forward it to the destination; a form of detour routing. Such *coordination-free scheduling*, first proposed by Chang et al. [9] as an extension of Valiant’s method [50], obviates the complexity and latency associated with centralized schedulers while guaranteeing the worst-case network throughput across *any* traffic pattern [9]. Such scheduling, however, requires that all nodes are connected through what looks like a single non-blocking switch. To achieve this, Shoal’s fabric uses many low port-count circuit switches connected in a Clos topology. When reconfigured synchronously, the switches operate like a single circuit switch. Further, we decompose the static, equal-rate schedule for the fabric into static schedules for the constituent switches.

Overall, this paper makes the following contributions:

- We present a network architecture for disaggregated racks that couples fast circuit switches with the servers’ network stack to achieve low and predictable latency at low cost and power.
- We designed a fabric that uses low port-count circuit switches to offer the abstraction of a rack-wide circuit switch. We also scaled the coordination-free scheduling technique to operate across the fabric.
- We devised an efficient congestion control mechanism to run atop Shoal’s fabric. This is particularly challenging to achieve due to high multi-pathing—traffic between a pair of nodes is routed through all rack nodes. Shoal leverages the observation that the static schedule creates a periodic connection between any pair of rack nodes to implement an efficient backpressure-based congestion control,

amenable to hardware implementation.

- We implemented Shoal’s NIC and circuit switch on an FPGA; our prototype achieves small reconfiguration delay (6.4 ns) for the circuit switches and is a faithful implementation of our entire design including the scheduling and the congestion control mechanisms.
- We incorporated the NIC and the switch prototype into an end-to-end small-scale rack testbed that comprises six FPGA-based circuit switches in a leaf-spine topology connecting eight FPGA-based NICs at end hosts.

Experiments on this small-scale testbed shows that Shoal offers high bandwidth and low latency; yet our analysis indicates that its power can be 71% lower than an equivalent packet-switched network. Using a cross-validated simulator, we show that Shoal’s properties hold at scale too. Across datacenter-like workloads, Shoal achieves comparable or higher performance than a packet-switched network using state-of-the-art protocols [2, 25, 54], with improved tail latency (up to 2× lower as compared to NDP [25]). Further, through simulations based on real traces [19], we also demonstrate that Shoal can cater to the demands of emerging disaggregated workloads.

2 Motivation

We first consider how conventional datacenter networks could be adapted for disaggregated racks and the shortcomings of such an approach.

Strawman 1. Chassis switches with hundreds of ports, often used at higher levels of a datacenter’s network hierarchy, could connect all rack nodes but at significant cost, power, and space. For example, the Cisco Nexus 7700 switch can support 768 ports at 10 Gbps (only 192 at 100 Gbps). Yet, it consumes 4 KW power and occupies 26 RU [61], which is 26% and 54% of the rack’s power and space budget respectively. A rack’s total power has a hard limit of around 15 KW due to constraints on power supply density, rack cooling and heat dissipation [35, 60, 66]. We also considered a custom solution involving commodity switches arranged in a Clos topology, which would still consume around 8.72 KW to connect 512 nodes (§ 5). The key reason for this is that packet switching necessitates buffers and arbitration inside each switch and serialization-deserialization at each switch port, which are major contributors (up to 70%) to the switch’s chip area and package complexity [34, 62], and in turn, its power.

Strawman 2. Motivated by the observation that enabling high-density racks requires a step change in the power-efficiency of the network, practitioners have attempted to integrate several very low-port (typically four or six ports) packet switches in the system-on-chip (SoC) of the microserver. Thus, instead of building a ToR-based network, the microservers can be connected to each other using direct-connect topologies prevalent in HPC and super-computing

systems, e.g., a 3D torus [41, 68, 79]. This design significantly reduces the overall network power consumption as the additional logic per SoC is small. However, a key drawback of direct-connect networks is that they have a static topology which cannot be reconfigured based on current traffic pattern. Hence their performance is workload dependent—for dynamically changing workloads such as datacenter workloads, it results in routing traffic across several rack nodes, which hurts network throughput and latency (§7.3) and complicates routing and congestion control [15].

Circuit switching. These strawmans lead to the question whether packet-switched networks are well-suited to support high-density racks. On the upside, packet-switched networks offer excellent performance and allow the network core to be loosely coupled with the servers’ network stack. In datacenters and WANs, this has been a good trade-off—the increased power of switches is not a key concern yet loose coupling has allowed the core network technologies to evolve independent of the servers’ network stack. This also allows the network to be asynchronous, which helps scaling. These benefits, however, do not hold up inside a rack. The physical size of a rack means that achieving rack-wide synchronization is feasible. Further, many density and cost benefits of disaggregated racks come from the co-design of servers and the network, so independent evolution is not critical.

Instead, we argue that a circuit-switched network offers a different set of trade-offs that are more suited to disaggregated racks. Compared to a packet switch, circuit switches can draw less power and be cheaper due to their simplicity, and these gains could grow with future optical switches (§5). Thus, they can better accommodate higher density. On the flip side, circuit switching does necessitate a tight coupling where all nodes are synchronized and traffic is explicitly scheduled. Further, past solutions with slow circuit switches have had to rely on a separate packet-switched network to support low latency workloads which increases complexity and hurts network manageability. Using fast circuit switches helps on the performance front yet makes the scheduling harder. We show that these challenges can be solved at the scale of a rack and it is feasible to build a rack network that satisfies its power constraints while achieving performance on par with a packet-switched network.

3 Design

Shoal is a network architecture for disaggregated racks. It comprises a network stack at the rack nodes which is tightly coupled with a circuit-switched physical fabric.

3.1 Design overview

Shoal’s architecture is shown in Fig. 1. Each rack node is equipped with a network interface connecting it to the Shoal fabric. The fabric comprises a hierarchical collection of smaller circuit switches, electrical or optical, that are reconfigured synchronously. Hence, the fabric operates

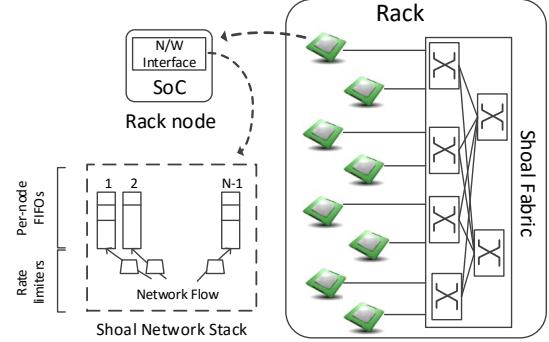


Figure 1: Shoal architecture.

like a single, giant circuit switch (§3.2). The use of a circuit switched fabric means that we need to schedule it. One possible approach is to schedule it *on-demand*, i.e., connect nodes depending on the rack’s traffic matrix. However, such on-demand scheduling requires complicated scheduling algorithms and demand estimation, and would make it hard to meet low-latency constraints.

Instead, Shoal uses *coordination-free* scheduling [9]. Specifically, each circuit switch forwards fixed-sized packets or “cells” between its ports based on a predefined “schedule”. These per-switch schedules, when taken together, yield a schedule for the fabric which dictates when different node pairs are connected to each other. The schedule for individual switches is chosen such that the fabric’s schedule provides equal rate connectivity between each pair of nodes. To accommodate any traffic pattern atop the equal rate connectivity offered by the fabric, each node spreads its traffic uniformly across all other rack nodes, which then forward it to the destination (§3.3.1).

The second mechanism implemented in Shoal’s network stack is a congestion control technique that ensures that network flows converge to their max-min fair rates, while bounding the maximum queuing at all rack nodes. Our main insight here is that the periodic connection of rack nodes by the fabric enables backpressure-based congestion control amenable to hardware implementation. One of the main challenges in implementing backpressure-based mechanisms over multi-hop networks is instability for dynamic traffic [26]. In Shoal, we restrict the backpressure mechanism to a *single hop*, avoiding the instability issue altogether.

3.2 Shoal fabric

Shoal uses a predefined, static schedule to reconfigure the fabric such that the rack nodes are connected at an equal rate. Fig. 3 shows an example schedule with $N = 8$ nodes. Thus, in a rack with N nodes, each pair of nodes is directly connected by the fabric once every $N - 1$ time slots, where a slot refers to the cell transmission time.

However, constructing a monolithic switch, electrical or optical, with hundreds of high-bandwidth ports and fast re-

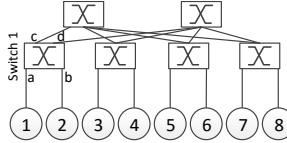


Figure 2: Circuit switches in a two-stage Clos topology.

Node	Time slot						
	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	1
.
8	1	2	3	4	5	6	7

Figure 3: Fabric schedule for a rack with 8 nodes.

Port	Time slot						
	1	2	3	4	5	6	7
a	d	c	d	c	d	c	d
b	c	d	c	d	c	d	c
c	b	a	b	a	b	a	b
d	a	b	a	b	a	b	a

Figure 4: Switch 1’s schedule (see Fig. 2 for topology).

configuration is intractable due to fabrication constraints. Instead, Shoal’s fabric comprises low port-count circuit switches connected in a non-blocking Clos topology. Arranging k -port circuit switches in a two-stage Clos topology allows the fabric to connect $\frac{k^2}{2}$ nodes. For e.g., using 64-port electrical circuit switches allows us to connect a rack with 2,048 nodes. Fig. 2 shows six 4-port circuit switches arranged in such a topology to implement an 8-port fabric. Packets between any two nodes are always routed through both stages of the topology, even if the nodes are connected to the same switch (like nodes 1 and 2 in the figure). Since the topology is non-blocking, this does not impact network throughput. It ensures, however, that the distance between any two nodes is the same which, in turn, aids rack-wide time synchronization (§3.4).

We decompose the schedule of the overall fabric into the schedule for each constituent circuit switch. Consider the example fabric shown in Fig. 2. Fig. 3 shows the schedule for this fabric while Fig. 4 shows the schedule for switch 1. Each switch’s schedule is contention-free, i.e., at a given instant, any port is connected to only one port. This allows the switch to do away with any buffers and any mechanisms for packet inspection or packet arbitration.

3.3 Shoal network stack

Shoal’s mechanisms operate at the data link layer (layer-2) of the network stack. At each node, Shoal spreads its layer-2 traffic uniformly across the rack to ensure guaranteed network throughput and implements a congestion control technique that ensures fair bandwidth sharing and low latency.

3.3.1 Forwarding plane

Rack nodes send and receive fixed-sized cells. Packets received from higher layers are thus fragmented into cells at the source node and reassembled at the destination. Each cell has a header (Fig. 5) that contains the corresponding packet’s destination and other control information.

Cells sourced by a node, irrespective of their destination, are sent to the next node the source is connected to. This uniformly spreads traffic across all rack nodes. Each node has a set of FIFO queues, one for every node in the rack. Cells arriving at an intermediate node are put into the queue corresponding to their final destination. This ensures traffic is detoured through at most one intermediate node. These queues are served according to the node’s transmission schedule.

We highlight two key aspects of this simple design. First,

uniformly distributing traffic is perfectly suited to the equal rate connectivity provided by the Shoal fabric. This guarantees the worst-case throughput across *any* traffic pattern [9]—Shoal’s network throughput can be at most $2\times$ worse than that achieved by a hypothetical, rack-wide ideal packet switch. To compensate for this throughput reduction due to detouring, we double the aggregate bisection bandwidth of the fabric for Shoal. This is a good trade-off as circuit switches are expected to be cheaper and hence, adding fabric bandwidth is inexpensive; in §5, the cost of the resulting network is still estimated to be lower than the cost of a traditional packet-switched network.

Second, when the fabric’s schedule connects node i to node j , the former *always* transmits a cell; the cell at the head of the queue $i \rightarrow j$ is transmitted, otherwise an empty cell is sent. This ensures that each node periodically receives a cell from every other node, which enables implementing an efficient backpressure-based congestion control (§3.3.2) and simple failure detection (§3.5).

3.3.2 Congestion control

Each node sending traffic computes the appropriate rate for its traffic to avoid congesting the network. We begin with a discussion of the network topology resulting from periodic reconfiguration of the Shoal fabric and its implications for congestion control, followed by the details of our design.

High Multi-pathing. The periodic reconfiguration of Shoal’s fabric means that the entire network can be seen as an all-to-all mesh with virtual links between each pair of nodes. For e.g., consider a rack with 8 nodes whose schedule is shown in Fig. 3. Since each node is connected to every node $1/7^{th}$ of the time, the network provides the illusion of a complete mesh with virtual links whose capacity is $1/7^{th}$ of each node’s total network bandwidth.

Shoal’s use of detouring means that each node’s traffic is routed through all the rack nodes on their way to their destination, resulting in very high multi-pathing. In contrast, the TCP suite of protocols, including protocols tailored for datacenters [2, 51] and recent protocols for RDMA networks [39, 54] only use a single path. Even multi-path extensions like MPTCP [44] target scenarios with tens of paths, which is an order of magnitude less than the number of paths used by traffic in our fabric.

Design insights. Shoal’s congestion control design is based on three key insights. First, we leverage the fact that the fab-

ric in an N -node rack directly connects each pair of nodes once every $N - 1$ time slots. We refer to this interval as an *epoch*. This means that, when the queues at an intermediate node grow, it can send a timely backpressure signal to the sender. As we detail below, the periodic nature of this signal coupled with careful design of how a sender reacts to it allows us to bound the queue size across rack nodes.

Second, achieving per-flow fairness with backpressure mechanisms is challenging [54], especially in multi-path scenarios. In Shoal, a *flow* refers to all layer-2 packets being exchanged between a pair of nodes. For network traffic, this includes all transport connections between the nodes. For storage traffic, this includes all IO between them. Each flow comprises $N - 1$ *subflows*, one corresponding to each intermediate node. Shoal achieves max-min fairness across flows by leveraging the fact that each flow comprises an equal number of subflows that are routed uniformly across a symmetric network topology, so we can achieve per-flow fairness by ensuring per-subflow fairness. We thus treat each subflow independently and aim to determine their fair sending rates. The mechanism can also be extended to accomodate other flow-level sharing policies.

Finally, each subflow traverses two virtual links, either of which can be the bottleneck. For e.g., a subflow $i \rightarrow j \rightarrow k$ can either be bottlenecked at the virtual link between nodes i and j , or between nodes j and k . Shoal maintains a queue, Q_{ij} , at node i to store cells destined to node j . We use the length of the queue Q_{ij} as an indication of the load on the virtual link between nodes i and j . Note that the node sourcing the traffic, node i , can observe the size of the local queue Q_{ij} . It, however, also needs to obtain information about the size of the remote queue Q_{jk} that resides at node j .

Congestion control mechanism. We use a subflow from source i to destination k through intermediate node j , $i \rightarrow j \rightarrow k$, as a running example to explain Shoal’s congestion control. When node i sends a cell to node j , it records the subflow that the cell belongs to. Similarly, when node j receives the cell, it records the index k of the queue that the cell is added to. The next time node j is connected to node i , it embeds the current length of queue Q_{jk} into the cell header:

$$\text{rate limit feedback}_{ji} = \text{len}(Q_{jk}) \quad (1)$$

Each pair of nodes in the rack exchange a cell every epoch, even if there is no actual traffic to be sent. Thus, when node i sends a cell to node j , it gets feedback regarding the relevant queue at j within the next epoch. Let us assume that node i receives this feedback at time T . At time $t (\geq T)$, it knows the instantaneous length of its local queue to node j , $Q_{ij}(t)$, and a sample of the length of the remote queue between nodes j and k , $Q_{jk}(T)$. The max-min fair sending rate for a subflow is governed by the most bottlenecked link on its path, i.e., the link with the maximum queuing. As a result, the next cell for this subflow should only be sent after both the queues have had time to drain, i.e.,

at least, $\max(\text{len}(Q_{ij}(T)), \text{len}(Q_{jk}(T)))$ epochs have passed since the feedback was received. To achieve this, node i releases a cell for this subflow into its local queue for j only when the current length of the queue, after accounting for the time since the last feedback, exceeds the size of the remote queue Q_{jk} , i.e., a cell is released into Q_{ij} at time t when,

$$\text{len}(Q_{ij}(t)) + (t - T) \geq \text{len}(Q_{jk}(T)) \quad (2)$$

Thus, when a new cell is released into the queue at its source, the previous cell in that queue is guaranteed to have been sent to the remote queue while the previous cell in the remote queue is guaranteed to have been sent to the destination. This ensures the *invariant* that at any given time a subflow has at most one cell each in both the queue at its source and the queue at its intermediate node. As a consequence, at any given time, the size of each queue Q_{ij} is bounded by:

$$\text{len}(Q_{ij}) \leq \text{outcast degree}(i) + \text{incast degree}(j) \quad (3)$$

Thus, this mechanism ensures that, for each virtual link, Shoal performs *fair queuing* at cell granularity across all the subflows sharing that link. This, in turn, results in a tighter distribution of flow completion times.

Note that while Shoal’s basic design assumes a single traffic class for the flows, it can be easily extended to support multiple traffic classes as explained in Appendix C.

3.3.3 Improving network latency

While Eq. 3 bounds the queue size, it also highlights one of the challenges of detouring: network latency experienced by a cell, while bounded, is impacted by cross-traffic — traffic from remote nodes at the cell’s source node and traffic from local node at the cell’s intermediate node. To reduce this impact of detouring, we introduce following optimizations:

Reducing cell latency at the intermediate node. In addition to queue Q_{jk} , node j also maintains a ready queue R_{jk} . Instead of adding cells to Q_{jk} from local flows that satisfy Eq. 2, Shoal adds the corresponding flow ids into the ready queue R_{jk} . Thus,

$$\text{len}(R_{jk}) \leq \text{outcast degree}(j) \leq N - 1 \quad (4)$$

Shoal then scans the local flow ids in R_{jk} , and adds the corresponding cells into the queue Q_{jk} such that at any given time there is at most one local cell in Q_{jk} . Thus Eq. 3 changes to:

$$\text{len}(Q_{jk}) \leq 1 + \text{incast degree}(k) \leq N \quad (5)$$

However, to ensure that the rate limit feedback accounts for the local subflows, Eq. 1 needs to be updated accordingly:

$$\text{rate limit feedback}_{ji} = \text{len}(Q_{jk}) + \text{len}(R_{jk}) - 1 \quad (6)$$

The rack network is thus still shared in a max-min fashion, while simultaneously reducing the impact of local traffic on the latency of remote cells — the latency experienced by a remote cell at any intermediate node is determined only by the incast degree of cell’s destination.

Reducing cell latency at the source node. While Eq. 5 reduces the impact of detouring at the intermediate node, at

the source node i , the latency for a local cell in Q_{ij} is governed by incast degree of intermediate node j . To reduce the impact of cross traffic (i.e., non-local traffic), Shoal selectively adds cells from a new flow to queue Q_{ij} only if $\text{len}(Q_{ij}) \leq 2^{\text{age}}$, where age is measured in epochs since the flow started. Thus, for the first few epochs, cells will be released to queues over virtual links with low contention, and afterwards will quickly converge to uniform load-balancing using all virtual links after a max of $\log(N)$ epochs. This achieves uniform load-balancing for long flows, and hence preserves Shoal’s throughput bounds, while reducing completion time for short flows.

The impact of these optimizations is evaluated in Fig. 14.

3.3.4 Bounded queuing

Eq. 5 guarantees that at any given time, the size of each queue Q_{ij} at node i is bounded by the instantaneous number of flows destined to destination j plus one, with at most one cell per flow. This queue bound can be used to determine the maximum buffering needed at each node’s network interface to accomodate even the worst-case traffic pattern of all-to-one incast. In a rack with 512 nodes and 64 B cells, this requires a total buffering per node of 17 MB. Importantly, since Shoal accesses a queue only once every epoch for transmission, and assuming the access latency of off-chip memory is less than an epoch, Shoal only needs to buffer one cell from each queue Q_{ij} on the on-chip memory, resulting in $N - 1$ total cells. Using the example above, this leads to on-chip cell buffer size of just 32 KB per node.

3.4 Shoal slots and guard band

Shoal operates in a time-slotted fashion. Slots are separated by a “guard band” during which the switches are reconfigured. The guard band also accounts for any errors in rack synchronization.

Circuit switch reconfiguration. Shoal uses fast reconfigurable circuit switches. For example, our prototype implements an FPGA-based circuit switch that can be reconfigured in 6.4 ns (§4.1). Electrical circuit switches with fast reconfiguration are also commercially available [76] while fast optical circuit switches with nanosecond-reconfiguration time have also been demonstrated [12, 16, 17, 30, 36, 48, 52].

Time synchronization. Shoal’s slotted operation requires that all rack nodes and switches are time synchronized, i.e., they agree on when a slot begins and ends. Synchronizing large networks is hard, primarily because of high propagation delay and the variability in it. In contrast, fine-grained rack-wide synchronization is tractable due to their size—a typical rack is only a few meters high which means that, even when using optical transmission with a propagation delay of 5 ns/m, the maximum propagation latency across a rack is about 10-15 ns. Furthermore, the rack can be constructed with tight tolerances to aid synchronization. For example, if all links are the same length with a tolerance of ± 2 cm,

the propagation delay would vary by a maximum of 0.2 ns. Small physical distance also mitigates the impact of temperature variations that could lead to variable propagation delay.

Shoal leverages the WhiteRabbit synchronization technique [32, 37, 40, 45] to achieve synchronization with bit-level precision. WhiteRabbit has been shown to achieve sub-50 picoseconds of synchronization precision [45]. The main idea is to couple frequency synchronization with a time synchronization protocol (§6.1).

Frequency synchronization is achieved by distributing a global clock to all the nodes and switches in the rack. This global clock is generally derived from one of the rack nodes, designated as the clock master. The clock can be distributed explicitly, or implicitly through Synchronized Ethernet (SyncE) [75] whereby nodes derive a clock from the data they receive and use this clock for their transmissions.

In Shoal, time synchronization protocol like PTP [70] or DTP [33] need to run only between the end nodes (and not the switches). At bootstrap, each switch’s circuits are configured according to their respective schedule’s configuration at time slot 1 (e.g. Fig. 4) and they do not change. End nodes then start running the time synchronization protocol. Once all the nodes are synchronized to a desired level of precision, they send a bootstrap signal to the switches, followed by actual data according to the fabric schedule (Fig. 3). Switches on receiving the bootstrap signal start reconfiguring their circuits according to their respective schedules (Fig. 4).

Slot size configuration. Overall, the guard band size is the sum of the reconfiguration delay, variability in propagation and the precision of synchronization. Given the guard band size, the slot size can be configured to balance the trade-off between latency and throughput: a smaller slot reduces epoch size resulting in smaller latency, yet it imposes higher guard band overhead resulting in smaller duty cycle and hence lower throughput.

Epoch size and multiple channels. In Shoal, two nodes exchange cells at the interval of an epoch. Therefore, each queue drains at the rate of one cell per epoch, meaning a smaller epoch size results in smaller queuing delay. We can reduce the epoch size by taking advantage of the fact that network links comprise multiple channels. For e.g., 100 Gbps links actually comprise four 25 Gbps channels, which can be switched independently. Thus, in Shoal, each channel is used to send cells to a quarter of the rack nodes in parallel. Given a fixed slot size (as determined based on guard band size), this shrinks the epoch size by a quarter ($\text{epoch} = \frac{N-1 \text{ slots}}{\text{num of channels}}$). Finally, the actual cell size is determined by the slot size and channel speed, for e.g. a slot size of 20.5 ns (without guard band) will correspond to 64 B cells over a 25 Gbps channel.

3.5 Practical concerns

We now discuss a few practical concerns of the design.

Clock and data recovery (CDR). A key challenge for any network relying on fast circuit switches is that each node

source id	10 bits	destination id	10 bits
sequence number	22 bits	rate limit feedback	11 bits
start-of-packet	1 bit	end-of-packet	1 bit
last-cell-dropped	1 bit	CRC checksum	8 bits

Figure 5: Header fields in the 8 B cell header carried by each cell. Header field sizes assume a max of 1024 rack nodes.

needs to be able to receive traffic from different senders at each time slot. This requires that, at each time slot, the incoming bits are sampled appropriately so as to achieve error-free reception. The sampling is done by the Clock and Data Recovery (CDR) circuitry at the receiver and typically takes a few hundred microseconds [46]. However, we note that this is only a problem when using layer 0 circuit switches that operate at the raw physical layer, e.g., when using an optical circuit switch. Such a switch imposes no latency overhead but requires very fast CDR at the receiver in order to achieve a reasonable guard band. Recent work has shown that subnanosecond CDR is achievable in datacenter settings [13].

Electrical circuit switch can also operate at layer 1 [76]. When a circuit is established between ports $i \rightarrow j$, the switch retimes data received on port i before sending it to port j . With such switches, each link in the network is a point-to-point link and thus, fast CDR is not needed. Each switch, however, does need to be equipped with a small buffer to account for any differences in the clocks associated with ports i and j . For Shoal, only a few bits worth of buffering is required since the entire rack is frequency synchronized and the buffer is only needed to absorb any clock jitter.

Cell reordering and reassembly. The sequence number field in each cell’s header is used to assemble cells in-order at the destination. Note that Shoal’s congestion control is robust to reordering, as it operates at the granularity of individual subflows with a congestion window of size 1. Once the cells have been reordered, the `start-of-packet` and `end-of-packet` fields in the cell header are used to figure out the packet boundary, and the cells within each packet boundary are then assembled together to re-construct the original packet.

Failures. To detect failures, Shoal relies on the fact that a node sends a cell to every other node in the rack, even if there is no traffic to send, once every epoch. A path refers to the set of links and switches through which a node j sends a cell to some other node i once every epoch.

When a node i does not receive a cell from a node j in its corresponding slot, either due to path failures or node failure, it conservatively infers that node j has failed, and i) stops sending any further cells to j , ii) notifies other nodes that it can no longer communicate with j , so other nodes stop forwarding cells destined to j via i , iii) forwards the last cell (if it happens to be i ’s local cell) it sent to j via some other node, and iv) discards all the outstanding cells it was supposed to

1 node	1 node-leaf link	1 leaf switch	1 leaf-spine link	1 spine switch
$\approx 1/N$	$\approx 1/N$	$\approx 1/\sqrt{2N}$	$\approx 2/N$	$\approx 2\sqrt{2}/\sqrt{N}$

Table 1: Fraction of failed slots against different failed components, for a two-stage clos topology. N = no of rack nodes.

forward to j . Shoal relies on a higher layer end-to-end transport protocol to recover from the loss of those outstanding cells. Finally, in case of an actual node failure, Shoal again relies on the transport protocol to recover all the cells that were queued to be forwarded at the failed node. If the failed node was the primary clock reference for synchronization, another node needs to take over and remaining nodes seamlessly switch to it as the new reference. ITU standard for SyncE [75] already supports this.

Note that the failure detection mechanism is symmetric — when node i infers that node j has failed, it immediately stops sending cells to j , causing j to infer that i has failed, and hence immediately stop sending cells to i . This ensures the consistency of Shoal’s closed-loop congestion control mechanism (§3.3.2), even in the face of failures.

One of the consequences of Shoal’s design is if a node can no longer “directly” communicate with some other node, either because the other node or the path to it has failed, it hurts node’s throughput as the corresponding slot is marked as failed and hence goes unused (Table 1). We evaluate network performance against fraction of failed nodes in §7.3.

Scalability. Shoal’s scalability is mainly limited by two factors: i) On-chip resource consumption on the NIC, in particular on-chip memory, and ii) epoch size, which contributes to network latency. The on-chip memory consumption for Shoal scales as $\Theta(N^2 \log(N))$ bits (§4). Even for a very dense rack comprising ~ 1000 nodes, this results in a memory consumption only of the order of a megabyte. On the other hand, epoch size increases linearly with the number of nodes (§3.4). The impact of increasing epoch size on network latency is evaluated in §7.3.

4 Implementation

In this section, we discuss our FPGA-based implementation of Shoal’s switch and NIC. We used Bluespec System Verilog [57] ($\sim 1,000$ LOCs). Our design runs at a clock speed of 156.25 MHz, thus each clock cycle is 6.4 ns.

4.1 Switch design

Our circuit switch operates at layer 1, i.e., data traversing the switch is routed through the PHY block at the ingress and egress ports (Fig. 6). The mapping between the ingress and egress ports varies at every time slot according to the static schedule. This mapping is implemented using p different $p:1$ multiplexers, where p is the number of ports in the switch. The control signals to these multiplexers are driven by p registers, one per multiplexer. In each time slot, all the p registers are configured in parallel according to the schedule.

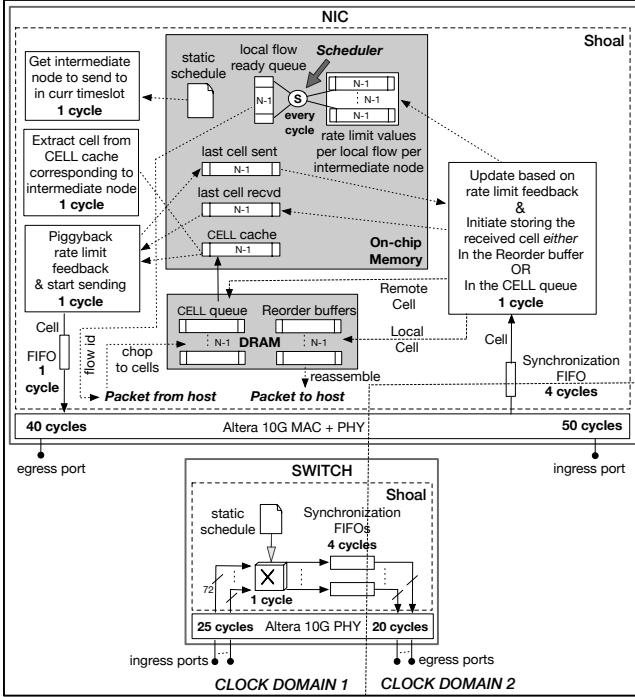


Figure 6: Switch and NIC implementation with the latency of each block. Clock cycle is 6.4 ns. N = num of rack nodes.

Hence, the switch reconfiguration delay is simply the time it takes to update the registers, which can be done in one clock cycle. Our switch is driven by the clock that drives the interface to PHY. The interface to 10G Ethernet PHY (XGMII) runs at 156.25 MHz, resulting in reconfiguration delay of 6.4 ns for our switch. However, for higher link speeds the clock frequency can be higher, for e.g., at 50 Gbps, the interface to PHY (LGMII) runs at 390.625 MHz [55], yielding a reconfiguration delay of 2.5 ns.

The transmit and receive paths of the switch are located in two separate clock domains: the transmit path is driven by the clock distributed throughout the rack, while the receive path is driven by the clock recovered from the incoming bits. To move data safely across clock domains, we use synchronization FIFOs. The total port-to-port latency of our switch is 50 cycles (320 ns): PHY block (45 cycles) + switching (1 cycle) + synchronization FIFO (4 cycles).

4.2 NIC design

Fig. 6. shows the routing and congestion control pipelines implemented in Shoal’s NIC. Each NIC maintains a cell cache on the on-chip memory, of size $N - 1$ cells, which stores the next cell to forward per intermediate node. Remaining cells sit in the DRAM. The backpressure-based mechanism underpinning Shoal’s congestion control (§3.3.2) is implemented using two vectors of size $N - 1$ each, that record the last cell sent (received) to (from) each intermediate node, and a $(N - 1) \times (N - 1)$ matrix that stores the

rate limit feedback received from each intermediate node for each active local flow. The scheduler uses these data structures to schedule local cells into a ready queue in accordance with the logic described in § 3.3.2 and § 3.3.3.

NIC latency is dominated by the PHY and MAC IP blocks, with the routing and congestion control logic only adding 4 and 5 cycles on the transmit and receive paths, resp. Thus, Shoal’s additional mechanisms impose low overhead.

Appendix B details the resource consumption for Shoal’s FPGA-based implementation.

5 Power and cost implications

We now compare the power and cost of a Shoal network to that of a packet-switched network (PSN). Along with the performance evaluation in §7, we demonstrate that for 71% lower power and an estimated cost reduction of up to 40%, Shoal’s circuit-switched fabric can reduce tail latency by up to 2× as compared to state-of-the-art congestion control protocols such as NDP [25] atop a PSN.

We analyze a 512-node rack. For a PSN, we consider today’s packet switches [58, 81], which support 64 ports at 50 Gbps and consume a maximum of 350 W [47, 49]. Nodes have 50 Gbps NICs with copper cables (i.e., no optoelectronic transceivers) and connecting them using a non-blocking Folded Clos topology requires 24 such switches. For Shoal, extrapolating from today’s circuit switches [76], we estimate that a 64×50 Gbps circuit switch would consume 38.5 W. To compensate for the throughput overhead of detouring packets, each node is equipped with 100 Gbps links. So the Shoal network has 48 circuit switches. The small physical size of the actual circuit switch ASIC means that the space required for the extra switches is manageable. Based on current SoC trends [59, 73, 79], we expect the NIC to be integrated with the CPU on a single SoC and to benefit from the same 10× reduction in power consumption. Given a typical power consumption of 12.4 W for today’s 100 Gbps NICs [77], this would lead to an estimated power consumption of 1.37 W for the Shoal’s NIC (including 11% overhead as computed in Appendix B) and of 0.62 W for PSN’s. Thus, the total power of the Shoal network is 2.55 KW, 71% lower than PSN (resp. 8.72 KW). Lower power density is critical because a rack’s total power has a hard limit around 15 KW [35, 60, 66].

Quantifying the cost of the Shoal network is harder as it requires determining the at-volume cost of circuit switches. Circuit switches can be electrical or optical; today, electrical circuit switches are commercially used in scenarios like HDTV [76] and are capable of fast switching while fast optical switches only exist as research prototypes [16, 17, 30, 36, 48, 52]. Thus, instead of focussing on absolute costs, we ask: *how cheap would circuit switches need to be, relative to equivalent packet switches, for Shoal to offer cost benefits over PSN?* We assume Shoal NICs cost between 2 and 3× PSN NICs to account for the 2× bandwidth and extra func-

tionality they provide. Fig. 11 shows how the relative cost of the Shoal network varies as a function of the relative cost of circuit switches to packet switches. A Shoal network would cost the same as a PSN as long as circuit switches are 33.3–41.6% the cost of packet switches while providing power and performance gains. If the cost of circuit switches was 9.6–18.3% of the cost of packet switches, then Shoal would offer a 40% cost reduction. While absolute costs are hard to compare as they also depend on several non-technical factors, the analysis below and our estimations based on hardware costs indicate that at-volume circuit switches could cost as low as 15% of equivalent packet switches.

The lack of buffering, arbitration and packet inspection in circuit switches means that they are fundamentally simpler than packet switches which should mean lower cost. For electrical switches, designers often use the switch package area as a first-order approximation of switch cost—the actual chip area dictates yield during fabrication and therefore fabrication cost while the total package area dictates the assembly and packaging cost. In today’s packet switches 50% of the total area is attributed to memory, 20% to packet processing logic while 30% is due to serial I/O (SerDes) [62]. Electrical circuit switches, by contrast, have no memory and packet processing, so the first two components have negligible contribution. While the amount of I/O bandwidth in a circuit switch remains the same, the actual SerDes is much smaller because they are only retiming the signals, instead of serializing and deserializing data from a high-rate serial channel to lower-rate parallel channels. Even assuming the SerDes are only halved in size, the total packaged area for circuit switching could be as low as 15% that of a packet switch. Overall, this analysis indicates that, with volume manufacturing, the relative cost of electrical circuit switches can be low enough for Shoal to simultaneously reduce both power and cost as compared to PSN.

Looking ahead, optical circuit switches hold even more promise: as bandwidth increases beyond 100Gbps per channel, copper transmission becomes noise limited even at intra-rack distances and optical transmission becomes necessary [64]. Optical circuit switches further reduce cost and power because they obviate the need for expensive transceivers for opto-electronic conversions. However, a few technical challenges need to be solved for optical switches to be used in Shoal [7]. For example, while several technologies being studied in the optics community can achieve nanosecond switching, practical demonstrations have been limited to 64–128 ports [12]. Another longstanding challenge is to achieve fast CDR at layer 0 although recent work has shown the feasibility of such CDR within 625ps [13].

6 Prototype

In this section, we evaluate our FPGA-based implementation of Shoal through a 8-node prototype, shown in Fig. 7.

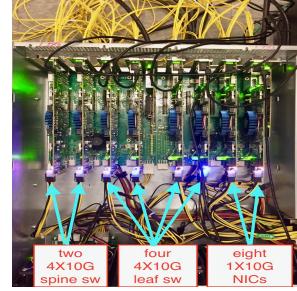


Figure 7: Shoal’s FPGA-based prototype.

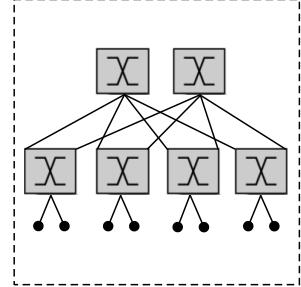


Figure 8: Shoal prototype’s physical topology.

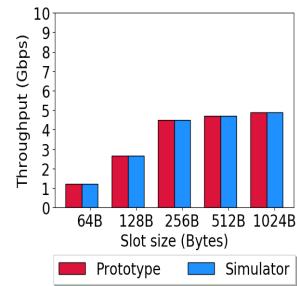


Figure 9: [Prototype] Avg destination throughput for full permutation matrix.

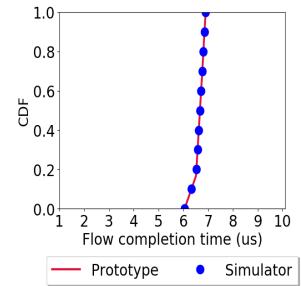


Figure 10: [Prototype] Flow completion time for 7:1 synchronized incast.

6.1 Prototype setup

Our prototype comprises eight Terasic DE5-Net boards [63], each with an Altera Stratix V FPGA [80] and four 10 Gbps SFP+ transceiver modules. Two FPGAs are used to implement eight NICs, one per port. The remaining six FPGAs implement six 4-port circuit switches. The switches are connected in a leaf-spine topology and the NICs are connected to the leaf switches as shown in Fig. 8. We connect all eight FPGAs to a Dell T720 server. We use the PCIe clock as the global clock and distribute it to the Phase-locked loop (PLL) circuit running on each FPGA. Thus all the local clocks derived from the respective PLL circuits on each FPGA are frequency synchronized. For time synchronization we use DTP [33].

Guard band. Our prototype achieves synchronization precision of less than a clock cycle. Further, the switch reconfiguration delay is one clock cycle (§4.1), and all wires are of same length. Hence a guard band of one clock cycle (6.4 ns) is sufficient.

Slot size. To keep the guard band overhead to around 10%, we select a slot size of 12 clock cycles (76.8 ns). This includes 1 cycle of guard band overhead and 24 B (3 cycles) of Altera MAC overhead. Thus the usable slot size equals 8 cycles (51.2 ns), which translates to 64 B cells at 10 Gbps link speed. The epoch size equals 0.53 us.

6.2 Prototype experiments

We used the prototype to verify that our implementation achieves throughput and latency in accordance with the design. We also use it to cross-validate our simulator which, in turn, is used for a large-scale evaluation regarding the viability and benefits of a real world deployment of Shoal.

Throughput. We consider a permutation matrix with $N = 8$ flows: each node starts a single long-running flow to another random node such that each node has exactly one incoming and outgoing flow. For throughput, this is the worst-case traffic matrix. In Fig. 9, we show performance in terms of *destination throughput*, measured as the amount of “useful” cells (i.e., excluding the cells to forward and the empty ones) received by each destination. For full permutation matrix, the throughput for Shoal is expected to converge to $\sim 50\%$ of the ideal throughput. Interestingly, however, the throughput is significantly lower for smaller slot sizes, and it converges towards 50% only for larger slot sizes. This is an artifact of the small scale of our prototype, which causes the node-to-node cell propagation latency (1.57 us: 40 ns of wire propagation latency + 3×320 ns of switching latency (dominated mostly by Altera 10G PHY latency) for three switches along the path + 576 ns of Altera 10G MAC and PHY latency at the two end nodes (Fig. 6)) to be higher than the epoch size (0.53 us for 64 B cells). The problem is that Shoal’s congestion control mechanism prevents a node from sending its next cell to an intermediate node until it has received feedback from it. Therefore, if the cell propagation latency spans multiple epochs, the overall throughput suffers as senders cannot fully utilize their outgoing bandwidth. As the slot size increases, the ratio between the cell propagation latency and the epoch size decreases, and this explains why in our prototype the throughput improves with larger slots. In practice, however, even for modest-sized racks, this issue will not occur as the cell propagation latency will be much smaller than the epoch size, and can be easily accommodated in the schedule as explained in Appendix A.

Latency. We consider all-to-one incast, where seven nodes each send 448 B of data (seven 64 B cells) to the same destination at the same time. Fig. 10 shows the distribution of flow completion time (FCT) of all seven flows. The queue at each node corresponding to the destination node grows upto a maximum of 7 (Eq. 5). This results in maximum FCT of 6.9 us : 3.76 us of queuing delay plus 2×1.57 us of propagation latency. Also note that the difference between the fastest and slowest flow is fairly small (6.05 us vs. 6.9 us), highlighting Shoal’s fair queuing.

Overall, across all experiments, the prototype and simulation results were in agreement.

7 Simulation

We complement the prototype experiments in §6 with simulations to investigate the scalability of Shoal.

7.1 Simulation setup

We use the packet-level simulator that was cross-validated against our prototype (§6). We simulate a 512-node rack, where each node is equipped with an interface bandwidth of 100 Gbps, connected using a full bisection bandwidth Clos topology comprising circuit switches.

Guard band. We assume a guard band of 2.75 ns, based on a 2.5 ns switch reconfiguration delay (§4.1) and 0.25 ns to account for any variability in propagation and synchronization imprecision (§3.4).

Slot size. To keep the guard band overhead to around 10% (resulting in max throughput of 90 Gbps), we select the slot size of 23.25 ns. This results in 20.5 ns of usable slot size. As explained in §3.4, we use the fact that existing 100 Gbps links comprise 4×25 Gbps channels, resulting in 4 parallel uplinks and an epoch size of 2.9 us. Finally, usable slot size of 20.5 ns translates to 64 B cells at 25 Gbps channel speed.

7.2 Microbenchmarks

We start with a set of microbenchmarks to verify that the behavior observed in our testbed holds at large scale too.

Throughput. In Fig. 12, we plot the average destination throughput, as defined in §6.2, for the permutation traffic matrix: each communicating node sends and receives one flow. We vary the number of communicating pairs from 1 to 512. As there is no contention at any of the source and destination nodes, the ideal destination throughput equals the maximum interface bandwidth. However, for Shoal, as the number of communicating pairs increases, so does the amount of detouring traffic, resulting in the expected throughput trend: it starts from the peak value for a single flow and then monotonically decreases until it halves when all pairs are communicating (full permutation traffic matrix).

Fairness. To verify Shoal’s fairness, we ran several workloads comprising a variable number of flows from 50 to 1,024 with randomly selected sources and destinations. We compared the throughput achieved by each flow against its ideal throughput computed using the max-min water-filling algorithm [8]. Across all workloads, 99% of the flows achieve a throughput within 10% of the ideal one. This shows that, despite the simplicity of its mechanisms, Shoal closely approximates max-min fairness.

Latency under Incast. We evaluate Shoal under incast, the most challenging traffic pattern for low latency. A set of nodes send a small flow of size 130 KB each, to the same destination at the same time. In Fig. 13, we plot the flow completion time (FCT) of the slowest flow as well as the mean completion time, against increasing number of sending nodes. As expected, at each intermediate node, the queue corresponding to the destination node grows linearly with increasing number of sending nodes, but bounded by the incast degree of the destination (Eq. 5). Hence the FCT for the slowest flow increases linearly and is also the optimal

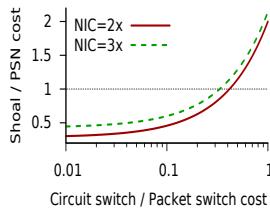


Figure 11: Relative cost of Shoal network vs. packet switch network (PSN).

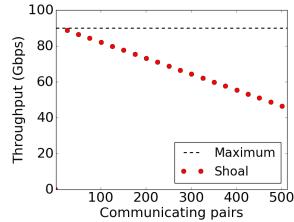


Figure 12: Average destination throughput vs. size of permutation matrix.

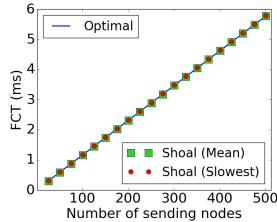


Figure 13: Flow completion times against synchronized short flow incast.

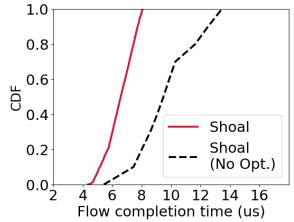


Figure 14: Reduced impact of detouring on latency via optimizations in §3.3.3.

maximum FCT under such incast. The mean completion time coincides with the slowest flow’s FCT, thus highlighting Shoal’s fair queuing.

Reducing the impact of detouring on network latency. To show that the optimizations described in §3.3.3 indeed improve the network latency, we choose two nodes, Node-511 (source) and Node-0 (destination), to exchange short flows (20 KB) at regular intervals, and generate random background traffic amongst the remaining nodes. We plot the distribution of flow completion time (FCT) of short flows exchanged between nodes 511 and 0 in Fig. 14. The optimizations in §3.3.3 enable Shoal to achieve much smaller and predictable FCT for target short flows—at the source, Shoal selectively adds cells to local queues where there is low contention, and at the intermediate node the queue length is bounded to two regardless of the cross-traffic, as the incast degree of Node-0 is one (Eq. 5). However, without the optimizations, the cross-traffic due to detouring significantly increases the FCT of target short flows.

7.3 Datacenter workloads

We now investigate the performance of Shoal in dynamic settings, using more realistic workloads.

Workload. We generate a synthetic workload, modeled after published datacenter traces [2, 22]. Flow sizes are heavy tailed, drawn from a Pareto distribution with shape parameter 1.05 and mean 100 KB [3, 4]. Flows arrive according to a Poisson process and each simulation ends when one million flows have completed. Flow sources and destinations are chosen with uniform probability across all nodes (we will study the impact of skewed workloads in §7.4).

Network load. We define network load $L = \frac{F}{R \cdot N \cdot \tau}$ where F is the mean flow size, R is the per-node bandwidth, N is the number of nodes, and τ is the mean inter-arrival flow time, e.g., $L = 1$ means that, on average, there are N active flows.

Evaluation metric. We evaluate Shoal based on the flow completion time (FCT) for short flows (≤ 100 KB) and average goodput (i.e., throughput after accounting for the 8 B cell header overhead (§3.5)) for long flows (≥ 1 MB).

Baseline 1: Direct-connect network. We start with comparing Shoal against a rack-scale network using a direct-connect

topology. We arranged the 512 nodes into a 3D torus, which is the topology used in the AMD SeaMicro 15000-OP [79]. As with the Shoal network, we assume an aggregate node bandwidth of 100 Gbps. We use R2C2 [15] for congestion control. For all values of load, Shoal consistently outperforms the rack-scale setup up to a factor of 14.9 for tail FCT for short flows (resp. a factor of 4.8 for avg goodput for long flows). This is due to the multi-hop nature of direct-connect topologies; it significantly increases the end-to-end latency as queuing can occur at any hop. Further, node bandwidth is also used to forward traffic originating several hops away, which reduces the overall throughput. This does not occur in Shoal as packets only traverse one hop and the congestion control guarantees bounded queues.

Baseline 2: Packet-switched network. Now we compare Shoal against a 512-node packet-switched network (PSN), that connects the nodes using Clos topology with full-bisection bandwidth. The interface bandwidth of each node is 50 Gbps. Thus Shoal is provisioned with $2 \times$ bandwidth, to compensate for the throughput overhead of detouring packets. Note that despite the extra bandwidth, Shoal’s power is still estimated to be lower than that of PSN with a comparable or lower cost (§5). As baselines, we use DCTCP [2], NDP [25] and DCQCN [54] as the three state-of-the-art congestion control algorithms atop a packet-switched network. The baselines are based on the simulator used in [25]. We assume 1500 B packets for all of them. DCTCP and DCQCN use standard ECMP routing, with the congestion window size of 35 packets and queue size of 100 packets. NDP uses packet spraying for routing with initial window size of 35 packets and queue size of 12 packets.

As shown in Fig. 15a, at low to moderate load, Shoal exhibits an average FCT comparable to DCQCN and DCTCP and slightly higher than NDP. This increase is a consequence of the use of detouring due to the static schedule (3.3.1). However, Shoal outperforms DCTCP and DCQCN in terms of tail FCT for short flows by a factor of $3 \times$ at low load and $2 \times$ at high load (resp. outperforms NDP by a factor of $1.5 \times$ and $2 \times$). The reason for this is three-fold: i) $2 \times$ bandwidth per node in Shoal reduces the serialization delay, ii) selectively adding cells from new flows to local queues with low contention reduces queuing delay at the source, and iii)

Shoal’s congestion control ensures small and bounded queuing at intermediate nodes, thus reducing the queuing delay at intermediate nodes. Shoal also outperforms all three baselines in terms of long flow goodput (Fig. 15b) by a factor of $1.7\times$, even at high load. This is primarily due to the fact that each Shoal node is equipped with more bandwidth.

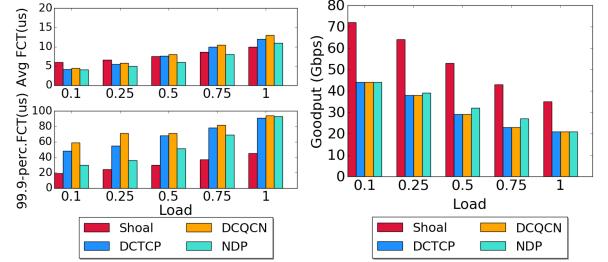
Queueing and reordering. To validate our claim that Shoal operates with very small queues, we plot the maximum queuing observed across all nodes in Fig. 16. Even at high load, the maximum queue size is 11 cells (704 B) and the maximum aggregate queue per node is 336 cells (21 KB). Maximum cell reordering within a flow across all nodes and across all values of load is 200 KB (Fig. 16).

Node failures. We now focus our attention to the impact of node failures. We ran the same workload as in the previous experiments ($L = 1$) but at the beginning of each experiment we fail an increasing fraction of nodes (up to 50%). As expected, the goodput decreases linearly ($2\times$ worse for 50% failure rate, Fig. 17) because the slots corresponding to the failed nodes are wasted. We can alleviate this with a more sophisticated mechanism that, on detecting long-term failures, updates the schedule of both rack nodes and switches to discount the failed nodes. FCT also increases with increasing failed nodes, as the number of paths along which cells from a flow can be sent is reduced, resulting in higher subflow collision and increased queuing. However, Fig. 17 shows that even for high failure rate the increase in completion time is rather marginal, e.g., $1.5\times$ for a 40% failure rate (resp. $1.2\times$ for 20% failure rate), thus making Shoal amenable even for sealed rack-scale deployments in which replacing failed nodes is not possible.

Impact of epoch size on network latency. Next, we study the impact of epoch size on the FCT. Larger epoch size results in higher latency (§3.4). In the first experiment, we reduced the number of channels from 4×25 Gbps to 2×50 Gbps, thus doubling the epoch size. This increased tail FCT by $1.26\times$ at low load (resp. $1.15\times$ at high load). In the second experiment, we kept the number of channels constant at 4, and increased the number of nodes to 1,024, again doubling the epoch size. In this case, tail FCT at low load grew by $1.28\times$ (resp. $1.2\times$ at high load). This, in turn, shows that Shoal’s performance scales reasonably well with number of nodes, making it suitable even for very dense racks.

7.4 Disaggregated workloads

Finally, we evaluate the performance of Shoal on disaggregated workloads, based on recently published traces [19]. These traces comprise a variety of real-world applications, including batch processing, graph processing, interactive queries, and relational queries. To generate the workloads, we mapped each rack node to one of the server resources (CPU, memory, and storage) and created flows between them following the distributions observed in these traces. This yielded a much more skewed workload than the one in §7.3



(a) Flow completion time. (b) Average flow goodput.

Figure 15: Flow completion time (short flows ≤ 100 KB) and avg flow goodput (long flows ≥ 1 MB) vs. traffic load.

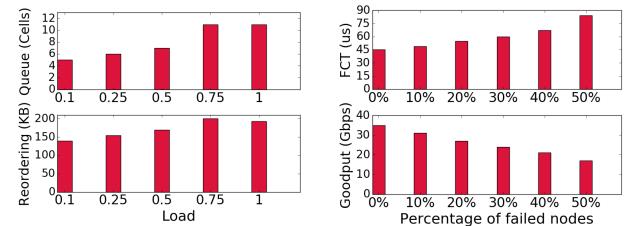


Figure 16: Max queue size and max cell reordering vs. traffic load.

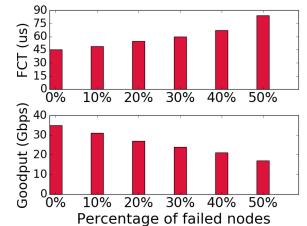
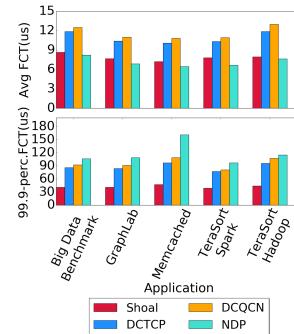


Figure 17: Short flow 99.9p FCT and long flow avg goodput vs. failure.



(a) Flow completion time. (b) Average flow goodput.

Figure 18: Flow completion time (short flows ≤ 100 KB) and avg flow goodput (long flows ≥ 1 MB) for different applications with disaggregated workload.

with more than 84% of the flows being generated among a third of the nodes.

Fig. 18 shows the results for all the six applications, assuming a mean inter-arrival time of 12.65 ns. Shoal significantly outperforms the baselines in terms of both the tail FCT for short flows (factor of $2\times$ or more) and avg goodput for long flows (factor of $2.5\times$). As explained in §7.3, this is due to higher bandwidth provisioning in Shoal in combination with its highly effective scheduling and congestion control mechanisms (resulting in maximum queue size of just 10 cells across all applications).

These results show the versatility of Shoal and its ability to carry different types of traffic, including disaggregated workloads, with high throughput and low latency.

8 Discussion

The focus of this paper is on the design of a network for disaggregated racks. Here we discuss a few open questions.

Integrating a Shoal rack with rest of the datacenter. A key question is how to seamlessly integrate a rack-scale network, such as Shoal, with the rest of the datacenter, which might consist of both disaggregated and traditional racks. Existing rack-scale designs [35] typically use a few rack nodes as gateways that are then directly connected to the datacenter network. Such a design could be adapted for Shoal as well—the gateway nodes would act as a bridge between Shoal’s network stack and datacenter-wide network stack, such as TCP/IP over Ethernet. There are, however, several key challenges that still remain to be addressed such as the interplay between different congestion controls and how to design the interface between IP packets and Shoal cells (e.g., packet reassembly/fragmentation and encapsulation).

Running applications on top of Shoal. Shoal is a link layer architecture with support for congestion control. Running applications on top of Shoal, however, requires a transport layer providing application multiplexing, reliability, and flow control. One option would be to re-use an existing transport layer such as TCP, although the impact of the interaction of its congestion-control mechanism with Shoal’s remains an open question. Another approach would be to design a Shoal-specific transport layer. This would be significantly simpler as congestion control is already handled by Shoal. We leave the exploration of these options for future work.

9 Related work

Disaggregated architectures promise significant cost, power, and performance gains [56, 67, 71]. However, unlocking these benefits requires solving numerous challenges.

Topology and technology. Several topologies have been investigated for disaggregated racks. Direct-connect topologies whereby each node is connected to a small subset of other rack nodes through point-to-point links are common in super computers and have been adopted in some commercial disaggregated racks. For example, both AMD SeaMicro [79] and HP Moonshot [69] use the 3D Torus topology and custom routing. Motivated by the fact that the best direct-connect topology is workload-dependent, reconfigurable networks have emerged as an attractive alternative. At rack scale, XFabric [35] combines circuit switches with SoC-based packet switches to reconfigure the rack’s topology on the fly, while for datacenter networks, electrical [10], optical [11, 21, 24, 42, 53] and wireless technologies [23] that operate like a circuit switch have been proposed. These solutions typically rely on a centralized controller to schedule traffic. This imposes significant communication and computation overhead and requires accurate demand estimation.

Congestion control through tight network-host coupling. Shoal’s congestion control tightly couples the network fab-

ric with host network stack. For packet-switched networks, there is already a trend towards tighter coupling between the network and servers for low latency congestion control in datacenters; for example, using ECN as a feedback signal in DCTCP [2]. Congestion control mechanisms specialized for RDMA over converged Ethernet, such as DCQCN [54] and TIMELY [39], also rely on a closer coupling with the network. Shoal is an extreme design point in this direction as the coupling of its congestion mechanism to its fabric achieves bounded queuing and fairness despite very high multi-pathing. For direct-connect topologies, R2C2 [15] is a recently proposed congestion control that relies on broadcasting of flow events across the rack. It achieves computation tractability at the expense of network utilization.

Load-balanced switch. In 2002, load-balanced switches [9, 29] were proposed as a way to obviate arbitration in monolithic switches. Shoal’s fabric operates like a load-balanced switch. However, instead of using an explicit intermediate stage (i.e., special nodes for detouring) as in the original proposal, Shoal detours cells through other rack nodes. Furthermore, while the original technique focused on monolithic switches, we scale it to a hierarchy of switches.

The load-balancing approach is also at the core of RotorNet [38], an optical network for datacenters that does not require a centralized controller. By leveraging multiple rack uplinks, RotorNet reduces epoch duration and proposes a novel indirection technique that lowers the throughput impact of load balancing. However, it uses optical circuit switches with a relatively high reconfiguration delay ($20\ \mu s$) and hence, requires a separate packet-switched network for low-latency traffic. It also does not ensure bounded queuing. In contrast, Shoal works atop circuit switches with nanosecond-reconfiguration, and proposes novel congestion control and scheduling mechanisms that achieve high throughput, low latency, fairness and bounded queuing for all flows atop a purely circuit-switched fabric.

10 Summary

We presented Shoal, a network architecture for disaggregated racks that couples a circuit-switched fabric with the nodes’ network stack. The fabric operates like a rack-wide switch with a static schedule. Rack nodes achieve coordination-free scheduling by detouring their traffic uniformly, and implement backpressure-based congestion control which achieves fairness and bounded queuing. Our FPGA-based prototype achieves good performance and illustrates that Shoal’s mechanisms are amenable to hardware implementation. Our results show that Shoal can achieve high throughput and low latency across diverse workloads while operating comfortably within the rack’s power budget. This demonstrates that disaggregated architectures can be deployed using today’s technologies and not need be gated on the viability of future advancements in low-power technologies for packet-switches.

Acknowledgments

We thank the anonymous reviewers, George Porter, Michael Schapira, and our shepherd, Alex Snoeren, for their useful comments and suggestions. This research was partially supported by DARPA CSSG (D11AP00266), NSF (1053757, 1440744, 1422544, 1413972, and 1704742), European Unions Horizon 2020 research and innovation programme under the SSICLOPS project (agreement No. 644866), Google Faculty Research Award, Microsoft Research PhD scholarship, and gifts from Cisco, Altera and Bluespec.

References

- [1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
- [2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.
- [3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.
- [4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*, 2013.
- [5] K. Asanovic. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *FAST*, 2014. Keynote.
- [6] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogas, E. Peterson, and A. Rowstron. Pelican: A Building Block for Exascale Cold Data Storage. In *OSDI*, 2014.
- [7] H. Ballani, P. Costa, I. Haller, K. Jozwik, K. Shi, B. Thomsen, and H. Williams. Bridging the Last Mile for Optical Switching in Data Centers. In *Optical Fiber Communication Conference (OFC)*, 2018.
- [8] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.
- [9] C.-S. Chang, D.-S. Lee, and Y.-S. Jou. Load Balanced Birkhoff-von Neumann Switches, Part I: One-stage Buffering. *Comput. Commun.*, 25(6), Apr. 2002.
- [10] A. Chatzieleftheriou, S. Legtchenko, H. Williams, and A. Rowstron. Larry: Practical Network Reconfigurability in the Data Center. In *NSDI*, 2018.
- [11] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *NSDI*, 2012.
- [12] Q. Cheng, A. Wonfor, J. L. Wei, R. V. Penty, and I. H. White. Demonstration of the feasibility of large-port-count optical switching using a hybrid Mach-Zehnder interferometer-semiconductor optical amplifier switch module in a recirculating loop. *Optics Letters*, 39(18), 2014.
- [13] K. Clark, H. Ballani, P. Bayvel, D. Cletheroe, T. Gerard, I. Haller, K. Jozwik, K. Shi, B. Thomsen, P. Watts, H. Williams, G. Zervas, P. Costa, and Z. Liu. Sub-Nanosecond Clock and Data Recovery in an Optically-Switched Data Centre Network. In *European Conference on Optical Communication (ECOC), Post-deadline paper*, 2018.
- [14] R. Colwell. The chip design game at the end of Moore's law. In *HotChips*, 2013.
- [15] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A Network Stack for Rack-scale Computers. In *SIGCOMM*, 2015.
- [16] M. Ding, A. Wonfor, Q. Cheng, R. V. Penty, and I. H. White. Scalable, Low-Power-Penalty Nanosecond Reconfigurable Hybrid Optical Switches for Data Centre Networks. In *Proceedings of the Conference on Lasers and Electro-Optics (CLEO)*, 2017.
- [17] V. Eramo and M. Listanti. Power Consumption in Bufferless Optical Packet Switches in SOA Technology. *IEEE/OSA Journal of Optical Communications and Networking*, 1(3), 2009.
- [18] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.
- [19] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Nov. 2016.
- [20] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, page 1. ACM, 2015.
- [21] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*, 2016.
- [22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.
- [23] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *SIGCOMM*, 2011.
- [24] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *SIGCOMM*, 2014.
- [25] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. Moore, G. Antichi, and M. Wojcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.
- [26] B. Ji, C. Joo, and N. B. Shroff. Exploring the inefficiency and instability of Back-Pressure algorithms. In *INFOCOM*, 2013.
- [27] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirosky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *DATE*, 2016.
- [28] K. Keeton. The Machine: An Architecture for Memory-centric Computing. <http://www.mcs.anl.gov/events/workshops/ross/2015/slides/ross2015-keeton.pdf>.
- [29] I. Keslassy, S.-T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, and N. McKeown. Scaling Internet Routers Using Optics. In *SIGCOMM*, 2003.
- [30] J. H. Kim, W. S. Kwon, H. Lee, K.-S. Kim, and S. Kim. A novel method to acquire ring-down interferograms using a double-looped mach-zehnder interferometer. In *Lasers and Electro-Optics (CLEO)*, 2014.
- [31] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2), Feb. 2007.
- [32] M. Lapinski, T. Wlostowski, J. Serrano, and P. Alvarez. White Rabbit: a PTP Application for Robust Sub-nanosecond Synchronization. In *Proceedings of the International IEEE Symposium*

- posium on Precision Clock Synchronization for Measurement Control and Communication, 2011.
- [33] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *SIGCOMM*, 2016.
- [34] S.-J. Lee, K. Lee, and H.-J. Yoo. Analysis and Implementation of Practical, Cost-Effective Networks on Chips. *IEEE Des. Test*, 22(5), Sept. 2005.
- [35] S. Legtchenko, N. Chen, D. Cletheroe, A. Rowstron, H. Williams, and X. Zhao. XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers. In *NSDI*, 2016.
- [36] O. Liboiron-Ladouceur, A. Shacham, B. Small, B. Lee, H. Wang, C. Lai, A. Biberman, and K. Bergman. The Data Vortex Optical Packet Switched Interconnection Network. *Journal of Lightwave Technology*, 26(13), 2008.
- [37] M. Lipinski, T. Wlostowski, J. Serrano, P. Alvarez, J. D. G. Cobas, A. Rubini, and P. Moreira. Performance results of the first White Rabbit installation for CNGS time transfer. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2012.
- [38] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. RotorNet: A Scalable, Low-complexity, Optical Datacenter Network. In *SIGCOMM*, 2017.
- [39] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Sigcomm*, 2015.
- [40] P. Moreira, J. Serrano, T. Wlostowski, P. Loschmidt, and G. Gaderer. White Rabbit: Sub-Nanosecond Timing Distribution over Ethernet. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2009.
- [41] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLoS*, 2014.
- [42] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *SIGCOMM*, 2013.
- [43] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantines, J. Demme, H. Esmailzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*, 2014.
- [44] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.
- [45] M. Rizzi, M. Lipinski, T. Wlostowski, J. Serrano, G. Daniluk, P. Ferrari, and S. Rinaldi. White Rabbit Clock Characteristics. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2016.
- [46] A. Rylyakov, J. E. Proesel, S. Rylov, B. G. Lee, J. F. Bulzacchelli, A. Ardey, B. Parker, M. Beakes, C. L. S. Christian W. Baks, and M. Meghelli. A 25 Gb/s Burst-Mode Receiver for Low Latency Photonic Switch Networks. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 50(12), Dec. 2015.
- [47] Spectrum. 32-port non-blocking 100gbe open ethernet switch system. http://format.com.pl/site/wp-content/uploads/2015/09/pb_sn2700.pdf.
- [48] F. Testa and L. Pavese, editors. *Optical Switching in Next Generation Data Centers*. Springer, 2017.
- [49] Tomahawk. As7712-32x/as7716-32x series switch. http://www.edge-core.com/_upload/images/AS7712_AS7716-32X_DS_R09_20170607.pdf.
- [50] L. G. Valiant and G. J. Brebner. Universal Schemes for Parallel Communication. In *ACM Symposium on Theory of Computing*, 1981.
- [51] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNEXT*, 2010.
- [52] X. Ye, Y. Yin, S. Yoo, P. Mejia, R. Proietti, and V. Akella. DOS - A scalable optical switch for datacenters. In *ANCS*, 2010.
- [53] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *SIGCOMM*, 2012.
- [54] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments . In *SIGCOMM*, 2015.
- [55] 40g/50g high speed ethernet subsystem v1.0. https://www.xilinx.com/support/documentation/ip_documentation/l Ethernet/v1_0/pg211-50g-ethernet.pdf.
- [56] Amazon joins other web giants trying to design its own chips. <http://bit.ly/1J5t0fE>.
- [57] Bluespec. www.bluespec.com.
- [58] Broadcom BCM56960 series (Tomahawk). <http://bit.ly/2wPnJHI>.
- [59] Calxeda EnergyCore ECX-1000. <http://bit.ly/1nCgdH0>.
- [60] Choosing the Optimal Data Center Power Density. http://www.apc.com/salestools/VAVR-8B3VJQ/VAVR-8B3VJQ_R0_EN.pdf.
- [61] Cisco Nexus 7700 Switches Data Sheet. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/data_sheet_c78-728187.html.
- [62] Data Plane Programming at Terabit speeds. https://p4.org/assets/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf.
- [63] DE5-Net FPGA development kit. <http://de5-net.terasic.com.tw>.
- [64] Electronic Design Blog. <http://bit.ly/2xsIhaU>.
- [65] Facebook Engineering Blog. Introducing "Yosemite": the first open source modular chassis for high-powered microservers. <http://bit.ly/1MpHjuW>.
- [66] How is a Mega Data Center Different from a Massive One? <http://www.datacenterknowledge.com/archives/2014/10/15/how-is-a-mega-data-center-different-from-a-massive-one>.
- [67] How Microsoft Designs its Cloud-Scale Servers. <http://bit.ly/1HKCy27>.
- [68] HP Labs. The Machine. <https://www.labs.hpe.com/the-machine>.
- [69] HP Moonshot System. <http://bit.ly/1mZD4yJ>.
- [70] IEEE Standard 1588-2008. <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757>.
- [71] Intel, Facebook Collaborate on Future Data Center Rack Technologies. <http://intel.ly/MRpOMO>.
- [72] Intel Rack Scale Architecture. <http://ubm.io/1iejjx5>.
- [73] Intel Xeon Processor D-1500. <http://intel.ly/2hrn1hR>.
- [74] Introducing Big Basin: Our next-generation AI hardware.

- <http://bit.ly/2pQPU2S>.
- [75] ITU-T Rec. G.8262. <http://www.itu.int/rec/T-REC-G.8262>.
 - [76] Macom M21605 Crosspoint Switch. <http://www.macom.com/products/product-detail/M21605/>.
 - [77] Mellanox ConnectX-4 Ethernet Single and Dual QSFP28 Port Adapter Card User Manual. <http://bit.ly/2f1b0uZ>.
 - [78] NVIDIA DGX-1 System Architecture. <http://bit.ly/2xjV54K>.
 - [79] SeaMicro SM15000 Fabric Compute Systems. <http://bit.ly/1hQepIh>.
 - [80] Stratix V FPGA. <http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp>.
 - [81] The Next Platform Blog. Mellanox Comes Out Swinging With 100G Spectrum Ethernet. <http://bit.ly/2cZXrwo>.
 - [82] Under The Hood Of Googles TPU2 Machine Learning Clusters. <http://bit.ly/2qfaUDd>.

Appendix

A Accounting for propagation delay

Even at the scale of a rack, the propagation delay is not negligible as compared to the transmission time of a cell. This means that a cell sent at time slot t will not be received within the same slot at the receiver. More generally, say that the cell is received at slot $t + k$. For the feedback mechanism described in § 3.3.2 to work optimally in the face of such propagation delay, there should be at least k slots from the time node i transmits to node j and the time node j transmits to i , as j needs to know the destination of the last cell that i sent to j to send back the right rate limit feedback. We can easily re-arrange any cyclic permutation schedule such as in Fig. 3 to ensure this property, as long as k is less than half the number of slots in an epoch—when number of rack nodes, N , is odd, this constraint can be easily accommodated by inverting the order of the last $\frac{N-1}{2}$ columns. This would ensure that there are exactly $\frac{N-1}{2}$ slots between the slot in which i communicates with j and the one in which j communicates with i . This is shown in Fig. 19(a). For even N , this is slightly more complicated as it requires to introduce an additional empty slot per node to satisfy this requirement as demonstrated in Fig. 19(b), consequently resulting in a throughput reduction by a factor of $\frac{1}{N}$ for each node.

B Resource consumption for Shoal’s FPGA-based implementation

To understand the resource utilization at scale, we synthesize our design onto a Stratix-V FPGA [80], which comprises 234,720 adaptive logic modules (ALMs) and 52 Mbits of BRAM. Assuming 500 nodes and 64-port switches, our NIC logic consumes about 84% of ALMs and 13% of BRAM (resp. 2% and 0.2% for the switch). Finally, we did a power analysis to quantify the overhead of Shoal’s additional functionalities. We leverage the study done in [31] to translate the power consumption of our FPGA-based design into an equivalent ASIC design. Assuming a 500-node rack, this re-

Node	Time slot					
	1	2	3	4	5	6
1	2	3	4	7	6	5
2	3	4	5	1	7	6
3	4	5	6	2	1	7
4	5	6	7	3	2	1
5	6	7	1	4	3	2
6	7	1	2	5	4	3
7	1	2	3	6	5	4

(a) Fabric schedule for odd N

Node	Time slot							
	1	2	3	4	5	6	7	8
1	2	3	4	5	8	7	6	
2	3	4	5	6	1	8	7	
3	4	5	6	7	2	1	8	
4	5	6	7	8	3	2	1	
5	6	7	8		4	3	2	
6	7	8	1		5	4	3	2
7	8	1	2		6	5	4	3
8	1	2	3		7	6	5	4

(b) Fabric schedule for even N

Figure 19: Fabric schedule accounting for propagation delay.

sulted in Shoal NIC’s on-chip extra functionality – routing and congestion control – consuming up to 11% of the power consumed by commercial ASIC NICs, and Shoal switch’s extra functionality – reconfiguration using a static schedule – consuming just 0.1% of the power consumed by commercial ASIC circuit switches.

C Quality-of-Service

By default, Shoal assumes that each cell belongs to the same traffic class, and schedules them using a single per destination FIFO queue. However, Shoal can be easily extended to support multiple traffic classes by maintaining multiple per destination FIFO queues, one per traffic class, and scheduling cells from the FIFOs in the order of their priority. Note that in this case the queue bound (§3.3.4) will only hold for the highest priority traffic, while the lower priority traffic can see tail drops. To notify the sender of the tail drop, the node sets the `last-cell-dropped` field to 1 in the header of the next cell it sends to the sender. And finally, the rate limit feedback for a single traffic class in Eq 6 is updated for multiple traffic classes as follows—for a cell in traffic class c ,

$$\text{rate limit feedback}_{ji}^c = \sum_p [\text{len}(Q_{jk}^p) + \text{len}(R_{jk}^p) - 1] \\ \forall \text{ traffic class } p \text{ s.t. } \text{priority}(p) \succeq \text{priority}(c)$$

D Recovering from cell corruption

Shoal uses the CRC field in the cell header to check for cell corruption. If node j receives a corrupted cell from node i , it first extracts the header fields corresponding to congestion control, namely the `rate limit feedback` and `last-cell-dropped`, and then discards the cell. It also signals the sender i that the cell was dropped by setting the `last-cell-dropped` field to 1 in the header of the next cell sent to node i . Node i then re-transmits the last cell it sent to node j . Since the `last-cell-dropped` field in the corrupted cell might also have been corrupted, node j takes the conservative approach of assuming the field was set to 1 and re-transmits the last cell it sent to node i . Further, if the `rate limit feedback` field also happens to be corrupted, the queue bound as described in §3.3.4 might get violated and there could be tail drops in the worst case. Shoal again uses the `last-cell-dropped` field in the cell header to notify the sender of any tail drop.

NetScatter: Enabling Large-Scale Backscatter Networks

Mehrdad Hessar[†], Ali Najafi[†] and Shyamnath Gollakota

University of Washington

[†]Co-primary Student Authors

Abstract – We present the first wireless protocol that scales to hundreds of concurrent transmissions from backscatter devices. Our key innovation is a distributed coding mechanism that works below the noise floor, operates on backscatter devices and can decode all the concurrent transmissions at the receiver using a single FFT operation. Our design addresses practical issues such as timing and frequency synchronization as well as the near-far problem. We deploy our design using a testbed of backscatter hardware and show that our protocol scales to concurrent transmissions from 256 devices using a bandwidth of only 500 kHz. Our results show throughput and latency improvements of 14–62x and 15–67x over existing approaches and 1–2 orders of magnitude higher transmission concurrency.

1 Introduction

The last few years have seen rapid innovations in low-power backscatter communication [20, 31, 18, 15, 17, 21], culminating in long range and reliable backscatter systems [26, 23, 28]. These designs enable wireless devices to communicate at microwatts of power and operate reliably at long ranges to provide whole-home or warehouse coverage. To achieve this, they employ low-power coding techniques such as chirp spread spectrum, to decode weak backscatter signals below the noise floor [23, 26] and deliver long ranges.

While these long range backscatter systems are promising for enabling power harvesting devices (e.g., solar and vibrations) as well as cheap and small Internet-connected devices that operate on button-cells or flexible printed batteries, they primarily work at the link layer and are not designed to scale with the number of devices — all these prior designs [26, 23, 28] are evaluated in a network of 1–2 devices.

Our goal in this paper is to design a network protocol that enables these low-power backscatter networks to support hundreds to thousands of concurrent transmissions. This is challenging because the resulting design must operate reliably with weak backscatter signals that can be close to or below the noise floor. To this end, we present *NetScatter*, the first wireless protocol that can scale to hundreds and thou-

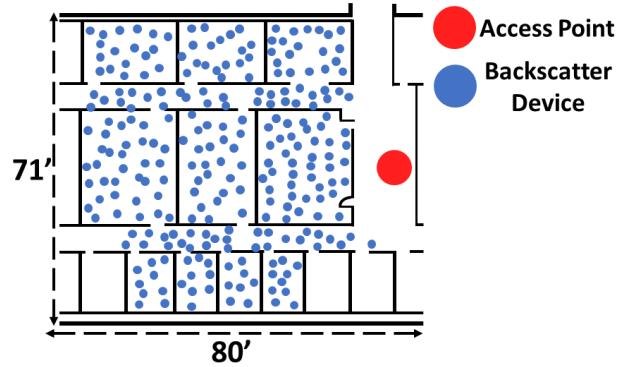


Figure 1: **Large-Scale Network Deployment of Backscatter Devices.** We deploy 256 backscatter devices across a floor of an office building covering multiple rooms.

sands of concurrent transmissions from backscatter devices. Our design enables concurrent transmissions from 256 devices over a bandwidth of 500 kHz. Consequently, it can support transmissions from a thousand concurrent backscatter devices using a total bandwidth of only 2 MHz.

Our key innovation is a distributed coding mechanism that satisfies four key constraints: i) it enables hundreds of devices to concurrently transmit on the same frequency band, ii) it can operate below the noise floor while achieving reasonable bitrates, iii) its coding operation can be performed by low-power backscatter devices, and iv) it can decode all the transmissions at the receiver using a single FFT operation, thus minimizing the receiver complexity.

We introduce *distributed chirp spread spectrum coding*, which uses a combination of chirp spread spectrum (CSS) modulation and ON-OFF keying. In existing CSS systems (e.g., LoRa backscatter [26]), the AP transmits a continuous wave signal which each device backscatters and encodes bits using different cyclic shifts of a chirp signal. In contrast, in our distributed CSS coding, we assign a different cyclic shift of the chirp to each of the concurrent devices. Each device uses ON-OFF keying over these cyclic shifted chirps to convey bits, i.e., the presence and absence of the corresponding cyclic shifted chirp correspond to a ‘1’ and ‘0’ bit respectively, as shown in Fig. 2. Note that in comparison to

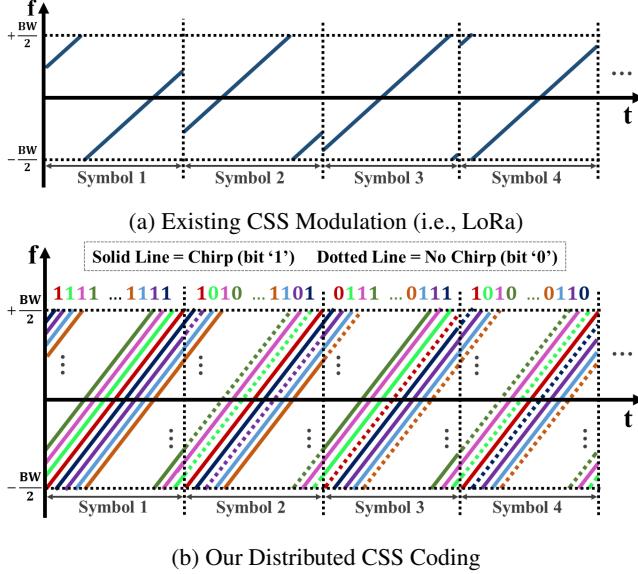


Figure 2: **NetScatter Overview.** In traditional CSS systems, a single device uses different cyclic shifts to convey bits. In distributed CSS coding, each cyclic shift is assigned to a different backscatter device. Each device then uses the presence and absence of cyclic shift to send ‘1’ and ‘0’ bits.

existing CSS systems where each device transmits $\log_2 N$ bits using N cyclic shifts, our distributed design enables N concurrent devices, each of which transmits a single bit, using ON-OFF keying. Thus, our design transmits a total of N bits within a chirp duration, providing a theoretical gain of $\frac{N}{\log_2 N}$.

Our design leverages the fact that creating concurrent cyclic-shifted chirps at a single device requires distributing its transmit power amongst all the cyclic shifts, which reduces the ability of the receiver to decode each chirp. Instead we generate concurrent cyclic-shifted chirps across a distributed set of low-power devices in the network. This allows us to efficiently leverage the coding gain provided by chirp spread spectrum under the noise floor [9]. Further, we can decode all the concurrent transmissions using a single FFT operation, since cyclic shifting the chirps in the time domain translates to offsets in the frequency domain.

Using the above distributed coding mechanism in practice, however, is challenging for two key reasons.

- *Near-far problem.* A fundamental problem with enabling concurrent transmissions is that signals from a nearby backscatter device can overpower a farther concurrent device. To address this issue, we introduce two main techniques. First, we present a power-aware cyclic shift allocation technique in §3.2.3, where lower SNR devices use much different cyclic shifts than higher SNR devices. We show that such an allocation can allow backscatter devices that have an SNR difference of up to 35 dB to be concurrently decoded. Second, to account for channel variations over time, we develop a zero-overhead power adaptation algorithm where backscatter devices use reciprocity to esti-

mate their SNR at the AP, using the signal strength of the AP’s query message. The backscatter devices then adjust their transmission power to fall within the tolerable SNR difference. Since this calibration is done independently at each backscatter device using the AP’s query, it does not require additional communication overhead at the AP.

- *Timing synchronization.* The above design requires all the devices to start transmitting at the same time so as to enable concurrent decoding. However, hardware variations and propagation delays of different devices can make it challenging for hundreds of devices to be tightly synchronized in time. To avoid this coordination overhead, we leave gaps between cyclic shifts to ensure that concurrent devices can be decoded. We explore the trade-off between the required gaps and the chirp bandwidths in §3.2.1.

We implement *NetScatter* on a testbed of backscatter devices. We create backscatter hardware that implements *NetScatter* and includes circuits to perform automatic power adaptation before each transmission. We deploy our backscatter testbed with 256 devices in an office building spanning multiple rooms as shown in Fig. 1. We implement our receiver algorithm using USRP X-300 software-defined radios. Our results reveal that over a 256 node backscatter deployment, *NetScatter* achieves a 14–62x gain over prior long-range backscatter systems [26] for its end-to-end link layer data rates. The key benefit however is in the network latency which sees a reduction of 15–67x.

Contributions. Our paper demonstrates, to the best of our knowledge, the first network protocol that achieves orders of magnitude more concurrent transmissions than existing backscatter systems. The closest work to our design is *Choir* [12] *in the radio domain*, which decodes concurrent transmissions from 5–10 LoRa radios at a software radio. *Choir* leverages frequency imperfections to disambiguate between LoRa radios. However, backscatter devices achieve low-power operations by running at a lower frequency (1–10 MHz) than radios (900 MHz) and thus have much smaller frequency differences between backscatter devices. This severely limits the ability to rely on frequency imperfections to disambiguate between a large number of backscatter devices (see §2.2). In contrast, our distributed chirp spread spectrum coding mechanism provides a systematic approach to enable large scale backscatter networks.

2 CSS Primer & Existing Approaches

2.1 Primer on Chirp Spread Spectrum

In CSS, data is modulated using linearly increasing frequency signals or upchirps. The receiver demodulates these symbols in a two step process. First, it de-spreads these upchirp symbols by multiplying them by a downchirp and it then performs an FFT on the de-spread signal. Since the slope of the downchirp is the inverse of the slope of the upchirp, multiplication results in a constant frequency signal,

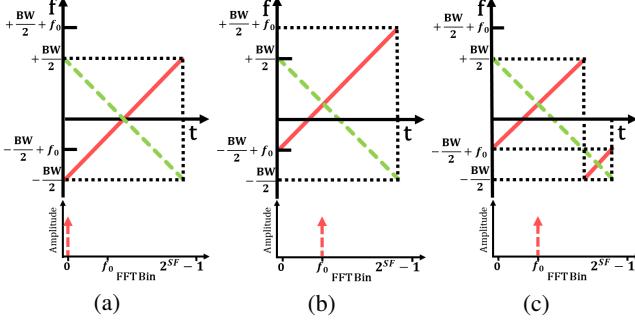


Figure 3: **CSS Primer.** We show upchirp and downchirp symbols and FFT results of their multiplication. (a) Baseline upchirp symbol, (b) frequency shifted upchirp symbol and (c) cyclically shifted upchirp symbol.

as shown in Fig. 3(a). Thus, taking an FFT on this will lead to a peak in an associated FFT bin. Changing the initial frequency of an upchirp will result in a change in the demodulated signal’s FFT bin peak index which corresponds to the initial change in frequency, as shown in Fig. 3(b). This property is used to convey information. When the sampling rate is equal to chirp bandwidth (BW), frequencies higher than $\frac{BW}{2}$ will alias down to $-\frac{BW}{2}$ as shown in Fig. 3(c). This means cyclically shifting in time is equivalent to changing the initial frequency and thus to conserve bandwidth, CSS uses cyclic shifts of the chirp in the time-domain instead of frequency shifts. This means that to modulate the data we just need to cyclically shift the baseline upchirp in time. Note that one can transmit multiple bits within each upchirp symbol. In particular, say the receiver performs an N point FFT. It can distinguish between N different cyclic shifts each of which corresponds to a peak in one of the N FFT bins. Thus, we can transmit $SF = \log_2 N$ bits within each upchirp symbol, where SF is called the spreading factor.

Based on above explanations, CSS can be characterized by two parameters: chirp bandwidth/sampling rate and spreading factor. Thus, each chirp symbol duration is equal to $\frac{2^SF}{BW}$ and the symbol rate is $\frac{BW}{2^SF}$. Since CSS sends SF bits per symbol, the bitrate is equal to $\frac{BW}{2^SF} SF$. This means increasing SF or decreasing BW decreases the bitrate. Further, the sensitivity of the system depends on the symbol chirp duration and increases with SF and decreases with BW.

2.2 Existing Collision Approaches

While existing CSS-based backscatter systems do not support collision decoding, we outline potential approaches to deal with collisions in CSS *radio* systems, i.e. LoRa, and explore whether they can be adopted for backscatter.

Using different spreading factors. One way to enable concurrent transmissions is to assign different spreading factors to each device. There are three problems with using multiple spreading factors in the same network: i) the receiver needs to use multiple FFTs and downchirps with different spreading factors to despread upchirp symbols of different devices,

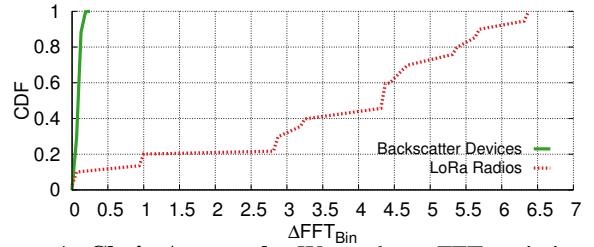


Figure 4: **Choir Approach.** We evaluate FFT variation of chirp symbols when $BW = 500$ kHz and $SF = 9$ for both active LoRa radios and backscatter devices.

which increases the receiver complexity with the number of concurrent transmissions, ii) in LoRa, different BW and SF can be concurrently decoded without sensitivity degradation, only if the chirp slope is different [25]. Specifically, if two chirp symbols transmitted concurrently with different BW and different SF, which result in the same chirp slope, $\frac{BW^2}{SF}$ (shown in Fig. 6 as well), the receiver cannot decode their concurrent transmissions. This results in only 19 different BW and SF pairs that could be used concurrently, iii) further, requiring receiver sensitivity better than -123 dBm and bitrates of at least 1 kbps limits these concurrent configurations to only 8, which does not support hundreds of concurrent devices on a 500 kHz band. Note that ignoring the receiver complexity, this approach is orthogonal to our design since we could in principle run multiple concurrent NetScatter networks with the above 8 SF and BW pairs. Evaluating this is not in the scope of this paper.

Choir [12]. Recent work on decoding concurrent LoRa transmissions leverages the hardware imperfections in radios to disambiguate between multiple transmissions. Specifically, radios have slight variations which result in timing and frequency offsets, which translate to fractional shifts in the FFT indexes. Choir [12] uses these fractional shifts, with a resolution of one-tenth of an FFT bin, to map the bits to each transmitter. However, as demonstrated in [12], in practice this approach does not scale to more than 5 to 10 concurrent devices. To understand this limitation in theory, consider N concurrent devices. The probability that each of these transmitters has a different FFT peak index fraction, given the resolution of one-tenth of an FFT bin, is equal to $\frac{10!}{(10-N)!10^N}$. When N is 5 this probability is only 30%. Moreover, if any two transmitters use the same cyclic shifted upchirp symbol at the same time, it will result in a collision that cannot be decoded. In the case of LoRa modulation, if there are N transmitters and assuming each device transmits a random set of bits during each symbol interval, the probability of two transmitters using the same cyclic shift is equal to: $1 - \prod_{i=1}^N (1 - \frac{i-1}{2^SF})$ which is approximately $\frac{N(N-1)}{2^{SF+1}}$.

For $SF = 9$ and $N = 10$, this probability is around 9%. This means that there is around 9% probability that within each CSS symbol, two transmitters will use the same upchirp cyclic shift, which the receiver cannot disambiguate.

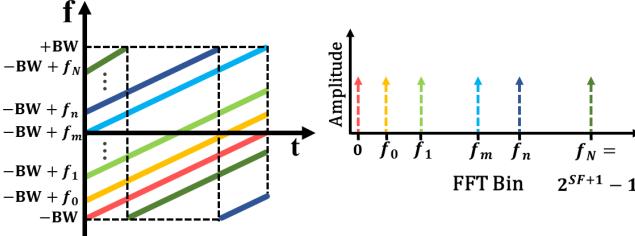


Figure 5: **Bandwidth Aggregation.** Here we use an aggregate bandwidth of $2BW$ but each device transmits only using BW . Upchirps with different cyclic shifts shown in different colors. Each upchirp is assigned to a device.

This probability increases to 32% with 20 devices, preventing concurrent decoding of a large number of transmitters.

Moreover, Choir is based on oscillator imperfection causing frequency variation on different devices, and Choir cannot differentiate two concurrent transmissions if both transmissions fall into same FFT bin fraction. Choir uses an active radio system which generates frequencies in 900 MHz band. However, since backscatter systems are designed to consume less power and only generate baseband signals, their output frequency is less than 10 MHz. Now, in the ideal scenario where the same crystal oscillator is used for both radios and backscatter devices, the frequency variation of the backscatter devices is 90 times smaller than radios and can be even less than 1 FFT bin depending on the SF and BW . This means a backscatter network cannot use all 10 different FFT bin fractions that Choir have used. Fig. 4 shows CDF of FFT bin variation for our actual backscatter hardware which are recorded over time. This results show that FFT variation is always less than a third of an FFT bin. Thus, Choir cannot enable large concurrent transmissions with backscatter.

In conclusion, the desired solution must satisfy three constraints: 1) ability to differentiate between FFT peaks corresponding to different backscatter devices, 2) ability to associate the FFT peaks to the corresponding devices, and 3) ensure that two devices do not use the same FFT peak at the same time. NetScatter design satisfies all these constraints.

3 NetScatter Design

3.1 Distributed CSS Coding

Our approach is to take advantage of low-power and high sensitivity of CSS modulation to design a communication and networking system that enables hundreds of backscatter devices to transmit at the same time.

At a high level, we use a combination of CSS modulation and ON-OFF keying to enable concurrent transmissions. Our intuition is as follows: if we look at the FFT plots of Fig. 3, all the FFT bins except one bin are empty; however these empty bins could be utilized for orthogonal transmissions. While it is difficult to design low-power backscatter devices that can transmit multiple cyclic shifts at the same

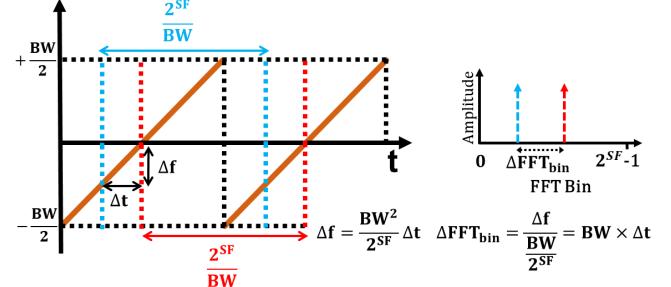


Figure 6: **Timing Mismatch**, in detecting beginning of a chirp symbol and its translation to FFT bin variation.

time, we can leverage all these empty bins by having different devices transmit different shifts and make use of the unused FFT bins. In particular, each device is assigned to a particular cyclic shifted upchirp symbol. It sends data by either sending the upchirp symbol or not sending it, i.e., by using ON-OFF keying of its assigned cyclic shifted chirp. Since, there are 2^{SF} FFT bins, ideally we can support 2^{SF} concurrent transmissions. This modulation will satisfy the above three requirements. The peaks can be differentiated and assigned to their corresponding devices. Moreover, none of them will use the same FFT bin at the same time.

We note the following about our distributed design.

- **Receiver complexity.** The received signal is composed of multiple transmissions. They can be demodulated by despreading with a baseline downchirp multiplication and performing an FFT operation. Then, we can determine the presence and absence of a peak in each FFT bin and find if the corresponding backscatter device is sending ‘0’ or ‘1’. The key point is that the process of despreading and performing FFT, which are the major contributors of the demodulation process and provide a coding gain for each of the backscatter devices enabling them to operate below the noise floor, are being done once and do not depend on the number of concurrent transmissions. This means that the receiver complexity is nearly constant with the number of devices.

- **Throughput gain.** In our approach, ideally there can be as many as 2^{SF} transmissions at each symbol period. Since each backscatter device uses ON-OFF keying over a symbol, their individual data rate is $\frac{BW}{2^{SF}}$. Thus, the aggregate network throughput is equal to BW . In comparison, LoRa have a throughput of $\frac{BW}{2^{SF}}SF$. Thus, we can achieve a throughput gain of $\frac{2^{SF}}{SF}$, which shows that the gain exponentially increases with the SF value used in the system. This is expected since the number of concurrent devices we can support is an exponential function of SF, i.e., 2^{SF} .

- **NetScatter and CDMA.** Our distributed CSS coding can be thought of as code-division multiplexing mechanism that is low-power and where each of the 2^{SF} cyclic shifts is in an orthogonal set of codes in a CDMA system. These orthogonal codes are then assigned to 2^{SF} different backscatter devices which enables 2^{SF} concurrent transmissions.

- **Gain in the context of Shannon capacity.** A key gain we are achieving in our design stems from using the power across all the concurrent backscatter devices. Specifically we note that the Shannon capacity of a multi-user network that operates under the noise floor linearly increases with the number of devices. Said differently, the multi-user capacity of an access point network is given as [27], $C = BW \log_2(1 + \frac{NP_S}{P_N})$. Here BW is the channel bandwidth, P_N and P_S are the noise and signal power and N is the number of concurrent devices. At SNRs below the noise floor, the above equation can be approximated as $\frac{BW}{\ln(2)} \frac{NP_S}{P_N}$, since $\ln(1+x) \approx x$ when x is small. This means that for systems that operate below the noise-floor, the network capacity scales linearly with the number of users. This linear increase stems from the fact that the N backscatter devices put in N times more power back to the AP than a single backscatter device.

- **Bandwidth aggregation.** The bitrate achieved by each backscatter device in our distributed design is given by $\frac{BW}{2^{SF}}$ and the number of concurrent devices is 2^{SF} . Thus, while we can increase the number of devices by increasing SF, it would decrease the bitrate of each device. Thus, to increase both the bitrate and the number of device we should increase the bandwidth, BW . Say, we want to support twice the number of devices while maintaining the same bitrate by using twice the bandwidth. This can be achieved in two ways. First, we can use two filters and independently operate two sets of devices across the two bands. This approach requires two different FFTs to be performed independently across the bands. The second approach is to use one aggregate band with twice the bandwidth, $2BW$, but use the same SF and chirp BW as before and alias down to $-BW$ whenever the chirp frequency hits the maximum as shown in Fig. 5. To demodulate this signal, we just need to multiply the signal which is composed of the aggregate band by the downchirp and perform 2×2^{SF} FFT operation once. The complexity of this method is lower than the former since there is no need to use filters and separate the bands.

3.2 Addressing Practical Issues

3.2.1 Timing Mismatch

The above design requires all the backscatter devices to be time synchronized. To understand why, consider two consecutive upchirps being sent by a device, as shown in Fig. 6. Now say that we demodulate the signal in these two timing durations, shown in blue and red, we will get different FFT peak locations. Specifically, with a Δt time difference between these durations, the corresponding FFT bin peak location would change by, $\Delta FFT_{bin} = \Delta t BW$. When this change is greater than a single FFT bin, backscatter devices that are assigned to consecutive cyclic shifts interfere with each other and cannot be decoded. Thus, all the devices should be time synchronized. In our design the access point sends a query message telling devices to transmit concurrently. The de-

vices use this query to synchronize and respond concurrently. First, we explain the sources of time delay in our system and then we explain our solutions. There are multiple factors that can contribute to time delays introduced in practice and can be different for different backscatter devices.

- **Hardware delay.** Unlike Wi-Fi devices which use much higher clock frequencies for processors, backscatter devices use low-power microcontrollers (MCUs) that can introduce a variable delay into the system. For backscatter devices, the source of these hardware delay variations come from the time the envelope detector receives the query message from the access point, communicates it to the MCU and then the device backscatters the chirp signal. As we show in §4.2, this hardware delay variations can be as high as $3.5 \mu s$, which can translate to more than one FFT bin at 500 kHz bandwidth.

- **Propagation delay and multipath.** Since backscatter devices can be at different distances to the access point, their time of flight (TOF) can be different. However, since our target application is for whole-home or warehouse sensing, the propagation distance is less than 100 m which translates to a $ToF < 666ns = \frac{2 \times 100}{3 \times 10^8}$ and corresponds to only a 0.33 FFT bin change, assuming a bandwidth of 500 kHz. The multipath delay spread for indoor environments is between 50 to 300 ns [24, 11]. For 500 kHz, this delay spread translates to less than 0.15 FFT bin change, which is negligible.

Our solution: Bandwidth-based cyclic-shift assignment. Hardware delay variations over time are hard to correct for. As described above, by nature of operating on MCUs and other low-power computational platforms, these devices have a hardware delay variation over time that changes between packets. Our solution to this problem is to put a few empty FFT bins adjacent to each FFT bin assigned to a device. That is, if FFT bin i is assigned to a device, the adjacent $SKIP - 1$ FFT bins are empty and not assigned to any device. This can be done by using only every $SKIP^{th}$ cyclic shift of the chirp. This ensures that the hardware delay does not result in interference between adjacent devices.

Achieving such an assignment requires us to answer the following key question: how do we pick the value $SKIP$? As described earlier, given the hardware delay variation Δt , the shift in the number of FFT bins is $\Delta t BW$. This means that there is a trade-off in our system regarding the total network throughput, bitrate for each device and sensitivity. In particular, increasing BW increases the number of FFT bins that have to be left empty and decreases the total network throughput. On the other hand, decreasing BW reduces the number of FFT bins but decreases the bitrate per device with the same SF . To compensate for the decreased device's bitrate, we can decrease the SF . Note that, we can choose total bandwidth, chirp BW and SF of the system by considering the hardware delay variations, required bitrate per device, sensitivity for each device and total number of devices. For our implementation, we pick the same total bandwidth and

Table 1: NetScatter Different Modulation Configurations, with maximum time/freq. mismatch that can be tolerated.

BW [kHz]	SF	Time Variation	Frequency Variation	Bit Rate [bps]	Sensitivity [dBm]
500	9	$2 \mu s$	976 Hz	976	-123
500	8	$2 \mu s$	1953 Hz	1953	-120
250	8	$4 \mu s$	976 Hz	976	-123
250	7	$4 \mu s$	1953 Hz	1953	-120
125	7	$8 \mu s$	976 Hz	976	-123
125	6	$8 \mu s$	1953 Hz	1953	-118

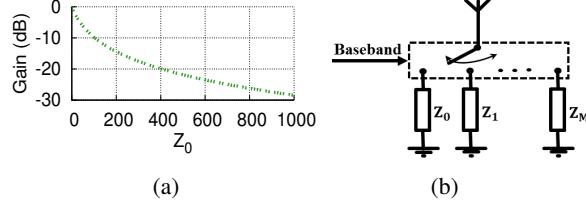


Figure 7: **Power Adjustment for Backscatter.** (a) gain normalized to maximum power as a function of Z_0 impedance and (b) switch network to support multiple power levels.

chirp BW of 500 kHz and $SF = 9$ which supports around 1 kbps (976 bps) bitrate at each device while ensuring that the number of empty bins between devices, $SKIP$, is two.

3.2.2 Frequency Mismatch

The devices experience frequency offsets because of hardware variations in the crystals used in their oscillators. As explained in §2.1, change in frequency translates to FFT bin change of the demodulated device packet. This again, causes one device to be misinterpreted as other device. Considering a bandwidth of BW and spreading factor of SF , the frequency difference between FFT bins is equal to $\frac{BW}{2SF}$. This means that a Δf frequency offset results in a change in the FFT bin of $\Delta FFT_{bin} = \frac{2SF\Delta f}{BW}$. Therefore, either increasing the spreading factor SF or decreasing the BW can increase the shift in the FFT bin. Crystals' frequency tolerance can be as high as 100 ppm [2]. Since backscatter devices run at a few MHz frequencies, this frequency variation translates to less than one FFT bin for the bandwidths and spreading factors in this paper which makes it negligible for our backscatter network.

Table 1 shows the timing and frequency mismatch that can be tolerated for different modulation configurations. As can be seen, there are multiple options for achieving the same bitrate and sensitivity. These options will result in different tolerable timing and frequency mismatch, requiring a different $SKIP$ value; this is validated using experiments in §4.2.

3.2.3 Near-Far Problem

Since our network are designed to work in below-noise conditions, we need to address the near-far problem in our decoding process at the receiver. Specifically, to account for the residual timing and frequency offsets, a CSS receiver has to achieve a sub-FFT bin resolution. To do so without increasing the sampling rate, the receiver uses zero-padding which adds zeros at the end of the time domain samples of

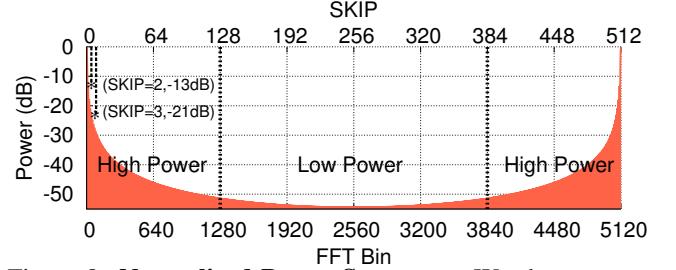


Figure 8: **Normalized Power Spectrum.** We show power spectrum of an upchirp multiplied by a baseline downchirp in FFT domain. This plot shows the main lobe and side lobes of a single chirp transmission. We assign devices to high and low power regions based on their power level.

the single chirp [12]. Zero-padding operation in the time domain is effectively a multiplication operation with a pulse which translates to convolution with a sinc function in the FFT domain. This makes it easier to locate the FFT peak location. However, convolving with a sinc function introduces side lobes as shown in Fig. 8. Assume that there are two devices with cyclic shifts $C_1 = 0$ and C_2 . If the power of C_2 is lower than power of C_1 's side lobes, it cannot be decoded.

Our solution. To address this issue, we propose two techniques that work together to increase our dynamic range.

Coarse-grained power-aware cyclic shift assignment. Our intuition here is as follows: Fig. 8 suggests that we should assign adjacent FFT bins to devices that have a small SNR difference. In particular, when $SKIP$ is 2, for two neighboring backscatter devices with an SNR difference greater than 13 dB, the lower power device cannot be decoded. Further, it shows that the side-lobe power of a high SNR device decreases as we go to farther FFT bins. Thus, we need to ensure that a lower SNR device has to correspond to FFT bins that are farther from the FFT bins corresponding to higher SNR devices. This ensures that the side-lobes of the high-SNR device do not affect the decoding of the low-SNR devices. Specifically, we assign different cyclic shifts to different devices at association phase to ensure that the FFT bins corresponding to the lower-SNR devices are close to each other and are far from higher-SNR devices. To do this, the AP computes the signal strength of the incoming device in the association phase (see §3.3.2) and assigns its cyclic shift based on its signal strength and also the strengths of the devices that are already in the network.

We run simulations to understand the benefits of this allocation. Specifically, we assign two devices to FFT bins 2 and 258, with $SF = 9$ and $BW = 500$ kHz. To be realistic, we added Gaussian frequency mismatch with variance of 300 Hz to each device to account for timing and frequency mismatches between them. We change the power of the second device and measure the bit error rate (BER) for the first device. Fig. 12 shows the BER over 10^4 symbols, for different power differences between the two devices. As can

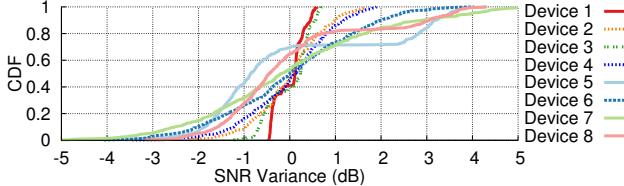


Figure 9: **Backscatter Devices SNR Variance.** CDF of SNR variance of backscatter devices in an office environment, when people were walking around, over 30 mins.

be seen, the BER remains unaffected even when the second device is around 40 dB stronger than the first device. This shows that our power-aware allocation can in theory tolerate power difference of 40 dB between devices. In practice however this is a little lower at 35 dB (see §4.3).

Fine-grained self-aware power-adjustment. While the above assignment is determined at association, mobility in the environment and fading will change the SNR of each of the devices over time (see Fig. 9). To address this, each device adjusts its power over time using the signal strength of the query message from the AP, using three different levels. We define the maximum power of the device as 0 dB power gain. First, during association, we consider two cases for the associating device. If it sees a low received signal strength for the AP’s query packet, it sets its power gain to the maximum. Otherwise, it sets its gain to the middle level. This gives the higher signal strength backscatter devices leeway to both increase and decrease their power, after association. The AP uses the resulting backscatter signal strengths during association to assign a corresponding cyclic shift. The backscatter devices use the signal strength at association as a baseline and either increase or decrease their power gains for the rest of the concurrent transmissions, i.e., if the signal strength for the AP’s query message increases (decreases), the backscatter devices decrease (increase) their power gain. If the device cannot meet its expected SNR requirements given its limited power levels and assigned cyclic shift, it does not join the concurrent transmissions. If this happens more than twice, the backscatter device re-initiates association after which the AP reassigned the cyclic shifts to account for the new significantly different power value (see §3.3.2).

The key question however is: how can a low-power backscatter device change its transmission power gain? This is interesting since power adaptation has not been used before in the network of backscatter devices. In backscatter, the transmit power gain, $Gain_{power}$, is equal to $\frac{|\Gamma_0 - \Gamma_1|^2}{4}$. Here Γ_0 and Γ_1 are reflection coefficients for switching between two impedance value, Z_0 and Z_1 . Backscatter hardware is designed to maximize the difference between reflection coefficients to maximize their transmission power. This corresponds to $Gain_{power} = 0 \text{ dB}$. One way to achieve this is to switch between extreme impedance values, $Z_0 = 0\Omega$ and $Z_1 = \infty\Omega$. To achieve power adaptation, in contrast, we pick impedance values that correspond to multiple power set-

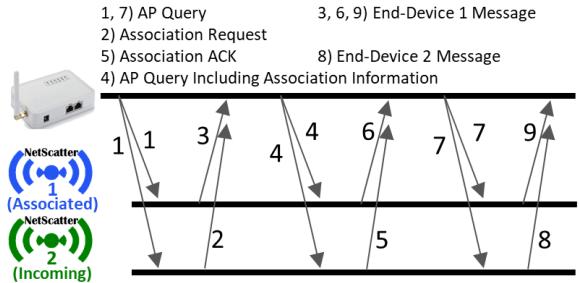


Figure 10: **NetScatter Network Association Process.** We show the association process of an incoming NetScatter device (#2) to the network, while there are existing devices associated with the network (i.e., device #1).

tings. In particular, as shown in Fig. 7a, instead of switching from $Z_0 = 0\Omega$, we switch from intermediary impedances and hence achieve lower power gains. Our hardware implementation achieves three power gains of 0 dB, -4 dB and -10 dB to achieve power adaptation. Note that [26] uses a similar circuit structure as Fig. 7b to cancel higher order harmonics. We instead design this circuit structure to control the power.

Design trade-off. Readers might wonder if reducing the power of high SNR devices would decrease the network throughput, since high SNR devices in traditional LoRa backscatter designs can achieve a higher bitrate. In contrast, by reducing their power we are enabling a large number of concurrent transmissions with a fixed bitrate. Thus, we are encouraging concurrency by reducing the bitrate of high SNR devices. §4.4 compares the results for NetScatter with one where each backscatter device uses rate adaptation to pick its ideal bitrate, while transmitting alone using LoRa backscatter [26]. The results show that the network throughput and latency gains due to large scale concurrency outweigh the reduction in the power for high SNR devices.

3.3 NetScatter Protocol & Receiver Details

Putting it together, the AP transmits an ASK modulated query message which is used to synchronize all the participating concurrent devices. This message conveys information about cyclic shift assignment which are based on the devices’ signal strength at the AP. The devices measure the query message’s signal strength using the envelope detector and use it to fine-tune their transmit power gain. In the rest of this section, we describe various protocol details required to make our design work in practice. Note that our focus in the protocol design is about scheduling a set of concurrent transmissions. Typically networks could have more devices than concurrent transmitters supported by our design. Since the AP knows the duty-cycle of each device from the association phase (see §3.3.2), it can i) assign the cyclic shifts and ii) schedule the devices involved in concurrent transmissions.

3.3.1 Link-Layer Backscatter Packet Structure

Similar to LoRa, the device packet starts with upchirp and downchirp preambles. They are designed to serve two pur-

poses: i) finding the start of the packet and ii) detecting the transmissions. We emphasize here that the device transmits the same assigned cyclic shift for both upchirps and downchirps in the preamble as well as the payload. The preamble consists of six upchirps followed by two downchirps. This is then followed by the payload and the checksum. We note that in our design, all the devices send their preambles concurrently. This reduces the overhead of transmitting preambles for each device, which in turn increase the end-to-end throughput gain achieved by NetScatter. The AP uses the above structure to achieve two goals.

i) Finding the exact packet start. We use the downchirp in the preamble to find the start of the packet transmission. Specifically, we use the middle point between an upchirp and downchirp and switch by six upchirp symbols, number of preamble symbols in our implementation, to the left to find the packet beginning. We suspect that the LoRa preamble has a downchirp for this exact purpose. We note that in our case, since the upchirp and downchirp in the preamble from each of the devices uses the same cyclic shifts, they are symmetric around the middle point and hence the same algorithm for estimating the packet beginning is applied.

ii) Detecting and decoding each concurrent transmitter. Now that we found the packet start, we need to find out which transmitters are in the network. To do so, for each preamble symbol, we demodulate it and look at the peaks in FFT domain. If there is an FFT peak in the demodulator output which repeats in all the preamble symbols, we conclude that the device corresponding to that cyclic shift is sending data. After finding current devices in the network, we compute the average power over the six preamble symbols for each device. This average power is used as a threshold to demodulate the payload of each device. In particular, if the power of the device’s FFT peak for each payload symbol is more than half this average, we interpret that as 1 and 0 otherwise.

3.3.2 Network Association

Say the network already has N devices associated to the AP and the $N + 1^{\text{th}}$ device wants to join the network. A naïve approach is to periodically dedicate time periods for association. This however can lead to high association delays depending on the frequency of the association periods. Our approach instead is to reserve N_{assoc} cyclic shifts and the corresponding FFT bins for association and use the rest for communication. In other words, all the devices transmit at the same time but the ones who want to enter the network transmit with the N_{assoc} association cyclic shifts.

To address the near-far problem, we reserve two cyclic shifts, one in high-SNR and the other one in the low-SNR cyclic shift regions. The incoming device would choose which association region to transmit based on the signal strength of the AP’s query message, calculated using the envelope detector. However to account for the hardware delay variations, as before, we skip two cyclic shifts to ensure that

Preamble [16]	Packet Length [8]	Group ID [4]	Device ID [8] (Optional)	Cyclic Shift [8] (Optional)	CRC
------------------	----------------------	-----------------	-----------------------------	--------------------------------	-----

Figure 11: Structure of AP’s Query Message.

the association packets from the devices can be decoded and will not interfere with communication cyclic shifts. Finally, to support scenarios where more than one device want to associate at the same time, one can use Aloha protocol with binary exponential back-off in the association process. Our deployment does not implement this option and turns ON the backscatter devices one at a time and runs the network only after all the devices are associated.

After the incoming device sends its packet to the AP in association process using the association cyclic shifts, the AP computes its signal strength and decides which cyclic shift and timing schedule it should be assigned to. The AP piggybacks these assignments in its query messages.

3.3.3 AP Query Message

Fig. 11 shows the ASK-modulated query message that the AP sends. The message has a group ID which identifies the set of 256 devices that should concurrently transmit. In our implementation, since there are only 256 devices, we set this group ID to 0. In a larger network, the AP can assign different sets of devices to different groups depending on their signal strengths, i.e., devices that have a similar signal strength are grouped into the same group to enable concurrent transmissions while further minimizing the near-far problem.

This is then followed by an optional association response payload that assigns an 8-bit network ID and a 8-bit cyclic shift. Note that prior LoRa backscatter designs are request-response systems that query each backscatter device sequentially and need most of the fields in Fig. 11 other than the group ID and cyclic shift assignment. Since these additional 12 bits is transmitted using 160 kbps ASK downlink, the overhead is negligible compared to the 1 kbps backscatter uplink. Finally, we note that if the AP is unable to assign a new device given the existing assignments, the AP updates the cyclic shift assignments for all the devices in the network. It does so by transmitting the identifier for one of the the 256! orderings, which requires $\log_2(256!)$ (≤ 1700) bits. This occupies less than 11 ms using our 160 kbps downlink.

3.3.4 Network Protocol

Fig. 10 summarizes our network protocol. First the AP broadcasts its query. Device 1, which is already associated to the network receives the query and sends its data using its assigned cyclic shift after performing any necessary power control. Concurrently, device 2 sends a *Association Request* using one of the N_{assoc} cyclic shifts. The AP receives these two messages and broadcast another query which includes association information for device 2. Upon receiving this query, Device 1 continues to send its data, however, device 2 extract cyclic shift assignment from the query and then transmits *Association ACK* to the AP in the assigned cyclic shift.

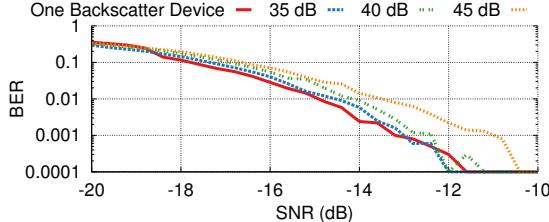


Figure 12: **Near-Far BER Results.** We show the effect of the second device’s power on the first device’s BER vs. SNR for different ratios of the second device’s to first device’s power with power aware cyclic shift assignments.

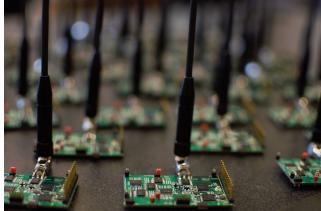


Figure 13: **Our Backscatter Devices.** They are arranged closely for this picture. They are spread out across more than ten rooms in our deployment.

If AP receives *Association ACK*, it adds device 2 to associated devices. Otherwise, it will repeat the association information in the following queries. After association, each device uses its assigned cyclic shift for sending data.

4 Evaluation

4.1 Hardware Implementation

Implementation Using Discrete Hardware Components. Our discrete hardware implementation shown in Fig. 13 consists of RF section and baseband section, both implemented on a four layers FR4 PCB. On RF receive side, we implemented envelope detector similar to [18] but at 900 MHz and it has a sensitivity of -49 dBm to receive downlink query messages from AP.¹ RF transmit side consists of five ADG904 [3] switches cascaded in three levels to build an impedance switch network for backscatter, power gain control and also switching between transmit and receive modes. Our backscatter device uses a 2 dBi whip antenna to transmit packets and receive query messages in the 900 MHz ISM band. The baseband side is implemented using an IGLOO nano AGLN250 FPGA [1] and an MSP430FR5969 [5]. We generate CSS packets on the FPGA and output real and imaginary components of the square wave signal to the backscatter switch network. The envelope detector is controlled by the MCU. Downlink receiver algorithm is implemented on MCU. To be resilient to self-interference caused by the AP’s single-tone, the baseband at the backscatter device shifts the AP’s signal by 3 MHz. Note that the discrete

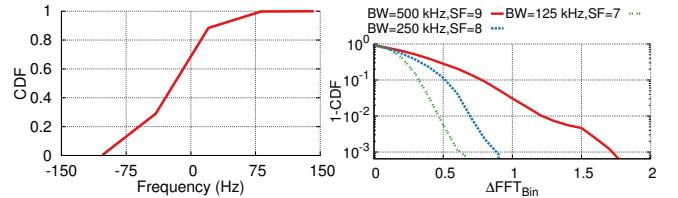


Figure 14: **Frequency Offset FFT Bin Variation.** (a) frequency offset of backscatter devices, and (b) effect of residual time and frequency offset for different configurations.

implementation is for prototyping and proof-of-concept; an ASIC is typically required to achieve the orders of magnitude power benefits of backscatter communication. We use a battery to power each backscatter device for our evaluations.

IC Simulation. We design and simulate an IC for our backscatter device using TSMC 65nm LP process. It consists of four blocks with total power consumption of $45.2 \mu W$: i) An envelope detector that demodulates the APs ASK query messages and consumes less than $1 \mu W$. ii) Baseband processor for processing and extracting AP data from envelope detector, interfacing with sensors and sending the chirp specifications and sequence of data to chirp generator consuming $5.7 \mu W$ of power. iii) A chirp generator that takes SF, BW, cyclic shift assignment and data sequence from the baseband processor to generate the sequence of ON-OFF keying chirps. We use Verilog code to describe the baseband signal’s phase behavior and generate assigned cyclic-shift with required frequency offset. We use Synthesis, Auto-Place and Route (SAPR) to simulate Verilog code on chip. The power consumption of this block is $36 \mu W$. iv) We simulate a Switch network including three resistors that are connected to NMOS switches to generate backscatter signal with three power gain levels. Note that since these resistors and NMOS switches consume minimal area, more power gain levels can be added at almost no cost. The power consumption of the switch network is $2.5 \mu W$ with 3 MHz frequency offset.

Access Point Implementation. We implement the access point on the X-300 USRP software-defined radio platform by Ettus Research [8]. We use a mono-static radar configuration with two co-located antennas separated by 3 feet. The transmit antenna is connected to a UBX-40 daughterboard, which transmits the query message and the single-tone signal. The USRP output power is set at 0 dBm and we use an RF5110 RF power amplifier [6] to amplify the transmit signal to 30 dBm. The receiver antenna is connected to another UBX-40 daughterboard, which down-converts the NetScatter packets to baseband signal and samples them at 4 Msps.

4.2 Frequency and Timing Mismatch

Measurements 1: Hardware frequency variations. We measure the frequency offsets of our hardware by recording thousand packets for each device. Using the method de-

¹Note that since ASK-modulated AP query received by backscatter device experiences one-way path loss, its required sensitivity is only -44 dBm in contrast to the -120 dBm sensitivity for the backscatter signals.

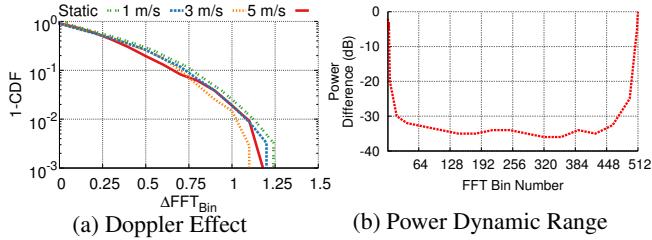


Figure 15: **Doppler Effect and Power Dynamic Range Evaluation.**

Evaluation. We evaluate (a) Doppler effect, and (b) we show power difference between two concurrent transmissions at different locations of FFT domain. One transmission is fixed and the other is sweeping across different chirp symbols.

scribed in §3.3.3, we compute the frequency offset for the 256 backscatter devices in our network deployment which we show in Fig. 14a. The variations of backscatter devices are less than 150 Hz which is nearly 0.15th of one FFT bin when $BW = 500\text{kHz}$ and $SF = 9$. Therefore, our system is not affected by frequency variation of different devices.

Measurements 2: Timing offsets. Next, we characterize how the timing offsets affect ΔFFT_{bin} . This helps us understand how many empty cyclic shifts, $SKIP - 1$, we need to put for each occupied cyclic shift. To do this, we setup a wireless experiment sending query messages from the AP and receiving transmissions from the backscatter devices deployed in our system. By decoding these transmissions and comparing the received cyclic shifts with what we have programmed the devices to send, we can find the ΔFFT_{bin} for each device; this measurement is a combination of both timing and the small frequency variations on the hardware.

Fig. 14b shows residual ΔFFT_{bin} for backscatter devices. The plots show that the ΔFFT_{bin} is considerable. This is because in backscatter devices, the energy detector receives the amplitude modulated query message and sends interrupt to initiate backscatter transmission. Both these steps add to the timing variations. Specifically, the hardware delay variation comes from variation in receiving query message and initiating the transmission on FPGA which can vary from packet to packet. In our deployment in §4.4 with backscatter devices, we use $BW=500\text{ kHz}$, $SF=9$ and leave one FFT bin between occupied cyclic shifts ($SKIP = 2$). This translates to supporting 256 devices with an aggregate throughput of around 250 kbps and bitrate per tag of around 1 kbps.

Measurements 3: Doppler effects. Other than hardware frequency offsets, Doppler effect can cause changes in frequency as well. However, this effect will be much less than 1 FFT bin, $\frac{BW}{2SF}$, for most cases. As an example, assume a backscatter device is moving with a speed of 10 m/s. Considering the carrier frequency is 900 MHz, the Doppler effect induced frequency change would be 30 Hz which is much less than 1 kHz, the FFT bin frequency, assuming $BW=500\text{ kHz}$ and $SF=9$. To confirm this, we run various mobility experiments where a subject holds a backscatter device

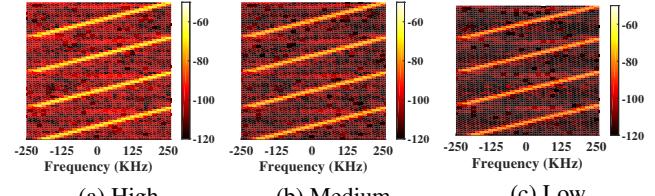


Figure 16: **Spectrogram of Backscattered Signal at the Different Power Levels.**

and moves with different average speeds which we measure using an accelerometer. We receive transmissions from the device and compute the ΔFFT_{bin} for different motion scenarios. Fig. 15a shows ΔFFT_{bin} for various speeds, which confirms that these speeds do not have an effect on ΔFFT_{bin} .

4.3 Near-Far Problem

Measurements 1: Power-aware cyclic shift assignment. As mentioned in §3.2.3, we assign cyclic shifts to devices depending on their signal strength values. To evaluate the effectiveness of this technique, we run experiments with two devices where one of them transmits at a high power (equivalent to being near the AP) with a cyclic shift corresponding to the beginning of the FFT spectrum. Then, we sweep the cyclic shift of the second device from small FFT bin difference cyclic shifts to high FFT bin difference ones. At each cyclic shift, we decrease the power of the second device using an attenuator up to when it has packet error rates less than one percent. Fig. 15b shows the maximum power difference that can be tolerated between these two devices versus the assigned FFT bin difference. As can be seen, as we go further in FFT bin difference, we can tolerate more power difference between the two devices. Note that, because of aliasing Fig. 15b is symmetric around the center. The maximum happens in middle and is equal to 35 dB. This is the dynamic range that our system can support in practice. We also note that when the second device is assigned to an FFT bin 2 cyclic shifts away from the first device, it can be up to 5 dB below the latter and still correctly decoded. This means there is an in-built 5 dB dynamic range resilience to channel variations between devices that have close cyclic shifts.

Measurements 2: Self-aware power-adjustment. The second method to address the near-far problem and also increase the dynamic-range is power adjustments at the devices using the signal strength of the AP’s query message. To evaluate this, we first measure how well we can adjust power on the devices. We evaluate its efficacy in practical deployments. We use three different backscatter impedance values to be able to transmit packets in three different power gains. Fig. 16 shows the spectrum of backscattered signal at different power levels. These plots show that the hardware creates spectrum that is clean and does not introduce noticeable non-linearities into the backscattered signal. Furthermore, we can achieve three different power levels: 0, -4, and -10 dB.

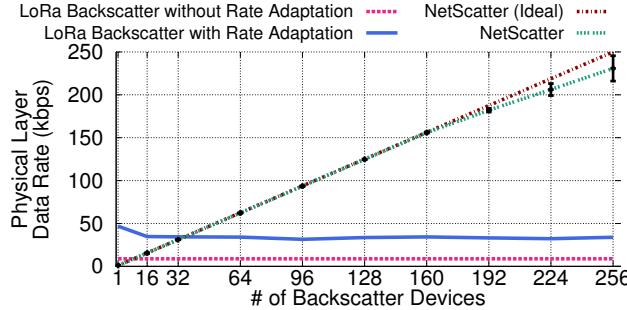


Figure 17: **Network Physical Rate.** We evaluate NetScatter network physical rate and compare it with other schemes.

4.4 Network Deployment

We evaluate three key network parameters:

- **Network PHY bitrate.** This is the bitrate achieved across all the devices during the payload part of the packet.
- **Link-layer data rate.** This is the data rate achieved in the network which is defined as the data rate for sending useful payload bits, after considering overheads including the AP’s query message and the preamble of the packet transmission.
- **Network latency.** This is the latency to get the payload bits from all the backscatter devices in the network.

We compare three schemes: i) LoRa backscatter [26] where all devices use a fixed bitrate of 8.7 kbps, ii) LoRa backscatter with rate adaptation where each device uses the best bitrate given its channel conditions and iii) NetScatter. Note that the authors of [26] did not publicly release the code and so, we replicate the implementation adding the missing details and using $BW = 500 \text{ kHz}$ and $SF = 9$. We also note that [26] is not designed to work with more than one to two users. Here, we use query-response design with scheduling when there are more users where the AP queries each device. While LoRa backscatter does not support rate adaptation, we want to compare with an ideal approach that maximizes the bitrate of each device by picking the optimal SF and BW . To do so, we measure the signal strength from each of the backscatter devices and compute the bitrate using the SNR table in [4]; this is the ideal performance a single-user LoRa backscatter design achieves with rate adaptation.

Network PHY bitrate. We set each device bitrate to 976 bps, $BW_{agg} = 500 \text{ kHz}$, $SF = 9$ and a payload size of five bytes. We deploy 256 backscatter devices across the floor of an office building with more than ten rooms. Fig. 1 shows our deployment in an office. We hard-code cyclic shift assignment on each device. Therefore, we skip the association phase in this deployment. Fig. 17 shows the results of network physical rate for our backscatter network deployment. The plot highlights the following key observations.

- The network data rate scales with the number of concurrent backscatter devices. When the number of concurrent devices is less than 128, the variance in the throughput is small.

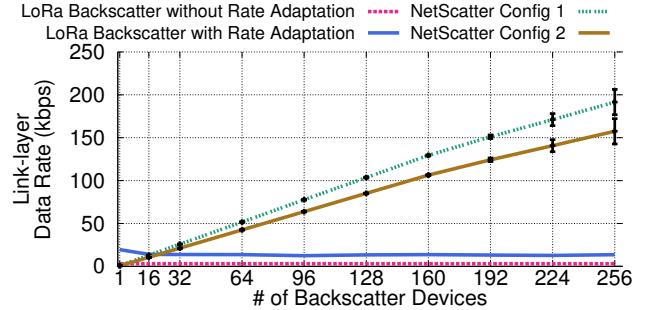


Figure 18: **Link-layer Data Rate.** We evaluate link-layer data rate for NetScatter and compare it with other schemes.

This is because in these scenarios effectively the backscatter devices are separated from each other by more than 2 cyclic shifts ($SKIP \geq 3$). As a result, the devices do not interfere with each other and hence can concurrently operate. As we increase the concurrent devices to 256, we are pushing the system to its theoretical limit (with $SKIP = 2$) and thus, we see larger variances in the network data rate.

- With 256 backscatter devices, NetScatter increases the PHY bitrate by 6.8x and 26.2x over LoRa backscatter with and without rate adaptation. The gains are lower with the ideal rate adaptation since with rate adaptation high-SNR devices could pick the maximum LoRa bitrate of 32 kbps.

Link-layer data rate. While the above plots measure the data rate improvements for the message payload, it does not account for the end-to-end overheads including pREAMbles and the AP’s query message to coordinate the concurrent transmissions. To see the effect of the AP query packet overhead for NetScatter, we consider two configurations.

- *NetScatter Config#1.* In this scenario the cyclic shifts are all assigned during the association phase and the AP query packet coordinating the concurrent transmissions is 32 bits long without the optional fields in Fig. 11.
- *NetScatter Config#2.* In this scenario, the AP query packet contain cyclic shift assignments for all the devices in the network and has a length of 1760 bits.

The above two configurations represent the two extremes of our deployment. We set the backscatter payload and CRC to 40 bits and use the total 8 upchirps and downchirps for preamble. For LoRa backscatter which queries each individual device sequentially, the AP query is 28 bits long.

Fig. 18 shows that the gains at the link-layer are higher for NetScatter over LoRa backscatter without and with rate adaptation by 61.9x (50.9x) and 14.1x (11.6x) respectively for config#1 (#2). This is because, in NetScatter, the added overhead of devices’ pREAMbles happen once and at the same time for all devices. But the other schemes need to do TDMA which means that sending preamble will not happen concurrently for all devices and these have to be sent individually for each backscatter device since in traditional designs the AP querying each of them sequentially. Further, in LoRa

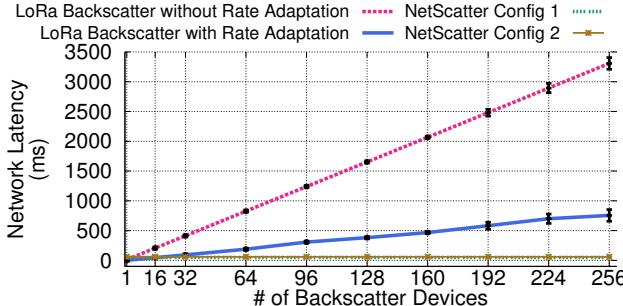


Figure 19: **Network Latency.** We evaluate the latency of NetScatter and compare it with other schemes. We define latency as total time for transmitting all the devices’ data.

backscatter which queries sequentially, the AP query message is transmitted once for each device in the network versus being transmitted once for all the devices in our design. Finally, since the downlink uses ASK at 160 kbps, the overhead of transmitting 1760 bits in config#2, while reducing the link-layer data rate over config#1, is still low because the backscatter links can only achieve a much lower bitrate.

Network latency. Finally, Fig. 19 shows that NetScatter has a latency reduction of 67.0x (55.1x) and 15.3x (12.6x) over prior LoRa backscatter without and with rate adaptation respectively in network config#1 (#2). This is the key advantage of using concurrent transmissions in low-power backscatter networks. It is noteworthy that since the downlink AP query bitrate is 160 kbps, AP query duration is negligible compared to duration of backscatter devices’ preamble for prior backscatter methods and also for config#1. For config#2, the AP query duration is significantly higher than the config#1. However, the total duration is still dominated by the backscatter payload + CRC and preamble. As a result, AP query is not the dominant factor in link-layer latency.

5 Related Work

Recent systems use backscatter with Wi-Fi signals [18, 31], have a receiver sensitivity of only -90 dBm and hence have a limited range and cannot work across rooms unless the RF source is placed close to the backscatter tag [19, 18]. LoRa backscatter [26] can achieve long ranges by generating LoRa-compliant chirp signals at the backscatter device. pLoRa [23] backscatters ambient LoRa signals in the environment in contrast to the single tone used as the RF source in NetScatter as well as [26]. We note that all SemTech LoRa chipsets have the capability in software to transmit single tone signals. All these prior long range systems are evaluated in a network of only 1–2 devices and propose to use time-division to support multiple backscatter devices. In contrast, our design enables large-scale concurrent transmissions and can achieve much higher link-layer data rates as well as lower latencies. We also note that these long range backscatter systems [26, 23] claim a kilometer range in outdoor scenarios such as open fields. This however requires

placing the RF source close to the backscatter devices. In indoor environments where the signal propagates through walls and the RF source is not placed close to the backscatter devices, our network operational range across ten different rooms is consistent with these prior work. Finally, we note that while prior work [26, 23] decodes the backscatter signal on Semtech LoRa chipsets, our distributed CSS protocol is decoded on a software radio. We however note that SemTech LoRa SX1257 [7] chipsets provide I-Q samples and hence our approach could also be implemented on these off-the-shelf chipsets together with a low power FPGA for baseband processing; this however is not in the scope of this paper.

In addition, prior work [10, 29] use FMCW techniques to multiplex sources of FMCW reflectometry. Specifically, [10] uses FMCW to multiplex Fiber Bragg Grating (FBG) sensors at different positions which results in different delays and different beat frequencies corresponding to each sensor’s reflection. In contrast, NetScatter generates chirp signals with different cyclic-shifts to modulate information on each backscatter device at the same time.

Finally, recent work on decoding concurrent transmissions from RFID tags, does not achieve the long range operations and below-noise operations of CSS based systems. Buzz [30], LF-Backscatter [13], and others [14, 22, 16] leverage the differences in the time domain signal transitions and changes in the constellation diagram to decode multiple RFIDs. However, the number of concurrent transmissions in the above designs is limited — the latest in this line of work, Fliptracer [16], can reliably decode up to five concurrent RFID tags. Further, these systems were tested with ranges of 0.5 to 6 feet [30, 13, 16] and in the same room. Finally, receiver sensitivity of even battery-powered backscatter tags for RFID EPC-GEN2 readers is around -85 dBm. So it cannot support the long ranges and whole-home deployments that CSS modulation based backscatter achieves.

6 Conclusion

We present a new wireless protocol for backscatter networks that scales to hundreds of concurrent transmissions. To this end, we introduce, distributed chirp spread spectrum coding, which uses a combination of chirp spread spectrum (CSS) modulation and ON-OFF keying. Further, we address practical issues including near-far problem and timing and frequency synchronization. Finally, we deploy our system in an indoor environment with 256 concurrent devices to demonstrate its throughput and latency performance.

7 Acknowledgments.

We thank Haitham Hassanieh, Vikram Iyer, Vamsi Talla, Justin Chan, Anran Wang, Rajalakshmi Nandakumar, and the anonymous reviewers for their helpful feedback on the paper. This work was funded in part by NSF awards CNS-1812554, CNS-1452494, CNS-1823148, Google Faculty Research Awards and a Sloan Fellowship.

References

- [1] Igloo nano fpga datasheet by, 2015. https://www.microsemi.com/document-portal/doc_download/130695-igloo-nano-low-power-flash-fpgas-datasheet.
- [2] Xrcha-f-a series, 2015. <https://www.murata.com/~media/webrenewal/products/timingdevice/crystalu/fliers/vppt-hcrj078-d.ashx?la=en-us>.
- [3] Adg904 datasheet by analog devices, 2016. <http://www.analog.com/media/en/technical-documentation/data-sheets/ADG904.pdf>.
- [4] Sx1276 datasheet by semtech, 2016. <https://www.semtech.com/uploads/documents/sx1276.pdf>.
- [5] Msp430fr5969 datasheet by ti, 2017. <http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf>.
- [6] Rf5110 rf power amplifier, 2018. http://www.rfmd.com/store/downloads/dl/file/id/30508/5110g_product_data_sheet.pdf.
- [7] Sx1257 datasheet by semtech, 2018. https://www.semtech.com/uploads/documents/DS_SX1257_V1.2.pdf.
- [8] Usrp x-300, 2018. <https://www.ettus.com/product/details/X300-KIT>.
- [9] A. Berni and W. Gregg. On the utility of chirp modulation for digital signaling. *IEEE Transactions on Communications*, 1973.
- [10] P. K. Chan, W. Jin, J. Gong, and N. Demokan. Multiplexing of fiber bragg grating sensors using a fmaw technique. *IEEE Photonics Technology Letters*, 11(11):1470–1472, 1999.
- [11] D. M. Devasivatham. Time delay spread measurements of wideband radio signals within a building. *Electronics Letters*, 20(23):950–951, 1984.
- [12] R. Eletreby, D. Zhang, S. Kumar, and O. Yağan. Empowering low-power wide area networks in urban settings. *SIGCOMM '17*.
- [13] P. Hu, P. Zhang, and D. Ganesan. Laissez-faire: Fully asymmetric backscatter communication. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15.
- [14] P. Hu, P. Zhang, and D. Ganesan. Leveraging interleaved signal edges for concurrent backscatter. In *hotWireless*, 2014.
- [15] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*.
- [16] M. Jin, Y. He, X. Meng, Y. Zheng, D. Fang, and X. Chen. Flptracer: Practical parallel decoding for backscatter communication. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17.
- [17] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*.
- [18] B. Kellogg, V. Talla, S. Gollakota, and J. R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *NSDI 16*.
- [19] M. Kotaru, P. Zhang, and S. Katti. Localizing low-power backscatter tags using commodity wifi. In *CoNext '17*.
- [20] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. *SIGCOMM '13*.
- [21] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith. Towards battery-free hd video streaming. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.
- [22] J. Ou, M. Li, and Y. Zheng. Come and be served: Parallel decoding for cots rfid tags. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 500–511. ACM, 2015.
- [23] Y. Peng, L. Shangguan, Y. Hu, Y. Qian, X. Lin, X. Chen, D. Fang, and K. Jamieson. Plora: a passive long-range data network from ambient lora transmissions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 147–160. ACM, 2018.
- [24] A. A. Saleh and R. Valenzuela. A statistical model for indoor multi-path propagation. *IEEE Journal on selected areas in communications*, 5(2):128–137, 1987.
- [25] N. Sornin and L. Champion. Signal concentrator device, Oct. 17 2017. US Patent 9,794,095.
- [26] V. Talla, M. Hessar, B. Kellogg, A. Najafi, J. R. Smith, and S. Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2017.
- [27] D. Tse and P. Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.
- [28] A. Varshney, O. Harms, C. M. Pérez-Penichet, C. Rohner, F. Hermans, and T. Voigt. Lorea: A backscatter architecture that achieves a long communication range. *Sensys'17*.
- [29] A. Vasilyev. *The optoelectronic swept-frequency laser and its applications in ranging, three-dimensional imaging, and coherent beam combining of chirped-seed amplifiers*. PhD thesis, California Institute of Technology, 2013.
- [30] J. Wang, H. Hassanieh, D. Katabi, and P. Indyk. Efficient and reliable low-power backscatter networks. In *SIGCOMM 2012*.
- [31] P. ZHANG, M. Rostami, P. Hu, and D. Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

Towards Programming the Radio Environment with Large Arrays of Inexpensive Antennas

Zhuqi Li¹, Yaxiong Xie¹, Longfei Shangguan¹, R. Ivan Zelaya²

Jeremy Gummesson³, Wenjun Hu², Kyle Jamieson¹

¹Princeton University, ²Yale University, ³UMass Amherst

Abstract

Conventional thinking treats the wireless channel as a constraint, so wireless network designs to date target endpoint designs that best utilize the channel. Examples include rate and power control at the transmitter, sophisticated receiver decoder designs, and high-performance forward error correction for the data itself. We instead explore whether it is possible to reconfigure the environment itself to facilitate wireless communication. In this work, we instrument the environment with a large array of inexpensive antenna (LAIA) elements, and design algorithms to configure LAIA elements in real time. Our system achieves a high level of programmability through rapid adjustments of an on-board phase shifter in each LAIA element. We design a channel decomposition algorithm to quickly estimate the wireless channel due to the environment alone, which leads us to a process to align the phases of the LAIA elements. Variations of our core algorithm then improve wireless channels on the fly for single- and multi-antenna links, as well as nearby networks operating on adjacent frequency bands. We implement and deploy a 36-element LAIA array in a real indoor home environment. Experiments in this setting show that, by reconfiguring the wireless environment, we can achieve a 24% TCP throughput improvement on average and a median improvement of 51.4% in Shannon capacity over baseline single-antenna links. Over baseline multi-antenna links, LAIA achieves an improvement of 12.23% to 18.95% in Shannon capacity.

1 Introduction

While the vision of the Internet of Things is rapidly becoming a reality, more and more wireless devices now crowd the wireless spectrum both at home and in enterprise. Largely unplanned wireless networks such as Wi-Fi and Zigbee have proven their utility, yet still degrade in performance when the number of radios operating in close proximity scales too far. Network designers are aware of this wireless “success disaster,” and have proposed many different approaches to

address the problem, but most—if not all—of the solutions thus far formulated treat the wireless channel as a constraint within which endpoint radios work to maximize throughput.

Prior work on improving wireless networks in isolation has followed two broad themes: (*i*) wireless modulation and channel coding schemes that aim to best utilize the wireless channel, and (*ii*) diversity schemes that aim to minimize periods of time when the wireless channel is inoperable, *i.e.*, outages. Modulation and channel coding have proved a remarkable success, achieving the information theoretic channel capacity both for static channel scenarios through fixed-rate LDPC codes [12], and for dynamic channel scenarios through rateless Spinal Codes [27]. Diversity schemes have taken on many forms in terms of time, space, and frequency diversity. Space diversity schemes generally fall into two categories. The first involves exploiting diversity among different communication endpoints to route data around failures of individual links. Schemes such as ExOR [7] leverage multiple relays, and MRD [26] and SOFT [39] improve performance through multiple APs serving mobile clients. Multiple Input, Multiple Output (MIMO) links [14, 28, 29, 34, 42] use multiple antennas to exploit spatial diversity across the multiple paths that make up a single wireless link. OFDM [35] makes efficient use of the wireless channel across different frequencies.

While the above ideas have reaped significant performance benefits, the vast majority of the design innovation has heretofore taken place at the *endpoints* of the wireless links, leaving the wireless channel itself unchanged. This paper explores a new approach: can we instead build a smarter radio environment, one that electronically reconfigures itself to the communication happening at any particular instant in time? After all, the wireless channel is the result of multipath signal propagation through the ambient environment. If we could configure signal propagation behavior at will, we could instead *create* more favorable channel conditions for wireless communication over the same spectrum. Since the number of reflectors, diffractors, and absorbers in the environment potentially dwarfs the number of antennas at the communication endpoints, more degrees of freedom may result from chang-

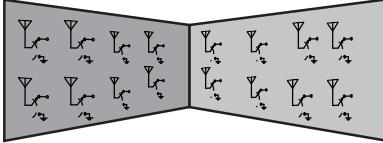


Figure 1: Building walls instrumented with LAIA, which modulate wireless signals incident on the wall, configuring the environment to improve wireless communication.

ing the environment itself rather than the communication endpoints. For this reason, our vision qualitatively differs from recent proposals to cover walls with conductive paint [43], add static reflectors [8, 16, 41, 44], or programmable phased-array reflectors [1, 2] for 60 GHz links in the environment. Furthermore, our vision raises new questions: is dynamically reconfiguring the environment even feasible—let alone in real time—in the face of constantly-changing channel conditions?

The approach we take in this paper is to augment the indoor environment with a *large array of inexpensive antennas (LAIA)*, as illustrated in Figure 1. Each array element is programmable and capable of dynamically shifting the phase of the wireless signal propagating through it. These elements form a *substrate* that can rapidly modulate the signal propagation characteristics of the *environment* itself as needed to improve communication. Since the main determinant of wireless multipath fading and interference are the phase offsets between signals arriving along different paths at each radio receiver, shifting the phases of the signals passing the LAIA array can generate desirable constructive or destructive signal superposition at the receiver. Therefore, the LAIA substrate is uniquely positioned to program these signals, and hence the radio environment.

Different LAIA configurations are desirable for different goals. For instance, if the desired outcome is to boost the performance of an individual link, LAIA will likely effect different phase shifts on the wireless signals passing through it compared to a scenario where the goal is to optimize two different links. Other possibilities include the handling of single-antenna (single-input, single-output, *i.e.*, *SISO*) links versus MIMO links. Therefore, if we can adjust the phase offsets on these paths, we can generate desirable signal alignment or separation—this process is explained further in §2.

In Section 3.1 we present the design of the LAIA element, which consists of two antennas and a programmable phase shifter. To control as many paths as possible, we deploy a large array of LAIA elements in the environment, connecting the array to a *controller*. We then tackle the central challenge of the LAIA architecture: how to design a control algorithm for the substrate, whose input consists of measurements at the endpoints and whose output consists of a time series of phase shifts (one such time series for each LAIA element) that the substrate should actuate. The design space of this control algorithm is significant, and so LAIA’s control algorithm

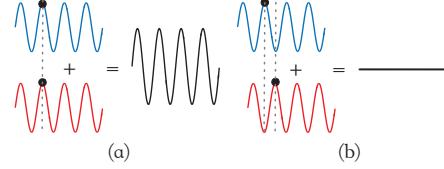


Figure 2: Signals propagating on two paths combine (a) constructively and (b) destructively based on their relative phase offsets.

focuses on a particular subset of that design space:

1. Scaling to large numbers of LAIA elements given the channel coherence time constraint;
2. Incorporating wireless channel measurements at communication endpoints, as supported by Wi-Fi standards, but piggybacking on regular data traffic;
3. Configuring the LAIA array to accomplish several different objectives simultaneously.

We describe our hardware and software implementation in Section 4. Section 5 describes a LAIA testbed comprised of 36 array elements deployed in a residential house. Extensive experiments on this 36-element testbed show LAIA can achieve a 24% TCP throughput improvement on average and a median improvement of 51.4% in Shannon capacity over the baseline single-antenna links. Over the baseline multi-antenna links, LAIA achieves an improvement of 12.23% to 18.95% in Shannon capacity.

2 Programming the radio environment

The indoor wireless channel is comprised of a collection of *propagation paths* in the environment. Radio signals radiated from the transmitter are reflected, diffracted, and scattered by multiple surfaces and objects, which cause them to traverse different propagation paths before reconvening at the receiver. The net effect of each path is captured by an attenuation in signal strength and a phase shift, and so can be modeled by $h_i = \alpha_i \cdot e^{j\phi_i}$ with amplitude α_i and the phase ϕ_i (for the i^{th} path). The L propagation paths superimpose at the receiver, constructively or destructively, based on their relative phase offsets, resulting in a wireless channel $h_{\text{env}} = \sum_{i=1}^L h_i$. Figure 2 shows two examples where two signals are completely in phase (add constructively) or out of phase (add destructively).

Consequently, if we can configure individual signal propagation paths, we can generate different channel profiles. This can be achieved by changing the phase of a path: as Figure 2 shows, when we change the phase of a propagation path, the amplitude of the combined signal at the receiver may change significantly, in the case that we turn destructive superposition into constructive superposition, for instance.

In other words, by reconfiguring the phase offset on each propagation path, we can in fact program the overall radio environment! This suggests deploying a collection of phase-

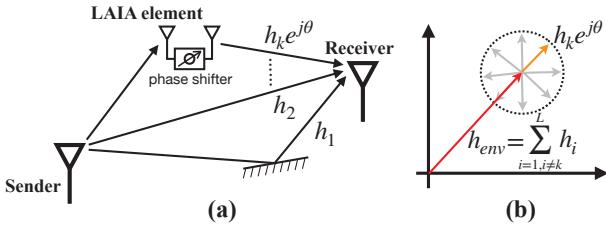


Figure 3: (a) The signal travels through the LAIA element superimposes with signals travels along all other propagation paths at the receiver. (b) By adjusting the phase shift $e^{j\theta}$, the signal from LAIA element to the receiver aligns completely with other signals from the sender to the receiver.

shifting signal relays (which we term LAIA *elements*), each with one antenna for transmission and reception, as shown in Figure 3(a). The effect of a LAIA element can be modeled by adding a phase shift of θ on the k^{th} propagation path term passing through the LAIA element between the sender and the receiver, *i.e.*,

$$h_{\text{comb}} = \left(\sum_{i=1, i \neq k}^L h_i \right) + h_k \cdot e^{j\theta} \quad (1)$$

as shown in Figure 3(b), we can adjust θ so that the signal from the LAIA element to the receiver aligns completely with the other signals from the sender to the receiver.¹

2.1 LAIA: System goals

With the ability to configure individual propagation paths, there is potential to address many link- or even network-level performance goals by adopting different signal alignment or separation strategies. LAIA takes the first step in this direction, highlighting the potential of programming the environment by addressing the following performance goals:

Goal 1: Removing the channel *null* that is associated with frequency-selective fading, for a single link, thus improving its throughput (§5.1).

Goal 2: Optimizing the throughput performance of individual single- (§5.2) and multi-antenna links (§5.3).

Goal 3: Jointly optimizing the throughput of two networks operating on different frequencies in close proximity (§5.4),

Goal 4: Substituting for and/or improving the performance of multi-AP diversity schemes (§5.2).

Discussion. Goal 1 is the least ambitious, simply requiring that different propagation paths be co-phased at a particular frequency, to fill a null at that frequency, analogous to transmit beamforming [4]. Goal 2, however, requires considering the different effects of each LAIA element on multiple

¹It is also possible to use single-antenna elements that alter the *reflected* propagation paths from the sender to the receiver, but we focus on a relay design in this work.

OFDM subcarriers (frequencies) at the same time, substantially complicating the problem. Multi-network and AP diversity scenarios (Goals 3 and 4) can be viewed as a problem of beamforming on multiple links (with different channels) simultaneously with a single beamforming matrix.

2.2 LAIA: Design challenges

A substrate design balancing cost and efficacy. As each LAIA element provides only limited programmability, we need a large array of inexpensive antennas to effectively program the whole environment. The minimalist element design of §3.1 addresses this challenge, and §4.1 gives an indication of the cost.

A scalable control plane design. The LAIA control plane orchestrates the many LAIA elements to achieve the desired effects. The first challenge here is accurate *estimation* of each LAIA element’s wireless channel h_{pi} as well as the environmental channel h_{env} between the two communication endpoints, where only measurements of the combined wireless channel are available at the receiver. The second challenge is calculating the most desirable individual element phase settings to *align* the wireless signals, given an enormous search space in the control plane. Further, both channel estimation and alignment challenges are exacerbated by the array size of LAIA and the need to complete both tasks within the channel coherence time. Our channel estimation, *flip-and-align*, and *channel alignment* algorithms (§3.2) address each of these challenges in turn.

Multi-objective control. At any given time instant, the LAIA substrate can be in at most one configuration. Therefore, when considering multiple subcarriers’, links’ and/or networks’ simultaneous operation, we require algorithms to address these joint optimization challenges. §3.3 explains how LAIA solves these.

3 Design

We first describe the individual element design for LAIA, followed by the overall system architecture including the hardware and control infrastructure (§3.1). We then delve into the control algorithm (§3.2), covering channel estimation (§3.2.1) and alignment of the entire LAIA array (§3.2.2). We conclude the section by explaining algorithms to optimize wireless channels on the fly for single- and multi-antenna links, as well as nearby networks operating on adjacent frequency bands (§3.3).

3.1 Element design and control architecture

Each LAIA element is minimalist and passive, simply consisting of two antennas connected to either end of a phase shifting device and without drawing any power, as shown in Figure 4.

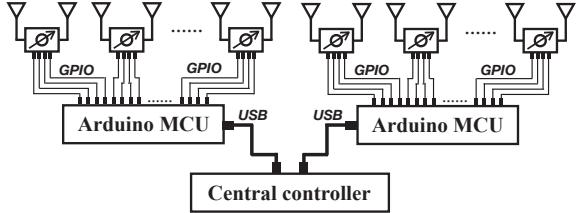


Figure 4: The architecture of the LAIA array. The LAIA element is controlled by Arduino MCU through GPIO. One Arduino is able to control up to 10 elements. A large array is controlled by multiple Arduino boards (two in the figure). All of the Arduino boards are connected to a central controller through USB 3.0 interface. The control algorithm is implemented on the central controller.

A wireless signal received by one antenna is thus shifted in phase and then emitted to the environment through the other antenna, and vice-versa for the wireless signal received by the other antenna. A purely passive radio chain significantly reduces the energy consumption and costs of each LAIA element. Each LAIA element is programmed by modulating the phase shifter, using the micro-controller (MCU) of an Arduino board as a controller to configure the phase shifter through general purpose I/O pins, as shown in Figure 4. Since each LAIA element is a passive device, it is low cost, but provides only a limited impact on the environmental channel, which we quantify in Section 5.1.

LAIA's control architecture is also shown in Figure 4. The phase shifters, which control up to 10 individual elements, are configured through the low-cost Arduino MCU (the figure gives an example of a LAIA array using two Arduino boards). Each Arduino board is connected to a central controller via a USB 3.0 interface. The central controller collects channel measurements from the AP, runs our control algorithm and distributes the control signals, *i.e.*, the amount of phase shift on each element, to the Arduino boards, which in turn configure the phase shifters of all the connected elements.

3.2 Control algorithm

We now present the control algorithm implemented on the LAIA controller. Our algorithm aims to find the configuration, *i.e.*, the phase shifts of LAIA elements in the array, that can accomplish a certain performance goal. We first introduce our channel estimation algorithm, followed by the channel alignment algorithm that aligns all the programmable element channels with the environmental channel for maximum capacity. Variations of the channel alignment algorithms optimize wireless channels on the fly for multi-antenna links, as well as nearby networks operating on adjacent frequency bands.

To program the environment, we need to first estimate the channels of all LAIA elements and the environmental channel and then align them, just as Figure 3(b) shows. The element

channels and environmental channel are linearly superimposed at the receiver. To estimate each individual channel using only the superimposed channel measurement, we can randomly configure the elements and measure the superimposed channel multiple times, using regular data frames from the on-going communication for overhead-free measurement. Individual channel measurements are uncorrelated, so we can estimate the element and environmental channel by solving the linear equations. On the other hand, channel alignment requires all elements to be configured with the optimal phase rotation so that they can add constructively. We implement a controller which resolves such a conflict and simultaneously estimate and align the channels.

3.2.1 Channel decomposition

Our first task is to separate (decompose) the ambient environment's channel h_{env} from the channel corresponding to the propagation paths traversing the i^{th} LAIA element. Suppose the i^{th} LAIA element is configured with phase setting θ_i . We refer to the channel through that LAIA element in isolation as $e^{j\theta_i} h_{p_i}$. With LAIA in mind, we can model the combined wireless channel differently than the traditional wireless channel model of Equation 1 and instead write the combined channel in terms of the ambient environment and each of M LAIA elements:

$$h_{comb} = h_{env} + \sum_{i=1}^M e^{j\theta_i} h_{p_i} \quad (2)$$

We start from a simplified algorithm to decompose the channel in the presence of only a single LAIA element before generalizing to the realistic, multi-element case.

Single LAIA element case. In the case of single element, the channel is comprised of the environment h_{env} , plus that element's contribution, h_{p_1} , supposing the element is initially configured at 0° phase. We first measure the channel with the element in the 0° state ($h_{env} + h_{p_1}$), then flip the element's phase to 180° and measure the channel again ($h_{env} - h_{p_1}$). The two resulting measured channels can be represented in matrix form as

$$\begin{bmatrix} h_{comb}^1 \\ h_{comb}^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} h_{p_1} \\ h_{env} \end{bmatrix} \quad (3)$$

and the resulting linear system solved for h_{env} and h_{p_1} . Once the environmental channel is known, we choose θ_1 such that $\angle e^{j\theta_1} h_{p_1} = \angle h_{env}$, thus phase-aligning the environment and LAIA element channels with each other.

Multiple LAIA elements. We now consider the realistic case where M LAIA elements are present. We form the channels to be decomposed into an $(M+1)$ -dimensional vector comprised of the M LAIA elements along with the environmental channel itself:

$$\mathbf{h} = [h_{p_1}, h_{p_2}, \dots, h_{p_M}, h_{env}]^\top \quad (4)$$

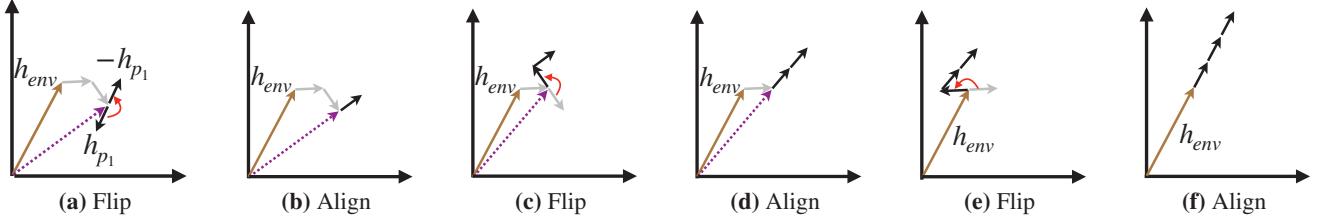


Figure 5: Channel estimation process for three LAIA elements. Without conscious alignment, the environmental channel and all LAIA elements superimpose randomly in (a). Instead, we estimate and then align the channels of elements 1, 2 and 3 in (b), (c), and (d), respectively.

Then $M+1$ channel measurements with different LAIA element phase settings can be expressed as another $M+1$ dimensional vector $\mathbf{h}_{\text{comb}} = [h_{\text{comb}}^1, h_{\text{comb}}^2, \dots, h_{\text{comb}}^{M+1}]^\top$, where

$$\mathbf{h}_{\text{comb}} = \mathbf{Q} \cdot \mathbf{h} \quad (5)$$

and \mathbf{Q} is a *control matrix* whose $M+1$ rows correspond to M different phase settings of each LAIA element (adjointed with 1 for the environmental channel which we cannot change) and whose $M+1$ columns correspond to each of $M+1$ channel measurements. We now decompose the channel, obtaining \mathbf{h} , by solving the linear system of Equation 5, and that this solution exists whenever matrix \mathbf{Q} is full rank.

We could construct a full rank control matrix by randomly configuring the LAIA elements for each of the $M+1$ measurements involved. But this would generate random contributions to the perceived \mathbf{h} , and so the intermediate combined channel \mathbf{h}_{comb} would be suboptimal. Hence we instead adopt a channel decomposition strategy such that we can achieve some alignment along with each additional channel measurement, even before we construct the entire control matrix \mathbf{Q} .

Flip-and-align algorithm. LAIA’s approach simultaneously decomposes the wireless channel and increasing the signal to noise ratio of the channel being decomposed while the measurements are taking place, thus making it more practical for networks where traffic is ongoing and present in the background of the measurement process. In fact, the channel measurements are piggybacked on ongoing traffic.

We use a three-element example to illustrate the flip-and-align algorithm. We begin with a “cold start” scenario where all element channels and the environmental channel are unknown. Since all channels are unknown, we set the phase shifter of all LAIA elements to 0° and collect the first channel measurement h_{comb}^1 . We flip the phase of the first element, just as Figure 5(a) shows and then collect the second channel measurement h_{comb}^2 . According to Equation 3, we can now calculate the channel h_{p_1} of the first element and the superimposed channel $h_{\text{env}} + h_{p_2} + h_{p_3}$, where h_{p_2} and h_{p_3} are the channels of the second and third elements respectively. Next, we rotate the channel h_{p_1} to align with the superimposed channel, as Figure 5(b) shows. We then flip the channel of the second element, as Figure 5(c) shows, and collect the third channel measurement h_{comb}^3 . Similarly, the channel h_{p_2} of

the second element and the superimposed channel $h_{\text{env}} + h_{p_3}$ is calculated using three channel measurements h_{comb}^1 , h_{comb}^2 , and h_{comb}^3 . All of the estimated channels, *i.e.*, h_{p_1} and h_{p_2} , are then aligned with the superimposed channel, as shown in Figure 5(d). We repeat until all channels are calculated and aligned, as Figures 5(e), (f) show.

The flip-and-align control matrix \mathbf{Q} is thus given by:²

$$\mathbf{Q} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ e^{j\theta_{1,1}} & -1 & 1 & 1 \\ e^{j\theta_{1,2}} & e^{j\theta_{2,1}} & -1 & 1 \end{bmatrix} \quad (6)$$

The control matrix \mathbf{Q} in Eq. 6 can be extended to include an arbitrary number of LAIA elements.³

When an ongoing round of channel measurements finishes, LAIA begins a new round, following the same flip-and-align algorithm with the only difference being the initial phase of each LAIA element. In the new round each LAIA element begins configured with its previously-computed phase rotation, to leverage any correlation in the wireless channel between the two measurement rounds that will be present if both rounds complete at time intervals close to the channel coherence time.

3.2.2 Channel alignment

Once we have decomposed the wireless channel, the LAIA *alignment* algorithm calculates the most desirable phase settings for individual elements to optimize a particular performance metric, *i.e.*, we find the solution Θ^* to the following optimization problem

$$\Theta^* = \arg \max_{\Theta} F(h_{\text{comb}}(\Theta)) \quad (7)$$

where $\Theta = [\theta_1, \theta_2, \dots, \theta_M]$ is the phase settings of all M elements, $h_{\text{comb}}(\Theta)$ ⁴ is the combined channel of applying Θ to

² $\theta_{1,1}$ is the phase rotation calculated to align the first element with the superimposed channel in the third channel measurement, as shown in Figure 5(b); $\theta_{1,2}$ and $\theta_{2,1}$ are phase rotations calculated to align the first and second elements with the superimposed channel, as shown in Figure 5(d).

³We prove that the control matrix \mathbf{Q} is full rank in the Appendix.

⁴ h_{comb} can be a scalar, a vector, or a matrix. We use the scalar notation here and explain the other scenarios later in the section.

LAIA, and $F(h_{\text{comb}}(\Theta))$ is the objective function characterizing the performance metric.

Example. Say we want to maximize the Shannon capacity of a *narrowband single-antenna* link. The combined channel $h_{\text{comb}}(\Theta)$ (a *scalar*, like h_{env}) after applying Θ is:

$$h_{\text{comb}}(\Theta) = e^{j\Theta} \cdot \mathbf{H}_p + h_{\text{env}} \quad (8)$$

where $\mathbf{H}_p = [h_{p_1}, h_{p_2}, \dots, h_{p_M}]^T$ represents all the LAIA element channels, and the objective function $F(h_{\text{comb}}(\Theta))$ (Shannon capacity in bits per second per Hertz) is

$$F(h_{\text{comb}}(\Theta)) = \log_2 (1 + |h_{\text{comb}}(\Theta)|^2 \rho) \quad (9)$$

where ρ is the signal-to-noise ratio (SNR) on the channel. Shannon capacity is maximized at the highest SNR for the combined channel, which can be achieved by aligning the phases of all LAIA element channels with the environmental channel h_{env} , as shown in Figure 5, to maximize $|h_{\text{comb}}|^2$. The i^{th} element in Θ , θ_i , is simply set to the phase difference between the channels h_{p_i} and h_{env} .

3.3 Multi-objective control

Wideband single-antenna links. Wireless links typically operate over a wide frequency band and therefore experience frequency selective fading. This means that even a single-antenna link is comprised of a set of distinct channels, corresponding to different frequency components (subcarriers). In other words, the environmental channel \mathbf{h}_{env} and the combined channel \mathbf{h}_{comb} become vectors, whose length is the number of subcarriers. We can still resort to Eq. 8, but now each element channel \mathbf{h}_{p_i} is also a vector. Since our phase shifters operate over the entire frequency band without regard for subcarriers, and so the same phase shift will be applied to all subcarriers. Clearly we cannot optimize for different channels simultaneously with one phase shift configuration. Therefore, while we can still define an objective function F over all subcarriers, for example, the total Shannon capacity (*i.e.*, summing the per-subcarrier Shannon capacity), F can no longer be maximized analytically.

Multi-antenna (MIMO) links. MIMO links, narrowband or wideband, are another form of "multi-channel" links. Therefore, following a similar approach used to handling wideband single-antenna links, we can derive the appropriate expressions for the channels (using Eq. 8) and the objective function, then again use the same optimization problem formulation (Eq. 7). For example, the situation for *narrowband MIMO* links (though not seen in practical wireless technologies) is analogous to that of *wideband single-antenna* links. The environmental channel \mathbf{h}_{env} and the combined channel \mathbf{h}_{comb} are now both *matrices*, whose dimensions are determined by the number of antennas at the wireless sender and receiver respectively. Eq. 8 is still valid, but each element channel \mathbf{h}_{p_i}

is now also a matrix of the same dimension as that of \mathbf{h}_{env} and \mathbf{h}_{comb} . The MIMO channel capacity is an example for the objective function.

Extending this further to *wideband MIMO* links used in Wi-Fi or LTE, the environmental channel \mathbf{h}_{env} , the combined channel \mathbf{h}_{comb} , and each element channel \mathbf{h}_{p_i} are all matrices of the form $N_{rx} \times N_{tx} \times N_{\text{subcarriers}}$, where N_{rx} , N_{tx} , and $N_{\text{subcarriers}}$ are the numbers of the receiver antennas, the sender antennas, and the subcarriers respectively.

Multiple links. Only single-link (SISO or MIMO) scenarios have been considered so far. Next, we extend LAIA to multi-link scenarios and discuss two representative cases: (i) a single cell with multiple clients associated with the same access point (AP) and on the same frequency; and (ii) multiple concurrent links on different frequencies.

In the first case, while the AP only communicates with one client at a time, LAIA cannot predict which client will be active and cannot configure the element array accordingly before the transmission. Therefore, LAIA configures the element array to maximize the total capacity of all AP-client links and keep the configuration until any client joins or leaves the network. In the second case, LAIA must find a configuration of the element array that works for multiple links simultaneously. This can again be reduced to maximizing the total capacity. In either case, we use the total capacity expression as the objective function in our optimization problem (Eq. 7). In addition, we adjust the objective function to consider fairness or other performance goals, *i.e.*, we maximize the aggregate capacity of all the links with the constraint that no individual link will be harmed in terms of capacity.

3.3.1 Iterative search

In above multi-channel cases, there is no analytic solution to the optimization problem in Eq. 7. Given a large array size and the number of possible per-element phase choices for LAIA, exhaustively searching for the optimal Θ that maximizes the objective function is too complex computationally. For example, a 4-bit phase shifter, used in LAIA, provides $2^4 = 16$ possible phase shifts per element and results in 16^M possible phase combinations for an M -element LAIA array! We therefore identify heuristics to prune the search space.

Basic version. We adopt a multi-round iterative algorithm to prune the search space. In each round, we sequentially align all element channels with the environmental channel. Specifically, in round one, we first align the channel h_{p_1} with h_{env} by searching through all possible phase settings of the first element that maximizes F , fixing the phases of the rest of the array. Once we find the result, θ'_1 , we then align the element channel h_{p_2} with the intermediate combined channel $e^{j\theta'_1} h_{p_1} + h_{\text{env}}$, again fixing the phases elsewhere in the array. We repeat the process for all LAIA elements until we obtain a vector $\Theta' = [\theta'_1, \theta'_2, \dots, \theta'_M]$, when this round completes.

With Θ' applied to the LAIA elements, the combined channel becomes $h_{\text{comb}}(\Theta')$: While $F(h_{\text{comb}}(\Theta'))$ usually improves over $F(h_{\text{env}})$, the improvement is not guaranteed to be optimal. Therefore, multiple rounds of the alignment may be needed. These rounds proceed in much the same way as the first round, except that in subsequent rounds of alignment, when we align LAIA element k , we leave the phase of all the other elements $\Theta^{(k)} = [\theta_1, \dots, \theta_{k-1}, \theta_{k+1}, \dots, \theta_M]$ constant so we align element k with the combined channel, $h_{\text{comb}}(\Theta^{(k)})$. We iterate until there is barely an additional improvement (*i.e.*, below a threshold) in F from a new round. The algorithm typically converges in two to three rounds in our experiments.

By tuning one element phase at a time, the iterative algorithm already reduces the search complexity dramatically from exponential to linear: $16M$ phase settings are checked instead of 16^M in each round. Strictly speaking this finds the local optimal result, but we find empirically that the local optimal is typically very close to the global optimal (Figure 11).

Prioritizing “influential” elements. Still, the execution time of our search algorithm may exceed the channel coherence time, especially when given a large LAIA array. Therefore, we further refine the basic iterative search as follows.

Recall that the number of *dominant multipath components* in a typical indoor environment is limited to a small number [10, 21]. This suggests that the number of *influential elements* in the whole LAIA array is also limited. Therefore, we can speed up the search and reap most of the benefit by prioritizing these influential elements. We assign a priority level, initialized to 0, to each LAIA element to indicate the order of search. In each round of iteration, the elements will be reconfigured to the optimal phases in the order of their priority levels. After an element is configured, its priority level is increased by its contribution to the overall channel improvement weighted by a random probability uniformly distributed between 0 and 1. The element with the highest contribution tends to get a large increment of its priority level and will likely be tuned again sooner in the next round.

The advantages of the prioritized search algorithm are two-fold. First, the priority level ensures the element that contributes most to the channel is configured first, within the channel coherence time, and thus adapting LAIA quickly to changing environmental channels, which helps to mitigate the effect of mobility. Second, the algorithm does not exclude less influential elements and can still converge to the result calculated by the basic iterative algorithm.

4 Implementation

In this section, we first introduce the LAIA element implementation (§4.1) and then describe the way to handle channel measurement errors (§4.2).

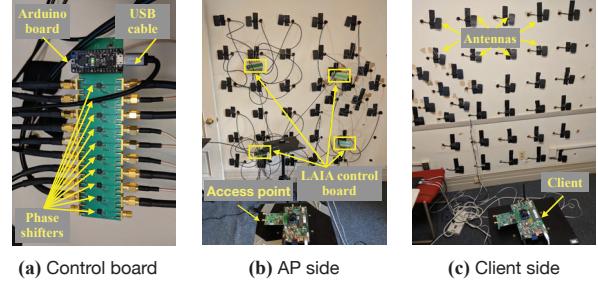


Figure 6: (a) A LAIA control board prototype, with ten phase shifters and one Arduino board. A real-world deployment of a 36-element LAIA array, one antenna of each LAIA element is on (b) the AP side of the wall and the other is on (c) the client side.

4.1 LAIA hardware

The hardware of LAIA includes two major parts: antenna and control board. The antenna we use is small panel antenna, with relatively narrow beam width (around 40 degree). Each antenna only costs about 1.75 USD. We build a prototype of LAIA control board using ten MACOM MAPS-010144 four-bit phase shifter [25] and an Arduino Adafruit Metro Mini MCU [3] on a four-layer printed circuit board (PCB), as shown in Figure 6(a). All the control boards are connected to the central controller (a laptop) via a USB 3.0 hub. The four-bit phase shifter can shift the phase of an incoming signal at a granularity of $\pi/8$.

4.2 Channel measurement error

Our control algorithm relies on a very accurate channel measurement at the receiver end, which is challenging to achieve in practice. First, *carrier frequency offset* (CFO) describes the frequency difference between the two transceivers’ oscillators. The phase offset caused by CFO changes over time, causing a random phase offset between channel measurements from different packets. The channel decomposition algorithm relies on multiple packets to decompose the channel, thus suffering from the random phase offset. We describe the detail of our CFO estimation algorithm in Appendix.

Second, *symbol timing offset* (STO) introduces a sub-sample delay in the time domain, equivalent to a phase slope in the frequency domain. As STO varies across different packets, it introduces a phase offset across packets and thus affects our channel decomposition algorithm. We leverage fractional interpolation [22, 40] to detect and remove the phase slope in channel readings, thus compensating for STO.

5 Evaluation

In this section, we evaluate LAIA’s performance, starting with microbenchmarks (§5.1) to characterize basic wireless

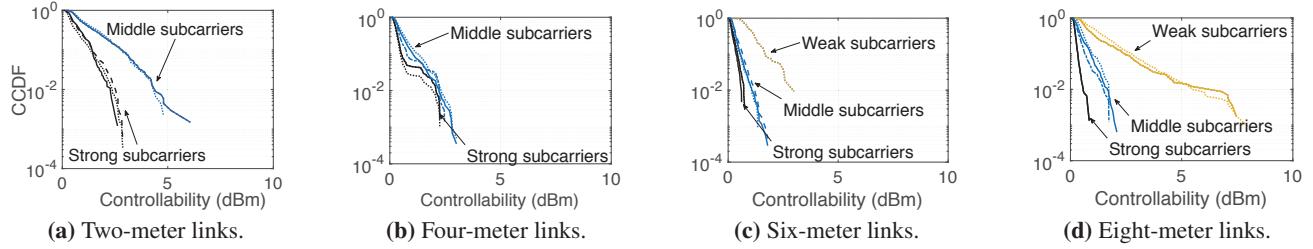


Figure 7: Distribution (across elements and subcarriers) of LAIA element controllability on strong (> -53 dBm), middle (> -65 dBm and not strong), and weak (< -65 dBm) subcarriers of three links, for various link lengths noted. LAIA controls weaker subcarriers more easily.

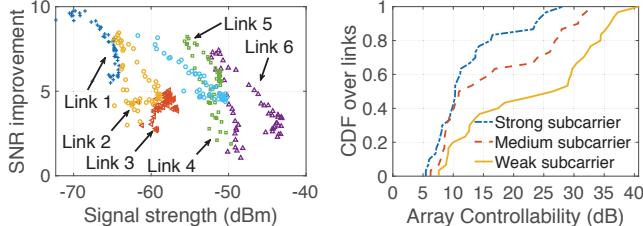


Figure 8: LAIA SNR improvement versus signal strength for six links (one point per subcarrier). **Figure 9:** Controllability distribution of the 36-element LAIA array, across 30 links.

channel controllability, accuracy of channel decomposition and the gap between the LAIA control algorithm and the optimal method. We then conduct field studies to quantify the gain of channel capacity and TCP throughput of LAIA for both SISO (§5.2) and MIMO (§5.3) links. An evaluation of multiple links using both the same and adjacent channels (§5.4) concludes our evaluation.

Experiment setup. We deploy 36 passive LAIA elements on an interior wall of a house, as shown in Figures 6(b) and (c). Each element is a passive relay with antennas attached to either side of the wall. The drywall provides a 1.5 dB attenuation for signal at 2.4GHz, which matches RADAR’s *Wall Attenuation Factor* [5] and the results of a construction material signal attenuation test [31, 38]. We use WARP v3 software-defined radios [37] as Wi-Fi senders and receivers on the 2.4 GHz Wi-Fi band with 20 MHz channel bandwidth. The sender and receiver are deployed in two different rooms with a wall in between blocking the line-of-sight (LoS) path.

5.1 Microbenchmarks

We conduct microbenchmarks to better understand the performance of each part of LAIA’s design. The experiments to evaluate the controllability of LAIA emulate link optimizations to remove (*i.e.*, “fill”) null subcarriers.

LAIA element controllability. We define the difference between the maximum and the minimum signal strength that one LAIA element can induce at a particular subcarrier as the *element controllability*. We vary the link distance from 2 m to 8 m, and measure the controllability of each individual

LAIA element, over three different links, for each link length. Due to frequency-selective fading, the environmental channel differs across subcarriers, which leads to differences in controllability. To investigate this effect, we separate subcarriers into *strong* ($[-53, +\infty]$ dBm), *middle* ($[-65, -53]$ dBm), and *weak* ($[-\infty, -65]$ dBm) strength ranges and test the controllability of each subcarrier. Figure 7 shows the distribution of controllability across different LAIA elements, with one curve per physical link and subcarrier strength combination. We see that two-meter links have no strong subcarriers, while six meter and eight-meter links have the most weakest subcarriers. We also see consistent controllability across link length after binning the data into the above subcarrier strength ranges. Overall the clear trend is for greater controllability for weak subcarriers than for medium and strong ones.

Considering absolute controllability, elements at different locations achieve from *ca.* 0 dB to 7.5 dB apiece, since their 40° directional antennas capture differing amounts of RF signal from different transmitter locations. As link distance increases, signal power received from each element decreases, resulting in generally higher controllability.

LAIA array controllability. Given the modest amount of controllability a single LAIA element provides, we investigate 36 LAIA elements working together. We configure the array to maximize link Shannon capacity. The SNR changes on each subcarrier of six representative links are shown on the y-axis of Figure 8. This scatter plot plots the original signal strength of each subcarrier on the x-axis. We see that LAIA improves the SNR for all subcarriers and generally the weaker the subcarrier, the larger the improvement.

In a separate larger-scale experiment involving 30 links, we bin the subcarriers of each individual link into the top, middle, and lowest thirds by SNR. The controllability distribution is shown for each bin (respectively, *strong*, *medium*, and *weak* in Figure 9). From the figure, we can see that LAIA provides up to 42 dB controllability. Median controllability for the strongest, medium, and weakest bins is about 10.4 dB, 11.8 dB, and 27.3 dB, respectively. In summary, multiple elements provide substantial controllability, especially when transceivers experience poor channel conditions.

Accuracy of channel decomposition. We next evaluate the accuracy of the channel decomposition algorithm. In these experiments, we randomly select a 2.4 GHz Wi-Fi channel

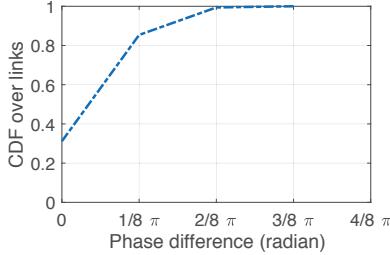


Figure 10: Predicted and measured phase difference for constructive superposition.

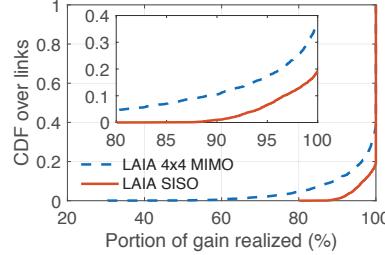


Figure 11: Percent improvement of LAIA compared with exhaustive search.

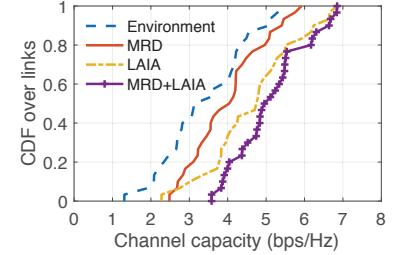


Figure 12: Channel capacity achieved by MRD [26], LAIA, and MRD+LAIA.

and estimate the CSI of an individual LAIA element channel and the environmental channel using the channel decomposition algorithm presented in §3.2.1. With the decomposed channel, we compute the phase setting for this LAIA element that can maximize the reference SNR. At the same time, we apply different phases to this LAIA element to measure the difference between the computed phase setting and the real phase setting that maximizes SNR. Since the accuracy of the computed phase setting is only determined by the accuracy of channel decomposition algorithm, we can use the deviation between the computed and real phase setting as a metric to measure channel decomposition accuracy.

We repeat the phase decomposition test 2,000 times with different LAIA elements and different transmitter/receiver locations and plot the absolute phase difference between the computed and real phase settings in Figure 10. We can see that 85.4% of the LAIA elements can be configured within an error of one phase shift step ($\pm 1/8\pi$) and 99.4% of the LAIA elements can be configured within an error of two phase shift steps ($\pm 1/4\pi$). Based on our experience, we attribute most of the error to the small phase oscillation of phase shifters.

LAIA control algorithm-optimal gap. We evaluate the gap between the LAIA control algorithm and optimal exhaustive search. We conduct extensive trace-based simulations on both SISO OFDM links and 4x4 MIMO OFDM links. We collect 5,000 traces for both cases, one trace for each link. Each trace includes the environmental channel and each LAIA element’s channel. In order to make the comparison feasible within reasonable time, we consider a subset of the search space: five LAIA elements, each having eight phases. Consequently, exhaustive search can find the answer in $8^5 = 32,768$ search combinations for each trace. We compute the Shannon capacity of the result of both the LAIA control algorithm and exhaustive search. Additionally, we use percent Shannon capacity increase $\frac{C(\text{LAIA}) - C(\text{Env})}{C(\text{Opt}) - C(\text{Env})}$ as a metric to evaluate how close the LAIA control algorithm approaches the optimal.

We observe in Figure 11 that the LAIA control algorithm provides a close approximation of the optimal. For SISO OFDM, LAIA finds the exact optimal in 80.76% of the cases. For 4x4 MIMO OFDM, LAIA finds the exact optimal in 61.6% of the cases. On average, LAIA can achieve 99.23%

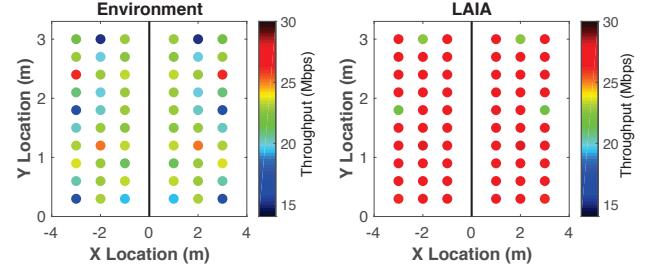


Figure 13: TCP throughput at different locations without (left) and with (right) LAIA. The location of the transmitter and receiver is symmetric to each other.

and 96.89% capacity increase compared to the optimal solution for SISO and 4x4 MIMO cases, respectively.

Element density. We evaluate the impact of LAIA element density in the array. Our 36-element array is deployed on a 2×1.8 m wall. We build a 9-element array by selecting nine adjacent elements from a 0.7×0.5 m wall area and measure the controllability of each such array. We then build another 9-element array by randomly selecting nine elements from all 36 elements. We repeat this random selection process 1,000 times. The controllability (averaged over subcarriers) achieved using nine adjacent and nine randomly selected elements (averaged over 1,000 experiments), is 2.57 dB and 2.50 dB, respectively. We see that the density of the LAIA elements does not significantly affect achieved controllability.

5.2 SISO performance

In this section, we evaluate the end-to-end performance of LAIA for single-antenna links.

Channel capacity. We first measure the channel capacity achieved by LAIA in SISO communication systems and compare it with that achieved by MRD [26], a link diversity scheme that deploys two APs and always connects the client to the AP with better channel. In these experiments, we place two receivers in one room and a transmitter in another room. The LAIA elements are deployed on the wall between these two rooms. We compare LAIA with MRD in the following

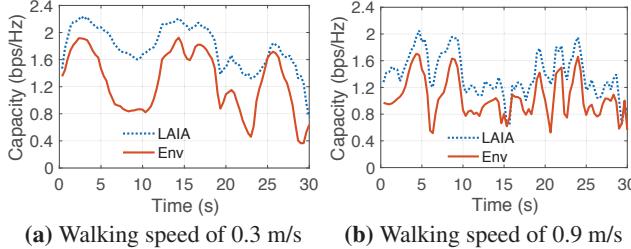


Figure 14: Snapshot of the channel capacity when walking with speed of (a) 0.3 m/s, and (b) 0.9 m/s.

way. We first connect the transmitter to the first receiver and measure the channel capacity achieved by LAIA. We next pick the better link between two transmitter-receiver links and measure the capacity achieved by the MRD algorithm. At last, we then pick the better link and measure the capacity achieved by running LAIA. As a baseline, we also measure the capacity of the environmental channel.

We repeat this experiment 30 times at different Tx/Rx locations and plot the CDF of the capacity of these four methods in Figure 12. We can see that the pure environmental channel achieves the lowest median capacity (3.13 bps/Hz), followed by the MRD algorithm (4.04 bps/Hz). In contrast, we can see that LAIA achieves a capacity of 4.74 bps/Hz, which is 51.4% higher than the pure environmental channel. This capacity further increases to 4.96 bps/Hz as we combine LAIA and the MDR algorithm.

TCP throughput of a single user. We further evaluate the TCP throughput achieved by LAIA using the WARP 802.11 reference design [36]. In these experiments, we set up a transmitter-receiver link across a wall, and measure the TCP throughput on different locations. The transmitter is placed on the location symmetric to the receiver. During the transmission, LAIA’s uses the channel alignment algorithm described in Section 3 to configure the phase of every element. The left figure of Figure 13 shows the TCP throughput of the environmental channel where we terminate all LAIA elements deployed on the wall. We can see that the TCP throughput varies significantly across locations. In contrast, from the right figure we can see that LAIA successfully improves the TCP throughput to above 26 Mbps for most locations. The average TCP throughput achieved by LAIA is 26.9 Mbps, a $1.24 \times$ improvement over the environmental channel (21.7 Mbps). Additionally, LAIA can achieve a maximum of $1.56 \times$ TCP throughput improvement.

Mobile case. The previous experiments demonstrate that LAIA can successfully improve the channel capacity and TCP throughput for static wireless links. In this section, we evaluate the performance of LAIA in a mobile environment. We place the receiver (AP) at a fixed location and move the transmitter at two constant speeds: slow (0.3 m/s) and normal (0.9 m/s). At the same time, the LAIA controller configures

the LAIA array in real time based on the CSI information from the AP. We then measure the SNR and estimate the Shannon capacity on the basis of that measurement during the transmitter’s movement. Figure 14 shows a snapshot of the real-time channel capacity. LAIA achieves consistently higher channel capacity than the pure environmental channel at both the slow and normal movement speeds. Specifically, LAIA successfully improves the average channel capacity from 1.28 bps/Hz to 1.79 bps/Hz for slow movement speed, and from 1.04 bps/Hz to 1.39 bps/Hz for normal movement speed, respectively, demonstrating that LAIA effectively improves the channel capacity in mobile environments. We notice that LAIA works slightly better when the moving speed is slower. This is because the slower movement gives LAIA more time to react to the channel change. We will further discuss how to scale LAIA to high speed in future work.

5.3 MIMO links

We next evaluate the capacity improvement achieved by LAIA for MIMO communication systems. We set up a multi-antenna transmitter-receiver system in our testbed and measure the channel capacity improvement at 20 different locations. Figure 15(a) shows the CDF of the channel capacity improvement LAIA achieves on 2×2 , 3×3 and 4×4 MIMO links. We see that the improvement increases with the number of antennas used in the MIMO system. Specifically, the median capacity improvement is 2.96 bps/Hz, 3.71 bps/Hz and 4.6 bps/Hz for 2×2 , 3×3 , 4×4 MIMO, respectively, which corresponds to 18.95%, 15.68% and 12.23% capacity improvement compared with the baseline.

Since both a received power increase and a better MIMO conditioning can lead to channel capacity improvement in MIMO systems, we plot the condition number and the total channel SNR (summation of SNR for all LAIA elements) in Figure 15(b) and Figure 15(c), respectively, to understand the root cause of the capacity improvement in our experiments. From Figure 15(b) we observe a significant condition number gap between the LAIA channel and the environmental channel, indicating that the channel conditioning makes a substantial contribution to the channel capacity gain. In contrast, Figure 15(c) shows that the difference of channel SNR between the LAIA channel and the environmental channel is small. Hence we conclude that most of the capacity improvement comes from a channel conditioning improvement.

5.4 Multiple links

We also evaluate LAIA’s performance in cases where multiple links operate in close proximity.

Multiple users. We first evaluate the performance in a Wi-Fi network consisting of five single-antenna clients and one single-antenna AP. Only one client can communicate with the AP at any given time instance. Since the LAIA controller

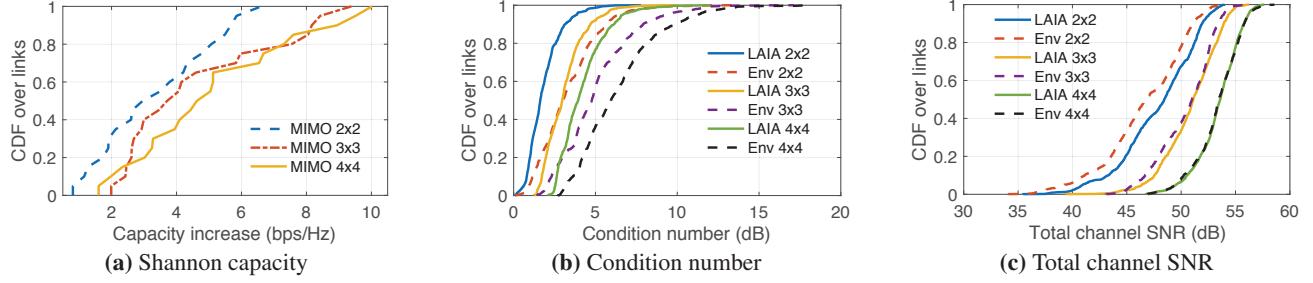


Figure 15: LAIA improvement on MIMO links (a) Shannon capacity increase achieved by LAIA, for 2×2 , 3×3 and 4×4 MIMO communications; (b) Condition number changed by LAIA, for 2×2 , 3×3 and 4×4 MIMO communications; and (c) Total channel SNR.

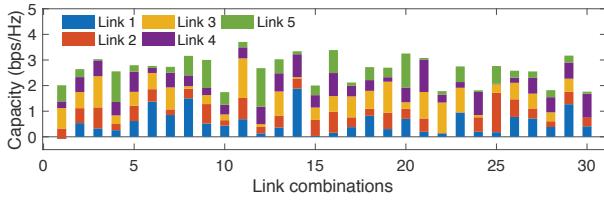


Figure 16: Shannon capacity improvement achieved by LAIA over five wireless links in 30 groups of experiments.

does not know which link is active, it configures the antenna array to maximize the total channel capacity of the five links, keeping the configuration until one user disassociates with the AP. We move the AP and all five clients to 30 different locations. At each location, we measure the CSI of all five links and calculate Shannon capacity.

Figure 16 shows the Shannon capacity improvement (with respect to the environmental channel) LAIA achieves. We observe that LAIA achieves a 3.43 bps/Hz capacity improvement on average over 30 locations. The maximum and minimum capacity improvement is 5.54 bps/Hz and 2.10 bps/Hz, respectively. We also observe diverse capacity improvements among different links, e.g., the capacity of Link 1 in Group 1 decreases, while Link 3 in Group 22 achieves a 72.9% capacity improvement. This is because our current control algorithm aims at maximizing the total capacity of all five links instead of optimizing any individual link.

We further sort the results of these 30 groups of experiments in ascending order of the capacity improvement over Link 3. In Figure 17(a), LAIA aims at maximizing the total capacity of all five links, while in Figure 17(b), LAIA aims at maximizing the capacity of Link 3 only. LAIA achieves much higher capacity improvement for a specific link when we target at maximizing that link alone. However, the capacity improvement of all the remaining links decreases significantly. Therefore, the control algorithm of LAIA should consider the fairness among all clients when serving them.

Concurrent transmissions. We next evaluate the performance of LAIA in the presence of concurrent wireless SISO links over two non-overlapping channels (channels 1 and 11 in the 2.4 GHz Wi-Fi band). LAIA’s controller optimizes the

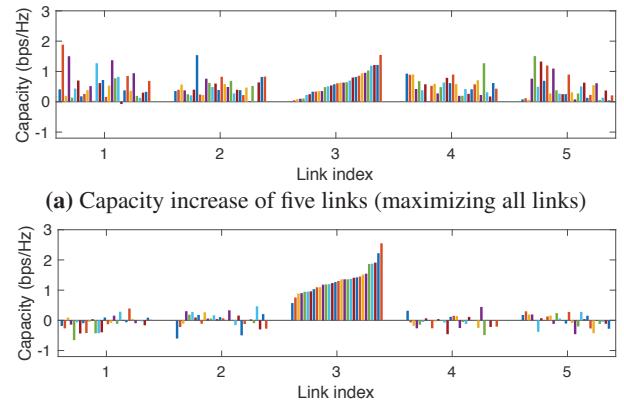


Figure 17: The sorted capacity improvement when LAIA aims at (a) maximizing the total capacity of five links; and (b) maximizing the capacity of link 3.

channel capacity of these two SISO links with the fairness constraint (denoted *LAIA*). We then measure the capacity of the environmental channel as the baseline (denoted *environment*). For comparison purposes, we also measure the channel capacity when LAIA aims to maximize the channel capacity of link one (*Max link 1*) and link two (*Max link 2*). We conduct the above experiments in 30 random locations and plot the CDF of the channel capacity in Figures 18(a) and (b). We observe that LAIA can improve the aggregate channel capacity of these two links without reducing the capacity of any link. These experimental results demonstrate the feasibility of improving the capacity of two non-overlapping links simultaneously with LAIA. When LAIA aims to maximize the channel capacity of a single link, we can see that the achievable capacity improvement over this link increases, but sometimes the capacity of the other link decreases, which is unfair to the user of the other link.

6 Related Work

Directional antennas and static Wi-Fi signal shapers. Use of directional antennas is perhaps one of the most straightfor-

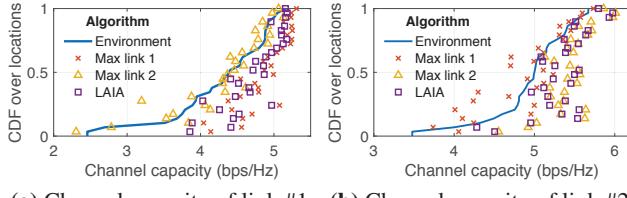


Figure 18: Channel capacity across locations when two wireless links transmit simultaneously on non-overlapping channels.

ward mechanisms to shape the wireless signals. Multiple of these antennas can be placed strategically to avoid interference between networks [24]) or generate desirable network coverage [30]. However, the granularity of the control is usually coarse, at the mercy of the beam specification and fluctuations in the environment. In a similar spirit, there have been several attempts at making the environment more amenable to wireless communications, for example, by adding 3D-printed static reflectors around the sender or receiver to shape outgoing or incoming signals [8, 41] or deploying static reflectors away from the AP [16]. However, these lack reconfigurability and cannot adapt to changing environment conditions. In contrast, LAIA aims to adapt to the environment in real time.

Millimeter-wave reflectors. Recent work uses reflectors, static mirrors [44], or programmable phased-array reflectors [1, 2] to generate alternate paths, which can circumvent obstacles blocking the direct path. While this is promising in the context of millimeter wave, the problem is qualitatively different for lower frequencies. For communications below 10 GHz, the number of reflectors, diffractors, and absorbers in the environment often creates a dense, high-dimensional channel, unlike the sparse matrix for 10 GHz and above. Optimizing this dense matrix is a qualitatively different problem than the one we undertake.

Wi-Fi extenders. Commercial Wi-Fi extenders are trending recently [11, 13, 32, 33], typically decoding packets from AP and forwarding them to clients located in longer ranges. This is again coarse-grained, however, and simply shifts the same coverage problem to a region further away from the transmitter. In contrast, LAIA controls the phase offset on different signal propagation paths to improve the SNR of the received signals and can be customized to practically any location within the antenna range. FastForward [6] uses a relay node to improve the communication channel. The fundamental difference between FastForward and LAIA is FastForward’s active versus LAIA’s passive design.

Backscatter systems. Much recent work [19, 20, 23] harnesses ambient signals in the environment as the power source for communication and computation. Another line of research [17, 18] presents active-passive hybrid designs (*i.e.*, traditional plus backscatter) for low-power radios as well as addressing power asymmetry between end-user devices.

LAIA is diametrically different since it alters communication signals instead of reusing them for another purpose. LAIA instead reconfigures signal propagation.

Massive MIMO. The advent of massive MIMO systems has ushered in a rapid growth in next generation wireless networks [14, 15, 28, 29, 42]. Massive MIMO leverages a large number of antennas to focus energy into ever smaller beams to serve multiple users simultaneously. These systems customize transmission based on the channels between the base station and the end users. In contrast, LAIA reconfigures the propagation paths of the wireless signals and can work synergistically with massive MIMO systems.

7 Discussion

Our current design merely scratches the surface of the design space. We briefly discuss limitations and future work.

Limitations. LAIA’s channel decomposition algorithm requires multiple packet exchanges to fully decompose wireless channels and hence takes time that scales with the number of elements. The channel decomposition requires highly accurate CSI, which is not provided by current commodity Wi-Fi devices, but may be in the future. Therefore, our current system only works on WARP software defined radio.

Future work. Many open questions remain and there are exciting avenues to explore. For example, what is the minimal subset of LAIA elements to control for competitive results? Further questions abound in the design of individual elements themselves: investigating the design spectrum between passive and active elements is ongoing work.

8 Conclusion

We have taken a first step towards programming the radio environment, a qualitatively different approach to the conventional strategy of optimizing communication endpoints. Our prototype implementation reconfigures the radio environment in real time, with an extensive evaluation demonstrating communications throughput enhancements that complement many other state of the art methods.

Acknowledgements

We thank the PAWS group, the reviewers and our shepherd, Xinyu Zhang for their insightful comments. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1617161, CNS-1763212, and CNS-1763309. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

- [1] ABARI, O., BHARADIA, D., DUFFIELD, A., AND KATABI, D. Cutting the Cord in Virtual Reality. In *HotNets* (2016).
- [2] ABARI, O., BHARADIA, D., DUFFIELD, A., AND KATABI, D. Enabling high-quality untethered virtual reality. In *NSDI* (2017).
- [3] Arduino Adafruit Metro Mini. [Website](#).
- [4] ARYAFAR, E., ANAND, N., SALONIDIS, T., AND KNIGHTLY, E. Design and experimental evaluation of multi-user beamforming in wireless LANs. In *MobiCom* (2010).
- [5] BAHL, P., AND PADMANABHAN, V. N. RADAR: an in-building rf-based user location and tracking system. In *IEEE INFOCOM* (2000).
- [6] BHARADIA, D., AND KATTI, S. Fastforward: Fast and constructive full duplex relays. *SIGCOMM* (2015).
- [7] BISWAS, S., AND MORRIS, R. ExOR: Opportunistic Multi-hop Routing for Wireless Networks. In *SIGCOMM* (2005).
- [8] CHAN, J., ZHENG, C., AND ZHOU, X. 3D Printing Your Wireless Coverage. In *HotWireless Workshop* (2015).
- [9] Chinese remainder theorem. https://en.wikipedia.org/wiki/Chinese_remainder_theorem.
- [10] CZINK, N., HERDIN, M., ÖZCELİK, H., AND BONEK, E. Number of multipath clusters in indoor MIMO propagation environments. *Electronics letters* 40, 23 (2004), 1498–1499.
- [11] NETGEAR N300 Wi-Fi Range Extender. <https://www.netgear.com/home/products/networking/wifi-range-extenders/WN3000RP.aspx>.
- [12] GALLAGER, R. G. *Low-Density Parity-Check Codes*. PhD thesis, MIT, 1963.
- [13] Google Wi-Fi. https://store.google.com/us/product/google_wifi.
- [14] HAMED, E., RAHUL, H., ABDELGHANY, M. A., AND KATABI, D. Real-time Distributed MIMO Systems. In *SIGCOMM* (2016).
- [15] HAMED, E., RAHUL, H., AND PARTOV, B. Chorus: truly distributed distributed-mimo. In *SIGCOMM* (2018).
- [16] HAN, S., AND SHIN, K. Enhancing wireless performance using reflectors. In *INFOCOM* (2017).
- [17] HU, P., ZHANG, P., AND GANESAN, D. Laissez-Faire: Fully asymmetric backscatter communication. In *SIGCOMM* (2015).
- [18] HU, P., ZHANG, P., ROSTAMI, M., AND GANESAN, D. Braidio: An integrated active-passive radio for mobile devices with asymmetric energy budgets. In *SIGCOMM* (2016).
- [19] IYER, V., TALLA, V., KELLOGG, B., GOLLAKOTA, S., AND SMITH, J. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *SIGCOMM* (2016).
- [20] KELLOGG, B., PARKS, A., GOLLAKOTA, S., SMITH, J. R., AND WETHERALL, D. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *SIGCOMM* (2014).
- [21] KOTARU, M., JOSHI, K., BHARADIA, D., AND KATTI, S. SpotFi: Decimeter level localization using WiFi. In *SIGCOMM* (2015).
- [22] LAAKSO, T. I., VALIMAKI, V., KARJALAINEN, M., AND LAINE, U. K. Splitting the unit delay [FIR/all pass filters design]. *IEEE Signal Processing Magazine* 13, 1 (1996), 30–60.
- [23] LIU, V., PARKS, A., TALLA, V., GOLLAKOTA, S., WETHERALL, D., AND SMITH, J. R. Ambient backscatter: wireless communication out of thin air. In *SIGCOMM* (2013).
- [24] LIU, X., SHETH, A., KAMINSKY, M., PAPAGIANNAKI, K., SESAN, S., AND STEENKISTE, P. DIRC: Increasing indoor wireless capacity using directional antennas. In *SIGCOMM* (2009).
- [25] MACOM Maps-010144 four-bits phase shifter. http://cdn.macom.com/datasheets/maps_010144.pdf.
- [26] MIU, A., BALAKRISHNAN, H., AND KOKSAL, C. E. Improving loss resilience with multi-radio diversity in wireless networks. In *MobiCom* (2005).
- [27] PERRY, J., IANNUCCI, P., FLEMING, K., BALAKRISHNAN, H., AND SHAH, D. Spinal codes. In *SIGCOMM* (2012).
- [28] RAHUL, H., KUMAR, S., AND KATABI, D. JMB: Scaling Wireless Capacity with User Demands. In *SIGCOMM* (2012).
- [29] SHEPARD, C., YU, H., ANAND, N., LI, L., MARZETTA, T., YANG, R., AND ZHONG, L. Argos: Practical many-antenna base stations. In *MobiCom* (2012).
- [30] SHETH, A., SESAN, S., AND WETHERALL, D. Geo-fencing: Confining wi-fi coverage to physical boundaries. In *PerCom* (2009).
- [31] STONE, W. C. Electromagnetic signal attenuation in construction materials. Tech. rep., National Institute of Standards and Technology, 1997.
- [32] Linksys - AC750 Boost Range Extender. <http://www.linksys.com/us/p/P-RE6300/>.

- [33] TP-Link RE450 1750 Wi-Fi Range Extender. https://www.tp-link.com/us/products/details/cat-5508_RE450.html.
- [34] TSE, D., AND VISWANATH, P. *Fundamentals of Wireless Communication*. Cambridge University Press, 2005.
- [35] VAN NEE, R. D., AND PRASAD, R. *OFDM for wireless multimedia communications*. Artech house, 2000.
- [36] Warp 802.11 throughput benchmarks. <https://warpproject.org/trac/wiki/802.11/Benchmarks/Throughput>.
- [37] Rice Univ. WARP platform (v. 3). <https://mangocomm.com/products/kits/warp-v3-kit>.
- [38] WILSON, R. Propagation losses through common building materials 2.4 ghz vs 5 ghz. *Magis Networks Inc.: San Diego, CA, USA* (2002).
- [39] WOO, G. R., KHERADPOUR, P., SHEN, D., AND KATABI, D. Beyond the bits: Cooperative packet recovery using physical layer information. In *MobiCom* (2007).
- [40] XIONG, J., SUNDARESAN, K., AND JAMIESON, K. ToneTrack: Leveraging frequency-agile radios for time-based indoor wireless localization. In *MobiCom* (2015).
- [41] XIONG, X., CHAN, J., YU, E., KUMARI, N., SANI, A. A., ZHENG, C., AND ZHOU, X. Customizing indoor wireless coverage via 3d-fabricated reflectors. In *BuildSys* (2017).
- [42] YANG, Q., LI, X., YAO, H., FANG, J., TAN, K., HU, W., ZHANG, J., AND ZHANG, Y. BigStation: Enabling Scalable Real-time Signal Processing in Large MU-MIMO systems. In *SIGCOMM* (2013).
- [43] ZHANG, Y., YANG, C. J., HUDSON, S. E., HARRISON, C., AND SAMPLE, A. Wall++: Room-scale interactive and context-aware sensing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (2018), ACM, p. 273.
- [44] ZHOU, X., ZHANG, Z., ZHU, Y., LI, Y., KUMAR, S., VAHDAT, A., ZHAO, B., AND ZHENG, H. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *SIGCOMM* (2012).

A Appendix

A.1 Probing Matrix Construction

Theorem 1. \mathbf{Q} in the form of

$$\mathbf{Q} = \begin{bmatrix} 1 & 1 & \dots & 1 & 1 \\ -1 & 1 & \dots & 1 & 1 \\ e^{j\theta_{2,0}} & -1 & \dots & 1 & 1 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ e^{j\theta_{M,0}} & e^{j\theta_{M,1}} & \dots & -1 & 1 \end{bmatrix} \quad (10)$$

is a full rank matrix regardless of the value of $\theta_{i,j}$

Proof. To prove matrix \mathbf{Q} is full rank, we try to prove the value of the determinant of \mathbf{Q} is non-zero. We first consider a transfer matrix \mathbf{Q}' ,

$$\mathbf{Q}' = \begin{bmatrix} -1 & 1 & \dots & 1 & 1 \\ e^{j\theta_{2,0}} & -1 & \dots & 1 & 1 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ e^{j\theta_{M,0}} & e^{j\theta_{M,1}} & \dots & -1 & 1 \\ 1 & 1 & \dots & 1 & 1 \end{bmatrix} \quad (11)$$

which swaps the first row of \mathbf{Q} with the last row. Notice that the absolute value of the determinant of \mathbf{Q} and \mathbf{Q}' are the same. $|\det \mathbf{Q}| = |\det \mathbf{Q}'|$. For $\det \mathbf{Q}'$, We can subtract the last row from the first $n-1$ rows, and we can get

$$\begin{aligned} \det \mathbf{Q}' &= \det \left(\begin{bmatrix} -2 & 0 & \dots & 0 & 0 \\ e^{j\theta_{2,0}} - 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ e^{j\theta_{M,0}} - 1 & e^{j\theta_{M,1}} - 1 & \dots & -2 & 0 \\ 1 & 1 & \dots & 1 & 1 \end{bmatrix} \right) \\ &= (-2)^{n-1}. \end{aligned} \quad (12)$$

So the absolute value of the determinant of \mathbf{Q} is not zero. Since $|\det \mathbf{Q}| = |\det \mathbf{Q}'| \neq 0$, \mathbf{Q} is a full rank matrix. \square

A.2 CFO Estimation

Figure 19 demonstrates why CFO can cause a random phase offset in CSI estimation. In the figure, the x-axis denotes time and y-axis denotes the phase difference between the carrier signals of sender and receiver. The slope of the green line is CFO, denoted by Δf . The blue boxes denote packets. In the figure, the time gap between the first packet and the second packet is Δt_1 . The time gap contributes to a phase difference of $\Delta\phi_1$ between the CSI of two packets. In addition, the phase difference will fold back modulo 2π . If ϕ_0 is phase difference between upconverter signal and downconverter signal at time 0, we can write the phase error of the i th packet as $\phi_i = (\phi_0 + \Delta f \cdot t_i) \bmod 2\pi$. Our goal is to estimate Δf accurately. To achieve this, we can change the formula to the

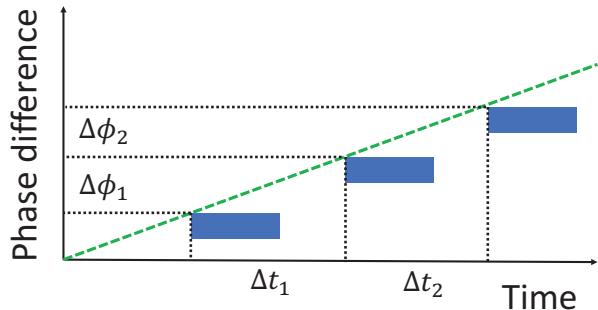


Figure 19: A example to demonstrate how LAIA estimate CFO.

form of the Chinese remainder theorem [9] $\phi_0 = (\phi_i + 2\pi \cdot n_k) \bmod t_i$, where n_k is a integer: we compute an accurate CFO and compensate for the random phase offset accordingly.

This method has a more accurate CFO estimation than the Schmidl-Cox method because it uses a much larger time window to estimate CFO. The time window that Schmidl-Cox uses to estimate CFO is typically a symbol time ($4\mu s$), while the time window that LAIA uses is typical several packet times (around 4 ms). Since phase errors due to CFO accumulate with time, a longer observation time window can provide better CFO estimation.

Pushing the Range Limits of Commercial Passive RFIDs

Jingxian Wang

Carnegie Mellon University

Junbo Zhang

Tsinghua University

Rajarshi Saha

IIT Kharagpur

Haojian Jin

Carnegie Mellon University

Swarun Kumar

Carnegie Mellon University

Abstract

This paper asks: “Can we push the prevailing range limits of commercial passive RFIDs?”. Today’s commercial passive RFIDs report ranges of 5-15 meters at best. This constrains RFIDs to be detected only at specific checkpoints in warehouses, stores and factories today, leaving them outside of communication range beyond these spaces. State-of-the-art approaches to improve the range of RFIDs develop new tag hardware that necessarily sacrifices some of the most attractive features of passive RFIDs such as their low cost, small form-factor or the absence of a battery.

We present PushID, a system that exploits collaboration between readers to enhance the range of commercial passive RFID tags, without altering the tags whatsoever. PushID uses distributed MIMO to coherently combine signals across geographically separated RFID readers at the tags. In doing so, it resolves the chicken-or-egg problem of inferring the optimal beamforming parameters to beam energy to a tag without any feedback from the tag itself, which needs this energy to respond in the first place. A prototype evaluation of PushID with 8 distributed RFID readers reveals a range of 64-meters to the closest reader and a $7.4 \times$, $1.2 \times$ and $1.6 \times$ improvement in range compared to state-of-the-art commercial readers and other two schemes [10, 33].

1 Introduction

Conventionally, passive commercial RFID tags have a maximum range of about 5-15 meters. Passive RFIDs are limited in range owing to their limited cost, form-factor, and the FCC-mandated power limits of the RFID readers they harvest energy from. Indeed, in much of today’s factories [30] and warehouses [7, 11], RFID-tagged products can only be detected around specific checkpoints in the vicinity of an RFID reader, and are virtually undetectable at other points in between [21, 31]. Further, recent innovation on RFID-based localization and sensing [24, 25, 35, 56] remain constrained to a few meters around each reader in these large spaces.

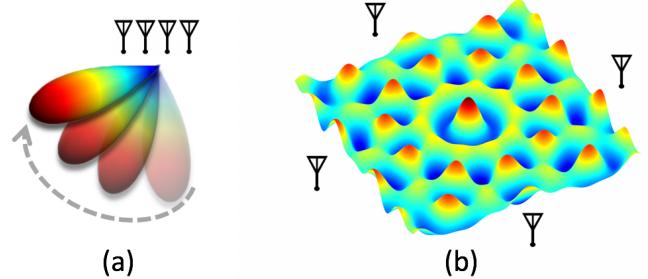


Figure 1: (a) In traditional multi-antenna beamforming, a beam forms towards a particular direction; (b) PushID’s distributed MIMO beams create multiple energy peaks and troughs over space resulting in complex energy distributions.

In this paper, we ask – “Can we push the range limits of today’s commercial passive RFID tags without increasing the prevailing density of deployment of RFID readers?”. In particular, we aim to do so without modifying the RFID tags in any way by adding to cost and complexity [27, 29, 49], relays [34] or requiring batteries [37]. We further avoid using sophisticated multi-antenna or directional RFID readers that can indeed expand range [10, 33], yet are vulnerable to obstacles and limited by FCC transmit power limits [1]. More importantly, such systems require commercial RFIDs that are linearly polarized to be carefully oriented towards their location [23] to harvest sufficient energy.

We present PushID, the first system that uses distributed MIMO to increase the communication range of commercial passive RFIDs. PushID synchronizes both transmissions and receptions from multiple, distributed RFID readers to beamform power to RFID tags that are several tens of meters away from any individual reader antenna. In addition, PushID exploits the diversity in location and polarization of reader antennas to further improve range. Our detailed experimental evaluation on an eight-distributed multi-antenna reader testbed reveals that PushID achieves a $7.4 \times$ improvement in range compared to the state-of-the-art commercial readers and 20% over the mean of distance improvement even when

compared to 8-antenna MIMO reader arrays [10], all while remaining compliant with FCC power limits for the readers.

PushID’s main goal is to find optimal beamforming weights to beam power to an RFID tag at an unknown location and orientation. Since RFID systems are back-scatter [4], optimal beamforming weights amplify *both* the transmitted and received signals to/from RFID tags. At first blush, one may consider using channel reciprocity [19], where one infers the optimal beamforming weights based on the wireless channels of signals from the RFID tag to the reader antennas. However, in the context of passive RFIDs, this leads to a chicken-or-egg problem. To emit a response, a passive RFID tag needs to harvest sufficient energy from the beamformed signal of the RFID reader antennas. Yet to perform accurate beamforming, the readers need a response from the RFID tag in the first place. Indeed, naively iterating over all possible beamforming weights would take days to simply beamform power to one tag from a long distance.

PushID resolves this dilemma by developing a novel distributed blind beamforming approach to efficiently search through the space of beamforming weights without a response from the tag. At a high level, PushID models the received signal power at each point in space for different beamforming weights applied across reader antennas. It then identifies subsets of 3-D space where RFIDs placed would receive sufficient energy to respond. Unlike traditional beams of a directional antenna, in the context of distributed MIMO, these regions of space that receive sufficient energy are quite complex and they span the entire 3-D space as shown in Fig. 1. PushID shows that finding the optimal beamforming weights while minimizing overlap is analogous to the weighted set-cover problem, a well-known NP-complete problem [5]. PushID then develops heuristic approximation algorithms to efficiently search the entire space for RFID tags under a limited time budget. A key challenge in ensuring minimal overlap between patterns is the unknown multipath characteristics of the environment which can completely change the energy patterns that beamforming weights produce (See Fig. 3). PushID’s approach to resolve this exploits responses from the RFID tags that are progressively detected as it applies various beamforming vectors. PushID uses these responses to better learn the nature and extent of multipath-richness in the environment. We show how this iteratively improves PushID’s ability to efficiently look for and power other tags in the environment.

A second challenge PushID must resolve is achieving time and frequency synchronization across multiple distributed RFID readers to beamform coherently. PushID borrows from classic distributed MIMO architectures in the Wi-Fi context [19, 40] that one can treat a transmitter as the master and apply phase shifts to the remaining slave transmitters to emulate signals from the master. Yet, a key challenge in the RFID context is that transmissions from the readers are significantly longer than Wi-Fi, causing phase drifts to accumulate significantly even within one packet from the reader. PushID

resolves this by leveraging the full-duplex nature of RFID readers. Specifically, each PushID slave transmitter subtracts its own signal and tracks the drift in phase of a carefully chosen subset of remaining transmitters. It then uses these phase drifts to account for phase errors that accumulate within a packet dynamically. We show how our system eventually converges to tightly synchronized transmissions and receptions, even if some RFID readers are not in the range of the master reader.

Limitations: We note that PushID has two important limitations common to RFID-systems: (1) First, despite significant range improvements, a small fraction of RFID tags (< 5%) are missed due to extreme shadowing or poor orientation; (2) Second, while PushID can handle modest mobility of tags (walking speeds), it struggles at higher speeds due to high dynamism in the multipath characteristics of the tags. We discuss and evaluate these limitations in Sec. 7.5- 7.6.

We implement PushID on eight USRP N210 software radio readers, each connected to separate Jackson Lab Fury clocks and commercial Alien passive RFID tags. We perform our experiments on a 140 x 140 meter outdoor space and a 20 x 40 meter indoor space in both line-of-sight and non-line-of-sight scenarios. Our experimental results reveal that:

- PushID achieves a maximum communication range of 64 m, an improvement of $7.4 \times$ that of commercial RFIDs and 20% over expensive 8-antenna MIMO.
- Even at short range, PushID achieves a mean throughput of 300 kbps at 8.5 meters ($2.6 \times$ vs. commercial RFID).
- Our system detects over 95% of the tags in a 140×140 m area, while commercial readers can detect tags no further than 8.5 m at best.

Contributions: To our knowledge, this paper presents the first distributed MIMO system to power commercial passive RFID tags. We present a novel blind distributed beamforming algorithm to efficiently search through the space of beamforming weights as well as novel phase synchronization for RFIDs. A detailed prototype evaluation on an eight-antenna distributed reader testbed reveals a $7.4 \times$ improvement in range compared to state-of-the-art commercial readers .

2 Related Work

RFID Communication and Sensing Systems: There has been much past research on RFID tags including ways to mitigate collisions [6], improve communication speed [41] and design a variety of localization and sensing solutions [24, 53]. However, all these solutions are limited to a range of at most 5-15 meters [32, 52] around the readers and thus have limited ability to locate, sense or communicate with RFID tags. Closely related to our system are recent solutions that use multi-antenna arrays connected to RFID readers to improve

range [10, 33]. While these systems improve range to commercial RFID tags (distances of up to 38 meters), our system varies in two ways: (1) First, systems with directional antennas are vulnerable to obstacles between the reader antennas and tags which can significantly attenuate the signal; (2) Second, they fail when the RFID tags are oriented poorly relative to the reader antennas. We see our system as an alternative approach using diversity of spatially distributed antennas within RFID ISM frequencies to extend the range of RFID and as complementary to [33]. We do think that both PushID and multi-antenna MIMO [42] as attractive solutions with different deployment cost/requirements. We show how by synchronizing signals across multiple RFID readers, PushID exploits the diversity in location and orientations of the reader antennas to significantly improve range compared to the state-of-the-art.

Wireless Power Transfer: Recent advances in wireless power transfer take two approaches: non-radiative near-field coupling and far-field RF radiation. Near-field coupling [22, 43] uses multiple coils to expand charging and communication range, yet is restricted to several tens of centimeters.

Far-field charging promises longer distance for wireless power transfer on the order of meters through innovative hardware design. Energy-harvesting RFID tags such as the WISP [46] promise a range of 18 meters. Past solutions [15, 16] have built wireless power transfer systems to deliver an enhanced energy in a targeted area. Recent work [48, 51] also demonstrates distances of kilometers by building custom backscatter tags with small batteries. Low-power WAN (LP-WAN) technologies [13, 14] including LoRa [47], SIGFOX [59] have explored battery-powered tags that can communicate over miles. In contrast, PushID strives to improve the range of current battery-free commercial RFID tags that cost a few cents by innovating at the RFID readers.

Blind Beamforming: In the RFID context, PushID needs to perform beamforming to tags whose wireless channels are unknown *a priori*. PushID builds upon blind beamforming [9], which is a set of theoretical beamforming solutions developed by the signal processing community in the presence of poor quality or even no channel state information available from the clients. Past work on blind beamforming with weak channel responses leverages its statistical features, for example, the cyclostationary property [45], spectral self-coherence [2]. In contrast, PushID does not have any channel information to leverage, given that tags are not *a priori* powered up. Other work on blind beamforming with zero channel feedback builds solutions without the need for carrier frequency synchronization [8, 33, 44]. PushID builds upon this past work but overcomes unique system-level challenges pertaining to the RFID context: (1) It accounts for feedback from neighboring RFID tags that are charged during the exploration of beamforming weights to refine the search in multipath-rich settings; (2) It deals with various synchronization challenges in distributed

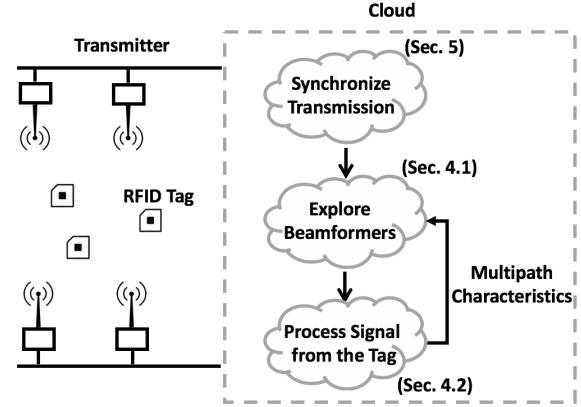


Figure 2: Architecture of PushID

beamforming with long RFID packets.

3 Overview

PushID aims to power and communicate with commercial passive RFID tags via RF-backscatter from a team of distributed commercial RFID readers, where tags are beyond the communication range of any single reader. PushID achieves this by coherently combining signals across distributed RFID readers to maximize received signal power at one or more RFID tags, whose location and orientation are *a priori* unknown. We note that since RFID systems are back-scatter, PushID applies beamforming weights *both* on the transmitted and received signals to maximize energy to/from RFID tags. We assume all PushID RFID readers are connected to a wired backhaul, which allows them to coordinate transmissions and data that needs to be transmitted on the downlink. We further assume that while the locations of the RFID readers are known, the number of RFID tags, their locations and their environment are unknown.

At a high level, PushID’s system design is as follows (see Fig. 2): All RFID readers time and phase synchronize their transmissions on the air and iteratively apply various beamforming vectors in the hope of receiving responses from RFID tags in the environment. The readers then collect responses from various RFID tags in the environment and use the wireless channels they perceive in improving the search for other tags. The readers continue this process until they believe (with sufficient confidence) that they have covered the entire desired coverage area. To achieve this, PushID optimizes the following (related) properties: (1) Maximize the total number of RFIDs found within the coverage area under an overall time budget (which limits the number of beamforming vectors that can be iteratively attempted); (2) Maximize the throughput of signals from each tag.

The rest of the paper addresses the key challenges in designing the two main aspects of PushID’s architecture:

(1) Searching through the Beamforming Space: First, our system needs to identify optimal beamforming weights to

iteratively search over in order to power RFID tags in the entire space. While this problem can be trivially addressed using channel reciprocity for RFID tags that are in range, the key challenge is that tags are outside the coverage area of any single reader. Therefore, PushID must identify the smallest set of beamforming weights that can provide sufficient energy to all tags over the entire area of interest. The key to this is to effectively model multipath in the environment, which would change the set of beamforming weights to search over. Sec. 4 describes our approach.

(2) Synchronizing Distributed RFID Readers: Second, PushID should efficiently synchronize RFID readers that are spatially distributed indoors, without a shared clock between them. In contrast to past work in the Wi-Fi domain [40], our key challenge comes from the longer duration of RFID transmissions, over which packets can quickly lose synchronization in phase. Further, RFID transmissions, unlike Wi-Fi, are narrowband, which makes time synchronization challenging as well. Sec. 5 addresses these challenges.

4 Blind Distributed Beamforming

This section describes how PushID enables a team of RFID readers with an arbitrary geometry to find the optimal beamforming weights and beam power to all RFID tags in their coverage area, including those beyond the range of any single reader. We aim to achieve this without any response from these RFID tags to begin with or prior knowledge of their locations and orientations. For ease of exposition, this section assumes that all RFID readers experience no time, carrier frequency and phase offsets. We will explicitly deal with synchronizing distributed RFID readers in Sec. 5.

We make a few key observations about the scope and goals of PushID’s approach:

- **Beamforming on both Downlink and Uplink:** We note that PushID seeks to amplify the received signal power from RFID readers to the tags and vice-versa. Specifically, since RFIDs operate on RF-backscatter, and owing to channel reciprocity [28], beamforming weights used on the transmit chain to power tags on the downlink can also be used to amplify their received signals on the uplink. For simplicity, the rest of this section discusses PushID in the context of maximizing energy on the downlink.
- **How much energy to beamform?:** We emphasize that the goal of PushID in this section is to simply beamform enough energy to detect an RFID tag. Once the RFID tag’s response is received, we can then use the reciprocal channel to obtain the optimal beamforming vector to maximize data rate to that tag in future transmissions. As a result, the rest of this section will favor PushID formulations that maximize the energy RFIDs require over the entire space, as opposed to focusing on individual tags.

4.1 Exploring the Beamforming Space

A naïve approach to perform blind beamforming is an exhaustive search of the space of beamforming vectors in hope of eliciting a response. Prior work in the context of cellular multi-antenna arrays [36] constructs codebooks that progressively steer the beam along various discrete directions in hope of covering an entire cell efficiently. For example, consider a phased array of antennas (see Fig. 1(a)) where a transmitter could simply beam power iteratively along discrete angles to cover the space of interest.

However, the distributed RFID context makes such an elegant design challenging. First, given that RFID readers are widely separated and they form an arbitrary geometry relative to each other, beamforming weights distribute energy over the space in very complex patterns. Further, it is challenging to find weights that both cover the entire space of interest with sufficient energy and minimize overlap.

What do beamforming energy patterns look like?: To better understand how different beamforming weights from a distributed array of antennas impact the distribution of energy over the space, we perform a simple simulation. We consider four transmitters in the corners of a square with a one meter diagonal length. For simplicity, we consider that the transmitters are in 2-D free space and use standard wireless channel models [50]. We first apply a beamforming weight that allows signals from the transmitters to add up coherently at the center of the square. We plot the distribution of energy over the entire 2-D space around the square encompassing the transmitters in Fig. 3 (a). We notice that while the center of the square indeed receives maximum energy (denoted by bright yellow), there are multiple spots around the center that are also energized with a similar received signal strength. This shows that applying beamforming weights in a distributed array also focuses energy on unintended points in the space. This means that simply iterating beamforming weights to focus on individual points in the space would lead to much unwanted overlap and be grossly inefficient. We therefore conclude that:

OBSERVATION 1: PushID must seek to minimize overlap between energy patterns of beamforming weights it applies.

Problem Formulation and Optimization: Based on the above observation, we will now formulate PushID’s core optimization problem that seeks to find the smallest group of beamforming vectors which energizes the entire space of interest with minimum overlap between them. At least to begin with, our system cannot rely on any feedback from RFID tags in the environment, given that none of them may have sufficient energy to respond. As a consequence, we have no prior information on the nature and extent of multipath in the environment. The rest of this section therefore assumes that line-of-sight paths to RFID readers dominate all other paths, and we explicitly account for multipath in Sec. 4.2.

At a high level, our approach shows that choosing the opti-

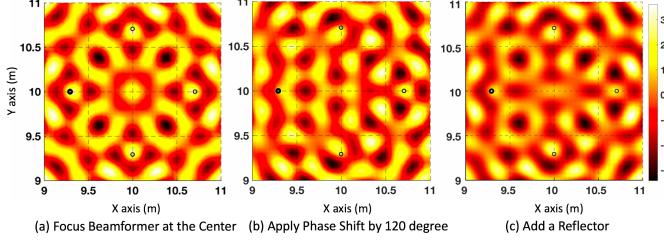


Figure 3: (a) Energy pattern when four transmitters focus beamforming at the center. (b) Energy pattern when we apply a phase shift ($\frac{2}{3}\pi$) to one of the transmitters. (c) Energy pattern when a strong reflector is placed along y-axis.

mal set of beamforming vectors is analogous to a well known combinatorial problem: the weighted set coverage problem. To see how, let us imagine that the 3-D space is divided into a grid of discrete blocks. Each beamforming vector effectively supplies sufficient energy to RFIDs in some subset of these blocks. Our goal is to find the smallest set of such beamforming vectors whose union is the universal set of all blocks in the grid. This is analogous to the weighted set cover problem, which seeks to find the smallest number of sets, each containing a few integers in the range $1, \dots, N$ whose union is precisely the universal set $\{1, 2, \dots, N\}$. Given that weighted set cover is NP-complete [55], we propose an efficient approximation algorithm, while presenting various optimization to reduce algorithmic complexity.

Mathematically, let us assume that the space of beamforming vectors has n discrete elements $\mathbf{B} = \{B_1, \dots, B_n\}$, and we aim to cover m discrete points in the space spanning the desired coverage area denoted by $\mathbf{G} = \{G_1, \dots, G_m\}$. Let the variable u_{ij} , $i = 1, \dots, n$ and $j = 1, \dots, m$ be one if the beamforming vector B_i provides energy to the point in space G_j . Given that we assume the L base station locations are known and no information on multipath is available (we discuss multipath in Sec. 4.2), we can determine u_{ij} as follows:

$$u_{ij} = \begin{cases} 1, & |B_i \cdot h_j|^2 > \tau \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$\text{where } h_j = \left[\frac{1}{d_{lj}} e^{-2\pi\sqrt{-1}\frac{d_{lj}}{\lambda}}, l = 1, \dots, L \right] \quad (2)$$

Where h_j is the vector of wireless channels from base stations to grid point j , λ is the wavelength, d_{lj} is the distance between the l^{th} base station and j^{th} grid point and τ is the minimum received energy required to energize an RFID tag.

Our objective is to find the smallest set of beamforming vectors that spans all m points in \mathbf{G} . We can state this mathematically as the following integer linear program based on the variable x_i which is 1 if and only if the i^{th} beamforming

vector is included in our optimal set:

$$\min \sum_{i=1}^n x_i$$

$$\text{s.t. } \sum_{i=1}^n x_i u_{ij} \geq 1 \quad \forall j \in \{1, \dots, m\} \quad (3)$$

$$x_i \in \{0, 1\} \quad \forall i \in \{1, \dots, n\} \quad (4)$$

The above formulation directly resembles the well-known weighted set cover problem [55], which processes a group of sets to find the smallest sub-collection whose union is also the union of the original group of sets. While this problem is known to be NP-complete [26], a reasonable polynomial-time approximation algorithm is to relax the above integer-program formulation into a linear program (LP). Specifically, we replace Eqn. 4 above by the following:

$$0 \leq x_i \leq 1 \quad \forall i \in \{1, \dots, n\} \quad (5)$$

We solve the optimization problem using standard linear programming [3] to obtain the optimal set $\{x_1^*, \dots, x_n^*\}$. We then output the chosen set of beamforming weights by applying randomized rounding [38] on the beamforming weights. This technique interprets the fractional part of the solution to the linear program as a probability distribution and then selects a solution by sampling this distribution. Randomized rounding is known to return a set of beamforming weights that is a valid set cover with probability at least $1/2$ [38]. Mathematically, to bound the probability, let ρ be a constant that satisfies: $e^{-\rho \log n} \leq \frac{1}{4n}$. Then randomized rounding proceeds for exactly $\rho \log n$ iterations, and in each iteration, it picks i^{th} beamforming weight with probability dictated by its respective solution x_i to the linear program.

Prior work has shown that the above approximation algorithm results in a set of beamforming vectors whose size is within a factor $O(\log n)$ of the optimum [17]. Our implementation uses the Ellipsoid LP-solver [57] with a worst-case complexity of $O(n^4)$, where n is the number of discrete beamforming vectors PushID's algorithm optimizes over.

Reducing Complexity and Search Space: To reduce the complexity of PushID, one must actively seek to reduce n , the number of beamforming weights that PushID considers in its optimization. Our key insight to this end is that while a large number of beamforming vectors are available, not all are created equally. To see why, let us revisit our example of the energy pattern from a beamforming vector that focuses energy at the center of a square in Fig. 3(a). We now slightly perturb the beamforming vector of one of the transmitters by choosing one of the transmitters and adding $\frac{2}{3}\pi$ to its phase, and we plot the updated energy pattern in Fig. 3(b). We make two observations about the resulting energy pattern. First, each local maximum of energy moves in different, complex ways. This is precisely why we need the optimization algorithm above to minimize overlap. Second, the size of each energized

region changes with maximum diffusing energy over wider spots. In practical terms, this means that the same amount of energy is spread out over a wider area than the previous case. Spreading energy over a wider space is good in that RFID tags over a wider region can be covered by a single beamforming weight. Yet, spreading energy over too thin over a wide area is likely to make the energy per unit area insufficient to activate tags. Indeed, the most ideal beamforming weights are those who diffuse energy in a Goldilocks zone between these two extremes. We observe the following:

OBSERVATION 2: PushID must favor beamforming weights with maximal total area where RFID tags remain powered.

PushID therefore aims to search over beamforming weights that meet the above criterion of maximizing area-of-coverage for RFID tags. Our approach begins with n beamforming weights chosen randomly, where n is dictated by available computing power. For each beamforming weight, we make incremental phase shifts and measure the gradient of net increase in coverage area. We then apply a gradient-based optimization that favors phase shifts which maximize coverage area. We implement Adadelta [58] to speed up the learning rate. The below algorithm summarizes our approach.

Algorithm 1 Gradient-Based Beamforming Vector Pruning

```

1:  $\mathbf{B}$  : random beamforming vectors.  $t = 1, \dots, N$ .  $D_i$ : energy
   of the set of points which could be covered by the  $i$ -th
   beamforming vector  $B_i$ .
Loop:
2:  $Q \leftarrow |\bigcup_i^n D_i|$ , where  $Q$  represents the number of points
   covered in  $\mathbf{G}$  by the beamforming vector set  $\mathbf{B}$ 
3:  $g^{(t)} \leftarrow \nabla_{\mathbf{B}}(Q)$ 
4:  $E[g^2]^{(t)} \leftarrow \gamma E[g^2]^{(t-1)} + (1 - \gamma)g^{(t)}$ 
5:  $\Delta\mathbf{B}^{(t)} \leftarrow -\frac{\text{RMS}[\Delta\mathbf{B}]^{(t-1)}}{\text{RMS}[g]^{(t)}} g^{(t)}$ 
6:  $\mathbf{B} \leftarrow \mathbf{B} + \Delta\mathbf{B}$ 
while  $Q < \text{threshold}$ 
return  $\arg \max_{\mathbf{B}} Q$ 
```

We then feed the above n set of beamforming weights into our optimization algorithm to find the optimal set.

Design Decisions: We emphasize a few key design choices:

(1) **HOW FINELY TO DIVIDE SPACE?:** First, PushID must choose discrete points \mathbf{G} in the space to capture the area covered by a beamforming vector. Choosing too few would lead to coverage holes, while choosing too many would waste computation. PushID therefore samples the space at an interval empirically measured to be below the minimum distance between two adjacent energized regions across beamforming vectors in \mathbf{B} . We empirically find that this corresponds to a sampling distance of $\lambda/3$ in our experiments.

(2) **PICKING THE ENERGY THRESHOLD:** PushID chooses the threshold τ empirically by measuring the smallest amount of energy needed for an RFID tag to respond at its smallest

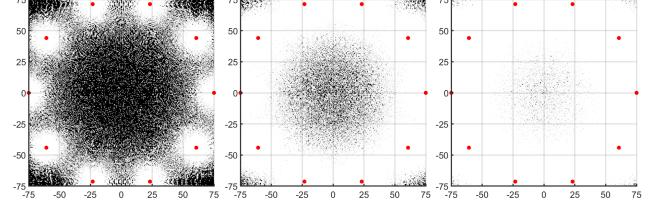


Figure 4: We simulate 10 transmitters deployed on a circle with a radius of 75 meters. Red points are the location of omni-directional transmitters. White points indicate the area that can activate the RFID tags (signal strength of energized area $\leq -12.8\text{dBm}$), black points are the opposite. From left to right, the plots represent energized pattern with 1, 330 and 450 beamforming vectors. The corresponding energized area is 34%, 80%, 97% of the enclosed area of transmitters.

data rate. Once the RFID tag is detected, future transmissions can use the reciprocal channel measurements from this tag to speed up data rates [50]. We note that τ must be calibrated conservatively to support all RFID tag models in the space.

(3) **IMPACT OF ORIENTATION:** PushID explicitly accounts for the reader’s antenna gain and polarization across spatial directions by applying a weight to each term of Eqn. 2: $\alpha_{l,j}$, which captures the attenuation in the l^{th} base station antenna when it faces the j^{th} grid point. We also account for the orientation of the tag by setting τ conservatively to the smallest amount of energy for a tag to respond should it be oriented most unfavorably relative to the readers.

(4) **RUN TIME:** We find that PushID’s run-time is primarily bottlenecked by the slow beamforming switch time of transmitters (4.5 ms for our hardware) as opposed to computation. PushID’s run time depends directly on the final number of beamforming vectors PushID must iterate over. This depends on the size of the space, placement of base stations and multipath. To get a sense for expected run time, we simulate ten RFID readers in a circle of radius 75 m (Fig. 4). We consider a threshold of RSSI $>-12.8\text{ dBm}$ for the RFID tags to respond. Without PushID, we find that only 33% of the total area of interest is covered. However, after only 450 iterations we find that nearly 97% of the area is covered by PushID. This maps to a total time of 2.0 s for 97% coverage. Sec. 7.6 discusses these tradeoffs in experiments (we observe 4 s run time due to multipath and change in layout).

(5) **ADAPTING TO NEW INFORMATION:** Our approach so far arrives at a static set of beamforming weights. However PushID can benefit from the new information in the channel response of RFID tags, as they are detected. In particular, we are interested in learning about the extent and nature of multipath, which can impact the optimal set of beamforming vectors. Sec. 4.2 below deals with this explicitly.

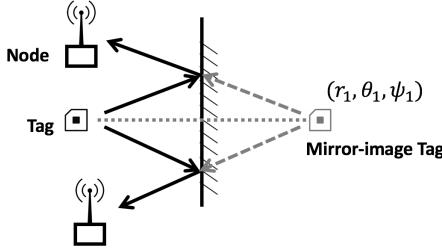


Figure 5: A reflected path can be modeled by a virtual source (r_1, θ_1, ψ_1) – the mirror image of the tag about the reflector.

4.2 Accounting for Multipath

While so far our discussion considers free space, the presence of multipath can considerably change the set of beamforming weights to efficiently search over a given area. To see why, we revisit our example in Fig. 3(a), and this time we add a strong reflector and re-evaluate the energized regions of space as shown in Fig. 3(c). We notice that the resulting energy heatmap varies considerably from the free-space heatmap, both in the number, size and placement of the hotspots. Note that the same set of reflectors can influence the energy perceived at different tags differently. As a result, we conclude:

OBSERVATION 3: PushID must account for multipath given its impact on the optimal set of beamforming vectors.

PushID’s high-level approach to do so uses the responses from RFID tags in the environment that are progressively detected. Indeed, in the absence of any response, PushID has no information about multipath to work with, and therefore assumes a free-space channel. As responses from RFID tags are collected, PushID progressively computes information of the location and orientation of dominant reflecting surfaces in the environment. It then uses this information to update its optimization algorithm, specifically, it modifies the energy patterns corresponding to our beamforming weights.

Finding Dominant Reflectors: To compute the location of dominant reflectors, PushID adapts the MUSIC algorithm while accounting for the arbitrary geometry of the RFID readers. Specifically, the algorithm takes as input wireless channels along the various frequencies of operation of an RFID tag (note that RFID tags naturally hop through a range of frequencies in the 900 MHz ISM band). It then measures the polar coordinates: (r, θ, ϕ) representing the mirror-image of the RFID tag along dominant reflectors by measuring the power of the received signal $P(r, \theta, \phi)$ of signals received from these coordinates. Mathematically, we write:

$$P(r, \theta, \phi) = \frac{1}{|a(r, \theta, \phi)^\dagger E_n E_n^\dagger a(r, \theta, \phi)|} \quad (6)$$

where: $a(r, \theta, \phi) = [e^{4\pi j|r-r_i| \cos(\theta-\alpha_i) \cos(\phi-\beta_i)/\lambda}]_{i=1,\dots,N}$

Where (r_i, α_i, β_i) are the polar coordinates of the transmitters, λ is the signal wavelength, j is the square root of -1 ,

E_n are the noise eigenvectors of $h_{obs} h_{obs}^\dagger$, h_{obs} represents the vector of observed wireless channels of the tags and $(.)^\dagger$ is the conjugate-transpose operator. Our algorithm computes the top- s ($s = 5$ in our implementation) local maxima in $P(r, \theta, \phi)$ to define the set of polar reflector coordinates: $\{(r_k, \theta_k, \phi_k), k = 1, \dots, s\}$.

Folding Multipath into the Optimization: At this point, we model how the energy patterns of beamforming weights change due to our knowledge of reflecting surfaces. We use a ray-tracing model [50] to account for how multipath changes received signal power. Mathematically, we rewrite Eqn. 2 in the definition of u_{ij} as:

$$h_j = \left[\sum_{k=1}^s 1/d_{ljk} e^{-2\pi\sqrt{-1}d_{ljk}/\lambda}, l = 1, \dots, L \right] \quad (7)$$

Where λ denotes the wavelength and d_{ljk} represents the distance traversed by the ray emanating from base station l to grid point j when reflecting off reflector at coordinates (r_k, θ_k, ϕ_k) . This formulation effectively removes the free-space assumption in our optimization to explicitly account for ambient reflectors.

Accounting for past vectors: We note that as new information about multipath emerges, one must account for how this impacts the coverage area of beamforming vectors used previously and therefore invoke the optimization to fill gaps in coverage. Mathematically, let us denote x_i^* as an indicator function on which beamforming weights were used previously. Then we can rewrite Eqn. 3 as:

$$\sum_{i=1}^n x_i u_{ij} \geq 1 - \sum_{i=1}^n x_i^* u_{ij} \quad \forall j \in \{1, \dots, m\} \quad (8)$$

Modeling fleeting and small reflectors: While the above formulation assumes reflectors impact all RFIDs in the coverage area equally, in practice, this may not be the case. Specifically, reflectors have a higher probability of impacting nearby RFIDs compared to RFID tags that are further away. Similarly, reflectors that were computed in the past may no longer exist at the same location and orientation in the future. To account for these effects, PushID employs the exponential weighting method [12] to progressively reduce the contribution of reflectors to the optimization with increasing distance from the reflector or time elapsed since detection. Specifically, we re-write Eqn. 7 as:

$$h_j = \left[\sum_{k=1}^s w_k / d_{ljk} e^{-2\pi\sqrt{-1}d_{ljk}/\lambda}, l = 1, \dots, L \right] \quad (9)$$

Where $w_k = f_1^{d_{ljk}} f_2^{t_k}$ and $f_1, f_2 < 1$ are constants (empirically set to 0.9 in our implementation) and t_k is the time elapsed since the measurement of reflector k was made.

5 Distributed Synchronization for RFIDs

In this section, we consider a classic challenge for distributed MIMO systems: accurate time and frequency synchronization with multiple distributed RFID readers for coherent beamforming. In particular, we actively estimate and correct for carrier frequency and timing offsets, which would otherwise cause transmissions across RFID readers to often combine incoherently. We build upon the classic distributed MIMO architecture used in the context of Wi-Fi [19, 40] while accounting for new challenges in the RFID context.

Quick Primer on Distributed MIMO: At a high level, past distributed MIMO systems for Wi-Fi [19, 40] use a master-slave architecture where multiple slave transmitters attempt to transmit in-phase with a master transmitter. Prior to transmitting each data packet, the master sends a short beacon containing a known preamble. Slave transmitters estimate their phase relative to this beacon to account for frequency offsets between the master’s clock and their own clocks. In addition, slaves exploit the relatively wide bandwidth of Wi-Fi to estimate phase shifts due to timing offsets. Slaves then apply phase shifts compensating for these offsets when they transmit data packets in-tandem with the master. Of course, during the data packet transmission itself, small additional phase drifts can accumulate owing to residual time and frequency offsets. As the duration of packets is short, the slope of such phase drifts can be readily corrected for.

Challenges in the RFID context: RFIDs bring two important challenges for distributed MIMO: (1) First, RFID packets last for a much longer time ($\sim 100\times$) than Wi-Fi packets [40], ensuring greater phase drift due to frequency offset. This is because tags need to harvest enough energy in order to respond to the readers’ queries, and this takes more time as the distance between the RFID tags and readers increases. (2) Second, RFID transmissions are narrowband (20 kHz), meaning that resolving timing offsets is extremely challenging.¹ The rest of this section tackles each of these challenges.

5.1 Frequency Offset Compensation

PushID’s key idea to compensate for frequency offsets leverages the full-duplex nature of RFID readers. Specifically, each PushID slave cancels out its own signal to recover the signal from the master reader. By measuring how the phase of this node drifts over time, PushID can correct for phase drifts that accumulate since the initial synchronization.

Correcting for Drift: For simplicity, let us begin with the case of two RFID readers – one master and one slave reader. To achieve that, initial synchronization in phase, we borrow from MegaMIMO [40]’s phase synchronization protocol

¹Past proposals for Wi-Fi like SourceSync [39] cannot be directly used in the narrow-band RFID context, as they assume availability of phase measurements over wide bandwidths (20 MHz) to estimate timing offsets.

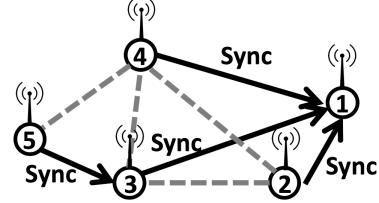


Figure 6: PushID constructs a spanning tree of the RFID readers in a distributed manner to guide synchronization.

where the RFID slave reader applies an initial phase shift to synchronize with the master (see Algm. 2). However, as frequency offset accumulates over time, the readers will notice that the phase of the master drifts.

PushID forms a closed loop to compensate for residual frequency offsets. Specifically, the slave RFID reader observes the change in phase from the master over a time interval $(t, t + \Delta t)$, where t is the most recent time of synchronization and Δt is the time elapsed since then. Should we observe a change in phase of the master transmitter since then, i.e. $\text{phase}(t + T) - \text{phase}(t)$, the slave applies the negative value of this phase offset to its own transmission. Note that this would, in effect, remove residual phase errors, allowing the two readers to combine their signals coherently at RFID tags. Further, note that due to channel reciprocity, the same phase shifts would ensure coherent combining on both the downlink and uplink.

Algorithm 2 Initial Frequency Synchronization

- 1: f_M : master’s oscillator frequency, f_{S_i} : slave i oscillator frequency.
 - 2: $\mathbf{H}(\mathbf{t}) = (h_1 e^{j2\pi(f_M-f_{S_1})t}, h_2 e^{j2\pi(f_M-f_{S_2})t})^T$
 - 3: Decompose $\mathbf{H}(\mathbf{t})$ we have $\mathbf{H}(\mathbf{t}) = \mathbf{R}(\mathbf{t})\mathbf{H}\mathbf{T}(\mathbf{t}) = \begin{pmatrix} e^{-j2\pi f_{S_1}t} & 0 \\ 0 & e^{-j2\pi f_{S_2}t} \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} e^{j2\pi f_M t}$
 - 4: $\mathbf{H}(\mathbf{t}) = e^{j2\pi f_M t} \mathbf{R}(\mathbf{t})\mathbf{H}\mathbf{T}(\mathbf{t}) e^{-j2\pi f_M t} = \begin{pmatrix} e^{j2\pi(f_M-f_{S_1})t} & 0 \\ 0 & e^{j2\pi(f_M-f_{S_2})t} \end{pmatrix} \mathbf{H} = \mathbf{A} \mathbf{H}$
 - 5: Apply frequency compensation $\mathbf{H}^{-1} \mathbf{A}^{-1}$ at slaves.
-

Scaling the system: A key challenge, however, is scaling the above system beyond two readers. This is because upon canceling one’s own transmission, each reader would perceive a linear combination of all other readers in its vicinity and not that of the master alone. This means that should a phase drift occur simultaneously for multiple readers (which is likely), these readers would be misled by out-of-sync transmissions from the others among them. To make matters worse, some readers may be beyond the communication range of the master and therefore may not be in a position to synchronize directly with the master as the last resort.

To mitigate this problem, we seek to assign to each RFID

reader a unique reference reader to which it may synchronize, should the master not be within its vicinity. Specifically, we assign indexes $1, \dots, n$ to each RFID reader, where RFID reader 1 denotes the master and all other readers denote slaves. We assume that these indexes are known a priori by the readers and can be constantly broadcasted so that any reader that fails can be removed from consideration in the optimization. Each RFID reader aims to synchronize its phase relative to the neighbor with the smallest index. In effect, our synchronization scheme creates a spanning tree of RFID readers (see Fig. 6), provided the graph of all readers is a connected one, which we assume. This spanning tree is designed to ensure that all readers eventually synchronizes to the root – the master RFID reader.

At this point each slave RFID reader subtracts its own signal and tracks the phase of the remaining linear combination. Should this change beyond a threshold, the RFID reader requests all its children and descendants in the spanning tree to cease transmission and then attempts to re-synchronize its transmission with the remaining active readers. We note that the master RFID reader never stops its transmission. It is easy to see that this scheme ensures that all RFID readers eventually transmit in-sync with the master.

5.2 Time Synchronization

PushID performs a two-step time synchronization process, a coarse synchronization to align symbols and a fine-synchronization that leverages the phase of signals from the master across frequency.

Coarse Synchronization: PushID synchronizes slaves with the master using the known preamble of the *Query* command transmitted by the reader to initialize an inventory. To maximize time resolution, slaves receive this signal from the master at a high sampling rate. Slaves then apply correlation with the known preamble to obtain the index of the master’s signal. We then only consider correlation coefficients above a threshold to reject outliers (we reject the bottom 6%). We repeat this process over five preambles and choose the result with the maximum correlation coefficient.

Fine Synchronization: To compensate for drift in timing offsets, PushID exploits the phase of signals from the master RFID reader a function of frequency. Specifically, recall that RFID transmissions hop between a wide range of frequencies in the 900 MHz ISM band spanning a total of 26 MHz. Each slave RFID reader estimates the phase of signals, having subtracted its own signal, across frequencies. PushID then monitors for any change in the slope of the phase of this signal across frequencies between measurements. Specifically, recall that any time offset of Δt between the two readers results in a frequency-dependent phase shift of $\Delta\phi = 2\pi f \Delta t$. As a result, PushID can estimate timing drifts by applying a least-squares linear regression [54] of ϕ as a function of t and

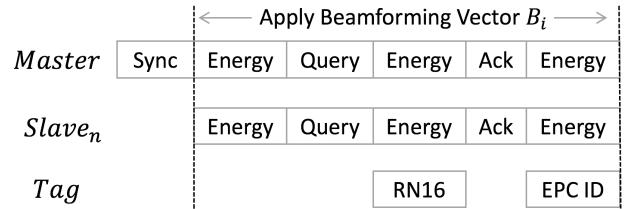


Figure 7: Depicts one round of PushID’s search for the unknown tags. Without a tag response, PushID starts another round by applying the next beamforming vector B_{i+1} .

obtain the resulting slope m . Any drift in timing offset can simply be computed as $m/2\pi$ and corrected for. We note that our system can scale akin to frequency offset compensation above, for more than two readers in the network. Specifically, when any RFID reader goes out-of-sync in time, it informs all its descendants to stop transmitting before attempting to re-synchronize.

6 Implementation and Evaluation

We implement PushID on a testbed of USRP N210 software radios with SBX/WBX daughterboards operating as RFID readers. We feed an omnidirectional and planar antenna to the antenna ports of each node for full-duplex use. All our readers are SISO, unless specified otherwise. Each USRP connects with an independent Jacksonlab Fury clock which could lead frequency and timing offset among the nodes. At the backend, each USRP is connected via Ethernet cables to a 64-bit Dell computer running Ubuntu 16.04. We also assume dedicated socket-based TCP connections between the reader nodes. Our RFID tags are commercial passive Alien Squiggle RFID tags. We measure a maximum range of up to 8.7 meters with our setup using one reader antenna.

PushID software: PushID is fully implemented in UHD/C++ including beamforming and distributed synchronization. In addition, we use an in-house UHD/Gnuradio based C++ RFID emulator to decode signals from the tags. We implement the set-cover based optimization in the cloud on a cluster of 64-bit core i7 Ubuntu machines and report the optimal beamforming weights to the reader nodes.

We ensure that all PushID RFID readers implement ASK modulation with PIE encoding to align with the specifications of the Gen2 RFID protocol. Apart from delivering energy, PushID readers also actively transmit messages which specify the tag’s modulation format, encoding scheme and backscatter frequency. The RFID tags in our experiments employ ASK modulation and FM0 encoding. The protocol flow of PushID is shown in Fig. 7.

Testbed: We evaluate PushID in two testbeds indoors and outdoors. (a) *Outdoor Testbed*: We deploy PushID around a football field (140 × 140 meters) with 8 transmitters. (b) *Indoor Testbed*: We deploy four-transmitter based PushID

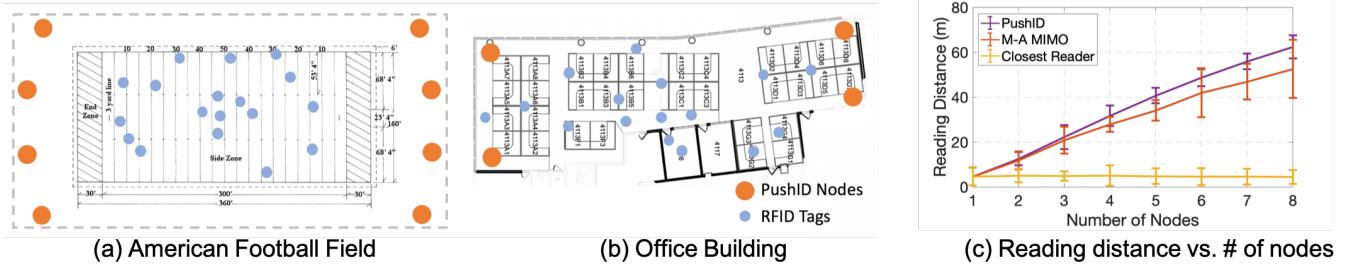


Figure 8: (a) **Outdoors:** We deploy PushID in a football field with one case of the transmitters placed as shown above. (b) **Indoors:** We deploy a four-transmitter based PushID on a floor (20×40 meters) of an office building covering multiple rooms and cubicles (c) Plots maximum reading range vs. # of readers. M-A MIMO – Multi-antenna MIMO.

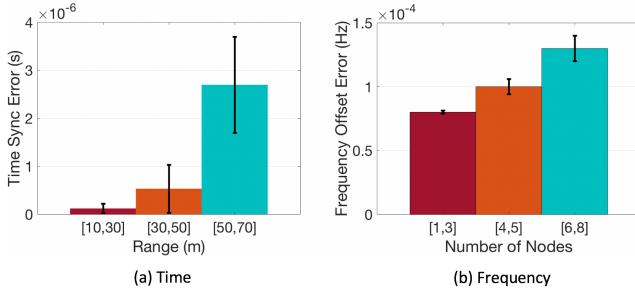


Figure 9: Time and Frequency Synchronization Accuracy

across a floor (20×40 meters) of an office building covering multiple rooms and cubicles. We note both testbeds are multipath rich due to stands/partitions in the former and cubicles/furniture in the latter blocking the direct path of some readers. We mount readers in various positions including different elevations. We put RFID tags in various positions and orientations that face towards different angles. Fig. 8(a) and (b) shows the candidate locations of RFID tags (represented by blue dots) and readers (represented by orange dots). We note that unless specified otherwise, all results incorporate an equal amount of data points (over 1000 RFID tag locations considered among them) from both testbeds with our core results evaluating how system accuracy changes in line-of-sight vs. non-line-of-sight relative to all readers.

Baseline: We compare PushID against two competing schemes: (1) *Closest Reader*: We assume that each reader independently decodes signals and each tag receives energy from the closest reader; (2) *Multi-antenna MIMO*: We assume that all reader antennas are co-located and synchronized by an external clock [10]. We note that unless specified otherwise, error bars in graphs denote standard deviation.

7 Results

7.1 Synchronization Accuracy

We evaluate the accuracy of PushID in achieving accurate frequency and time synchronization between base stations.

Method: We consider a testbed of up to eight USRP N210-

based RFID readers, one designated as the master and use PushID to synchronize the slaves to the master at high accuracy. The RFID readers are placed in various arbitrarily chosen geometries and different relative distances between the slaves and master reader. We measure two quantities of interest: (1) The error in time synchronization; (2) The error in frequency synchronization of signals at a USRP N210 receiver that compares the phase of the signals received from the master and slave(s) post PushID’s synchronization. We perform this experiment in both our indoor and outdoor testbed in which about half of the slaves on average are in non-line-of-sight relative to the master and some slaves (>50 m away from master) synchronize via multiple hops using PushID’s spanning tree approach.

Results: Fig. 9(a) shows the mean and standard deviation (error bars) in accuracy of time synchronization for different ranges of distance between a slave RFID reader and the master. We note that, as expected, the mean error with increasing distance also increases: PushID achieves a mean error of 0.12, 0.53, 2.69 μ s with the range of 10 to 30, 30 to 50 and 50 to 70 meters. However, we note that even the worst-case error is much smaller than one Nyquist time sample, given the narrow bandwidth of RFIDs (20 kHz). This means that PushID achieves the required level of time-synchronization accuracy to perform efficient distributed MIMO.

Next, Fig. 9(b) depicts the accuracy in frequency synchronization with increase in the number of RFID readers in the network. We find as expected the mean and variance of phase synchronization errors increase modestly as more readers join the network. PushID achieves a mean error of 0.0001 Hz in frequency offset overall across experiments. We note that this error corresponds to a phase shift of 0.0005° over the duration of a typical RFID packet and therefore minimally impacts the throughput of PushID’s distributed MIMO architecture, as observed in Sec. 7.3.

7.2 Range vs. Number of Nodes

In this experiment, we evaluate the maximum distance for a number of slaves that can detect the response from the tag.

Method: We deploy up to eight RFID readers in various geometries (starting from co-located and with progressively

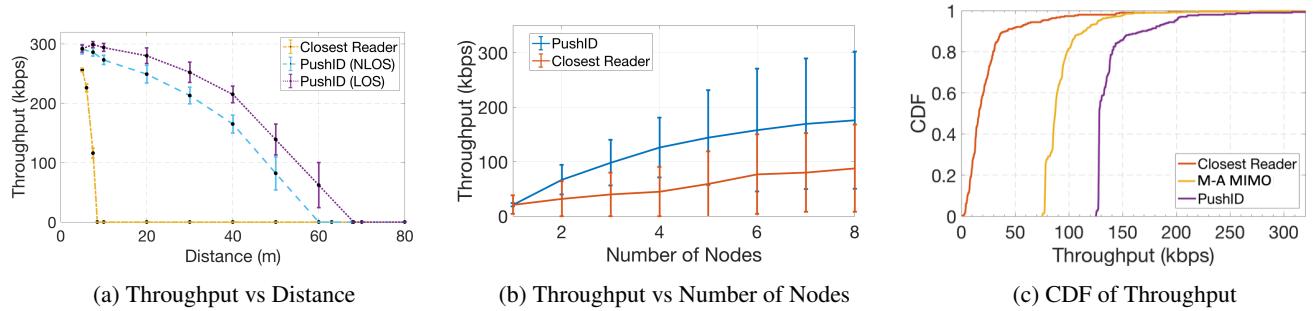


Figure 10: Throughput vs. (a) Distance of closest reader; (b) number of readers; (c) CDF with changing tag orientation.

increasing spacing) with tags placed in both involving line-of-sight and non-line-of-sight relative to the reader. Note that neither the location or number of RFID tags placed in the environment are known a priori to the readers. We consider, in aggregate over 1000 RFID tag locations across experiments. Across experiments, we note the distance between the RFID tag and its closest reader. Our experiments consider distances of up to 80 meters. Our goal is to estimate the maximum distance at which an RFID tag can be detected at the readers.

Results: Fig. 8(c) shows the maximum range of RFID tags with respect to the number of reader nodes (up to eight). As expected, PushID amplifies the received signal power from RFID readers to the tags and vice-versa, the PushID’s range increases quasi-linearly as the number of RFID readers increases. We note that the rate of increase does dip (gradually) with increasing number of readers due to the increasing impact of time and frequency synchronization errors as reported in Sec. 7.1 above. We further notice a surprising decreasing trend in the standard deviation with maximum distance. We find that this stems from the robustness of PushID to orientation in the presence of multiple distributed readers all oriented in diverse spatial directions. Our results show that PushID achieves a maximum range to an RFID tag of 64 meters, a gain of about $7.4 \times$ vs. commercial RFID and 20% over multi-antenna MIMO. We also note that the standard deviation of multi-antenna MIMO is large in various experimental settings. In contrast, PushID has better resilience and stability across experiments which gives more spatial diversity that benefits the poor polarization sensitivity of the RFID dipole antennas. We also notice that sometimes the multi-antenna MIMO has better performance than PushID when tag’s orientation favors the direction of collocated readers in the LOS setting.

7.3 Throughput vs. Distance and Scale

In this experiment, we evaluate the impact of PushID on the throughput as we vary the number of RFID readers and the distance between the tag and its closest reader.

Method: We measure the throughput by first measuring the SNR of each RFID tag measured from the eight RFID readers after coherently combining signals to and from the tag. We

then adapt the ESNR metric [18] to the RFID context to estimate the maximum data rate achievable for the received SNR. Note that once PushID’s algorithm is applied to detect a tag, we use channel reciprocity to maximize throughput to each detected tag in this experiment. We deploy PushID in both the outdoor (Fig.8(a)) and indoor scenario (Fig.8(b)) and consider tags in both line-of-sight and non-line-of-sight relative to the readers. Our RFID tags use FM0 modulation which allows for data rates over 45 kbps.

Throughput vs. Distance: Fig.10a shows the increase in throughput vs. distance in line-of-sight and non-line-of-sight settings and compares it against the baseline system that connects to the closest RFID reader. We observe that for the baseline, as expected, a reader has a maximum range of about 8.7 m across both line-of-sight and non-line-of-sight settings and we see that performance drops to zero throughput beyond this distance. PushID, with a 67.5 meter maximum range on average outperforms the baseline significantly in line-of-sight and 58.9 meters in non-line-of-sight. Further, as expected the throughput of PushID drops down as the distance increases due to lower signal-to-noise ratio. We note that quite significantly, PushID’s performance with eight transmitters increases the throughput of RFID tags $2.6 \times$ when compared to the baseline closest-RFID reader scheme at its maximum range of about 8.7 m.

Throughput vs. Number of Reader Nodes: As expected, with the increasing number of reader nodes we observe a gradual (logarithmic) increase in network throughput of covered RFID tags on average for PushID. There is a similar, although much more modest increase with reader nodes for the baseline owing to an increase in coverage area. However, our system observes a net mean throughput gain of $2.6 \times$ over a network of 8-nodes over the baseline closest-reader system.

7.4 Impact of Orientation

Method: In this experiment, we model the distribution of the throughput of an RFID tag progressively oriented along various directions in 100 locations with 8 readers and compare three schemes: (1) PushID; (2) A 8-antenna MIMO scheme; (3) The closest reader baseline.

Results: Fig. 10c plots the CDF of throughput across

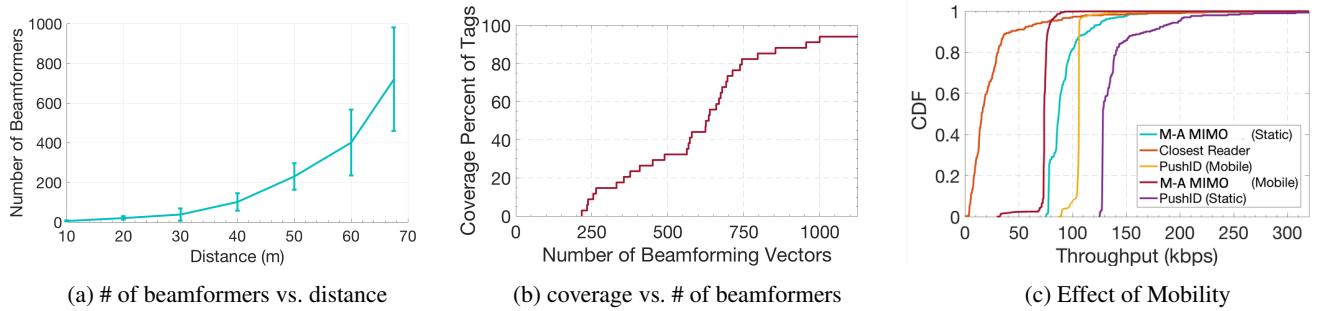


Figure 11: (a) Distance of closest reader; (b) % of tags covered; (c) CDF of throughput with mobility

schemes with changing tag orientation. We observe that PushID outperforms both multi-antenna MIMO (by 1.54× median) and the closest reader baseline (by 7.4× median). This is because, as explained in Sec. 4.1, PushID readers are oriented variously and therefore much more robust to change in orientations of the tags. In contrast, multi-antenna MIMO with reader antennas co-located loses performance when tags are oriented away from the MIMO reader, unlike PushID.

7.5 Impact of Mobility

Method: We deploy two tags in the environment in our indoor testbed, each placed initially at the same 100 randomly chosen initial locations at varying distances (up to 60 m) from the closest reader: (1) static RFID tags; and (2) an RFID tag moved around at walking speeds by volunteers. We measure the performance of PushID 8-antenna MIMO scheme and our closest reader baseline from eight RFID readers and compare performance.

Results: Fig. 11c plots the CDF of throughput across schemes for static and mobile tags. As expected, throughput dips in the presence of mobility across schemes. However, we note that PushID achieves gains over the baseline despite mobility (1.6× for static and 1.4× for mobile over multi-antenna MIMO). PushID’s robustness to mobility stems from two reasons: (1) The mobility of the RFID tag ensures that the tag is highly likely to move out of coverage holes. As a result, during its trajectory, PushID has a higher likelihood of detecting the tag, compared to a static tag. This counteracts to help recover some of the loss in performance owing to changing multipath in PushID’s algorithms. (2) Once the tag is first detected, PushID can use channel reciprocity to rapidly continue beamforming to the tag and thereby respond to its mobility. We however highlight (as stated in limitations in Sec. 1) that mobility at very high speeds would significantly deteriorate PushID’s performance and gains, just as it would deteriorate commercial RFID systems.

7.6 Convergence and Coverage

Method: In this experiment, we measure the convergence time of PushID’s algorithm and how it is impacted by the distance of RFID tags and its trade-off with total area covered.

We note that the initial set of beamforming vectors PushID uses can be found offline and future updates take minimal time overhead. PushID’s main computational bottleneck is the rate at which beamforming weights can be applied by the USRP hardware which is 4.5 milliseconds in our testbed. We therefore measure convergence time in terms of the number of beamforming vectors that needs to be applied. Once again, we consider RFID tags at a wide range of distances to the closest reader with eight RFID readers and run PushID.

Results: Fig. 11b shows that the percentage of tags discovered increases with increasing number of beamformers for distances from the closest reader over 60 m. We emphasize here that it is rare for our system to detect all tags, since some tags remain virtually undetectable due to their location, orientation or shadowing – a natural limitation of our system (highlighted in Sec. 1) and indeed most wireless systems (for e.g. even the best cellular networks have deadspots [20]). Beyond 95% coverage, we see diminishing returns upon applying more beamforming vectors. Fig. 11a measures the trade-off between the distance of the tag and the number of beamformers needed to find > 95% of tags in the area. We find that in the worst case at maximum distance, we need 980 beamformers (4.4 seconds for a USRP N210).

8 Conclusion and Future Work

This paper presents, to our knowledge, the first distributed MIMO system to power commercial passive RFID tags. PushID develops a blind distributed beamforming algorithm to efficiently search through the space of beamforming vectors. It further develops a novel phase synchronization algorithm to synchronize distributed RFIDs. A detailed prototype evaluation on an eight-antenna distributed reader testbed reveals a 7.4× improvement in range compared to state-of-the-art commercial readers. While this paper focuses on using existing commercial tags without modifications, we believe future work needs to explore algorithms that innovate on radio design and light-weight computation on the tags themselves to improve PushID’s performance.

Acknowledgements We thank anonymous NSDI reviewers and our shepherd, Fadel Adib, for their feedback and insights. We would like to thank NSF (grants 1718435, 1657318, 1823235 and 1837607) for their support.

References

- [1] Regulatory status for using rfid in the epc gen2 (860 to 960 mhz) band of the uhf spectrum. https://www.gs1.org/docs/epc/uhf_regulations.pdf, 2018. (Accessed on 08/27/2018).
- [2] AGEE, B. G., SCHELL, S. V., AND GARDNER, W. A. Spectral self-coherence restoral: a new approach to blind adaptive signal extraction using antenna arrays. *Proceedings of the IEEE* 78, 4 (April 1990), 753–767.
- [3] ALEVRAZ, D., PADBERG, M., AND PADBERG, M. W. *Linear optimization and extensions: problems and solutions*. Springer Science & Business Media, 2001.
- [4] ATLASRFIDSTORE. Impinj rhcp far field rfid antenna (fcc/etsi). <https://www.atlasrfidstore.com/impinj-rhcp-far-field-rfid-antenna-fcc-etsi/>. (Accessed on 04/30/2018).
- [5] BAR-YEHUDA, R., AND EVEN, S. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms* 2, 2 (1981), 198–203.
- [6] BIRARI, S. M., AND IYER, S. Mitigating the reader collision problem in rfid networks with mobile readers. In *Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication, 2005 13th IEEE International Conference on* (2005), vol. 1, IEEE, pp. 6–pp.
- [7] BISWAL, A. K., JENAMANI, M., AND KUMAR, S. K. Warehouse efficiency improvement using rfid in a humanitarian supply chain: Implications for indian food security system. *Transportation Research Part E: Logistics and Transportation Review* 109 (2018), 205–224.
- [8] BLETSAS, A., LIPPMAN, A., AND SAHALOS, J. N. Simple, zero-feedback, distributed beamforming with unsynchronized carriers. *IEEE journal on selected areas in communications* 28, 7 (2010).
- [9] CARDOSO, J.-F., AND SOULOUMIAC, A. Blind beamforming for non-gaussian signals. In *IEE proceedings F (radar and signal processing)* (1993), vol. 140, IET, pp. 362–370.
- [10] CHEN, S., ZHONG, S., YANG, S., AND WANG, X. A multiantenna rfid reader with blind adaptive beamforming. *IEEE Internet of Things Journal* 3, 6 (2016), 986–996.
- [11] CHEN, Y.-C., CHU, C., CHEN, R.-S., SUN, H. M., AND JU, P. Rfid-based bonded warehouse for science park. *International Journal of Radio Frequency Identification Technology and Applications* 5, 1 (2018), 1–23.
- [12] CROWDER, S. V. A simple method for studying run-length distributions of exponentially weighted moving average charts. *Technometrics* 29, 4 (1987), 401–407.
- [13] DONGARE, A., NARAYANAN, R., GADRE, A., LUONG, A., BALANUTA, A., KUMAR, S., IANNUCCI, B., AND ROWE, A. Charm: exploiting geographical diversity through coherent combining in low-power wide-area networks. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2018), IEEE Press, pp. 60–71.
- [14] ELETREBY, R., ZHANG, D., KUMAR, S., AND YAĞAN, O. Empowering low-power wide area networks in urban settings. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 309–321.
- [15] FAN, X., DING, H., LI, S., SANZARI, M., ZHANG, Y., TRAPPE, W., HAN, Z., AND HOWARD, R. E. Energy-ball: Wireless power transfer for batteryless internet of things through distributed beamforming. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 2 (July 2018), 65:1–65:22.
- [16] FAN, X., ZHANG, Z., TRAPPE, W., ZHANG, Y., HOWARD, R., AND HAN, Z. Secret-focus: A practical physical layer secret communication system by perturbing focused phases in distributed beamforming. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications* (April 2018), pp. 1781–1789.
- [17] FEIGE, U. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM (JACM)* 45, 4 (1998), 634–652.
- [18] HALPERIN, D., HU, W., SHETH, A., AND WETHERALL, D. Predictable 802.11 packet delivery from wireless channel measurements. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 159–170.
- [19] HAMED, E., RAHUL, H., ABDELGHANY, M. A., AND KATABI, D. Real-time distributed mimo systems. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 412–425.
- [20] HARROLD, T., AND NIX, A. Intelligent relaying for future personal communication systems.
- [21] HSIA, K.-H., WU, M.-G., LIN, J.-N., ZHONG, H.-J., AND ZHUANG, Z.-Y. Development of auto-stacking warehouse truck. *JRNAL* 4, 4 (2018), 334–337.
- [22] JADIDIAN, J., AND KATABI, D. Magnetic mimo: How to charge your phone in your pocket. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 495–506.

- [23] JIANG, C., HE, Y., ZHENG, X., AND LIU, Y. Orientation-aware rfid tracking with centimeter-level accuracy. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2018), IEEE Press, pp. 290–301.
- [24] JIN, H., WANG, J., YANG, Z., KUMAR, S., AND HONG, J. Rf-wear: Towards wearable everyday skeleton tracking using passive rfids. In *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers* (New York, NY, USA, 2018), UbiComp ’18, ACM, pp. 369–372.
- [25] JIN, H., WANG, J., YANG, Z., KUMAR, S., AND HONG, J. Wish: Towards a wireless shape-aware world using passive rfids. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2018), MobiSys ’18, ACM, pp. 428–441.
- [26] KARP, R. M. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972, pp. 85–103.
- [27] KIM, D., AND YEO, J. Dual-band long-range passive rfid tag antenna using an amc ground plane. *IEEE Transactions on Antennas and Propagation* 60, 6 (2012), 2620–2626.
- [28] KUESTER, D., AND POPOVIC, Z. How good is your tag?: Rfid backscatter metrics and measurements. *IEEE Microwave Magazine* 14, 5 (2013), 47–55.
- [29] LEE, J., KWON, H., AND LEE, B. Design consideration of uhf rfid tag for increased reading range. In *2006 IEEE MTT-S International Microwave Symposium Digest* (June 2006), pp. 1588–1591.
- [30] LIUKKONEN, M., AND TSAI, T.-N. Toward decentralized intelligence in manufacturing: recent trends in automatic identification of things. *The International Journal of Advanced Manufacturing Technology* 87, 9–12 (2016), 2509–2531.
- [31] LOO, C. H., ELSHERBENI, A. Z., YANG, F., AND KA-JFEZ, D. Experimental and simulation investigation of rfid blind spots. *Journal of Electromagnetic Waves and Applications* 23, 5-6 (2009), 747–760.
- [32] MA, Y., HUI, X., AND KAN, E. C. 3d real-time indoor localization via broadband nonlinear backscatter in passive devices with centimeter precision. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking* (2016), ACM, pp. 216–229.
- [33] MA, Y., LUO, Z., STEIGER, C., TRAVERSO, G., AND ADIB, F. Enabling deep-tissue networking for miniature medical devices. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 417–431.
- [34] MA, Y., SELBY, N., AND ADIB, F. Drone relays for battery-free networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 335–347.
- [35] MA, Y., SELBY, N., AND ADIB, F. Minding the billions: Ultra-wideband localization for deployed rfid tags. In *ACM MobiCom* (2017).
- [36] NADEEM, Q.-U.-A., KAMMOUN, A., AND ALOUINI, M.-S. Elevation beamforming with full dimension mimo architectures in 5g systems: A tutorial. *arXiv preprint arXiv:1805.00225* (2018).
- [37] PILLAI, V., HEINRICH, H., DIESKA, D., NIKITIN, P. V., MARTINEZ, R., AND RAO, K. S. An ultra-low-power long range battery/passive rfid tag for uhf and microwave bands with a current consumption of 700 na at 1.5 v. *IEEE Transactions on Circuits and Systems I: Regular Papers* 54, 7 (2007), 1500–1512.
- [38] RAGHAVAN, P. *Randomized Algorithms*.
- [39] RAHUL, H., HASSANIEH, H., AND KATABI, D. Sourcesync: a distributed wireless architecture for exploiting sender diversity. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 171–182.
- [40] RAHUL, H. S., KUMAR, S., AND KATABI, D. Jmb: Scaling wireless capacity with user demands. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM ’12, ACM, pp. 235–246.
- [41] SARANGAN, V., DEVARAPALLI, M. R., AND RADHAKRISHNAN, S. A framework for fast rfid tag reading in static and mobile environments. *Computer Networks* 52, 5 (2008), 1058–1073.
- [42] SHEPARD, C., YU, H., ANAND, N., LI, E., MARZETTA, T., YANG, R., AND ZHONG, L. Argos: Practical many-antenna base stations. In *Proceedings of the 18th annual international conference on Mobile computing and networking* (2012), ACM, pp. 53–64.
- [43] SHI, L., KABELAC, Z., KATABI, D., AND PERREAU, D. Wireless power hotspot that charges all of your devices. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 2–13.

- [44] SKLIVANITIS, G., ALEXANDRIS, K., AND BLETSAS, A. Testbed for non-coherent zero-feedback distributed beamforming. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), IEEE, pp. 2563–2567.
- [45] SLOCK, D. T. M. Blind fractionally-spaced equalization, perfect-reconstruction filter banks and multichannel linear prediction. In *Proceedings of ICASSP '94. IEEE International Conference on Acoustics, Speech and Signal Processing* (April 1994), vol. iv, pp. IV/585–IV/588 vol.4.
- [46] SMITH, J. R., SAMPLE, A. P., POWLEDGE, P. S., ROY, S., AND MAMISHEV, A. A wirelessly-powered platform for sensing and computation. In *International Conference on Ubiquitous Computing* (2006), Springer, pp. 495–506.
- [47] SORNIN, N., LUIS, M., EIRICH, T., KRAMP, T., AND HERSENT, O. Lorawan specification. *LoRa alliance* (2015).
- [48] TALLA, V., HESSAR, M., KELLOGG, B., NAJAFI, A., SMITH, J. R., AND GOLLAKOTA, S. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 105.
- [49] TRAN, N., LEE, B., AND LEE, J.-W. Development of long-range uhf-band rfid tag chip using schottky diodes in standard cmos technology. In *Radio Frequency Integrated Circuits (RFIC) Symposium, 2007 IEEE* (2007), IEEE, pp. 281–284.
- [50] TSE, D. Fundamentals of wireless communication.
- [51] VARSHNEY, A., HARMS, O., PÉREZ-PENICHET, C., ROHNER, C., HERMANS, F., AND VOIGT, T. Lorea: A backscatter architecture that achieves a long communication range. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (2017), ACM, p. 18.
- [52] WANG, J., AND KATABI, D. Dude, where's my card?: Rfid positioning that works with multipath and non-line of sight. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 51–62.
- [53] WANG, J., VASISHT, D., AND KATABI, D. Rf-idraw: virtual touch screen in the air using rf signals. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 235–246.
- [54] WOLD, S., RUHE, A., WOLD, H., AND DUNN, III, W. The collinearity problem in linear regression. the partial least squares (pls) approach to generalized inverses. *SIAM Journal on Scientific and Statistical Computing* 5, 3 (1984), 735–743.
- [55] YANG, J., AND LEUNG, J. Y.-T. A generalization of the weighted set covering problem. *Naval Research Logistics (NRL)* 52, 2 (2005), 142–149.
- [56] YANG, L., CHEN, Y., LI, X.-Y., XIAO, C., LI, M., AND LIU, Y. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 237–248.
- [57] ZAK, S. H., UPATISING, V., AND HUI, S. Solving linear programming problems with neural networks: a comparative study. *IEEE Transactions on Neural Networks* 6, 1 (1995), 94–104.
- [58] ZEILER, M. D. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).
- [59] ZUNIGA, J. C., AND PONSARD, B. Sigfox system description. *LPWAN@ IETF97, Nov. 14th* (2016).

SweepSense: Sensing 5 GHz in 5 Milliseconds with Low-Cost Radios

Yeswanth Gudde, Raghav Subbaraman[†], Moein Khazraee, Aaron Schulman, and Dinesh Bharadia
UC San Diego [†]*IIT Madras*

Abstract

Wireless transmissions occur intermittently across the entire spectrum. For example, WiFi and Bluetooth devices transmit frames across the 100 MHz-wide 2.4 GHz band, and LTE devices transmit frames between 700 MHz and 3.7 GHz). Today, only high-cost radios can sense across the spectrum with sufficient temporal resolution to observe these individual transmissions.

We present “SweepSense”, a low-cost radio architecture that senses the entire spectrum with high-temporal resolution by rapidly sweeping across it. Sweeping introduces new challenges for spectrum sensing: SweepSense radios only capture a small number of distorted samples of transmissions. To overcome this challenge, we correct the distortion with self-generated calibration data, and classify the protocol that originated each transmission with only a fraction of the transmission’s samples. We demonstrate that SweepSense can accurately identify four protocols transmitting simultaneously in the 2.4 GHz unlicensed band. We also demonstrate that it can simultaneously monitor the load of several LTE base stations operating in disjoint bands.

1 Introduction

High-time-resolution spectrum sensors [5, 18, 37, 32] enable new ways to share and manage the spectrum¹. For example, the FCC granted permission for LTE providers to share licensed spectrum in the 3.5 GHz CBRS band with military radars, only if spectrum sensors are installed that can detect the military’s millisecond-long military radar bursts anywhere within the 100 MHz bandwidth of the CBRS band [40]. In the future, we may even be able to improve co-existence of devices operating in the 5.8 GHz ISM band by performing high-time-resolution spectrum sensing of its 150 MHz band-

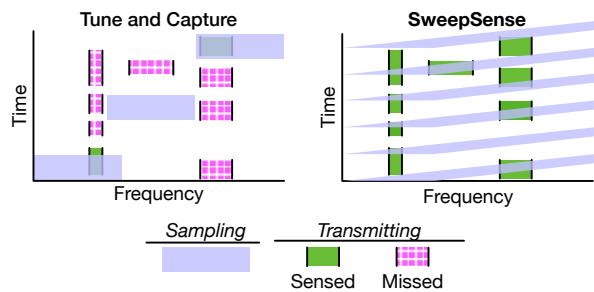


Figure 1: SweepSense rapidly sweeps its center frequency, rather than iteratively tuning and capturing the transmissions one frequency at a time.

width. For instance, a third-party high-time-resolution sensor can detect short intervals when WiFi devices are not using the spectrum, and inform unlicensed LTE base stations that they can operate without interfering [7].

Unfortunately, only complex and expensive spectrum sensors have both wide bandwidth and high time resolution. For example, there are radios that can sample several GHz of RF bandwidth continuously (e.g., On-eRadio [14]). However, they are expensive ($\sim \$500,000$) due to their high-speed Analog-to-Digital converters, and complex due to the heavy computational power needed to perform real-time signal processing on high sample rates (e.g., GPUs or FPGAs). On the other end of the spectrum are narrow-bandwidth (~ 50 MHz) radios (e.g., SDRs such as the USRP or HackRF [30, 37, 34]) that can not observe entire bands (e.g., 100 MHz) at once. The sensing bandwidth of these radios can be improved by intelligently tuning [37] but they are still likely to miss transmission due to their narrow bandwidth and the downtime they experience during tuning (as shown in Fig. 1).

We introduce a new paradigm in spectrum sensing, called SweepSense, which achieves both wide sensing bandwidth and high time resolution with off-the-shelf narrow-bandwidth radios. SweepSense introduces a fundamental shift in the receiver architecture of narrow-

¹High-time-resolution spectrum sensors are defined by their capability to observe a portion of every transmission (e.g., packet).

bandwidth radios: instead of tuning into each frequency, sampling for a short time, then switching to the next frequency, SweepSense rapidly sweeps the frequency of the receiver across the spectrum (Figure 1). By sweeping rapidly across the spectrum, SweepSense achieves high time resolution with a narrow bandwidth radio. However, there are several challenges that we must overcome to demonstrate that SweepSense is practical and feasible:

Off-the-shelf radios do not sweep: SweepSense is only practical if it can be deployed on existing radios, such as SDR-based spectrum sensors [35]. The RF signal path on the radio should not require extensive changes to make it sweep. Prior approaches to making radios sweep by adding an expensive high-sample rate Digital-to-Analog converter that acts as a rapidly sweeping local oscillator are impractical [9].

Sweeping radios distort samples: Rapidly sweeping the center frequency of a radio results in samples that are collected at an unknown, and changing, center frequency. These samples need to be mapped to a single center frequency, and corrected for distortions introduced by sweeping. Furthermore, the continuous changing of frequency may reduce the sensitivity of the radio, making it impossible to detect weak signals.

Sweeping radios only visit bands for a short time: Rapidly sweeping radios collect a small number of samples in each band. This may break typical spectrum sensing-related signal analysis, such as signal type identification and spectrum occupancy detection.

We make the following contributions that address each of these challenges:

1. Making off-the-shelf radios sweep (Section 3):

We show that with only a simple modification to the local oscillator circuit of a radio, we can make it rapidly sweep its center frequency. Specifically, we disconnect the feedback loop used to lock the receiver’s local oscillator onto a specific frequency, and replace it with a sawtooth signal, thus making the center frequency sweep. We demonstrate the generality of this simple modification, by performing it on three of the most popular RF frontends for the USRP SDR, the WBX (50 MHz–2.2 GHz), SBX (400 MHz–4.4 GHz), and CBX (1.2 GHz–6 GHz).

2. Unsweeping samples (Section 4): We present a novel calibration and recovery process that corrects the continuously changing frequency in samples captured by the sweeping radio receiver. Specifically, we created a mechanism that inverts the effects of the sweeping center frequency by mixing it with complex conjugate of a calibration signal. Generating the calibration signal

does not require any extra hardware: it is received through leakage from the radio’s own RF transmitter (As TX loopback mode was not supported in the SDR). The result of the unsweeping process is a stream of samples that look as if they were collected at a fixed center frequency.

3. Evaluating analysis of short captures (Section 5):

We demonstrate that even with the small number samples captured by SweepSense, the repeated patterns and unique features of the captured signals are retained. Specifically, we show that cyclo-stationary techniques when used in tandem with standard classification models need just 25 μ sec captures of signals to classify accurately. Previously it was assumed that these techniques required capturing the entire transmission (e.g., ~ 1 msec packet for WiFi).

We evaluate SweepSense by modifying a USRP N210 SDR to sweep, and performing experiments in both indoor and outdoor environments. We made the following observations: (1) SweepSense can classify signals with at least 90% accuracy (wideband DSSS and OFDM WiFi, as well as narrowband Zigbee and Bluetooth) with only 25 μ s of samples, (2) SweepSense can simultaneously measure the millisecond level utilization of multiple LTE downlink channels over a bandwidth of 200 MHz, , and (3) SweepSense can accurately detect fleeting radar bursts, required for serving as a spectrum sensor for the CBRS spectrum.

The SweepSense implementation for the USRP N210 is open source and available at:

<https://github.com/ucsdsysnet/sweepsense>

2 Related Work

Spectrum sensing is an extensively studied area [23, 15, 42, 33, 41, 28, 24, 31]. Recent innovations have been focusing on improving the time resolution of spectrum sensors. To the best of our knowledge, SweepSense is the first work to suggest improving the time resolution of narrow-band spectrum sensors by making them rapidly sweep—without sacrificing their ability to classify transmitter type and characterize utilization. In this section, we describe how SweepSense complements, compares, and improves upon prior approaches to improving the time resolution of spectrum sensors.

HIGH-SPEED SPECTRUM ANALYZERS: The most common RF equipment that can sweep the spectrum quickly (i.e., tens of milliseconds) are high-speed spectrum analyzers, such as the Oscor Blue [32, 3, 36]. These devices are expensive high-end test equipment, designed to accurately measure the absolute power of transmitters (e.g., for certification), or discover bugging devices that are transmitting in esoteric bands. Spectrum analyzers only measure the power of transmissions in the

frequency domain, they do not collect time-domain signals. Therefore, they cannot be used to perform signal analysis such as signal classification. For example, signals operating on the same frequency cannot be differentiated (e.g., in 2.4 GHz, antiquated DSSS 802.11b looks the same as modern OFDM 802.11g/n) on spectrum analyzer displays.

FMCW-BASED SPECTRUM SENSORS: As an improvement over spectrum analyzers which can only observe power, recent work by Cheema et al. [9] introduced receivers that can rapidly sweep over the spectrum to capture short time-domain samples across the spectrum. Their work is a proof-of-concept that demonstrates, with ideal hardware—namely, a costly signal generator—it is possible to perform high time resolution spectrum occupancy detection. This work inspired us to look into a practical modification for off-the-shelf radios that can make them sweep. However, unlike SweepSense, Cheema et al. only demonstrate using these samples to improve the time resolution of the spectrum occupancy. SweepSense is the first to demonstrate how to unsweep the samples to successfully perform signal analysis across GHz of spectrum, only with short captures of each band (Section 5). Prior to SweepSense, widebandwidth signal analysis was only considered possible with wide-bandwidth radios.

In summary, SweepSense demonstrates that narrow-bandwidth radios can be modified—with only the addition of an analog ramp generator fed to the VCO’s tuning input—to create a rapidly sweeping radio. SweepSense also introduces a novel algorithm to unsweep distorted samples captured by modified off-the-shelf radios (Section 4). SweepSense also demonstrates that these unswept samples can still be used to perform rigorous signal analysis such as signal classification (Section 5).

INTELLIGENT SCANNING FOR SDR-BASED SENSORS: SpecInsight [37] improves the time resolution of spectrum sensing with narrow-bandwidth SDR’s (~ 25 Msps) by intelligently scheduling when bands should be tuned into. Those that contain continuous transmitters (e.g., FM Radio) or predictable transmitters (e.g., airport RADAR) are tuned into infrequently, thereby improving the time resolution of narrow-bandwidth spectrum sensors. SpecInsight is complementary to SweepSense because it can use their band selection algorithm to intelligently select when to sweep particular bands. Therefore, other intelligent scanning algorithms [43, 26, 44, 25] can also be integrated into SweepSense to improve its time resolution.

SUB-NYQUIST SPECTRUM CAPTURE: Similar to SweepSense’s goal of modifying off-the-shelf radios to operate across a wide bandwidth, prior work [4, 18] demonstrates that an off-the-shelf SDR can sample outside of their Nyquist bandwidth by removing

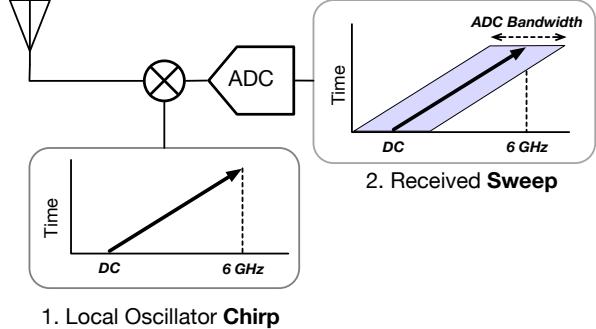


Figure 2: By chirping a receiver’s Local Oscillator, it will rapidly sweep the spectrum.

the anti-aliasing filter on the RF frontend. However, these techniques assume that spectrum is sparsely occupied, and make use of specialized techniques like sparseFFT [16, 13, 17], or compressed sensing [4, 2, 10, 39]. SweepSense does not make such assumptions about the power and frequency of the transmissions in the spectrum. However, given that these systems are built on the same inexpensive SDRs as SweepSense, we might be able to increase our instantaneous bandwidth by sampling at sub-Nyquist rate while sweeping.

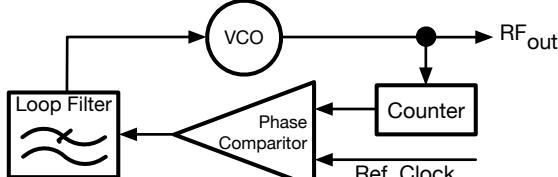
3 Making Off-the-Shelf Radios Sweep

In this section, we describe how we modify the oscillator in off-the-shelf radios so they can rapidly sweep across several GHz. Fig. 2 shows an overview of the operation of a SweepSense receiver. To make the radio sweep, we modify the behavior of the radio’s Local Oscillator (LO)—the device that tunes a radio into a particular frequency—to rapidly increase its frequency (chirp).

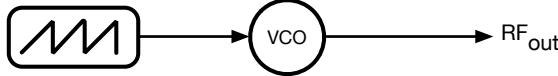
First, we describe how the LO in a radio can be modified to make it chirp. Then, we explain how to perform this modification on a USRP N210 SDR—a common off-the-shelf SDR with a wide tuning range.

3.1 How to make an LO chirp

To understand how to modify the LO to chirp, we must first explain how the LO operates in a radio. The LO is the hardware component in a radio that generates a tone which gives the receiver the ability to tune into a specific frequency. The tone from the LO is mixed with the amplified signal from the antenna to change the frequency of the received radio frequency (RF) signal and downconvert it to baseband. The baseband signal is then filtered and sampled by an ADC, and the raw digital samples are transferred to the host. Radios with a wide tuning range (e.g., SDRs) are built with a special LO that can generate tones across a wide frequency range; these LOs are called “wideband frequency synthesizers”. For instance, the MAX2870 [29] frequency synthesizer on



(a) **Normal config:** Closed-loop tuning



(b) **Sweeping config:** Open-loop chirping

Figure 3: Replacing a PLL-based LO’s tuning feedback loop with a sawtooth waveform makes it sweep.

the USRP CBX daughter card can generate tones ranging from 23.5 MHz to 6 GHz.

A wideband frequency synthesizer is commonly implemented using a Voltage Controlled Oscillator (VCO) in a highly integrated Phase Locked Loop (PLL). A simplified block diagram of a PLL is shown in Fig. 3 (a). The input voltage of the VCO determines its output frequency, and the PLL serves as the feedback loop that maintains control over the VCO input voltage to generate a fixed frequency tone. The feedback loop is driven by a phase comparator that compares the phase of the VCO output (divided by the counter), and the reference clock. The difference in phase is an indirect measure of the frequency error between the desired VCO output and its actual value. The external passive low-pass “loop filter” then filters the phase comparator output. The loop filter output drives the VCO input voltage, completing the control loop and “lock”ing the VCO output to the desired frequency. The loop filter characteristics and cutoff frequency determine the stability and accuracy of the frequency lock. Each time the frequency synthesizer is requested to generate a different frequency output, the PLL takes 10–100 μ s to lock, during which the radio is temporarily offline. It is this repeated downtime that SweepSense avoids by making the PLL sweep continuously across a wide frequency range.

There are two parts of such an LO design that make them amenable to sweeping (1) the ability to control output frequency by adjusting the input voltage to the VCO, and (2) the customizable loop filter that is implemented with external passive components.

An LO can be modified to sweep by first disconnecting (by desoldering) the loop filter components, giving direct access to the VCO control input. Then, the now-unconnected VCO control input is connected to an externally generated sawtooth voltage. As the sawtooth signal repeatedly ramps its voltage, the VCO repeatedly ramps its output frequency. As a result, the VCO out-

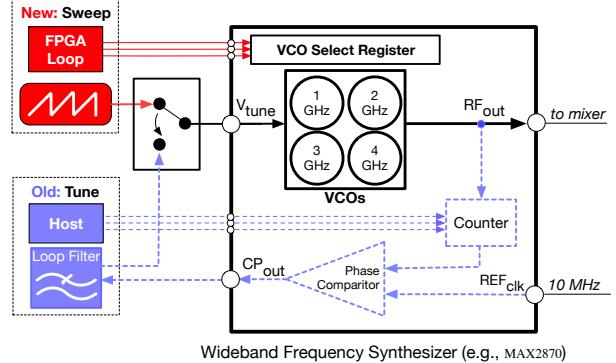


Figure 4: Modifications required to implement SweepSense on a COTS wideband frequency synthesizer.

put is a series of chirps (the modification is shown in Fig. 3 (b)). However, removing the feedback loop introduces several new challenges that we describe and address in Section 4.

Additionally, the wideband frequency synthesizers in off-the-shelf radios are particularly amenable to sweeping for spectrum sensing because they contain a bank of VCOs², each of which has a smaller frequency range (e.g., 100 MHz) that, put together, contribute to the LO’s wide frequency range (depicted in Fig. 4). This modular construction makes such synthesizers much less expensive as compared to a single VCO synthesizer that has comparable tuning range. Also, being able to select which VCOs are used is important for frequency planning, such as skipping entire VCO bands that do not have active transmitters (SpecInsight [37]). Many modern frequency synthesizers (like the MAX2870) provide an explicit control register to select a particular VCO. For such synthesizers, SweepSense can implement fine-grained VCO selection and sweep with virtually no delay introduced due to the selection process.

3.2 Proof of Concept: Sweeping USRP

We now describe the complete modification that makes the commonly available USRP N210 SDRs sweep. We demonstrate that these modifications are general by performing them on three popular RF frontends for the USRP: the WBX and SBX that have an older Analog Devices synthesizer, and the CBX that has a modern Maxim synthesizer. We also believe it is compatible with the HackRF One that has a modern synthesizer from Qorvo. There are two aspects to this modification: (1) a hardware modification to disconnect the VCO feedback loop and replace it with a sawtooth signal and (2) an FPGA

²VCOs are implemented as a set of LC circuits (VCO cores) each of which can switch in a set of varactors (bands) depending on the desired frequency range

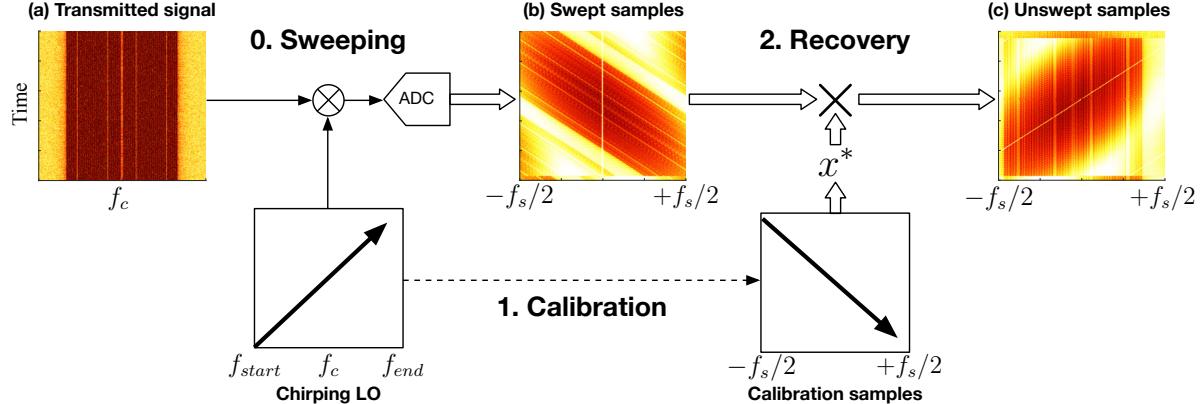


Figure 5: Illustration of the the signal captured by SweepSense at different stages in the receiver.

logic modification that makes the PLL cycle through its VCO bands, and generates the sawtooth waveform to sweep each VCO. Fig. 4 provides a visual overview of both modifications. The hardware schematics and Verilog needed to implement SweepSense will be made open source at the time of publication.

The hardware modification is straightforward for someone with surface mount soldering experience: you need to remove a single passive component from the SDR’s receiver RF frontend, and in its place connect a wire that connects to one of the USRP’s auxiliary Digital to Analog converters.

The FPGA’s frequency synthesizer control logic modification removes all tuning logic, and in its place we add logic to iteratively loop through the selected VCO bands. Also, a new logic module is added to generate a sawtooth waveform and send it to the auxiliary DAC. These two logic modules are designed to operate in sync with the USRP’s ADC sampling clock. This is required because unsweeping the samples requires knowing the configuration of the PLL, including its approximate tuning voltage, while the USRP is collecting each sample.

4 Unsweeping the Samples

Unlike a standard radio which samples with a local oscillator tuned to a fixed center frequency, SweepSense samples are distorted because they are captured while the center frequency is rapidly increasing. To aid in understanding the effect of a chirping local oscillator on captured samples, we begin with a primer on downconversion.

For a received signal $x(t)$ centered at frequency f_c as shown in Fig. 5(a), a standard fixed frequency direct IQ downconversion can be modeled as:

$$x_b(t) = x(t) \times e^{-j2\pi f_c t} \quad (1)$$

Where $x_b(t)$ is the downconverted signal (before base-

band filtering) and f_c is also the frequency of the oscillator. In SweepSense, the oscillator frequency varies with time as $f(t)$. In our implementation, $f(t)$ monotonically increases with time (chirp). Therefore, similar to Eq. 1, a chirp direct IQ downconversion can be modelled as:

$$x_c(t) = x(t) \times e^{-j2\pi f(t)t} \quad (2)$$

This equation shows how sweeping introduces a significant change to the received signal: the frequency with which $x(t)$ is multiplied in SweepSense changes at every instant, and is offset from a fixed frequency oscillator at f_c by $f_c - f(t)$. Since $f(t)$ monotonically increases with the sawtooth waveform connected to the VCO tuning input, the frequency offset continuously decreases with time as shown in Fig. 5(b). The problem is, standard digital signal processing techniques rely on the assumption that the signal is fixed around a constant frequency at all times; therefore, these techniques can not be applied directly to the swept samples captured by SweepSense.

Undoing the sweeping effect requires removing the time-varying frequency offset $f_c - f(t)$ from SweepSense samples at time t , for which $f(t)$ is required. We call this process of undoing the sweeping effects “unsweeping”. Unsweeping involves two steps:

1. Calibration: First, we extract the effect of sweeping ($f(t)$) by sending a known signal: we measure the frequency offset $f_c - f(t)$ introduced by SweepSense at time t .

2. Recovery: Then, we reverse the effect of sweeping by removing the offset $f_c - f(t)$ from the samples captured with SweepSense.

In summary, this method measures the sweeping center frequency, and uses it to recover signals as if they were captured at a fixed frequency.

4.1 Calibration

Why is calibration difficult?

The VCO's frequency increases as the voltage of the sawtooth waveform increases. Intuitively, one may expect that the VCO's frequency is directly related to the input voltage. However, this is not true for an open loop VCO (Section 3). An open loop VCO's frequency does not have a linear relationship with the input voltage : it is also dependent on temperature and other environmental conditions. However, we do know that the frequency increases monotonically as the input voltage increases. Therefore, to calibrate the VCO, we need to find another way to measure the center frequency $f(t)$ of SweepSense at each time instant in a sweep.

Insight and solution

Our insight is, we can calibrate the VCO by sweeping while capturing a tone transmitted at a known frequency. We measure the value of $f(t)$ by sending a tone at frequency f_c ($x_f(t) = e^{j2\pi f_c t}$) and collecting the received samples $x_{cal}(t)$ after the chirped direct downconversion, i.e., $x_{cal}(t) = e^{j2\pi(f_c - f(t))t}$ (Equation 2) as shown in Fig. 6. In summary, we directly capture the varying oscillator frequency in $x_{cal}(t)$. The implementation details of our calibration process appear in Section 6.

Calibration needs to be repeated at many reference tones due to the effect of the narrow-band radio's low-pass baseband filter on $x_{cal}(t)$. This filter suppresses the parts of $x_{cal}(t)$ whose frequencies lie outside the interval $[-F_s/2, F_s/2]$. Since the instantaneous frequency of $x_{cal}(t)$ is $f_c - f(t)$, it is detectable at time t only if $|f(t) - f_c| \leq F_s/2$. Therefore, for a specific tone, we can only calibrate the VCO behavior between $[f_c - F_s/2, f_c + F_s/2]$ using a tone of frequency f_c . To calibrate VCO's behavior at an arbitrary frequency interval $[f_{start}, f_{end}]$, we divide the calibration into chunks of bandwidth F_s and transmit a different reference tone for each chunk. Consecutive tones are each separated in frequency by F_s starting from $f_{start} + F_s/2$. We collect the received samples for all $x_{cal}(t) = e^{j2\pi(f_c - f(t))t}$ where $f_c = f_{start} + k * F_s$ where $k = 1, 2, \dots, (f_{end} - f_{start})/F_s$. This produces calibration data for the behavior of the VCO across the entire sensing bandwidth. This process only needs to be redone when temperature and environmental conditions change significantly.

4.2 Recovery

Next we describe how to use the data gathered in the calibration process to remove the time-varying frequency offset ($f_c - f(t)$). Recall that the downconversion in SweepSense VCO can be modeled as multiplying a chirp with the received signal. We observe that the frequency

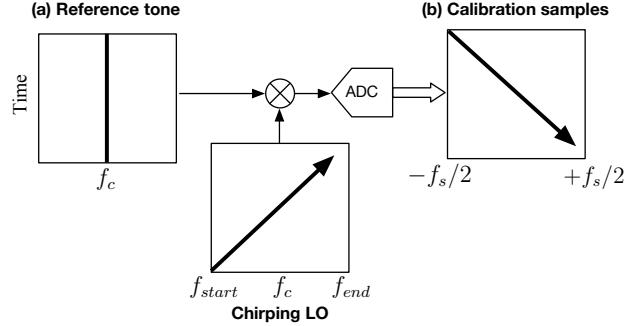


Figure 6: VCO behavior over F_s bandwidth is calibrated by sweeping over a reference frequency and collecting the samples.

of the calibration samples also varies similarly with time, motivating a similar multiplication to remove the effect of the chirp. Indeed, by multiplying the swept samples with the complex conjugate(\cdot^*) of the calibration samples $x_{cal}(t)$, it cancels out the frequency offset. Mathematically, the effect of sweeping cancels as follows:

$$x_c(t) \times x_{cal}^*(t) = [x(t) \times e^{-j2\pi f_c t}] \times e^{j2\pi f_c t - j2\pi f_c t} \\ = x(t) \times e^{-j2\pi f_c t} \quad (3)$$

This process converts a chirped direct downconversion to the corresponding fixed frequency downconversion as in Equation 1. Therefore, signals are recovered as if they were received by a standard fixed frequency receiver. We evaluate the performance of unsweeping in Section 7.

Fig. 7 shows an example of signals captured between 2.380 GHz and 2.480 GHz after their recovery using the calibration data. In this capture, we observe multiple OFDM packets centered at 2.412 GHz (even an acknowledgment packet around 400 μ sec) and a Bluetooth packet at 2.428 GHz. Unlike FMCW spectrum sensors which can only detect signal energy, SweepSense can capture short intervals of the time-domain samples of the transmitted signal. These samples enable SweepSense to distinguish different transmissions, even when they have a similar center frequency and bandwidth. Unsweeping therefore is an improvement to prior high-speed sweeping spectrum sensing architectures (Section 2).

5 Analysis and Inference

In this section, we describe a method to detect modulation scheme and protocol type from swept samples. Conventional detection algorithms for signal classification fixed frequency spectrum sensors rely on capturing a significant portion of the transmission, sometimes even requiring protocol-specific preambles [27]. However, SweepSense only captures a small number of samples for each frequency band. Hence, it is unlikely that

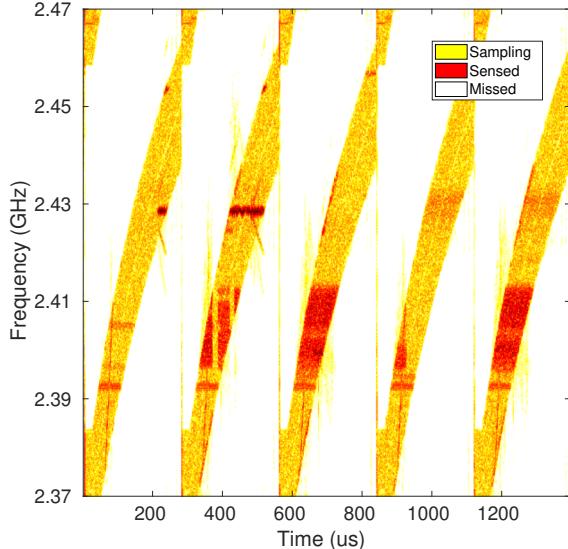


Figure 7: Example of ISM-band transmissions recovered from swept samples.

it will capture a preamble. Also, the open-loop operation of VCO during sweeping introduces additional noise into the signal, making it difficult to perform signal classification. Therefore, we designed a classification algorithm that is resilient to noise, and works even with only a short capture of the signals.

Our algorithm is inspired by cyclostationary analysis [12]. The basic premise behind cyclostationary analysis is that every human-made signal has inherent periodicity associated with it. This periodicity is unique to every protocol, independent of implementation or hardware used. It also can serve as a fingerprint for inference [19]. For example, in WiFi-OFDM, the cyclic prefix (CP) repeats at the start and end of a symbol. SweepSense’s key insight is that this periodicity is retained even when we receive a small portion of transmission filtered in time and frequency. Cyclostationary functions evaluate this periodicity as correlations in time and frequency domains. SweepSense uses these cyclostationarity signatures to build reliable ML models for signal classification. For all our analyses, we use two second-order cyclostationary functions: the Cyclic Auto-correlation Function (CAF) and the Spectral Correlation Function (SCF).

If $x[n]$ is the received signal, the CAF estimator is calculated as follows [8]:

$$R_x^\alpha(\tau) = \sum_{n=-\infty}^{\infty} x[n] [x^*[n-\tau]] e^{-j2\pi\alpha n} \quad (4)$$

The CAF is maximized when the choice of delay (τ) is equal to the time between consecutive repeating patterns in $x[n]$. This causes them to align in the correlation. These maxima occur periodically along n , and the

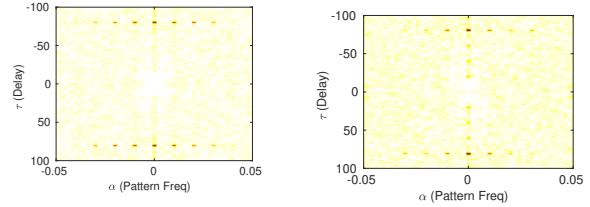


Figure 8: The CAF is visible in SweepSense captures.

term $e^{-j2\pi\alpha n}$ is a transform that brings out the frequency (α) of this periodicity. α may be interpreted as the frequency of repetition of hidden patterns, defined as the pattern frequency. Therefore, CAF peaks at values of τ and α that correspond respectively to the time period and repetition frequency of patterns in $x[n]$. The CAF is particularly useful in analyzing signals like OFDM with repetitive patterns in time (i.e., cyclic-prefixes [38]). The SCF is the Fourier transform of the CAF over τ , making them equivalent representations due to the unitary nature of the transform. The SCF peaks for the same values of α as the CAF and frequency f is the fourier dual of delay τ . The SCF can be efficiently computed due to its representation using FFTs as described below.

Consider L consecutive discrete time windows of $x[n]$, each of length N samples. $X_{lN}(f)$ is the FFT of $x[n]$ for the l^{th} time window. The time-smoothed SCF estimator for this signal is calculated as follows [8]:

$$S_x^\alpha(f) = \frac{1}{LN} \sum_{l=0}^{L-1} X_{IN}(f) X_{IN}^*(f - \alpha) \quad (5)$$

As an illustration, Fig. 8(a) shows the CAF plot of WiFi-OFDM. The x-axis represents pattern frequency (α) and the y-axis represents delay (τ). WiFi symbols are 80 samples long (of which 16 are CP) at 20 MHz sampling rate. Since we sample at 25 Msps, we get 100 samples per symbol (of which 20 are CP). Notice that the CAF peaks at a $\tau = 80$ samples and $\alpha = 0.01$ (normalized to 25 Msps). We also observe peaks in the SCF plot (not shown) at the same α values. Patterns such as these occur in every protocol, we do not need to capture the entire packet to identify them. Indeed, we see in Fig. 8(b) that the CAF of the unswept samples of WiFi-OFDM also exhibits the peaks at same points as the fixed frequency capture. The CAF and SCF are robust due to their highly signal selective nature, magnifying the signal's natural patterns while averaging and suppressing distortions introduced due to sweeping.

For ML-based classification, we extract CAF and SCF features from the unswept signal at a set of precomputed values of α , τ and f . Specifically, we to include values that are at the expected peaks for the protocols that we seek to detect.

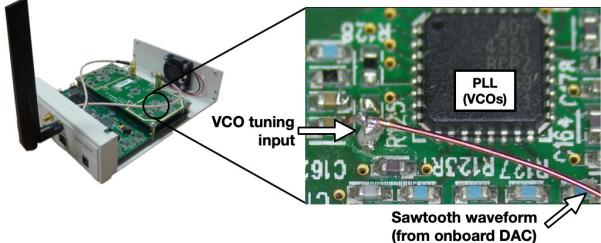


Figure 9: SweepSense requires a single-wire modification to the USRP’s RF frontend to make the PLL chirp.

6 Implementation

Our hardware setup for SweepSense uses a standard off-the-shelf USRP N210 SDR. We implement the LO modification as discussed in Section 3 on both the CBX daughter card which supports 1.2 GHz to 6 GHz and SBX daughter card which supports 400 MHz to 4.4 GHz (shown in Figure 6).

We then make the following modifications to the FPGA logic on the USRP. The voltage ramp used to control the VCO is generated using the AUX-DAC on the daughtercard, which is controlled by the FPGA. Special care is taken to ensure that the voltage generated by the AUX-DAC on the USRP is time synchronised with the baseband ADC samples. The added logic also selects the PLL’s VCO band and RF divider. The user can configure the sweeping bandwidth (VCO band and RF divider selection) and sweep rate (sawtooth voltage ramp slope) from the GNURadio python environment.

For our observations in the ISM band, we use a standard 2.4 GHz omnidirectional antenna, and for our wide-band captures, we use a discone antenna mounted on the roof of the CSE building at UC San Diego. We operate the USRP at a sampling rate of 25 MSps with 16-bit resolution. For the evaluation, the captured samples are streamed, stored on the PC and processed offline.

Calibration and Recovery

The calibration process is as follows: SweepSense transmits a reference tone from the (unmodified) transmit chain of USRP. It receives the tone with the (modified) sweeping receive chain indirectly from leakage between the transmitter and receiver RF paths³. To calibrate across the entire sensing bandwidth, we repeat this process with tones separated by the sampling bandwidth (Section 4.1). For example, we need to run the calibration process 200 times when the sampling bandwidth is 25 MHz and the sweeping bandwidth is 5 GHz. In each of these files consisting calibration data for a different tone, the samples where the sweeping of a VCO

³This is inspired by the USRP’s use of TX/RX leakage to calibrate for I/Q imbalance.

band starts and ends is deterministic since the voltage input to VCO is synchronized with start of ADC sampling. Further, since these tones are separated by F_s the time intervals during which they are received are non-overlapping. Therefore we can combine the calibration data from these multiple tones by just adding the data from each file.

Periodic re-calibration may be necessary due to frequency drift of the VCO, particularly when the ambient temperature significantly changes (details in Section 7.2). However, re-calibration only requires performing one sweep over each of the reference tones. For example, calibrating at a sweeping bandwidth of 5 GHz and rate of 125 μ sec/100 MHz only requires 6.25 milliseconds of downtime.

SweepSense recovers the time-domain samples from the swept samples in real time. This is feasible because recovery only requires performing conjugate multiplication of the swept samples with the calibration samples (Section 4.2).

7 Evaluation

To evaluate the performance of SweepSense as a spectrum sensor we first evaluate the performance of SweepSense with several high-time-resolution spectrum sensing case-studies that normally would require a wide-bandwidth spectrum sensor. Then, we evaluate the limitations of SweepSense with several micro-benchmarks.

We selected the case studies based on the results of a sample full spectrum (0–6 GHz) capture that we performed in the lab. Although there were many occupied bands in this capture, we observed that the 2–3 GHz spectrum was the most dynamic (shown in Figure 10) due to nearby WiFi, Bluetooth, and LTE deployments. In the ISM band (2.4 GHz), we demonstrate that we can detect and classify diverse protocols. In addition, we show how SweepSense can monitor the load on multiple LTE base stations (1.9–2.2 GHz) simultaneously. We conclude the case studies by evaluating the performance of SweepSense as an Environment Sensing Capability (ESC) sensor for the newly shared 3.5 GHz Citizens Broadband Radio Service (CBRS) spectrum [40].

The micro-benchmarks evaluate the frequency distortion and signal to noise ratio (SNR) loss due to the sweep and unsweep processes, and a demonstration of frequency stability across sweeps.

In summary, our evaluation contains the following the results:

- Protocols can be classified based on unswept samples containing partial packets or a few symbols, usually requiring only 25 μ s to classify the signal types in contrast to typical full packet lengths 1–10 ms, an improvement of over 40 ×.

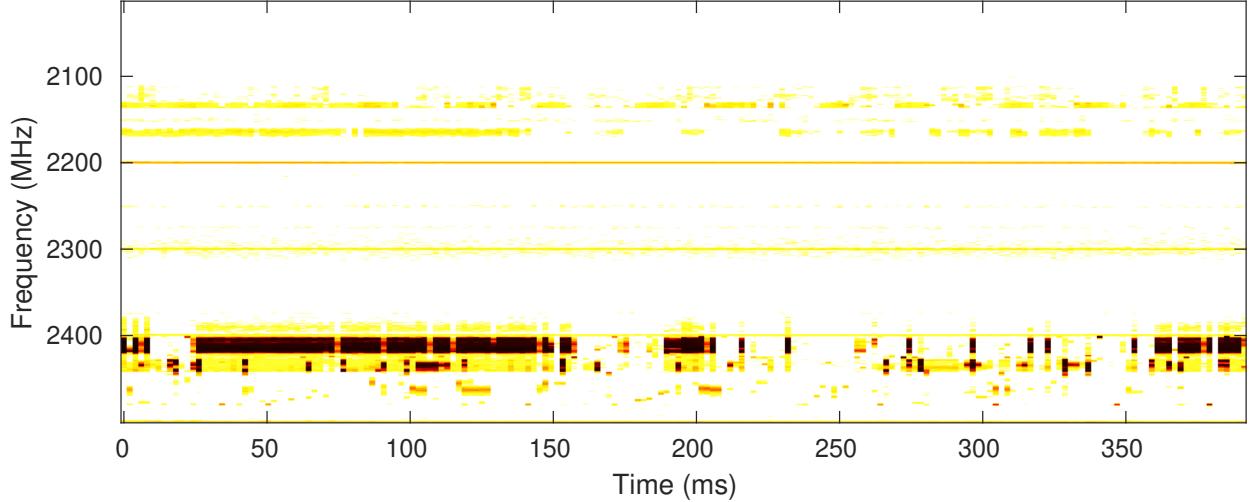


Figure 10: Example of transmissions between 2 and 2.5 GHz captured by SweepSense every 2 ms.

- In contrast to a standard CBX receiver, which takes $500 \mu\text{s}$ to monitor 100 MHz ($125\mu\text{s}$ to capture and retune four times), SweepSense can do it in $125 \mu\text{s}$, a $4\times$ improvement.
- Useful information such as channel utilization can be extracted with 1 ms resolution in highly dynamic and disjoint LTE distributed in a band of 200 MHz.
- Incumbent sensing can be reliably performed over 200 MHz of bandwidth for use in spectrum sharing architectures like CBRS.
- The loss of quality in received samples due to a free-running VCO and the unsweeping mechanism can be characterized and do not limit the use of SweepSense as a spectrum sensor

Our evaluation hardware setup is as described in the previous section. We select VCO bands and sweep rates that best suit the evaluation requirements. In situations where comparisons are required, we use an oracle to provide the ground truth. The oracle is an unmodified USRP (CBX frontend) synchronized with SweepSense using a MIMO cable. The oracle USRP is tuned to a particular frequency, while the SweepSense USRP continuously sweeps multiple bands. We then repeat the experiments while cycling the oracle through all of the relevant frequency bands.

7.1 Case Studies

7.1.1 ISM Protocol Classification

In the first case study we evaluate the performance of SweepSense in differentiating between four common protocols in the ISM band: WiFi-OFDM (802.11g/n),

WiFi-DSSS (802.11b), Bluetooth (BLE), Zigbee (ZB), and no transmission (Gaussian noise). These protocols are diverse in their bandwidth, modulation scheme, and behavior. Both WiFi-DSSS and WiFi-OFDM are relatively wideband but have the same bandwidth (20 MHz) and channel allocation. [20] BLE and ZB are relatively narrowband (2 MHz), and have overlapping, but different channel allocation, making the classification process more difficult [21, 22].

We used a two-level classifier to distinguish between the various protocols. The first level differentiates between narrowband and wideband signals using the Power Spectral Density (PSD). The second level then implements an SVM classifier for the wideband signals and a single layer neural network for narrow band signals [6]. Both of these classifiers take as input vectors the SCD and CAF of the unswept samples within each sweep. For wideband signals, CAF vectors are obtained at cyclic frequency shifts of $k * 0.01$; and for narrowband signals, they are obtained at cyclic frequency shift of $k * 0.0025$. The classifiers were trained using ground truth captures of each protocol captured over the air. The ground truth signals were generated using relevant MATLAB toolboxes or standard compliant scripts, and included signals at a wide range of SNRs. The first classifier (CLASSIFIER_1), differentiates between transmission (noise), ZB, and BLE. The second classifier (CLASSIFIER_2) differentiates between no transmission (noise), WiFi-OFDM, and WiFi-DSSS.

Classification accuracy is used as the primary metric in this evaluation, and is calculated as: the number of sweeps that were classified correctly, divided by the total number of sweeps where the signal was present. We performed the evaluation with a SweepSense receiver capturing signals over-the-air that we transmitted across the

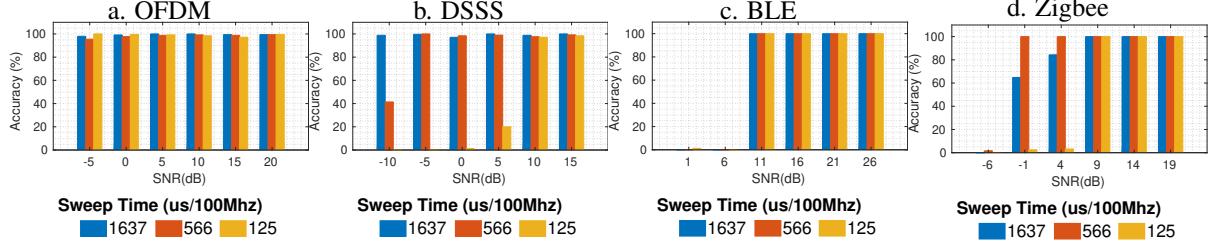


Figure 11: Classification accuracy for ISM protocols across SNRs.

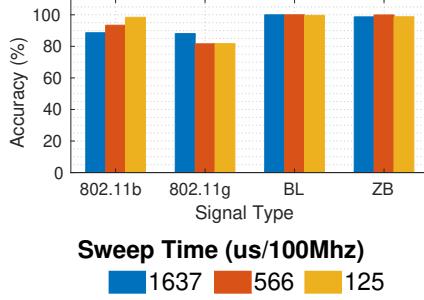


Figure 12: Classification accuracy for ISM protocols across multiple test locations.

entire 100 MHz wide 2.4 GHz ISM band. In each experiment, we transmit the ground truth signals containing a mix of protocols, and they are received simultaneously by co-located SweepSense and oracle USRP receivers. The classifier then operates on the unswept samples from the SweepSense receiver, and the ground truth samples from the oracle USRP. We then calculate the classification accuracy. The receiver setup is then moved around the lab to capture data from multiple locations.

Figure 12 shows the classification accuracy across all four protocols. The average classification accuracy for signals with the highest transmit SNR, across all protocols, is 95%. While operating on the fastest sweep rate of 125 μ s per 100 MHz, we repeat the experiment while varying the sweep rate and transmit SNR to understand the classification accuracy’s dependence on these parameters. Figure 11 shows that SweepSense classification accuracy is high for signals with decodable SNR even at the fastest sweep rate of 125 μ s per 100 MHz: CLASSIFIER_1 can detect and classify signals with 95% accuracy at even low SNR with sweep as fast as 125 μ s per 100 MHz. CLASSIFIER_2 can detect and classify signals with 90% accuracy at even low SNR with a sweep as fast as 125 μ s per 100 MHz.

We note that the noise suppression properties of cyclo-stationary analysis enables us to correctly classify signals even when they’re sometimes below the noise floor. The accuracy drops as the rate of sweep increases. We see that the drop in accuracy is because faster sweep rates lead to a smaller number of samples (the fastest sweep yields only 3125 samples in every 100 MHz). It also

leads to larger distortions, both of which negatively affect cyclo-stationary signatures. It should be noted that these signatures are preserved at lower sweep rates, despite the frequency distortions.

7.1.2 LTE Channel Utilization

The LTE bands are allocated to specific service providers, but even within a service provider, the bands are across a wide frequency range in the spectrum. Also, LTE base stations schedule traffic at a millisecond granularity. Therefore, monitoring the load across many LTE base stations demonstrates SweepSense’s ability to capture time dynamics of signals across a wide bandwidth. Specifically, we show that SweepSense can simultaneously monitor the load of a set of disjoint LTE downlink channels (with a total bandwidth of 75 MHz), spread over the 1.9 GHz and the 2.1 GHz bands.

Our experimental setup is as follows: we connect the SweepSense receiver to a wideband discone antenna on the roof of the building. SweepSense is configured to sense 1.9 GHz to 2.1 GHz spectrum in three sweeps, each is 80MHz at the rate of 375 μ s per 100MHz. We captured several seconds of sweeps during a peak hour in the evening.

Since the LTE protocol only puts energy on subcarriers when downlink traffic is transmitted, the energy of each subcarrier directly correlates with the downlink channel usage [1]. Therefore, we use a short-term Fourier transform on the unswept samples and report load as average power levels detected in the respective bands. The maximum power level obtained over all our experiments is used as the normalization factor to obtain the power corresponding to the maximum load. Fig. 13 shows a snapshot of simultaneously measured load of five LTE base stations with 0.9 ms granularity (less than the scheduling interval) per LTE base station. Surprisingly, even at peak hours, the load across base stations is very uneven.

7.1.3 CBRS ESC Sensor

The FCC requires spectrum sensing in the CBRS band to detect and avoid interfering with incumbent radar transmissions. Highly reliable ESC sensors that moni-

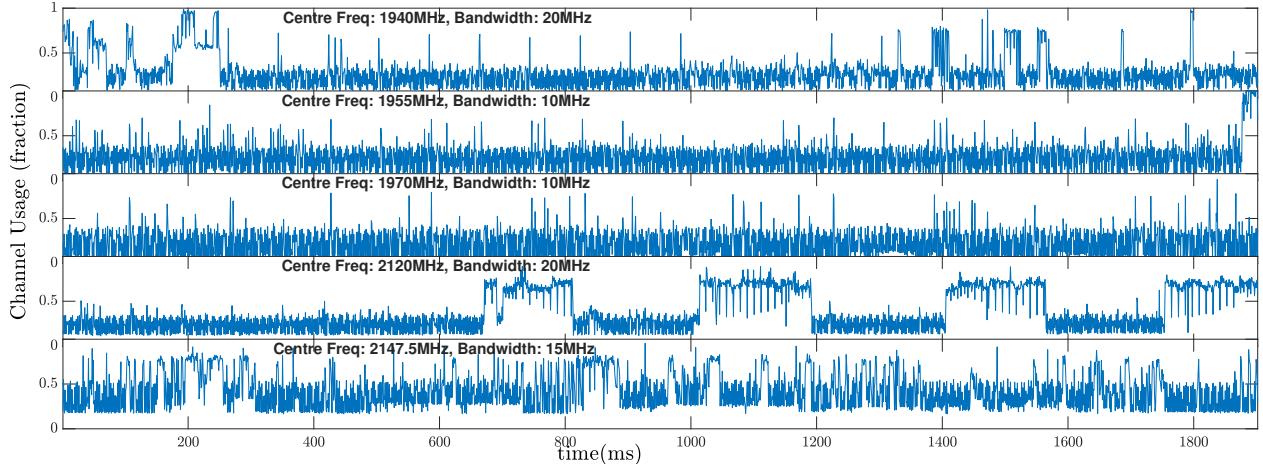


Figure 13: Downlink channel occupancy of five LTE base stations as observed simultaneously by SweepSense.

tor the entire spectrum for incumbent transmissions, are arguably one of the most critical parts of the rules for using the CBRS spectrum [40].

We evaluate the capability of SweepSense as an ESC sensor. Our experiment is to detect the “Bin 1 Lite” radar waveform as per the official testing and certification procedures for ESC sensors [11]: this radar type closely resembles widely deployed maritime pulse radar. We use MATLAB to generate the radar signals and add Gaussian noise (GN) according to the specified relative power levels in [11]. The samples are transmitted to the SweepSense USRP with a Vector Signal Generator (Keysight N5182B) at calibrated power levels. The signal generator is directly connected to the SweepSense receiver with RF coax. SweepSense is configured to sweep 3480 MHz - 3680 MHz every 1.3 ms. We sweep the spectrum multiple times within one radar burst interval, increasing chances of detection. In each experiment, we initiate the SweepSense capture for 10 seconds and then trigger the signal generator ten times. Our sensing algorithm declares radar events based on peaks in the short term Fourier transform of the unswept signal. Since the SweepSense USRP is not designed to have a low noise floor, the actual power levels used in this study are 9 dB/MHz higher (-80dBm/Hz for radar pulses and -100dBm/Hz for GN) than the respective values in [11].

Table 1 summarizes the radar detection performance of SweepSense. We observe that SweepSense can achieve 99.5% accuracy with a very simple receiver. Added to this, we also demonstrate that SweepSense can function as an ESC sensor over *double* the required bandwidth, motivating broader spectrum sharing applications in the future. In summary, SweepSense is effective in detecting fleeting signals (e.g., radar).

Radar Type	Pulse Width (μs)	Pulses per second	Pulses per burst	Detection accuracy
Bin 1 Lite	0.8	1000	19	99.5% (398/400)

Table 1: ESC radar classification accuracy

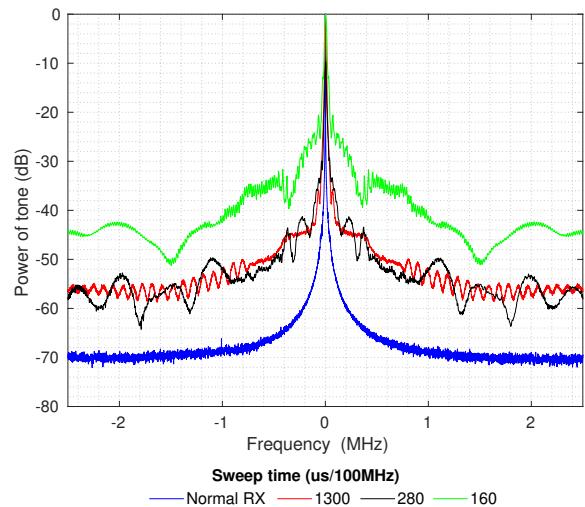


Figure 14: PSD characteristics of a fixed tone captured using SweepSense across multiple sweep rates.

7.2 Micro benchmarks

Frequency stability and phase noise are typical measurements used to characterize radios. Since the sweep-unsweep process recovers fixed frequency samples, we can benchmark the performance of SweepSense using

these standard metrics. We isolate the loss in performance due to sweeping by comparing with the performance of un-modified USRP SDR radios. In each of these evaluations, we connected a signal generator that outputs a single frequency tone into both the SweepSense USRP and the oracle USRP through identical RF paths. SweepSense sweeps the relevant band and uses pre-captured calibration data to obtain the unswept samples at different sweep rates.

Since the VCO in SweepSense is operating in open-loop mode, we observe a frequency drift over time (shown in Fig. 15). The rapid rise and subsequent settling of the frequency is due to the oscillator warming up and settling on its stable operating temperature after power-on. We observe that the settling time is consistent: it takes the same amount of time every time we power on the USRP (~ 1500 s), and it is also consistent across multiple VCO bands and sweep rates. Although the VCO takes many minutes to settle, this is only a one-time event at power-on and does not affect the performance of a SweepSense sensor after it has warmed up or switched bands.

Next, we characterize the performance of the SweepSense un-sweeping and noise distortion added due to un-sweeping compared to fixed frequency receiver. On a standard fixed frequency radio, the PLL reduces the phase noise of the VCO while it locks the frequency to the desired value. Since we removed the PLL lock loop for implementing SweepSense, it is essential to characterize the distortion created by the open loop VCO being controlled by the sawtooth signal from an external DAC. All measurements are taken after the frequency drift settles. We compare Power Spectral Density of the unswept tone at different sweep rates against samples received by the oracle USRP in Fig. 14. We see that the phase noise floor rises by ~ 10 dB for slower sweep rates and the skirt around 0 Hz starts increasing for higher sweep rates, compared to the oracle. An ideal response would have a clean tone with no skirt or spreading. Sweeping faster, therefore, comes at the cost of limited frequency resolution.

8 Limitation: SNR Loss and Inference

The phase noise of SweepSense will lead to a loss in signal quality. Phase noise is multiplicative noise, i.e., SNR loss due to phase noise depends on the signal strength of the transmission. If the transmission has 10 dB of SNR, i.e., the noise floor would be 10 dB lower than signal; then the effect of phase noise will be insignificant (less than 1 dB loss). Recall that the classification evaluation results demonstrate that even with a weak signal (e.g., 5 dB SNR), SweepSense can classify the 20 MHz OFDM signal with just a 25 μ sec capture (sampled at 25

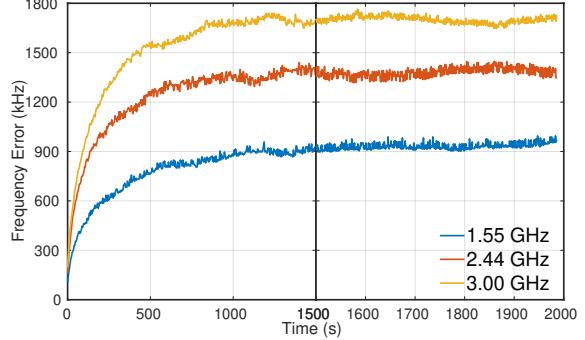


Figure 15: Frequency error in VCO output vs time of capture. The VCO reaches temperature stability in (~ 1500 s).

Msps). This means that even with such high phase noise, the inference algorithms still perform well. In summary, SweepSense has high distortion due to phase noise, but even then it still performs well for signal detection.

9 Conclusion

SweepSense presents the first spectrum sensor which can rapidly sweep the entire terrestrial spectrum with low-cost SDRs, while providing detailed measurements including transmitter classification and utilization. SweepSense achieves this by making a single-wire modification to the frontend of SDRs such as the USRP, allowing us to make this improvement to current deployments of USRP radios in multiple wide-scale deployments such as CityScape [35], and the Microsoft Spectrum Observatory [31].

In addition to spectrum sharing, SweepSense can be used for data mining, since communication signals are generated when humans, machines, and objects change their state. In the future we envision the community adding other spectrum analysis techniques beyond classifying communication protocol, namely transmitter localization.

References

- [1] 3GPP Consortium. 3GPP Specification series 36. <http://www.3gpp.org/dynareport/36-series.htm>.
- [2] O. Abari, F. Lim, F. Chen, and V. Stojanović. Why analog-to-information converters suffer in high-bandwidth sparse signal applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(9):2273–2284, Sep. 2013.
- [3] Anritsu. MS2840A Spectrum Analyzer. <https://www.anritsu.com/en-IN/test-measurement/solutions/ms2840a-066/>.

- [4] M. R. Avendi, K. Haghghi, A. Panahi, and M. Viberg. A NLLS based sub-nyquist rate spectrum sensing for wideband cognitive radio. *CoRR*, abs/1408.4544, 2014.
- [5] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White space networking with Wi-Fi like connectivity. In *Proc. ACM SIGCOMM*, 2009.
- [6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.
- [7] A. M. Cavalcante, E. Almeida, R. D. Vieira, S. Choudhury, E. Tuomaala, K. Doppler, F. Chaves, R. C. D. Paiva, and F. Abinader. Performance evaluation of lte and wi-fi coexistence in unlicensed bands. In *IEEE Vehicular Technology Conference (VTC Spring)*, June 2013.
- [8] chad spooner. Cyclostationarity blog. <https://cyclostationary.blog/>.
- [9] A. A. Cheema and S. Salous. Digital FMCW for ultrawideband spectrum sensing. *Radio Science*, 51(8):1413–1420, Aug 2016.
- [10] M. F. Duarte and R. G. Baraniuk. Spectral compressive sensing. *Applied and Computational Harmonic Analysis*, 35(1):111 – 129, 2013.
- [11] F. H. Sanders, J. E. Carroll, G. A. Sanders, R. L. Sole, J. S. Devereux, and E. F. Drocella. “Procedures for laboratory testing of environmental sensing capability sensor devices” National Telecommunications and Information Administration, Technical Memorandum TM 18-527”. <https://www.its.blrdoc.gov/publications/3184.aspx>, Nov. 2017.
- [12] W. A. Gardner. The spectral correlation theory of cyclostationary time-series. *Signal Processing*, 11(1), July 1986.
- [13] B. Ghazi, H. Hassanieh, P. Indyk, D. Katabi, E. Price, and L. Shi. Sample-optimal average-case sparse fourier transform in two dimensions. *CoRR*, abs/1303.1209, 2013.
- [14] A. P. Goodson. A multi-function, broad band, high dynamic range RF receiver. Technical report, On-eRadio, 2017.
- [15] T. Harrold, R. Cepeda, and M. Beach. Long-term measurements of spectrum occupancy characteristics. In *New Frontiers in Dynamic Spectrum Access Networks (DySPAN), 2011 IEEE Symposium on*, pages 83–89. IEEE, 2011.
- [16] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Nearly optimal sparse fourier transform. *CoRR*, abs/1201.2501, 2012.
- [17] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Simple and practical algorithm for sparse fourier transform. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’12, pages 1183–1194, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.
- [18] H. Hassanieh, L. Shi, O. Abari, E. Hamed, and D. Katabi. GHz-Wide sensing and decoding using the sparse fourier transform. In *Proc. IEEE Conference on Computer Communications (INFOCOM)*, 2014.
- [19] S. S. Hong and S. R. Katti. DOF: a local wireless information plane. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Toronto, ON, Canada, August 15-19, 2011, pages 230–241, 2011.
- [20] IEEE. IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”. (2016 revision). IEEE-SA. 14 December 2016. <http://ieeexplore.ieee.org/document/7786995/>.
- [21] IEEE Standard. ”802.15.4k-2013 - IEEE Standard for Local and metropolitan area networks—Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)—Amendment 5: Physical Layer Specifications for Low Energy, Critical Infrastructure Monitoring Networks.”. <https://ieeexplore.ieee.org/document/6581828/>.
- [22] IEEE Standard. ”IEEE Std 802.15.1-2005 – IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (W Pans)”. ieeexplore.ieee.org.
- [23] M. H. Islam, C. L. Koh, S. W. Oh, X. Qing, Y. Y. Lai, C. Wang, Y. Liang, B. E. Toh, F. Chin, G. L. Tan, and W. Toh. Spectrum survey in singapore: Occupancy measurements and analyses. In *2008 3rd International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CrownCom 2008)*, pages 1–7, May 2008.

- [24] A. P. Iyer, K. Chintalapudi, V. Navda, R. Ramjee, V. N. Padmanabhan, and C. R. Murthy. SpecNet: Spectrum sensing sans frontières. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
- [25] H. Kim and K. G. Shin. Efficient discovery of spectrum opportunities with mac-layer sensing in cognitive radio networks. *IEEE transactions on mobile computing*, 7(5):533–545, 2008.
- [26] H. Kim and K. G. Shin. Fast discovery of spectrum opportunities in cognitive radio networks. In *New Frontiers in Dynamic Spectrum Access Networks, 2008. DySPAN 2008. 3rd IEEE Symposium on*, pages 1–12. IEEE, 2008.
- [27] K. Lakshminarayanan, S. Sapra, S. Seshan, and P. Steenkiste. Rfdump: An architecture for monitoring the wireless ether. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, CoNEXT ’09*, pages 253–264, New York, NY, USA, 2009. ACM.
- [28] M. Lopez-Benitez, A. Umbert, and F. Casadevall. Evaluation of spectrum occupancy in spain for cognitive radio applications. In *Vehicular technology conference, 2009. VTC Spring 2009. IEEE 69th*, pages 1–5. IEEE, 2009.
- [29] Maxim Integrated. 23.5MHz to 6000 MHz Fractional Integer-N synthesizer/VCO. <http://hforsten.com/third-version-of-homemade-6-ghz-fmcw-radar.html>.
- [30] MetaGeek. Wi-Spy and Chanalyzer. <https://www.metageek.com/products/wi-spy/>.
- [31] Microsoft. Spectrum Observatory. <http://observatory.microsoftspectrum.com/>.
- [32] OSCOR. Blue Spectrum Analyzer. <https://reiusa.net/rf-detection/oscor-blue-spectrum-analyzer/>.
- [33] K. Qaraqe, H. Celebi, M. Alouini, A. El-Saigh, L. Abuhantash, M. Al-Mulla, O. Al-Mulla, A. Jolo, and A. Ahmed. Measurement and analysis of wideband spectrum utilization in indoor and outdoor environments. In *International Conference on Communications Technologies (ICCT 2010)*. Citeseer, 2010.
- [34] S. Rayanchu, A. Patro, and S. Banerjee. Airshark: Detecting non-WiFi RF devices using commodity WiFi hardware. In *Proc. ACM Internet Measurement Conference (IMC)*, 2011.
- [35] S. Roy, K. Shin, A. Ashok, M. McHenry, G. Vigil, S. Kannam, and D. Aragon. Cityscape: A metro-area spectrum observatory. In *Proc. IEEE International Conference on Computer Communication and Networks (ICCCN)*, 2017.
- [36] S. Salous, N. Nikandrou, and N. Bajj. Digital techniques for mobile radio chirp sounders. *IEE Proceedings-Communications*, 145(3):191–196, 1998.
- [37] L. Shi, P. Bahl, and D. Katabi. Beyond sensing: Multi-GHz realtime spectrum analytics. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.
- [38] P. D. Sutton, K. E. Nolan, and L. E. Doyle. Cyclostationary signatures in practical cognitive radio applications. *IEEE Journal on Selected Areas in Communications*, 26(1):13–24, Jan 2008.
- [39] Z. Tian and G. B. Giannakis. Compressed sensing for wideband cognitive radios. Technical report, MICHIGAN TECHNOLOGICAL UNIV HOUGHTON, 2007.
- [40] U.S. Government. CFR title 47 section 96.67 Environmental Sensing Capability.
- [41] M. Wellens, J. Wu, and P. Mähönen. Evaluation of spectrum occupancy in indoor and outdoor scenario in the context of cognitive radio. In *CrownCom*, pages 420–427, 2007.
- [42] J. Xue, Z. Feng, and P. Zhang. Spectrum occupancy measurements and analysis in beijing. *IERI Procedia*, 4:295–302, 2013.
- [43] S. Yoon, L. E. Li, S. C. Liew, R. R. Choudhury, I. Rhee, and K. Tan. Quicksense: Fast and energy-efficient channel sensing for dynamic spectrum access networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2247–2255. IEEE, 2013.
- [44] T. Yucek and H. Arslan. A survey of spectrum sensing algorithms for cognitive radio applications. *IEEE communications surveys & tutorials*, 11(1):116–130, 2009.

Slim: OS Kernel Support for a Low-Overhead Container Overlay Network

Danyang Zhuo Kaiyuan Zhang Yibo Zhu^{†*} Hongqiang Harry Liu[#]
Matthew Rockett Arvind Krishnamurthy Thomas Anderson

University of Washington

[†] Microsoft Research

[#] Alibaba

Abstract

Containers have become the de facto method for hosting large-scale distributed applications. Container overlay networks are essential to providing portability for containers, yet they impose significant overhead in terms of throughput, latency, and CPU utilization. The key problem is a reliance on packet transformation to implement network virtualization. As a result, each packet has to traverse the network stack twice in both the sender and the receiver's host OS kernel. We have designed and implemented Slim, a low-overhead container overlay network that implements network virtualization by manipulating connection-level metadata. Our solution maintains compatibility with today's containerized applications. Evaluation results show that Slim improves the throughput of an in-memory key-value store by 71% while reducing the latency by 42%. Slim reduces the CPU utilization of the in-memory key-value store by 56%. Slim also reduces the CPU utilization of a web server by 22%-24%, a database server by 22%, and a stream processing framework by 10%.

1 Introduction

Containers [6] have quickly become the de facto method to manage and deploy large-scale distributed applications, including in-memory key-value stores [32], web servers [36], databases [45], and data processing frameworks [1, 26]. Containers are attractive because they are lightweight and portable. A single physical machine can easily host more than ten times as many containers as standard virtual machines [30], resulting in substantial cost savings.

Container overlay networks—a key component in providing portability for distributed containerized applications—allow a set of containers to communicate using their own independent IP addresses and port numbers, no matter where they are assigned or which other containers reside on the same physical machines. The overlay network removes the burden of coordinating ports and IP addresses between application developers, and vastly simplifies migrating legacy enterprise applications to the cloud [14]. Today, container orchestrators, such as Docker Swarm [9], require the usage of overlay network for hosting containerized applications.

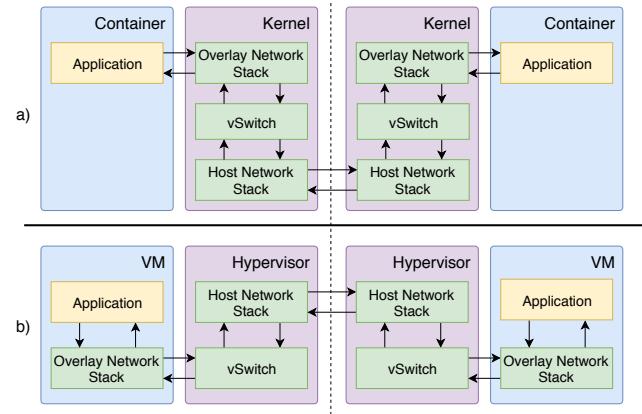


Figure 1: *Packet flow in: (a) today's container overlay networks, (b) overlay networks for virtual machines.*

However, container overlay networks impose significant overhead. Our benchmarks show that, compared to a host network connection, the throughput of an overlay network connection is 23-48% less, the packet-level latency is 34-85% higher, and the CPU utilization is 93% more. (See §2.2.) Known optimization techniques (e.g., packet steering [40] and hardware support for virtualization [22, 14]) only partly address these issues.

The key problem is that today's container overlay networks depend on multiple packet transformations within the OS for network virtualization (Figure 1a). This means each packet has to traverse network stack **twice** and also a virtual switch on both the sender and the receiver side. Take sending a packet as an example. A packet sent by a container application first traverses the overlay network stack on the virtual network interface. The packet then traverses a virtual switch for packet transformation (e.g., adding host network headers). Finally, the packet traverses the host network stack, and is sent out on the host network interface. On the receiving server, these layers are repeated in the opposite order.

This design largely resembles the overlay network for virtual machines (Figure 1b). Because a virtual machine has its own network stack, the hypervisor has to send/receive raw overlay packets without the context of network connections. However, for containers, the OS kernel has full knowledge of each network connection.

In this paper, we ask whether we can design and implement a container overlay network, where packets go through

*Yibo now works at Bytedance.

the OS kernel’s network stack **only once**. This requires us to remove packet transformation from the overlay network’s data-plane. Instead, we implement network virtualization by manipulating connection-level metadata at connection setup time, saving CPU cycles and reducing packet latency.

Realizing such a container overlay network is challenging because: (1) network virtualization has to be compatible with today’s unmodified containerized applications; (2) we need to support the same networking policies currently enforced by today’s container overlay network on the data-plane; and (3) we need to enforce the same security model as in today’s container overlay networks.

We design and implement Slim, a low-overhead container overlay network that provides network virtualization by manipulating connection-level metadata. Our evaluations show that Slim improves the throughput of an in-memory key-value store, Memcached [32], by 71% and reduces its latency by 42%, compared to a well-tuned container overlay network based on packet transformation. Slim reduces the CPU utilization of Memcached by 56%. Slim also reduces the CPU utilization of a web server, Nginx [36], by 22%-24%; a database server, PostgreSQL [45], by 22%; and a stream processing framework, Apache Kafka [1, 26], by 10%. However, Slim adds complexity to connection setup, resulting in 106% longer connection setup time. Other limitations of Slim: Slim supports quiescent container migration, but not container live migration; connection-based network policies but not packet-based network policies; and TCP, defaulting to standard processing for UDP sockets. (See §7.)

The paper makes the following contributions:

- Benchmarking of existing container overlay network with several data-plane optimizations. We identify per-packet processing costs (e.g., packet transformation, extra traversal of network stack) as the main bottleneck in today’s container overlay network. (See §2.2, §2.3.)
- Design and implementation of Slim, a solution that manipulates connection-level metadata to achieve network virtualization. Slim is compatible with today’s containerized applications and standard OS kernels. Slim supports various network policies and guarantees the same security model as that of today’s container overlay network. (See §4.)
- Demonstration of the benefits of Slim for a wide range of popular containerized applications, including an in-memory key-value store, a web server, a database server, and a stream processing framework. (See §6.)

Fundamentally, Slim integrates efficient virtualization into the OS kernel’s networking stack. A modern OS kernel already has efficient native support to virtualize file systems (using *mount* namespace) and other OS components (e.g., process id, user group). The network stack is the remaining performance gap for efficient container virtualization. Slim bridges this gap.

Mode	Applications use	Routing uses
Bridge	Container IP	—
Host	Host IP	Host IP
Macvlan	Container IP	Container IP
Overlay	Container IP	Host IP

Table 1: Container networking mode comparison.

2 Background

We first describe the architecture of traditional container overlay networks and why they are useful for containerized applications. We then quantify the overhead of today’s container overlay network solutions in terms of throughput, latency, and CPU utilization. Finally, we show that the overhead is significant even after applying known overhead reduction techniques (e.g., packet steering [40]).

2.1 Container Overlay Network

Containers typically have four options for communication: bridge mode, host mode, macvlan mode, and overlay mode. Table 1 shows the comparison between different modes in terms of the IP addresses used by containerized applications and routing in the host network. Bridge mode is used exclusively for containers communicating on the same host. With bridge mode, each container has an independent IP address, and the OS kernel routes traffic between different containers.

How can we enable communication between containers on different hosts? With host mode, containers directly use the IP address of their host network interface. The network performance of host mode is close to the performance of any process that directly uses the host OS’s network stack. However, host mode creates many management and deployment challenges. First, containers cannot be configured with their own IP addresses; they must use the IP address of the host network interface. This complicates porting: distributed applications must be re-written to discover and use the host IP addresses, and if containers can migrate (e.g., after a checkpoint), the application must be able to adapt to dynamic changes in their IP address. Worse, because all containers on the same host share the same host IP address, only one container can bind to a given port (e.g., port 80), resulting in complex coordination between different applications running on the same host. In fact, container orchestrators, such as Kubernetes, do not allow usage of host mode [27] due to these issues.

Macvlan mode or similar hardware mechanisms (e.g., SR-IOV) allow containers to have their own IP addresses different from their hosts. Macvlan or SR-IOV allow the physical NIC to emulate multiple NICs each with a different MAC address and IP address. Macvlan¹ extends the host network into the containers by making the container IP routable on the host network. However, this approach fundamentally

¹There are software approaches (e.g., Calico [3]) to extend the host network into containers. They have the same problem as macvlan.

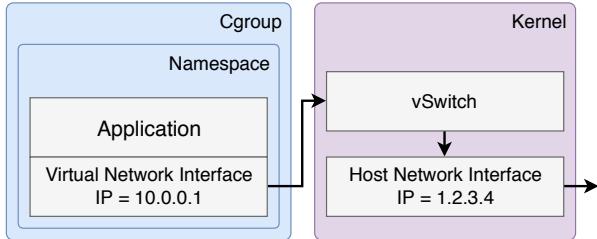


Figure 2: Architecture of container overlay network.

complicates data center network routing. Let’s say a distributed application with IP addresses IP 1.2.3.[1-10] is not co-located on the same rack, or starts co-located but then some containers are migrated. Then the host IP addresses will not be contiguous, e.g., one might be on host 5.6.7.8 and another might be on host 9.10.11.12. Macvlan requires the cloud provider to change its core network routing to redirect traffic with destination IP 1.2.3.[1-10] to 5.6.7.8 and 9.10.11.12, potentially requiring a separate routing table entry for each of the millions of containers running in the data center. Another limitation is that containers must choose IP addresses that do not overlap with the IP addresses of any other container (or host). Because of these complications, today, most cloud providers block macvlan mode [29].

To avoid interference with routing on the host network, the popular choice is to use overlay mode. This is the analog of a virtual machine, but for a group of containers—each application has its own network namespace with no impact or visibility into the choices made by other containers or the host network. A virtual network interface (assigned an IP address chosen by the application) is created per-container. The virtual network interface is connected to the outside world via a virtual switch (e.g., Open vSwitch [42]) inside the OS kernel. Overlay packets are encapsulated with host network headers when routed on the host network. This lets the container overlay network have its own IP address space and network configuration that is disjoint from that of the host network; each can be managed completely independently. Many container overlay network solutions are available today—such as Weave [52], Flannel [15], and Docker Overlay [8]—all of which share similar internal architectures.

Figure 2 presents a high-level system diagram of a container overlay network that uses packet transformation to implement network virtualization. It shows an OS kernel and a container built with namespaces and cgroups. Namespace isolation prevents a containerized application from accessing the host network interface. Cgroups allow fine-grained control on the total amount of resources (e.g., CPU, memory, and network) that the application inside the container can consume.

The key component of a container overlay network is a virtual switch inside the kernel (Figure 2). The virtual switch has two main functionalities: (1) network bridging, allowing containers on the same host to communicate, and (2) net-

work tunneling to enable overlay traffic to travel across the physical network. The virtual switch is typically configured using the Open vSwitch kernel module [42] with VXLAN as the tunneling protocol.

To enforce various network policies (e.g., access control, rate limiting, and quality of service), a network operator or a container orchestrator [27, 9, 18] issues policy updates to the virtual network interface or the virtual switch. For example, firewall rules are typically implemented via *iptables* [20], and rate limiting and quality of service (QoS) can also be configured inside the Open vSwitch kernel module. These rules are typically specified in terms of the application’s virtual IP addresses, rather than the host’s IP addresses which can change depending on where the container is assigned.

The hosts running a set of containers in an overlay network must maintain a consistent global network view (e.g., virtual to physical IP mappings) across hosts. They typically do this using an external, fault-tolerant distributed datastore [13] or gossiping protocols.

2.2 Overhead in Container Overlay Networks

The overhead of today’s container overlay networks comes from per-packet processing (e.g., packet transformation, extra traversal of the network stack) inside the OS kernel.

2.2.1 Journey of an Overlay Network Packet

In our example (Figure 2), assume that a TCP connection has previously been established between 10.0.0.1 and 10.0.0.2. Now, the container sends a packet to 10.0.0.2 through this connection. The OS kernel’s overlay network stack first writes the virtual destination IP address 10.0.0.2 and source IP address 10.0.0.1 on the packet header. The OS kernel also writes the Ethernet header of the packet to make the packet a proper Ethernet frame. The Ethernet frame traverses a virtual Ethernet link to the virtual switch’s input buffer.

The virtual switch recognizes the IP address 10.0.0.2 inside the Ethernet frame as that of a container on a remote host. It adds a physical IP header to the Ethernet frame using host source and destination addresses from its routing table. The packet now has both a physical and a virtual header. On the host network, the packet is simply a UDP packet (assuming the tunneling protocol is VXLAN) and its UDP payload is the Ethernet frame. The OS kernel then delivers the encapsulated packet to the wire using the host network stack.

The receiving pipeline is the same except that the virtual switch removes the host network header instead of adding one. The receiving side receives the exact same Ethernet frame from the sending side’s virtual network interface.

We can thus see why the overlay network is expensive: delivering a packet on the overlay network requires one extra traversal of the network stack and also packet encapsulation and decapsulation.

Setup	Throughput (Gbps)	RTT (μ s)
Intra, Host	48.4 ± 0.7	5.9 ± 0.2
Intra, Overlay	37.4 ± 0.8 (23%)	7.9 ± 0.2 (34%)
Inter, Host	26.8 ± 0.1	11.3 ± 0.2
Inter, Overlay	14.0 ± 0.4 (48%)	20.9 ± 0.3 (85%)

Table 2: Throughput and latency of a single TCP connection on a container overlay network, compared with that using host mode. Intra is a connection on the same physical machine; Inter is a connection between two different physical machines over a 40 Gbps link. The numbers followed by \pm show the standard deviations. The numbers in parentheses show the relative slowdown compared with using host mode.

2.2.2 Quantifying Overhead

We give a detailed breakdown of the overhead in one popular container overlay network implementation, Weave [52]. Our testbed consists of two machines with Intel Xeon E5-2680 (12 physical cores, 2.5 GHz). The machines use hyper-threading and therefore each has 24 virtual cores. Each machine runs Linux version 4.4 and has a 40 Gbps Intel XL710 NIC. The two machines are directly connected via a 40 Gbps link. The physical NIC is configured to use Receive Side Scaling (RSS). In all of our experiments, we do not change the configuration of the physical NICs.

We create an overlay network with Weave’s fast data-plane mode (similar to the architecture in Figure 2). We use *iperf3* [19] to create a single TCP connection and study TCP throughput atop the container overlay network. We use *NPtcp* [39] to measure packet-level latency. For comparison, we also perform the same test using host mode container networking. In all of our experiments, we keep the CPU in maximum clock frequency (using Intel P-State driver [47]).

The overhead of the container overlay network is significant. We compare TCP flow throughput and packet-level latency under four different settings. Table 2 shows average TCP flow throughput with maximum ethernet frame size over a 10-second interval and the round trip latency for 32-byte TCP packets for 10 tests. For two containers on the same host, TCP throughput reduces by 23% and latency increases by 34%. For containers across physical machines, TCP throughput reduces by almost half (48%) and latency increases by 85%. Intra-host container overlay network has lower overheads because packet encapsulation is not needed.

To understand the source of the main bottleneck, we measure CPU utilization with a standard Linux kernel CPU profiling tool, *mpstat*. We specifically inspect the overlay network across two different physical machines. We set the speed of the TCP connection to 10 Gbps and then use *mpstat* to identify where CPU cycles are spent for 10 tests where each test lasts 10 seconds. Figure 3 shows the overall CPU utilization and the breakdown. Compared with using a direct host connection, in the default mode (Random IRQ load bal-

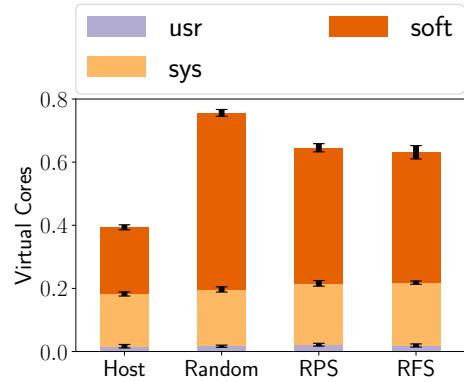


Figure 3: CPU utilization under different overlay network setups measured by number of virtual cores used for a single 10 Gbps TCP connection. The CPU cycles are spent: in user-level application (usr), inside kernel but excluding interrupt handling (sys), and serving software interrupts (soft). Error bars denote standard deviations.

ancing), the overlay network increases CPU utilization (relatively) by 93%. RPS (receive packet steering) and RFS (receive flow steering) are two optimizations we have done to Weave. (See §2.3.)

The main CPU overhead of the overlay network comes from serving software interrupts; in the default overlay setting, it corresponds to 0.56 virtual cores. The reason why the extra CPU utilization is in the software interrupt handling category is that packet transformation and the traversal of the extra network stack is not directly associated with a system call. These tasks are offloaded to per-core dedicated *softirq* thread. For comparison, using the host mode, only 0.21 virtual cores are spent on serving software interrupts. This difference in CPU utilization captures the extra CPU cycles wasted on traversing the network stack one extra time and packet transformation. Note here we do not separate the CPU utilization due to the virtual switch and due to the extra network stack traversal. Our solution, Slim, removes both these two components from the container overlay network data-plane at the same time, so understanding how much CPU utilization these two components consume combined is sufficient.

In §2.3, we show that existing techniques (e.g., packet steering) can address some of the performance issues of a container overlay network. However, significant overhead still remains.

2.3 Fine-Tuning Data-plane

There are several known techniques to reduce the data-plane overhead. Packet steering creates multiple queues, each per CPU core, for a network interface and uses consistent hashing to map packets to different queues. In this way, packets in the same network connection are processed only on a single core. Different cores therefore do not have to access the same queue, removing the overhead due to multi-core

Setup	Throughput (Gbps)	RTT (μs)
Random LB	14.0 ± 0.4 (48%)	20.9 ± 0.3 (85%)
RPS	24.1 ± 0.8 (10%)	20.8 ± 0.1 (84%)
RFS	24.5 ± 0.3 (9%)	21.2 ± 0.2 (88%)
Host	26.8 ± 0.1	11.3 ± 0.2

Table 3: TCP throughput and latency (round-trip time for 32-byte TCP packets) for different packet steering mechanisms atop a container overlay network across two physical hosts. The numbers followed by \pm show the standard deviations. The numbers in parentheses show the relative slowdown compared with using the host mode.

synchronization (e.g., cache-line conflicts, locking). Table 3 shows the changes to throughput and latency on a container overlay network using packet steering.

Packet steering improves TCP throughput to within 91% of using a host TCP connection, but it does not reduce packet-level latency. We experimented with two packet steering options, Receive Packet Steering (RPS) and Receive Flow Steering (RFS), for internal virtual network interfaces in the overlay network. RPS² ensures that packets in the same flow always hit the same core. RFS, an enhancement of RPS, ensures that software interrupt processing occurs on the same core as the application.

Although packet steering can improve throughput, it has a more modest impact on CPU utilization than throughput and almost no change to latency. Packets still have to go through the same packet transformations and traverse the network stack twice. Our design, Slim, focuses directly on removing this per-packet processing overhead in container overlay networks.

3 Overview

Slim provides a low-overhead container overlay network in which packets in the overlay network traverse the network stack exactly once. Like other container overlay network implementations [52, 8, 15], Slim creates a virtual network with a configuration completely decoupled from the host network’s. Containers have no visibility of host network interfaces, and they communicate only using virtual network interfaces that the OS kernel creates.

We require Slim to be (1) *readily deployable*, supporting unmodified application binaries; (2) *flexible*, supporting various network policies, such as access control, rate limiting, and quality of service (QoS), at both per-connection and per-container levels; and (3) *secure*, the container cannot learn information about the physical hosts, create connections directly on host network, or increase its traffic priority.

Figure 4 shows Slim’s architecture. It has three main components: (1) a user-space shim layer, *SlimSocket*, that is dy-

²RSS requires hardware NIC support. RPS is a software implementation of RSS that can be used on virtual network interfaces inside the OS kernel.

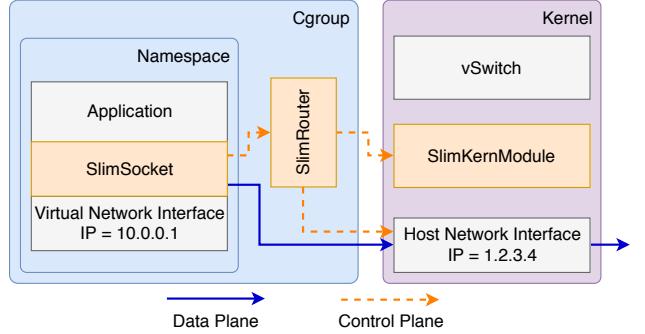


Figure 4: Architecture of Slim.

namically linked with application binaries; (2) a user-space router, *SlimRouter*, running in the host namespace; and (3) a small optional kernel module, *SlimKernModule*, which augments the OS kernel with advanced Slim features (e.g., dynamically changing access control rules, enforcing security).

Slim virtualizes the network by manipulating connection-level metadata. *SlimSocket* exposes the POSIX socket interface to application binaries to intercept invocations of socket-related system calls. When *SlimSocket* detects an application is trying to set up a connection, it sends a request to *SlimRouter*. After *SlimRouter* sets up the network connection, it passes access to the connection as a file descriptor to the process inside the container. The application inside the container then uses the host namespace file descriptor to send/receive packets directly to/from the host network. Because *SlimSocket* has the exact same interface as the POSIX socket, and Slim dynamically links *SlimSocket* into the application, the application binary need not be modified.

In Slim, packets go directly to the host network, circumventing the virtual network interface and the virtual switch; hence, a separate mechanism is needed to support various flexible control-plane policies (e.g., access control) and data-plane policies (e.g., rate limiting, QoS). Control-plane policies isolate different components of containerized applications. Data-plane policies limit a container’s network resource usage and allow prioritization of network traffic. In many current overlay network implementations, both types of policies are actually enforced inside the data-plane. For example, a typical network firewall inspects every packet to determine if it is blocked by an access control list.

SlimRouter stores control-plane policies and enforces them at connection setup time. This approach obviates the need to inspect every packet in the connection. Before creating a connection, *SlimRouter* checks whether the access control list permits the connection. When the policy changes, *SlimRouter* scans all existing connections and removes the file descriptors for any connection that violates the updated access control policy through *SlimKernModule*. Slim leverages existing kernel functionalities to enforce data-plane policies.

Sending a host namespace file descriptor directly to a ma-

licious container raises security concerns. For example, if a malicious container circumvents *SlimSocket* and invokes the *getpeername* call directly on the host namespace file descriptor, it would be able to learn the IP addresses of the host machines. A container could also call *connect* with a host network IP address to create a connection directly on the host network, circumventing the overlay network. Finally, a container could call *setsockopt* to increase its traffic priority.

To enforce the same security model as in today’s container overlay network, Slim offers a secure mode. When secure mode is on, Slim leverages a kernel module, *SlimKernModule*, to restrict the power of host namespace file descriptors inside containers. *SlimKernModule* implements a lightweight capability system for file descriptors. *SlimKernModule* has three roles: (1) track file descriptors as they propagate inside the container, (2) revoke file descriptors upon request from *SlimRouter*, and (3) prohibit a list of unsafe system calls using these file descriptors (e.g., *getpeername*, *connect*, *setsockopt*). *SlimSocket* emulates these system calls for non-malicious applications.

4 Slim

We first describe how to implement network virtualization without needing packet transformations in the data-plane while maintaining compatibility with current containerized applications. We then describe how to support flexible network policies and enforce security for malicious containers.

Slim does not change how virtual to physical IP mappings are stored. They can still be either stored in external storage or obtained through gossiping. As with today’s container overlay network, Slim relies on a consistent and current view of containers’ locations in the host network.

4.1 Connection-based Network Virtualization

Slim provides a connection-based network virtualization for containers. When a container is initiated on the host, Slim dispatches an instance of *SlimRouter* in the host namespace. Slim links a user-level shim layer, *SlimSocket*, to the container. When the process inside the container creates a connection, instead of making standard socket calls, *SlimSocket* sends a request to *SlimRouter* with the destination IP address and port number. *SlimRouter* creates a connection on behalf of the container and returns a host namespace file descriptor back to the container. We first present an example that shows how Slim supports traditional blocking I/O. We then describe how to additionally make Slim support non-blocking I/O.

Support for blocking I/O. Figure 5 shows how a TCP connection is created between a web client and a web server on Slim. Consider the web server side. The container first creates a socket object with the *socket* function call. This call is intercepted by *SlimSocket* and forwarded to *SlimRouter*, which creates a socket object in the host network. When the container calls *bind* on the socket with virtual network interface IP address 10.0.0.1 and port 80, *SlimRouter* also calls

bind on the host network interface IP address 1.2.3.5 and with some unused port 1234. The port translation is needed because a host can run multiple web servers binding on port 80, but the host network interface only has a single port 80. *SlimRouter* updates the port mapping. The web server then uses *accept* to wait for an incoming TCP connection. This function call is also forwarded to *SlimRouter*, which waits on the host socket.

We move next to the web client side. The client performs similar steps to create the socket object. When the client side connects the overlay socket to the server side at IP address 10.0.0.1 port 80, *SlimRouter* looks up the virtual IP address 10.0.0.1 and finds its corresponding host IP address 1.2.3.5. *SlimRouter* then contacts the *SlimRouter* for the destination container on 1.2.3.5 to locate the corresponding host port, 1234. *SlimRouter* sets up a direct connection to port 1234 on 1.2.3.5. After the TCP handshake is complete, *accept/connect* returns a file descriptor in which socket *send/recv* is enabled. *SlimRouter* passes the file descriptor back to the container, and *SlimSocket* replaces the overlay connection file descriptor with the host namespace file descriptor using system call *dup2*. From this point on, the application directly uses the host namespace file descriptor to send or receive packets.

To ensure compatibility with current containerized applications, *SlimSocket* exposes the same POSIX socket interface. Besides forwarding most socket-related system calls (e.g., *socket*, *bind*, *accept*, *connect*) to *SlimRouter*, *SlimSocket* also carefully maintains the expected POSIX socket semantics. For example, when a containerized application calls *getpeername* to get an IP address on the other side of the connection, *SlimSocket* returns the overlay IP address rather than the host IP address, even when the file descriptor for the overlay connection has already been replaced with the host namespace file descriptor.

Support for non-blocking I/O. Most of today’s applications [32, 36] use a non-blocking I/O API (e.g., *select*, *epoll*) to achieve high I/O performance. Slim must also intercept these calls because they interact with the socket interface. For example, *epoll* creates a meta file descriptor that denotes a set of file descriptors. An application uses *epoll_wait* to wait any event in the set, eliminating the need to create a separate thread to wait on an event in each file descriptor. On connection setup, we must change the corresponding file descriptor inside the *epoll*’s file descriptor set. *SlimSocket* keeps track of the mapping between the *epoll* file descriptor and *epoll*’s set of file descriptors by intercepting *epoll_ctl*. For an *accept* or *connect* on a file descriptor that is inside an *epoll* file descriptor set, *SlimSocket* removes the original overlay network file descriptor from the *epoll* file descriptor set and adds host namespace file descriptor into the set.

Service discovery. Our example in Figure 5 assumes that the *SlimRouter* on the client side knows the server side has bound to physical IP 1.2.3.4 and port 1234. To automatically

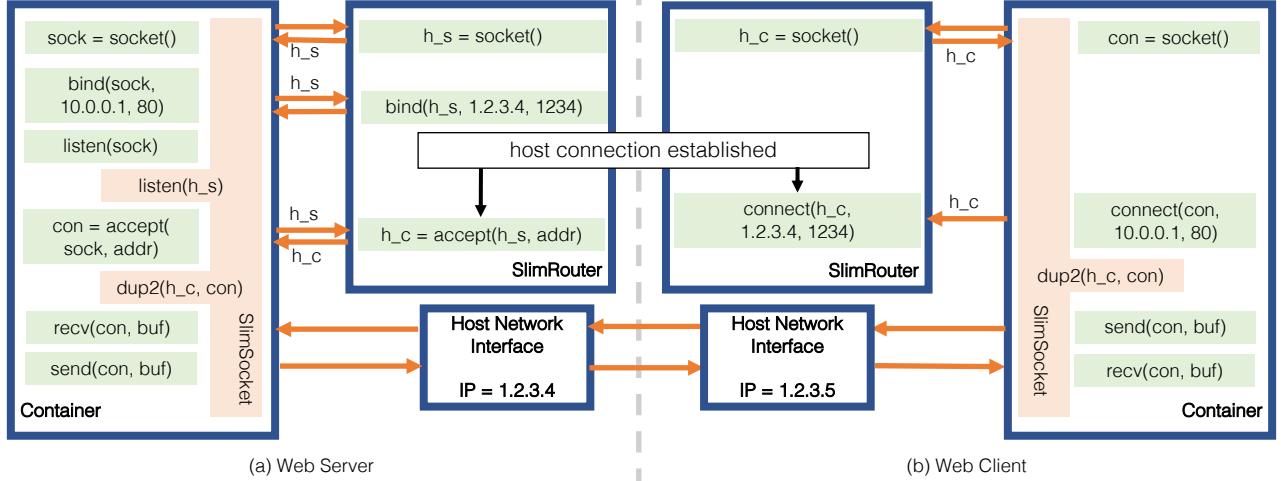


Figure 5: TCP connection setup between a web client and a web server atop Slim.

discover the server’s physical IP address and port, we could store a mapping from virtual IP/port to physical IP/port on every node in the virtual network. Unfortunately, this mapping has to change whenever a new connection is listened.

Instead, Slim uses a distributed mechanism for service discovery. Slim keeps a standard container overlay network running in the background. When the client calls *connect*, it actually creates an overlay network connection on the standard container overlay network. When the server receives an incoming connection on the standard overlay network, *SlimSocket* queries *SlimRouter* for the physical IP address and port and sends them to the client side inside the overlay connection. In secure mode (§4.3), the result queried from *SlimRouter* is encrypted. *SlimSocket* on the client side sends the physical IP address and port (encrypted if in secure mode) to its *SlimRouter* and the *SlimRouter* establishes the host connection. This means connection setup time is longer in Slim than that on container overlay networks based on packet transformation. (See §6.1.)

4.2 Supporting Flexible Network Policies

This section describes Slim’s support for both control- and data-plane policies.

Control-plane policies. Slim supports standard access control over overlay packet header fields, such as the source/destination IP addresses and ports. Access control can also filter specific types of traffic (e.g., SSH, FTP) or traffic from specific IP prefixes.

In the normal case where policies are static, Slim enforces access control at connection creation. *SlimRouter* maintains a copy of current access control policies from the container orchestrator or network operator. When a connection is created by *accept* or *connect*, *SlimRouter* checks whether the created connection violates any existing access control policy. If so, *SlimRouter* rejects the connection by returning -1 to *connect* or by ignoring the connection in *accept*.

Access control policies can change dynamically, and any

connection in violation of the updated access control policy must be aborted. *SlimRouter* keeps per-connection state, including source and destination IP addresses, ports, and the corresponding host namespace file descriptors. When access control policies change, *SlimRouter* iterates through all current connections to find connections that are forbidden in the updated policies. *SlimRouter* aborts those connections by removing the corresponding file descriptors from the container. Removing a file descriptor from a running process is not an existing feature in commodity operating systems such as Linux. We build this functionality in *SlimKernModule*. (See §4.3 for more details.)

Data-plane policies. Slim supports two types of data-plane policies: rate limiting and quality of service (QoS). Rate limiting limits the amount of resources that a container can use. QoS ensures that the performance of certain applications is favored over other applications.

Slim reuses an OS kernel’s existing features to support data-plane policies. A modern OS kernel has support for rate limiting and QoS for a single connection or a set of connections. Slim simply sets up the correct identifier to let the OS kernel recognize the container that generates the traffic.

In Slim, rate limits are enforced both at the per-connection and per-container level. Per-connection rate limits are set in a similar way as in today’s overlay network using Linux’s traffic control program, *tc*. For per-container rate limits, Slim first configures the *net_cls* cgroups to include the *SlimRouter* process. The *net_cls* cgroup tags traffic from the container or the corresponding *SlimRouter* with a unique identifier. *SlimRouter* then sets the rate limit for traffic with this identifier using *tc* on the host network interface. In this way, the network usage by *SlimRouter* is also restricted by the rate limit. Correct accounting of network usage is the fundamental reason why each container requires a separate *SlimRouter*.

Quality of service (QoS) also uses *tc*. *SlimRouter* uses socket options to set up the type of service (ToS) field (via

setsockopt). In this way, switches/routers on the physical network are notified of the priority of the container’s traffic.

Compatibility with existing IT tools. In general, IT tools³ need to be modified to interact with *SlimRouter* in order to function with Slim. IT tools usually use some user-kernel interface (e.g., iptables) to inject firewall and rate limits rules. When working with Slim, they should instead inject these rules to *SlimRouter*. Because Slim is fundamentally a connection-based virtualization approach, a limitation of our approach is that it cannot support packet-based network policy (e.g., drop an overlay packet if the hash of the packet matches a signature). (See §7.) If packet-based policies are needed, the standard Linux overlay should be used instead.

If static connection-based access control is the only network policy needed, then existing IT tools need not be modified. If an IT tool blocks a connection on a standard container overlay network, it also blocks the metadata for service discovery for that connection on Slim, thus it blocks the host connection from being created on Slim.

4.3 Addressing Security Concerns

Slim includes an optional kernel module, *SlimKernModule*, to ensure that Slim maintains the same security model as today’s container overlay networks. The issue concerns potentially malicious containers that want to circumvent *SlimSocket*. Slim exposes host namespace file descriptors to containers and therefore needs an extra mechanism inside the OS kernel to track and manage access.

SlimKernModule implements a lightweight and general capability system based on file descriptors. *SlimKernModule* tracks tagged file descriptors in a similar way as taint-tracking tools [12] and filters unsafe system calls on these file descriptors. We envision this kernel module could also be used by other systems to track and control file descriptors. For example, a file server might want to revoke access from a suspicious process if it triggers an alert. Slim cannot use existing kernel features like seccomp [50] because seccomp cannot track tagged file descriptors.

SlimKernModule monitors how host namespace file descriptors propagate inside containers. It lets *SlimRouter* or other privileged processes tag a file descriptor. It then interposes on system calls that may copy or remove tagged file descriptors, such as *dup*, *fork* and *close*—to track their propagation. If the container passes the file descriptor to other processes inside the container, the tag is also copied.

Tagged file descriptors have limited powers within a container. *SlimKernModule* disallows invocation of certain unsafe system calls using these file descriptors. For example, in the case of Slim, a tagged file descriptor cannot be used with the following system calls: *connect*, *bind*, *getsockname*, *getpeername*, *setsockopt*, etc. This prevents containers from

³We only consider IT tools that run on the host to manage containers but not those run inside containers. IT tools usually require root privilege to the kernel (e.g., iptables) and are thus disabled inside containers.

learning their host IP addresses or increasing their traffic priority. It also prevents containers from directly creating a host network connection. For a non-malicious container, *SlimSocket* and *SlimRouter* emulate the functionalities of these forbidden system calls.

SlimKernModule revokes tagged file descriptors upon request. To do so, it needs a process identifier (pid) and a file descriptor index. *SlimRouter* uses this functionality to implement dynamic access control. When the access control list changes for existing connections, *SlimRouter* removes the file descriptors through *SlimKernModule*. *SlimKernModule* revokes all the copies of the file descriptors.

Secure versus Non-secure mode. Whether to use Slim in secure mode (with *SlimKernModule*) or not depends on the use case. When containers and the physical infrastructure are under the same entity’s control, such as for a cloud provider’s own use [28], non-secure mode is sufficient. Non-secure mode is easier to deploy because it does not need kernel modification. When containers are potentially malicious to the physical infrastructure or containers of other entities, secure mode is required. Secure mode has slightly (~25%) longer connection setup time, making the overall connection setup time 106% longer than that of a traditional container overlay network. (See §6.1.)

5 Implementation

Our implementation of Slim is based on Linux and Docker. Our prototype includes all features described in §4. *SlimSocket*, *SlimRouter*, and *SlimKernModule* are implemented in 1184 lines of C, 1196 lines of C++ (excluding standard libraries), and 1438 lines of C, respectively.

Our prototype relies on a standard overlay network, Weave [52], for service discovery and packets that require data-plane handling (e.g., ICMP, UDP).

SlimSocket uses LD_PRELOAD to dynamically link to the application binary. Communication between *SlimSocket* and *SlimRouter* is via a Unix Domain Socket. In non-secure mode, file descriptors are passed between *SlimRouter* and *SlimSocket* by *sendmsg*. For secure mode, file descriptors are passed with *SlimKernModule*’s cross-process file descriptor duplication method.

SlimRouter allows a network operator to express the access control as a list of entries based on source/destination IP address prefixes and ports in a JSON file. *SlimRouter* has a command-line interface for network operators to issue changes in the access control list via reloading the JSON file. Slim rejects any connection matched in the list. *SlimRouter* uses *htb* qdisc to implement rate limits and *prio* qdisc for QoS with *tc*.

SlimRouter and *SlimKernModule* communicate via a dummy file in *procfs* [46] created by *SlimKernModule*. *SlimKernModule* treats writes to this file as requests. Accessing the dummy file requires host root privilege.

SlimKernModule interposes on system calls by replacing function pointers in the system call table. *SlimKernModule* stores tagged file descriptors in a hash table and a list of unsafe system calls. *SlimKernModule* rejects unsafe system calls on tagged file descriptors.

SlimKernModule also interposes on system calls such as *dup*, *dup2* and *close* to ensure that file descriptor tags are appropriately propagated. For process fork (e.g., *fork*, *vfork*, *clone* in Linux kernel), *SlimKernModule* uses the *sched_process_fork* as a callback function. Slim does not change the behavior of process forking. A forked process still has *SlimSocket* dynamically linked.

6 Evaluation

We first microbenchmark Slim’s performance and CPU utilization in both secure and non-secure mode and then with four popular containerized applications: an in-memory key-value store, Memcached [32]; a web server, Nginx [36]; a database, PostgreSQL [45]; and a stream processing framework, Apache Kafka [1, 26]. Finally, we show performance results for container migration. Our testbed setup is the same as that for our measurement study (§2.2). In all the experiments, we compare Slim with Weave [52] with its fast data-plane enabled and with RFS enabled by default. We use Docker [6] to create containers.

6.1 Microbenchmarks

Similar to the performance tests in §2.2, we use *iperf3* [19] and *NPtcp* [39] to measure performance of a TCP flow. We use *mpstat* to measure CPU utilization.

A single TCP flow running on our 40 Gbps testbed reaches 26.8 Gbps with 11.4 μ s latency in both secure and non-secure modes. Slim’s throughput is the same as the throughput on the host network and is 9% faster than Weave with RFS. Slim’s latency is also the same as using the host network, and it is 86% faster than Weave with RFS.

Using Slim, the creation of a TCP connection takes longer because of the need to invoke the user-space router. On our testbed, in a container with Weave, creating a TCP connection takes 270 μ s. With the non-secure mode of Slim, it takes 444 μ s. With the secure mode, it takes 556 μ s. As a reference, creation of a TCP connection on the host network takes 58 μ s. This means that Slim is not always better, e.g., if an application has many short-lived TCP connections. We did not observe this effect in the four applications studied because they support persistent connections [35, 37], a common design paradigm.

For long-lived connections, Slim reduces CPU utilization. We measure the CPU utilization using *mpstat* for Slim in secure mode and Weave with RFS when varying TCP throughput from 0 to 25 Gbps. RFS cannot reach 25 Gbps, so we omit that data point. Figure 6a shows the total CPU utilization in terms of number of virtual cores consumed. Compared to RFS, CPU overhead declines by 22-41% for Slim;

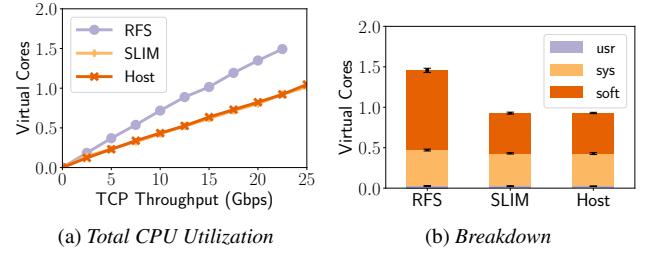


Figure 6: CPU utilization and breakdown for a TCP connection. In Figure 6a, the Slim and the host lines overlap. In Figure 6b, the *usr* bar is at the bottom and negligible. Error bars denote standard deviations.

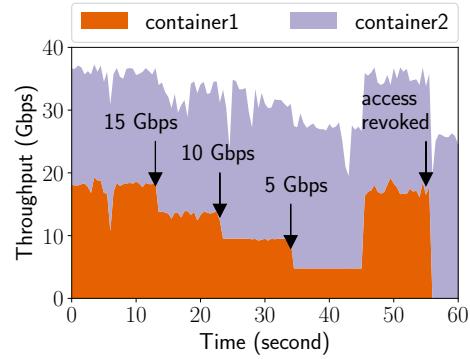


Figure 7: A bar graph of the combined throughput of two Slim containers, with rate limit and access control policy updates to one of the containers.

Slim’s CPU costs are the same as using the host network directly. To determine the source of this reduction, we break down different components using *mpstat* when TCP throughput is 22.5 Gbps. Figure 6b shows the result. As expected, the largest reduction in CPU costs comes from serving software interrupts. These decline 49%: Using Slim, a packet no longer needs data-plane transformations and traverses the host OS kernel’s network stack only once.

Network policies. Slim supports access control, rate limiting and QoS policies, including when applied to existing connections. We examine a set of example scenarios when rate limits and access control are used. We run two parallel containers, each with a TCP connection. The other end of those connections is a container on a different machine. We use *iperf* to report average throughput per half second. Figure 7 shows the result.

Without network policy, each container gets around 18-18.5 Gbps. We first set a rate limit of 15 Gbps on one container. The container’s throughput immediately drops to around 14 Gbps, and the other container’s throughput increases. A slight mismatch occurs between the rate limit we set and our observed throughput, which we suspect is due to *tc* being imperfect. We subsequently set the rate limit to 10 Gbps and 5 Gbps. As expected, the container’s throughput declines to 10 and 5 Gbps, respectively, while the other con-

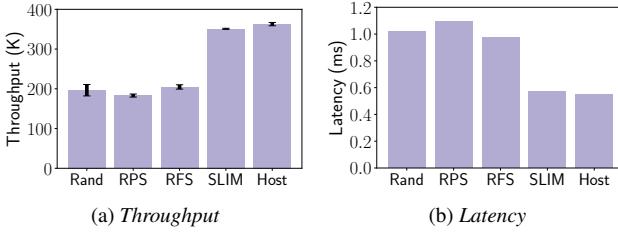


Figure 8: Throughput and latency of Memcached with Weave (in various configurations) and with Slim. Error bars in Figure 8a shows the standard deviation of completed Memcached operations per-second.

tainer’s throughput increases. Finally, Slim stops rate limiting and sets an ACL to bar the container from communicating with the destination. The affected connection is destroyed, and the connection from the other container speeds up to standard connection speed.

6.2 Applications

We evaluate Slim with four real world applications: Memcached, Nginx, PostgreSQL, and Apache Kafka. From this point on, our evaluation uses Slim running in secure mode.

6.2.1 Memcached

We measure the performance of Memcached [32] on Slim. We create one container on each of the two physical machines; one runs the Memcached (v1.5.6) server, and the other container runs a standard Memcached benchmark tool, *memtier_benchmark* [33] developed by redislabs [49]. The benchmark tool spawns 4 threads. Each thread creates 50 TCP connections to the Memcached server and reports the average number of responses per second, the average latency to respond to a memcache command, and the distribution of response latency (SET:GET ratio = 1:10).

Slim improves Memcached throughput (relative to Weave). Figure 8a shows the number of total Memcached operations per-second completed on Slim and Weave with different configurations. Receive Flow Steering (RFS) is our best-tuned configuration, yet Slim still outperforms it by 71%. With the default overlay network setting (random IRQ load balancing), Slim outperforms Weave by 79%. Slim’s throughput is within 3% of host mode.

Slim also reduces Memcached latency. Figure 8b shows the average latency to complete a memcache operation. The average latency reduces by 42% using Slim compared to RFS. The latency of the default setting (random IRQ load balancing), RPS, and RFS are not significantly different (within 5%). Slim’s latency is exactly the same as host mode.

Slim also reduces Memcached tail latency. Figure 9 shows the CDF of latency for SET and GET operations. The default configuration (i.e., IRQ load balancing) has the worst tail latency behavior. Synchronization overhead depends on temporal kernel state and thus makes latency less predictable.

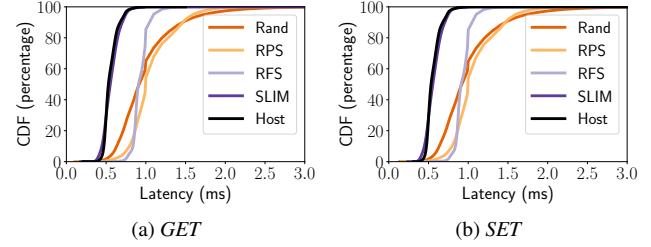


Figure 9: Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The Slim and Host lines overlap.

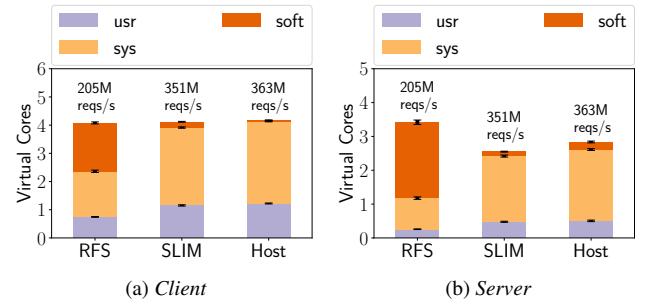


Figure 10: CPU utilization of Memcached client and server. Error bars denote standard deviations.

RPS and RFS partially remove the synchronization overhead, improving predictability. Compared to the best configuration, RFS, Slim reduces the 99.9% tail latency by 41%.

Slim reduces the CPU utilization per operation. We measure average CPU utilization on both the client and the Memcached server when *memtier_benchmark* is running. Figure 10 shows the result. The total CPU utilization is similar on the client side, while the utilization is 25% lower with Slim on the server compared to RFS. Remember that Slim performs 71% more operations/second. As expected, the amount of CPU utilization in serving software interrupts declines in Slim. We also compare CPU utilization when the throughput is constrained to be identical. Slim reduces CPU utilization by 41% on the Memcached client and 56% on the Memcached server, relative to Weave.

6.2.2 Nginx

We run one Nginx (v1.10.3) server in one container and a standard web server benchmarking tool, *wrk2* [53], in another container. Both containers are on two different physical machines. The tool, *wrk2*, spawns 2 threads to create a total of 100 connections to the Nginx server to request an HTML file. This tool lets us set throughput (requests/sec), and it outputs latency. We set up two HTML files (1KB, 1MB) on the Nginx server.

Nginx server’s CPU utilization is significantly reduced with Slim. We use *mpstat* to break down the CPU utilization of the Nginx server for scenarios when RFS, Slim, and host

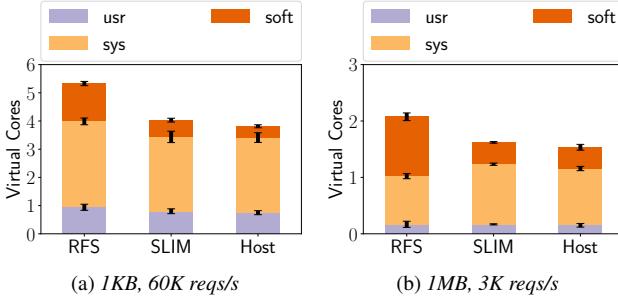


Figure 11: CPU utilization breakdown of Nginx. Error bars denote standard deviations.

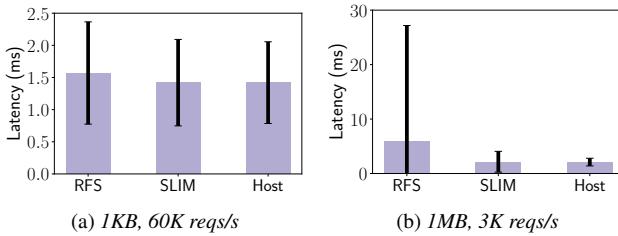


Figure 12: Latency of Nginx server. Error bars denote standard deviations.

can serve the throughput. Figure 11 shows the CPU utilization breakdown when the file size is 1KB and the throughput is 60K reqs/second, and also when the file size is 1MB and the throughput is 3K reqs/second. (We choose 60K reqs/second and 3K reqs/second because they are close to the limit of what RFS can handle.). For the 1 KB file, the CPU utilization reduction is 24% compared with RFS. For the 1 MB file, the CPU utilization reduction is 22% compared with RFS. Note that much of the CPU utilization reduction comes from reduced CPU cycles spent in serving software interrupts in the kernel. The CPU utilization still has a 5%-6% gap between Slim and host. We expect this gap is from the longer connection setup time in Slim. Unlike our Memcached benchmark, where connections are pre-established, we observe that *wrk2* creates TCP connections on the fly to send HTTP requests.

While Slim improves the CPU utilization, the improvements to latency are lost in the noise of the natural variance in latency for Nginx. The benchmark tool, *wrk2*, reports the average and the standard deviation of Nginx’s latency. Figure 12 shows the result. The standard deviation is much larger than the difference of the average latencies.

6.2.3 PostgreSQL

We deploy a relational database, PostgreSQL [45] (version 9.5), in a container and then use its default benchmark tool, *pgbench* [43], to benchmark its performance in another container. The tool, *pgbench*, implements the standard TPC-B benchmark. It creates a database with 1 million banking accounts and executes transactions with 4 threads and a total of

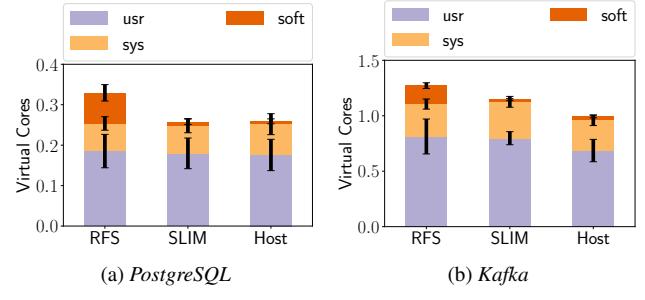


Figure 13: CPU utilization of PostgreSQL and Kafka. Error bars denote standard deviations.

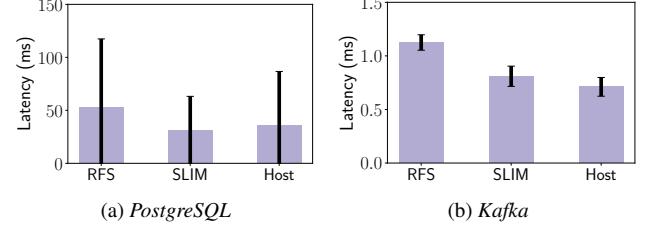


Figure 14: Latency of PostgreSQL and Kafka. Error bars denote standard deviations.

100 connections.

Slim reduces the CPU utilization of PostgreSQL server. We set up *pgbench* to generate 300 transactions per second. (We choose 300 transactions per second because it is close to what RFS can handle.) Figure 13a shows the CPU utilization breakdown of the PostgreSQL server. Compared with RFS, Slim reduces the CPU utilization by 22% and the CPU utilization is exactly the same as using host mode networking. Note here, the reduction in CPU utilization is much less than in Memcached and Nginx. The reason is that the PostgreSQL server spends a larger fraction of its CPU cycles in user space, processing SQL queries. Thus, the fraction of CPU cycles consumed by the overlay network is less.

Similar to Nginx, the latency of PostgreSQL naturally has a high variance because of the involvement of disk operations, and it is difficult to conclude any latency benefit of Slim. The benchmark tool, *pgbench*, reports the average and the standard deviation of PostgreSQL’s latency. Figure 14a shows the results. The standard deviation is much larger than the difference of the mean latencies.

6.2.4 Apache Kafka

We now evaluate a popular data streaming framework, Apache Kafka [1, 26]. It is used to build real-time data processing applications. We run Kafka (version 2.0.0) inside one container, and Kafka’s default benchmarking tool, *kafka-producer-perf-test*, in another container on a different physical machine.

Slim reduces the CPU utilization of both the Kafka server and the Kafka client. We use *kafka-producer-perf-test* to set

	Weave	Slim
Stop running container	0.75 ± 0.02	0.75 ± 0.02
Extract fs into image	0.43 ± 0.01	0.43 ± 0.01
Transfer container image	0.44 ± 0.01	0.44 ± 0.01
Restore fs	0.82 ± 0.09	1.20 ± 0.10
Start <i>SlimRouter</i>	-	0.003 ± 0.001
Start container	0.90 ± 0.09	0.94 ± 0.17
Total	3.34 ± 0.12 s	3.76 ± 0.20 s

Table 4: Time to migrate a Memcached container. The numbers followed by \pm show the standard deviations.

throughput to be 500K messages per second. (We choose 500K messages per second because it is close to what RFS can handle.) Each message is 100 bytes and the batch size is 8192. The tool spawns 10 threads that generate messages in parallel. Figure 13b shows the breakdown of CPU utilization. The total CPU utilization of the Kafka server reduces by 10% with Slim. The CPU utilization reduction is even smaller than PostgreSQL because Kafka spends more time in user space processing.

Slim reduces message latencies in Kafka. The benchmark tool, *kafka-producer-perf-test*, reports the latency of Kafka. Figure 14b shows the results. Kafka’s latency reduces by 0.28 ms (28%), compared with RFS. There is still a 0.09 ms latency gap between using Slim and the host mode.

6.3 Container Migration

Slim supports container migration. On our testbed, we migrate a Memcached container from one physical server to another physical server on the 40 Gbps network. We test migration 20 times with/without Slim. The container’s IP address is kept the same across the migration. Likewise, we do not change the host network’s routing table. The container image extracted from the file system is 58 Mbytes.

Using Slim marginally slows down container migration. Table 4 is the breakdown of the average container migration time on Weave and on Slim. In total, Slim slows down container migration from 3.34 s to 3.76 s. Slim does not change most of the migration process. The extra overhead is introduced mainly in restoring the file system. With Slim, a container has an additional disk volume containing *SlimSocket* and also a dummy file to support communication over UNIX domain socket between *SlimSocket* and *SlimRouter*. We suspect that the additional disk volume slows down the file system restoration process. Further, starting a container with Slim adds a small amount of additional overhead.

7 Discussion

Connection Setup Time. One drawback of Slim is that connection setup time is significantly longer (§6.1). This can penalize applications with many short connections. Slim allows individual applications in a container to opt out by detaching *SlimSocket*. In the future, we want the choice of opt-

ing out to be at a per-connection level. We can either (1) allow developers to specify which connection to opt out, or (2) automatically opt out based on predicted flow sizes [10].

Container Live Migration. Although Slim does support quiescent container migration, it does not currently support container live migration. All the TCP connections are disconnected during the migration process, and memory states are not migrated. However, in live migration, live application state has to be fully restored, including state such as application threads waiting on events inside the OS kernel. Docker is currently experimenting with live checkpointing and restoration with *criu* [4], but it is focused on the simpler case of a single host [7]. Provided a practical live container migration system could be built, Slim would make that more difficult because: (1) the state of the container now includes host namespace file descriptors and (2) data-plane policies (e.g., rate limits) are enforced on host connection identifiers (i.e., five tuples) that would need to be properly translated when migrated.

UDP. The focus of this paper has been on improving the container communication performance of connection-oriented protocols, such as TCP, by moving operations from the data-plane to connection setup. This poses a challenge for connectionless protocols such as UDP. Slim potentially could support UDP using similar mechanisms as for TCP, by intercepting *socket*, *bind*, *connect*, *sendto*, and *recvfrom*. However, we chose not to do this in our prototype because of two reasons. First, we do not have a good mechanism to support flexible network policy for UDP. In UDP, a file descriptor does not describe a single network pair, but rather an open port to which every node in the virtual network can send packets. Second, the most common use case for UDP in data centers is to avoid the overhead of connection setup; since Slim makes connection setup more expensive, it would subvert some of those benefits. Instead, to work with unmodified applications that may use a mixture of TCP and UDP packets, our prototype simply directs UDP traffic to Weave.

Packet-based Network Policy. A limitation of Slim is that it supports connection-based network policy and not packet-based network policy. For example, a virtual network can be set up to prevent access to a backend database, except from certain containers; Slim supports this kind of access control. Packet-based filters allow the system drop packets if the hash of the overlay packet matches a signature. On Slim, the virtual overlay packet is never constructed and so checking against a signature would be prohibitively expensive. If packet-based network policy is needed, a standard overlay network should be used instead.

LD_PRELOAD. Our prototype uses LD_PRELOAD to dynamically link *SlimSocket* into unmodified application binaries. Some systems assume statically linked application binaries (e.g., applications written in Go). These can benefit from Slim by patching the application binaries to use *SlimSocket* instead of POSIX sockets; we do not implement this

support in our prototype.

Error Code. Our current prototype implementation is not transparent in one significant way. When an access control list changes, requiring Slim to revoke a file descriptor, the application receives a different error code when it used that file descriptor, relative to Weave. In Slim, a send on a revoked file descriptor returns a bad file descriptor error code, while in Weave the packet would be silently discarded. We believe it is possible to address this but it was not needed for our benchmark applications.

SmartNICs. A recent research trend has been to explore moving common case network data-plane operations to hardware. Catapult [48], for example, moves packet encapsulation required for virtual machine emulation to hardware. Catapult runs as a bump on the wire, however, so in order to offload overlay network processing, Linux would need to be modified to accept virtual network packets on its physical network interface. SR-IOV is commodity hardware, but it suffers from the same problem as macvlan mode. (See §2.1.) FlexNIC [22] has proposed a flexible model that can incorporate application, guest OS, and virtual machine packet management, but to date it is only experimental hardware. While new hardware support is likely to become increasingly available, what we have shown is that such hardware support is not necessary for efficient virtual overlay networks for containers; the container OS has all the information it needs to perform virtualization at connection setup.

8 Related Work

Network namespace. Mapping resources from a host into a container is not a new idea. In Plan9 [44], resources, such as directories in the file system or process identifiers, are directly mapped between namespaces. Our work revisits Plan9’s idea in the networking context, but with performance as a goal, rather than portability. Today’s Linux network namespace works at a per-device level, and so is not strong enough for supporting connection-based network virtualization. Slim uses the Linux networking namespace to isolate the container from using the host network interface.

Host support for efficient virtual networking. Host support for efficient virtual networking is an old topic, mostly in the context of VMs. Menon et al. co-design the driver of the virtual network interface and the hypervisor for efficient virtual network interface emulation [34]. Socket-outsourcing [11], VMCI socket [51], and Slipstream [5] improve intra-host networking. FreeFlow [23] redirects RDMA library calls to create a fast container RDMA network. To the best of our knowledge, Slim is first work that provides network virtualization at TCP connection setup time for unmodified containerized applications.

Redirecting system calls. Redirecting system calls is a useful technique for many purposes, such as taint tracking [12], building user-level file systems [16] and performing other advanced OS kernel features (e.g., sandboxing [24],

record and replay [17], and intrusion detection [25]). In a networking context, mTCP [21] redirects socket calls to construct a user-level networking stack. NetKernel [38] redirects socket calls to decouple networking stack from virtual machine images.

Separation of control- and data-plane. The performance gain of Slim comes from moving network virtualization logic from the data- to the control-plane. Separation of the control- and the data-plane is a well-known technique to improve system performance in building fast data-plane operating systems [41, 2] and routing in flexible networks [31].

9 Conclusion

Containers have become the de facto method for hosting distributed applications. The key component for providing portability, the container overlay network, imposes significant overhead in terms of throughput, latency, and CPU utilizations, because it adds a layer to the data-plane. We propose Slim, a low-overhead container overlay network that implements network virtualization by manipulating connection-level metadata. Slim transparently supports unmodified, potentially malicious, applications. Slim improves throughput of an in-memory key-value store by 71% and reduces latency by 42%. Slim reduces CPU utilization of the in-memory key-value store by 56%, a web server by 22%-24%, a database server by 22%, and a stream processing framework by 10%. Slim’s source code is publicly available at <https://github.com/danyangz/slim>.

Acknowledgments

We thank Antoine Kaufmann, Jialin Li, Ming Liu, Jitu Padhye, and Xi Wang for their feedback on earlier versions of the paper. We thank our shepherd Jon Howell and the anonymous reviewers for their helpful feedback on the paper. This work was partially supported by the National Science Foundation (CNS-1518702 and CNS-1616774) and by gifts from Google, Facebook, and Huawei.

References

- [1] Apache Kafka. <https://kafka.apache.org/>.
- [2] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI* (2014).
- [3] Calico. <https://www.projectcalico.org/>.
- [4] CRIU. https://criu.org/Main_Page.
- [5] DIETZ, W., CRANMER, J., DAUTENHAHN, N., AND ADVE, V. Slipstream: Automatic Interprocess Communication Optimization. In *USENIX ATC* (2015).
- [6] Docker. <http://www.docker.com>.
- [7] Docker Checkpoint and Restore. <https://github.com/docker/cli/blob/master/experimental/checkpoint-restore.md>.
- [8] Docker container networking. <https://docs.docker.com/engine/userguide/networking/>.
- [9] Docker Swarm. <https://docs.docker.com/engine/swarm/>.

- [10] DUKIC, V., JYOTHI, S. A., KARLAS, B., OWAIDA, M., ZHANG, C., AND SINGLA, A. Is advance knowledge of flow sizes a plausible assumption? In *NSDI* (2019).
- [11] EIRAKU, H., SHINJO, Y., PU, C., KOH, Y., AND KATO, K. Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments. In *ACM Symposium on Applied Computing* (2009).
- [12] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MC-DANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010).
- [13] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. <https://coreos.com/etcd/>.
- [14] FIRESTONE, D., PUTNAM, A., ANGEPAT, H., CHIOU, D., CAULFIELD, A., CHUNG, E., HUMPHREY, M., OVTCHAROV, K., PADHYE, J., BURGER, D., MALTZ, D., GREENBERG, A., MUNDKUR, S., DABAGH, A., ANDREWARTHA, M., BHANU, V., CHANDRAPPA, H. K., CHATURMOHTA, S., LAVIER, J., LAM, N., LIU, F., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., VAID, K., AND MALTZ, D. A. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI* (2018).
- [15] Flannel. <https://github.com/coreos/flannel>.
- [16] Filesystem in userspace. <https://github.com/libfuse/libfuse>.
- [17] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An Application-level Kernel for Record and Replay. In *OSDI* (2008).
- [18] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI* (2011).
- [19] iperf. <https://iperf.fr/>.
- [20] iptables. <https://linux.die.net/man/8/iptables>.
- [21] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI* (2014).
- [22] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High Performance Packet Processing with FlexNIC. In *ASPLOS* (2016).
- [23] KIM, D., YU, T., LIU, H. H., ZHU, Y., PADHYE, J., RAINDEL, S., GUO, C., SEKAR, V., AND SESHA, S. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI* (2019).
- [24] KIM, T., AND ZELDOVICH, N. Practical and effective sandboxing for non-root users. In *USENIX ATC* (2013).
- [25] KING, S. T., AND CHEN, P. M. Backtracking Intrusions. In *SOSP* (2003).
- [26] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: a Distributed Messaging System for Log Processing. In *NetDB* (2016).
- [27] Kubernetes: Cluster Networking. <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.
- [28] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPEZ, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. CrystalNet: Faithfully Emulating Large Production Networks. In *SOSP* (2017).
- [29] Networking using a macvlan network. <https://docs.docker.com/network/network-tutorial-macvlan/>.
- [30] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My VM is Lighter (and Safer) Than Your Container. In *SOSP* (2017).
- [31] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. In *SIGCOMM CCR* (2008).
- [32] Memcached. <https://memcached.org/>.
- [33] memtier.benchmark. https://github.com/RedisLabs/memtier_benchmark.
- [34] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing Network Virtualization in Xen. In *USENIX ATC* (2006).
- [35] MOGUL, J. C. The Case for Persistent-connection HTTP. In *SIGCOMM* (1995).
- [36] Nginx. <https://nginx.org/>.
- [37] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MC ELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *NSDI* (2013).
- [38] NIU, Z., XU, H., HAN, D., CHENG, P., XIONG, Y., CHEN, G., AND WINSTEIN, K. Network stack as a service in the cloud. In *HotNets* (2017).
- [39] netpipe(1) - Linux man page. <https://linux.die.net/man/1/netpipe>.
- [40] Scaling in the Linux Networking Stack. <https://www.kernel.org/doc/Documentation/networking/scaling.txt>.
- [41] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *OSDI* (2014).
- [42] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The Design and Implementation of Open vSwitch. In *NSDI* (2015).
- [43] pgbench. <https://www.postgresql.org/docs/9.5/static/pgbench.html>.
- [44] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The Use of Name Spaces in Plan 9. In *SIGOPS OSR* (1993).
- [45] PostgreSQL. <https://www.postgresql.org/>.
- [46] proc - process information pseudo-filesystem. <http://man7.org/linux/man-pages/man5/proc.5.html>.
- [47] Intel P-State driver. <https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt>.
- [48] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., HASelman, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., XIAO, P. Y., AND BURGER, D. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA* (2014).
- [49] redislab. <https://redislabs.com/>.
- [50] SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
- [51] VMCI Socket Performance. <https://www.vmware.com/techpapers/2009/vmci-socket-performance-10075.html>.
- [52] Weave. <https://www.weave.works/>.
- [53] wrk2. <https://github.com/giltene/wrk2>.

Shinjuku: Preemptive Scheduling for μ second-scale Tail Latency

Kostis Kaffes¹

Timothy Chong¹

Jack Tigar Humphries¹

Adam Belay²

David Mazières¹

Christos Kozyrakis¹

¹ Stanford University ² Massachusetts Institute of Technology

Abstract

The recently proposed dataplanes for microsecond scale applications, such as IX and ZygOS, use non-preemptive policies to schedule requests to cores. For the many real-world scenarios where request service times follow distributions with high dispersion or a heavy tail, they allow short requests to be blocked behind long requests, which leads to poor tail latency.

Shinjuku is a single-address space operating system that uses hardware support for virtualization to make preemption practical at the microsecond scale. This allows Shinjuku to implement centralized scheduling policies that preempt requests as often as every 5μ sec and work well for both light and heavy tailed request service time distributions. We demonstrate that Shinjuku provides significant tail latency and throughput improvements over IX and ZygOS for a wide range of workload scenarios. For the case of a RocksDB server processing both point and range queries, Shinjuku achieves up to $6.6 \times$ higher throughput and 88% lower tail latency.

1 Introduction

Popular cloud applications such as web search, social networks, and machine translation fan out requests to hundreds of communicating services running on thousands of machines. End-to-end response times are then dominated by the slowest machine to respond [23]. Reacting to user actions within tens of milliseconds requires that each participating service process requests with *tail latency* in the range of ten to a few hundred microseconds [14]. Unfortunately, thread management in modern operating systems such as Linux is not designed for microsecond-scale tasks and frequently produces long, unpredictable scheduling delays resulting in millisecond-scale tail latencies [36, 37].

To compensate, researchers have developed network stacks, dataplanes, and full applications that bypass the operating system [44, 31, 39, 32, 16, 45, 22]. Most of these systems operate in a similar way: the NIC uses receive-side scaling (RSS) [21] to distribute

incoming requests across multiple queues in a flow-consistent manner; a polling thread serves requests in each queue in a first-come-first-served manner (FCFS) without scheduling interruptions; optimizations such as zero copy, run-to-completion, adaptive batching, and cache-friendly and thread-private data structures reduce overheads. The resulting request scheduling is known as *distributed queuing and FCFS scheduling*, or *d-FCFS*.

d-FCFS is effective when request service times exhibit low dispersion [57], as is the case for get/put requests to simple in-memory key-value stores (KVS) such as Memcached [43]. d-FCFS fares poorly under high dispersion or heavy-tailed request distributions (e.g., bimodal, log-normal, Zipf, or Pareto distributions), as short requests get stuck behind older long ones assigned to the same queue. d-FCFS is also not work-conserving, an effect exacerbated by implementations based on RSS’s flow-consistent hashing, which approximates true d-FCFS only with high numbers of client connections spreading requests out evenly over queues.

ZygOS [46] improved on d-FCFS by implementing low-overhead task stealing: threads that complete short requests steal work from threads tied up by longer ones. It approximates *centralized FCFS scheduling (c-FCFS)*, in which all threads serve a single queue. Work stealing is not free. It requires scanning queues cached on non-local cores and forwarding system calls back to a request’s home core. However, if service times exhibit low dispersion and there are enough client connections for RSS to spread requests evenly across queues, stealing happens infrequently.

Unfortunately, c-FCFS is also inefficient for workloads with request times that follow heavy-tailed distributions or even light-tailed distributions with high dispersion. These workloads include search engines that score and rank a number of items depending on the popularity of search terms [13]; microservices and function-as-a-service (FaaS) frameworks [17]; and in-memory stores or databases, such as RocksDB [26], Redis [35], and Silo [54], that support both simple get/put requests

and complex range or SQL queries, and use slower non-volatile memory in addition to fast DRAM. Theory tells us that such workloads do best in terms of tail latency under *processor sharing (PS)* [57], where all requests receive a fine-grain, fair fraction of the available processing capacity.

To approximate PS, we need *preemption*, as built into any modern kernel scheduler including Linux. However, any service that uses one thread per request or connection and allows Linux to manage threads will experience *millisecond*-scale tail latencies, because the kernel employs preemption at millisecond granularities and its policies are not optimized for microsecond-scale tail latency [36, 37]. User-level libraries for cooperative threading can avoid the overheads of kernel scheduling [56]. However, it is difficult to yield often enough during requests with longer processing times and without many blocking I/O calls, which are precisely the requests impacting tail latency.

This paper presents *Shinjuku*, a single-address space operating system that implements *preemptive scheduling at the microsecond-scale* and improves tail latency and throughput for *both light- and heavy-tailed service time distributions*. Shinjuku deemphasizes RSS in favor of true centralized scheduling by one or more dedicated dispatcher threads with centralized knowledge of load and service time distribution. It leverages hardware support for virtualization—specifically posted interrupts—to achieve preemption overheads of 298 cycles in the dispatcher core and 1,212 cycles in worker cores. The single address space architecture allows us to optimize context switches down to 110 cycles.

Fast preemption enables scheduling policies that switch between requests as often as every 5 μ sec when needed. We developed two policies. The first assumes no prior knowledge of request service times and uses preemption to select at fine granularity between FCFS or PS based on observed service times. The second policy assumes we can segregate requests with different service level objectives (SLO) in order to ensure good tail latency for both short and long requests. Both policies are work conserving and work well across multiple distributions of service times (light-tailed, heavy-tailed, bimodal, or multimodal). The two policies make Shinjuku the first system to support microsecond-scale tail latency for workloads beyond those with fixed or low-dispersion service time distributions.

We compare Shinjuku with IX [16] and ZygOS [46], two state-of-the-art dataplane operating systems. Using synthetic workloads, we show that Shinjuku matches IX’s and ZygOS’ performance for light-tailed workloads

while it supports up to 5x larger load for heavy-tailed and multi-modal distributions. Using RocksDB, a popular key-value store that also supports range queries, we show that Shinjuku improves upon ZygOS by up to 6.6 \times in terms of throughput at a given 99th percentile latency. We show that Shinjuku scales well with the number of cores available, can saturate high speed network links, and is efficient even with small connection counts.

The rest of the paper is organized as follows. §2 motivates the need for preemptive scheduling at microsecond-scale. §3 discusses the design and implementation of Shinjuku. §4 presents a thorough quantitative evaluation while §5 discusses related work.

Shinjuku is open-source software. The code is available at <https://github.com/stanford-mast/shinjuku>.

2 Motivation

Background: We aim to improve the SLO of latency-critical services on a single server. For cloud services and microservices with high fan-out, Shinjuku must achieve *low tail latency* at the microsecond scale [14]. Low average or median latency is not sufficient [23]. While tail latency can be improved through overprovisioning, doing so is not economical for services with millions of users. To be cost-effective, Shinjuku must maintain low tail latency in the face of *high request throughput*. Finally, it must be *practical* for a wide range of workloads and support intuitive APIs that simplify development and maintenance of large code bases.

The key to achieving low tail latency and high throughput is effective request scheduling, which requires both *good policies* and *low-overhead mechanisms* that operate at the microsecond scale. Good policies are easy to achieve in isolation. Linux already supplies approximations of the optimal policies for workloads we target. Unfortunately, the Linux kernel scheduler operates at the millisecond scale because of preemption and context switch overheads and the complexity of simultaneously accommodating batch, background, and interactive tasks at different time scales [36, 37].

Recent proposals for user-level networking stacks, dataplanes, RPC protocols, and applications [22, 44, 16, 45, 31, 39, 32] sidestep the bloated kernel networking and thread management stacks in order to optimize tail latency and throughput. Most of these systems use RSS to approximate a d-FCFS scheduling policy [21], the IX dataplane being a canonical example [16]. ZygOS improves on IX by using work stealing to approximate c-FCFS [46]. Linux applications built with libevent [47] or libuv [5] also implement c-FCFS,

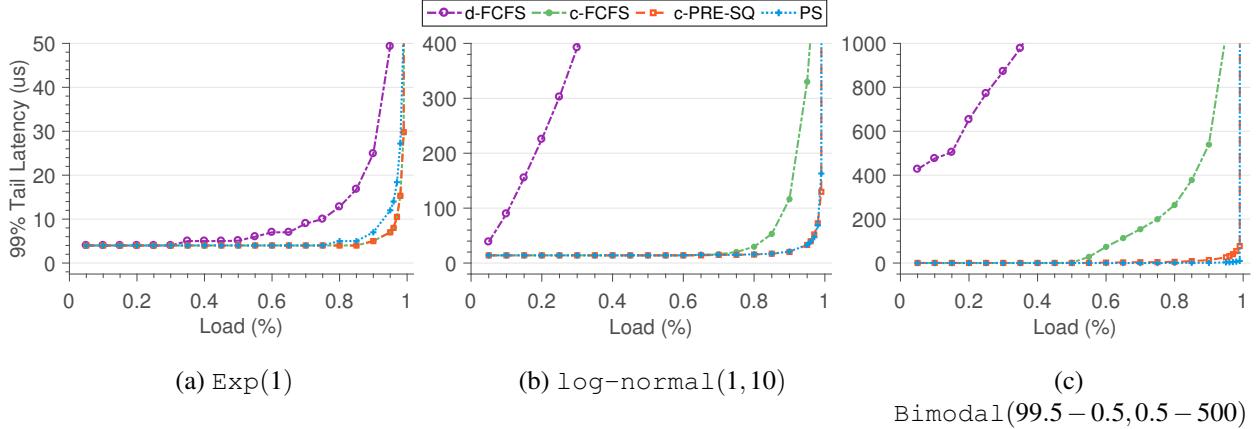


Figure 1: Simulation results for different workloads and scheduling policies for a 16-core system.

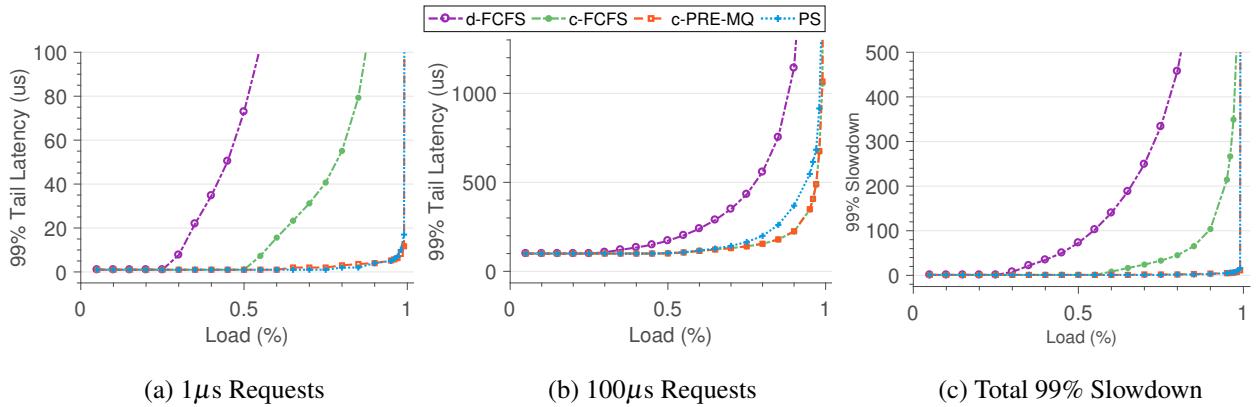


Figure 2: Simulation results for $\text{Bimodal}(50 - 1, 50 - 100)$ for a 16-core system.

but at much higher overheads due to the use of interrupts for request distribution instead of RSS and polling.

Policy comparison: In order to quantify the differences between different scheduling policies, we developed a discrete event simulator. The simulator allowed us to configure parameters such as scheduling policy, number of host cores, system load, service and inter-arrival time distributions as well as various system-related overheads. Figure 1 compares idealized versions of scheduling policies—i.e., no stealing or pre-emption overhead—using the simulator. Plot (a) shows a light-tailed exponential distribution of service times with mean $\mu = 1 \mu\text{sec}$, representative of workloads such as the get/set requests of in-memory key-value stores. d-FCFS is arguably tolerable under such simple workloads, but suffers at moderate and high load as requests are not perfectly distributed across workers. c-FCFS is optimal under such workloads, while PS is slightly worse because it preempts even short requests. The PS time slice used for all simulations is $0.1 \mu\text{sec}$.

d-FCFS is a poor option for heavy-tailed request distributions [42], as found in search engines [38] or induced by activities such as garbage collection or compaction [23, 6, 26]. Plot (b) shows performance under a heavy-tailed log-normal distribution with mean $\mu = 1 \mu\text{sec}$ and standard deviation $\sigma = 10 \mu\text{sec}$. Any long request blocks every short request assigned to the same queue in d-FCFS. c-FCFS performs significantly better as a worker can service any request; short requests are only delayed when most workers simultaneously process older long requests, which is uncommon for the log-normal distribution.

c-FCFS performs significantly worse under a light-tailed bimodal distribution, commonly found in object stores and databases that mix simple get/put requests with complex range or relational queries [35, 26, 54]. Plot (c) shows such a distribution in which 99.5% of requests take $0.5 \mu\text{sec}$ and 0.5% take $500 \mu\text{sec}$. Compared to a heavy-tailed case, the bimodal distribution’s long requests are not as long but far more frequent. PS han-

dles both cases in Plots (b) and (c) well by preempting long requests to interleave execution of short ones.

Figure 2 provides further insights by separating the performance of short and long requests in a bimodal workload with service times evenly split between 1 μ sec and 100 μ sec. This approximates a KVS in which half of the requests are get/put requests and the other half are range queries. The tail latency for the two request types is drawn separately in Plots (a) and (b), and Plot (c) shows the 99th percentile of the request slowdown for all requests, which is the ratio of a request’s overall latency to its service time. This ratio is a useful metric for measuring how well we achieve our goal of reducing queuing time for all request types: if this ratio is small, it means that queuing time is small for all types and no requests are affected by the requests of different types.

Plot 2a shows that both d-FCFS and c-FCFS heavily penalize 1- μ sec requests. Plot 2b shows that c-FCFS is marginally better than PS for 100 μ sec requests, as it effectively prioritizes older, long requests that would be preempted by PS. Plot 2c shows that, in relative terms, the penalty c-FCFS inflicts on short requests dwarfs any benefit to long requests.

Shinjuku approach: Shinjuku implements the c-PRE policies (see §3.4) that achieve the best of both worlds between PS and c-FCFS as shown in Figures 1 and 2. The reason other recent systems cannot implement similar policies is that these policies require preemption at arbitrary execution points. Preemption typically involves interrupts and kernel threads whose overheads are incompatible with microsecond-scale latencies. Therefore, Shinjuku aims to achieve the following goals: 1) Implement low-overhead preemption and context switching mechanisms for user-level threads. 2) Use these mechanisms to build scheduling policies that work well across all possible distributions of service times for microsecond-scale workloads.

3 Shinjuku

Shinjuku¹ is a single-address space operating system for low-latency applications. Shinjuku is a significant departure from the common pattern in IX [16] and ZygoOS [46], which rely heavily on RSS to distribute incoming requests to workers that process them without interruption. Instead, Shinjuku uses a centralized queuing and scheduling architecture and relies on low overhead and frequent preemption in order to ensure low tail latency for a wide variety of service time distributions.

¹Shinjuku (新宿駅) is a major train station in Tokyo that serves millions traveling on 12 lines of various types and speeds.

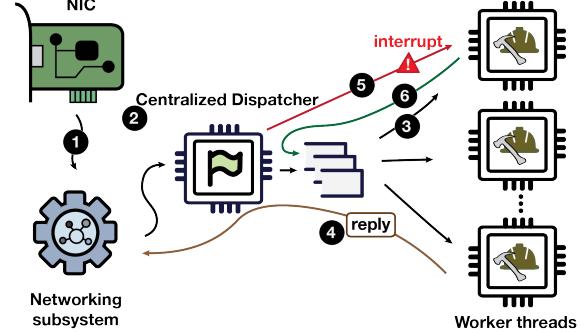


Figure 3: Shinjuku system design.

3.1 Design Overview

Figure 3 summarizes the key components in Shinjuku and the typical request flow. Incoming requests are first processed by the **networking subsystem** that handles all network protocol processing and identifies request boundaries ①. The networking subsystem can be implemented using one or more dedicated cores or hyper-threads [22], a smartNIC [33, 53], or a combination of the two. By separating network processing from request scheduling, Shinjuku can be combined with a range of networking protocols that optimize for different conditions (UDP, TCP, ROCE [52], TIMELY [41], etc.) and various optimized network stacks [31, 28, 22]. The networking subsystem passes requests to a **centralized dispatcher** ② that will queue and schedule them to **worker threads**. The dispatcher generates a context for each incoming request in order to support preemption and rescheduling. In its simplest form, the dispatcher maintains a single queue for all pending requests. The dispatcher sends requests to worker threads ③, each using a dedicated hardware core or hyperthread. Most requests will complete their execution without interruption. Network processing for any replies can take place either at the networking subsystem or the worker thread itself to optimize for latency. At a minimum, the worker thread notifies the networking subsystem to free any buffer space allocated for the incoming request ④.

The dispatcher uses timestamps to identify long running requests that should be preempted based on the scheduling policy. Assuming there are queued requests, we preempt running requests after 5 μ sec to 15 μ sec for the workloads we studied (see §4), which is extremely frequent compared to the time slice in the Linux kernel. For example, the CFS scheduler has a target preemption latency of 6ms and a minimum one of 0.75ms. The dispatcher sends an interrupt to the worker thread ⑤, which performs a context switch and receives a different request to run from the dispatcher. The long request is re-enqueued ⑥ in the dispatcher and processed later us-

Preemption Mechanism	Sender Cost	Receiver Cost	Total Latency
Linux Signal	2084	2523	4950
Vanilla IPI	2081	2662	4219
IPI Sender-only Exit	2081	1212	2768
IPI Receiver-only Exit	298	2662	3433
IPI No Exits	298	1212	1993

Table 1: Average preemption overhead in cycles. Sender/receiver cost refers to cycles consumed in the sending/receiving core, including the receiver overhead of invoking an empty interrupt handler. Total cost includes interrupt propagation through the system bus. Hence, it is not equal to the sum of sender and receiver overhead. For “IPI Sender-only Exit” and “Vanilla IPI,” the receiver starts interrupt processing before the sender returns from its VM exit.

ing steps ②-⑥ as many times as needed.

Since Shinjuku is a single-address space operating system, communication between its components occurs over shared memory. We use dedicated pairs of cache lines for each pair of communicating threads (see §3.5).

Similar to IX and ZygOS, Shinjuku leverages the Dune system for process virtualization [15]. With Dune, Linux and the Dune kernel module run in VMX root mode ring 0, where a hypervisor would run in a virtualized system. Shinjuku runs in VMX non-root mode ring 0, where a guest OS would run. This allows it to use very low overhead interrupts while separating the control from the data plane. The application context that uses Shinjuku can run in VMX non-root mode ring 0 or ring 3. For the results in §4, we run applications in VMX non-root mode ring 0 to avoid the address space crossings between Shinjuku and the application code. There is a separate instance of Shinjuku for each low-latency application running on the server.

3.2 Fast Preemption

To use preemptive scheduling at microsecond latencies, Shinjuku requires fast preemption. A naive approach would be for the dispatcher to notify workers using Linux signals. As we show in Table 1, however, signals incur high overheads for both the sender and the receiver (roughly 2.5 μ sec on a 2GHz machine). They require user- to kernel-space transitions plus some kernel processing.

Preemption through interrupts. Direct use of inter-processor interrupts (IPIs) is potentially faster than signals. x86 processors implement IPIs using the Advanced Programmable Interrupt Controller (APIC).

Each core has a local APIC and an I/O APIC is attached to the system bus. To send an IPI, the sending core writes registers in its local APIC which propagates the interrupt via the I/O APIC to the destination core’s APIC, which in turn vectors execution to an interrupt handler.

We extended Dune to support IPIs by virtualizing the local APIC registers. When a non-root thread on core *A* writes its virtual APIC to send interrupt number *V* to core *B*, this causes a VM exit to Dune running in root mode. Dune writes *V* to core *B*’s *posted interrupt descriptor*, and then uses the real APIC to send interrupt 242 to core *B*. That causes core *B* to perform a VM exit to an interrupt handler in Dune, which injects interrupt number *V* into non-root mode on resuming the application.

As Table 1 shows, this vanilla implementation of pre-emption using IPIs is slightly faster than Linux signals but still suffers from significant overheads due to the cost of VM exits in both the sender and the receiver.

Optimized interrupt delivery. We first focus on removing the VM exit on the receiving core *B* (the Shinjuku worker) using *posted interrupts*, an x86 feature for receiving interrupts without a VM exit. To enable posted interrupts, Dune on *B* configures its hardware-defined VM control structure (VMCS) to recognize interrupt 242 as the special *posted interrupt notification vector*. *B* also registers its posted interrupt descriptor with the VMCS. Core *A* still performs a VM exit upon writing the virtual APIC. Dune code on *A* writes *V* into *B*’s posted interrupt descriptor and sends interrupt 242 to *B*. However, *B* then directly injects interrupt *V* without a VM exit. Table 1 shows that eliminating the receiver-side VM exit reduces receiver overhead by 54% (from 2662 to 1212 cycles). This allows frequent preemption of worker threads without significant reduction in useful worker throughput. This receiver overhead consists of modifications to hardware structures, and it cannot be significantly improved without hardware changes, such as support for lightweight user-level interrupts [51].

Optimized interrupt sending. Finally, we remove the VM exit on the sending core (dispatcher thread) by trusting the Shinjuku dispatcher with direct access to the real (non-virtual) APIC. Using the extended page table (EPT), we map both the posted interrupt descriptors of other cores and the local APIC’s registers into the guest physical address space of the Shinjuku dispatcher. Hence, the dispatcher can directly send an IPI without incurring a VM exit. Table 1 shows that eliminating the sender-side VM exit reduces sender overheads down to 298 cycles (149ns in a 2GHz system). This improves

Mechanism	Linux process	Dune process
swapcontext	985	2290
No signal mask	140	140
No FP-restore	36	36
No FP-save	109	109

Table 2: Average overhead in clock cycles of different context-switch mechanisms for both an ordinary Linux process and the Dune process used by Shinjuku.

dispatcher scalability and allows it to serve more requests per second and/or more worker threads (cores).

Table 1 presents the result of combining the sender-side and receiver-side optimization for the interrupt delivery used to support preemption in Shinjuku. The low sender-side overhead (298 cycles) makes it practical to build a centralized, preemptive dispatcher that handles millions of scheduling actions per second. The low receiver-side overhead (1212 cycles) makes it practical to preempt requests as often as every 5 μ sec in order to schedule longer requests without wasting more than 10% of the workers’ throughput.

3.3 Low-overhead Context Switch

When a request is scheduled to an idling core or upon preemption, we context switch between the main context in each worker and the request handling context. The direct approach would be to use the `swapcontext` function in the Linux `ucontext` library. According to Table 2, the overhead is significant in an ordinary Linux process and doubles when used in a Dune process. `swapcontext` requires a system call to set the signal masks during the switch, which requires a VM exit in Dune. The rest of the work in `swapcontext`—i.e., saving/restoring register state and the stack pointer—does not require system calls.

Table 2 evaluates context switch optimizations. First, we skip setting the signal mask which eliminates the system call and brings Dune to parity with ordinary Linux. This introduces the limitation that all tasks belonging to the same application need to share the same signal mask. Next, we exploit that the main worker context does not use floating (FP) instructions. When switching from a request context to the worker context, we must save FP registers as they may have been used in request processing, but we do not need to restore them for the worker context. When switching from the worker context to a request context, we skip saving FP registers and just restore them for the request context. Shinjuku uses the last two options in Table 2 for context switching in worker cores. The overall cost ranges from 36 to 109

cycles (18 to 55ns for a 2GHz system).

3.4 Preemptive Scheduling

The centralized dispatcher and fast preemption and context switch mechanisms allow Shinjuku to implement preemptive scheduling policies. We developed two policies that differ on whether we can differentiate *a priori* between requests types. The policies rely on frequent preemption to provide near-optimal tail latency for any workload, approximating c-FCFS for low dispersion workloads and PS for all other cases.

Single queue (SQ) policy: This policy assumes that we do not differentiate *a priori* between request types and that there is a single service-level agreement (SLO) for tail latency. This is the case, for example, in a search service where we cannot know *a priori* which requests will have longer service times. All incoming requests are placed in a single FCFS queue. When a worker is idle, the dispatcher assigns to it the request at the head of the queue. If requests are processed quickly, this policy operates as centralized FCFS. The dispatcher uses timestamps to identify any request running for more than a predefined quantum (5 to 15 μ sec in our experiments) and, assuming the queue is not empty, preempts it. The request is placed back in the queue and the worker is assigned the request at the current head of the queue. The *c-PRE-SQ* policy evaluated through simulation in Figure 1 is this single queue policy.

Multi queue (MQ) policy: This policy assumes that the network subsystem can identify different request types. For example, it can parse the request header for KVS like Redis and RocksDB and separate simple get/put requests from complex range query requests [33] or use different ports for different request types. Linux already supports peeking into packets with eBPF [2]. Each request type can have a different tail latency SLO. The dispatcher maintains one queue per request type. If only one queue has pending requests, this policy operates just like the single queue policy described above. If more than one queue is non empty, the dispatcher must select a queue to serve when a worker becomes idle or a request is preempted. Once the queue is selected, the dispatcher always takes the request at the head.

The queue selection algorithm is inspired by BVT [24], a process scheduling algorithm for latency sensitive tasks. In BVT, each process has a *warp factor* that quantifies its priority compared to other processes. For Shinjuku, we need a similar warp factor that favors requests with smaller target latency in the short term, but also considers aging of requests with longer latency targets. Since Shinjuku schedules requests and not long running processes with priorities like BVT, the selec-

tion algorithm shown below uses as input the target SLO latency for each queue (e.g., target 99th percentile latency). For the request at the head of each queue, the algorithm uses timestamps to calculate the ratio of the time it has already spent in the system (queuing time) to the SLO target latency for this request type. The queue with the highest such ratio is selected. The algorithm initially favors short requests that can only tolerate short queuing times, but eventually selects long requests that may have been waiting for a while. The per-queue SLO is a user-set parameter. In our experiments, we set it by running each request type individually using the single queue policy and use the observed 99% latency. This captures the requirement that the performance of a request type should not be affected by the existence of requests with different service time distributions.

1 Queue Selection Policy

```

1: procedure QUEUESELECTION(QUEUES):
2:   max  $\leftarrow$  0
3:   max_queue  $\leftarrow$  -1
4:   time  $\leftarrow$  timestamp()
5:   for queue in queues do
6:     cur_ratio  $\leftarrow$   $\frac{\text{time} - \text{queue}[0].\text{timestamp}}{\text{queue}.SLO}$ 
7:     if cur_ratio > max then
8:       max  $\leftarrow$  cur_ratio
9:       max_queue  $\leftarrow$  queue
10:  return max_queue

```

A preempted request can be placed either at the tail of its queue to approximate PS or at the head of the queue to approximate c-FCFS. This choice can be set by the application or based on online measurements of service time statistics. The rule of thumb we use is that for multi-modal or heavy-tailed workloads, the requests should be placed at the tail of the queue, while for light-tailed ones at the head. Frequent preemption is needed even with light-tailed distributions in order to allow Shinjuku to serve the queues for other request types. The *c-PRE-MQ* policy evaluated through simulation in Figure 2 is this multi-queue policy, where both request types are placed at the head of their corresponding queues when preempted.

3.5 Implementation

The current Shinjuku implementation is based on Dune and requires the VT-x virtualization features in x86-64 systems [20]. Dune can be ported to other architectures with similar virtualization support. Our modifications to Dune involve 1365 SLOC. The Shinjuku dispatcher and worker code are 2535 SLOC. The network subsystem we used in §4 is based on IX [16]. All the aforementioned codebases are in C.

API: To use Shinjuku, applications need to register three callback functions: the `init()` function that initializes global application state; the `init_per_core(int core_num)` function that initializes application state for each worker thread (e.g. local variables or configuration options); the `reply * handle_request(request *)` function that handles a single application-level request and returns a pointer to the reply data.

Context management: We use a modified version of the Linux `ucontext` library for context management. The context structure consists of a machine-specific representation of the saved state, the signal mask, a pointer to the context stack, and a pointer to the context that will be resumed when this context finishes execution. The dispatcher allocates context objects and stack space for each request from a memory pool. They are freed by the dispatcher when the request context completes execution and is returned by a worker thread.

Inter-thread communication: In addition to preemption, we use a low-overhead, shared memory communication scheme similar to that used in [50]. Each pair of threads, running on dedicated cores or hyperthreads, communicates over shared pairs of cache lines, one for each direction of communication. The sending thread fills the cache line with the data it wants to send, e.g. request or context pointers, as shown in Figure 3. Then, it sets the value of the byte the receiver polls to notify it that the cache line is ready for reading. This approach requires two cache line state transitions, one from shared to exclusive state which takes place when the sender writes the data and one from exclusive to shared state when the receiver reads the data. The average roundtrip latency for a message sent and received over a cache line is 211 cycles. The dispatcher’s minimum work for sending a message is approximately 70 cycles, i.e. 35ns in a 2GHz machine. This sets a theoretical upper bound of 28 MRPS for the number of requests the dispatcher can handle, assuming all it has to do is to place pointers to the requests in shared memory locations and notify idling workers.

3.6 Discussion

Hardware constraints: §4 shows that a single dispatcher thread can process at least 5M requests per second and comfortably saturate a full socket with 12 cores and 24 hyperthreads. To scale a single application to higher core and/or socket counts, we must improve the dispatcher throughput. The approach we use is to have each dispatcher thread handle a subset of the worker threads and steer requests to different dispatchers using the NIC RSS feature. A relatively simple hardware fea-

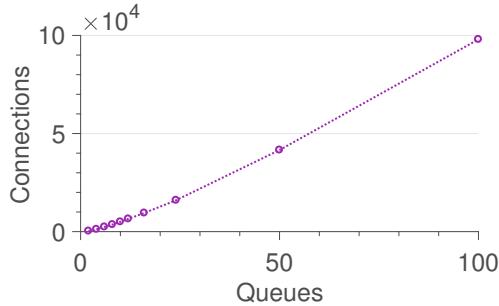


Figure 4: Number of concurrent connections needed for load imbalance among queues to be less than 10% with probability greater than 90%.

ture that would vastly improve the dispatcher scalability would be a low-overhead message passing mechanism among different cores [34, 51]. Ideally, such a mechanism would offer two variations, a preemptive one that would be used for scheduling and a non-preemptive one where messages are added to per core queues and would be used for work assignment.

Connection counts: IX and ZygOS use RSS to distribute requests to workers. Using a Monte-Carlo simulation, we calculate the connection count needed for RSS to keep imbalance below 10% with high probability as we increase the number of cores. As shown in Figure 4, they need 16,000 connections (clients or flows) to avoid imbalance on a server with 24 hyperthreads. High connection counts are common for public facing services (e.g., public load balancer or HTTP server), but not for internal ones. The DCTCP project [9] found at most a few hundred connections to back-end servers over each 1 msec window. In contrast, Shinjuku uses RSS to distribute requests to dispatchers. Since each dispatcher can manage tens of cores, Shinjuku is not subject to the requirement of high connection (clients or flows) counts discussed in §2. For example, 300 connections are sufficient to load balance across 2 dispatchers. When a single dispatcher suffices, Shinjuku will operate efficiently even with a single connection.

Alternative scheduling policies: Shinjuku can support more scheduling policies in addition to the two we presented. In future work, we will explore integrating Shinjuku with datacenter-wide profiling tools [49] and online experimentation tools [55] in order to dynamically infer the service time distributions and adjust the policy accordingly. We will also explore microsecond-scale scheduling policies that are locality- and heterogeneity-aware [30, 27]. For example, consider an application which creates a large memory footprint before responding to a client request. In such

cases, we will want to avoid preempting and context switching as multiple cache lines will have to move to a different core, which can be very expensive.

Control plane: Online services experience load variations, such as a diurnal load patterns and spikes. Hence, it makes sense to adjust over time the number of workers a Shinjuku process uses. Shenango [7] solves this problem by adjusting core allocation between applications in microsecond timescales. We plan to explore the possibility of integrating the two systems.

Security model: The Dune kernel module [15] uses hardware support for virtualization to isolate a Shinjuku process from the Linux kernel and any other process, ordinary Linux or Shinjuku based. Linux can also remove cores and network queues from a Shinjuku process at any time. Within a Shinjuku process, the application code must trust the Shinjuku runtime and, if the application contexts execute in VMX non-root ring 0, the Shinjuku runtime must trust the application code. For example, the fact that APIC registers are mapped in the process address space means that one process could launch a denial-of-service attack on another process by issuing a large number of interrupts to a specific core.

We measured the cost of a ring 3 → ring 0 → ring 3 transition to be only 84 cycles. Future versions of Shinjuku will run application code in ring 3 while the Shinjuku runtime will be running in VMX non-root ring 0 eliminating this attack vector with very small overhead. Moreover, with this approach, bugs in application code will only cause contexts to crash, not affecting the runtime system.

Synchronization in user code: Online services are designed to run well on multiple cores. They synchronize across requests, but synchronization is short and infrequent to achieve scalability. Scalable applications will perform with Shinjuku regardless of whether we disable or allow preemption around read/write locks. We currently disable interrupts during any non thread-safe code, using a `call_safe(fn)` API call to simplify application porting. The runtime overhead of the instructions that are used to disable interrupts is only a few clock cycles and they do not affect the Linux kernel’s abilities to reclaim the cores. Memory allocation code is a special case that often optimizes away locks using thread-local storage. We preload our own version of the C and C++ libraries that disable interrupts (and hence preemption) during the execution of allocation functions. If these functions take a long time, it will affect the tail latency observed with Shinjuku.

Any application that frequently uses coarse-grain or contested locks within requests will scale poorly regard-

less of scheduling policy on any system, including Shinjuku.

4 Evaluation

We compare Shinjuku to IX [16] and ZygOS [46], two recent systems that use d-FCFS and approximate c-FCFS respectively to improve tail latency. All three systems are built on top of Dune [15]. We use the latest IX and ZygOS versions available at [4].

4.1 Experimental Methodology

Machines: We use a cluster of 6 client and one server machines, connected through an Arista 7050-S switch with 48 10GbE ports. The client machines each include two Intel Xeon E5-2630 CPUs operating at 2.3GHz. Their NICs are a mixture of Intel 82599ES and Solarflare SFC9020 10GbE NICs. The server machine that runs IX, ZygOS, or Shinjuku includes two Intel E5-2658 CPUs operating at 2.3GHz, 128GB of DRAM, and an Intel 82599ES 10Gb NIC. All machines run Ubuntu LTS 16.0.4 with the 4.4.0 Linux kernel. Hyperthreading is always enabled unless noted. NICs are configured as half-duplex by the IX and ZygOS drivers and we use the same setting for Shinjuku. To perform scalability experiments, we also use the server machine with a 40Gb Intel XL710-QDA2 NIC and an identical E5-2658 two-socket machine as the client.

Each of the two server CPUs has 12 cores and 24 hyperthreads. However, ZygOS and IX can only support up to 16 hyperthreads as their network drivers are limited to 16 RSS RX queues. Hence, we use an 8-core (16-hyperthread) configuration for most experiments. Shinjuku always uses two of the available hyperthreads for the networking subsystem and dispatcher. Hence, our results use the notation $\text{Shinjuku}(x)$ to specify that Shinjuku uses $x-2$ hyperthreads for workers and a total of x hyperthreads. The notation $\text{IX}(x)$ and $\text{ZygOS}(x)$ specify that IX and ZygOS use x hyperthreads, all for d-FCFS or c-FCFS processing respectively.

Networking: We use the following networking subsystem with Shinjuku. A single hyperthread, co-located on the same physical core with the dispatcher, polls the NIC queue and processes raw packets. It performs UDP processing, identifies requests, and optionally parses the request header to identify types. The Shinjuku workers process network replies. This simple subsystem is sufficient to evaluate Shinjuku. Since Shinjuku decouples network processing from request scheduling, we can combine Shinjuku in the future with alternative systems that implement other transport protocols and use optimizations such as multithreaded stacks [31, 22] or stacks

that offload networking to a SmartNIC [33, 18, 53, 19]. The latter will free x86 hyperthreads for Shinjuku workers. If the SmartNIC is connected to the processor chip through a coherent interconnect like Intel’s UPI, we can also offload the Shinjuku dispatcher to the NIC cores.

IX supports both UDP and TCP networking. We use it with UDP and a batch size of 64. ZygOS supports only TCP networking [4], but is configured to use exactly one TCP segment per request and reply. Hence, ZygOS requests have some additional service time for TCP processing ($< 0.25\mu\text{sec}$), but are otherwise similar to UDP-based IX and Shinjuku requests.

Workloads: We use one synthetic and one real workload. The synthetic workload is a server application where requests perform dummy work that we can control in order to emulate any target distribution of service times. This synthetic server allows us to derive insights about how the three systems compare across a large application space.

We also use RocksDB version 5.13 [26], a popular and widely deployed key-value store developed by Facebook. The IX, ZygOS, and Shinjuku servers handle RocksDB queries that may be simple get/put requests or range scans. We configure RocksDB to keep all data in DRAM in order to evaluate all three systems under the lowest latency requirements possible. If some RocksDB requests had to access data in Flash, the variability of service times would be even higher, and the preemptive Shinjuku would perform even better than the non-preemptive IX and ZygOS.

We developed an **open loop** load generator similar to mutilate [36] that transmits requests over either TCP or UDP. The load generator starts a large number of connections in a set of client machines, while it measures latency from a single unloaded machine. Unless otherwise noted, we use 1920 persistent TCP connections (ZygOS) and 1920 distinct UDP 5-tuples (IX and Shinjuku). Using fewer connections significantly affected the performance of IX and ZygOS (see §3.6).

4.2 Synthetic Workload Comparison

Figure 5 compares Shinjuku to IX and ZygOS for three service time distributions. Figure 5a uses a fixed service time of $1\mu\text{sec}$, while Figure 5b uses an exponential distribution with a mean of $1\mu\text{sec}$. These two cases are ideal for IX that uses d-FCFS. IX benefits further from its ability to batch similarly sized requests. Shinjuku (SQ) performs close to IX, despite exclusively using two hyperthreads for networking and the dispatcher and despite preempting requests that exceed $5\mu\text{sec}$. In this case, Shinjuku places preempted requests at the head of the queues. Moreover, preemption is fast and for light-

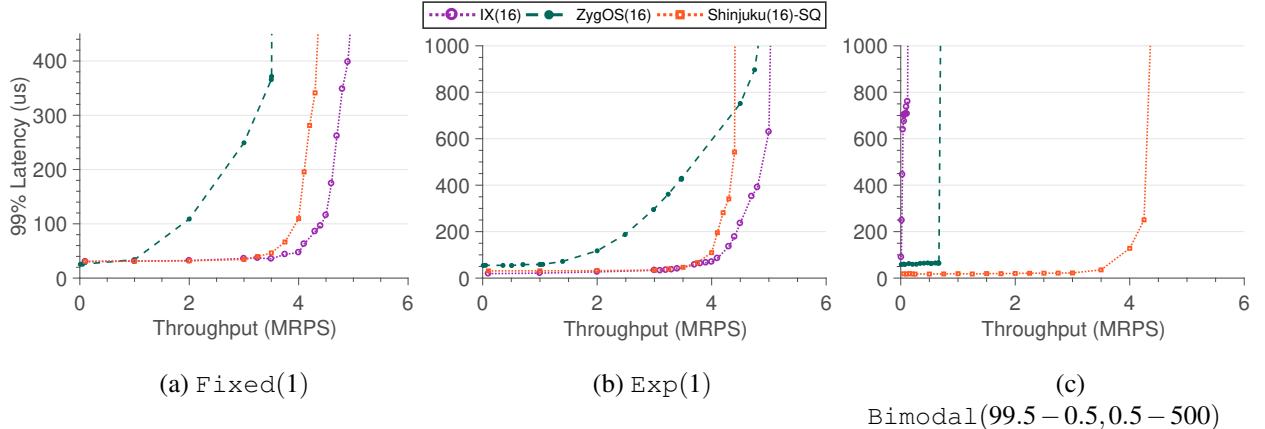


Figure 5: Systems comparison with synthetic workloads. Shinjuku uses the single queue policy.

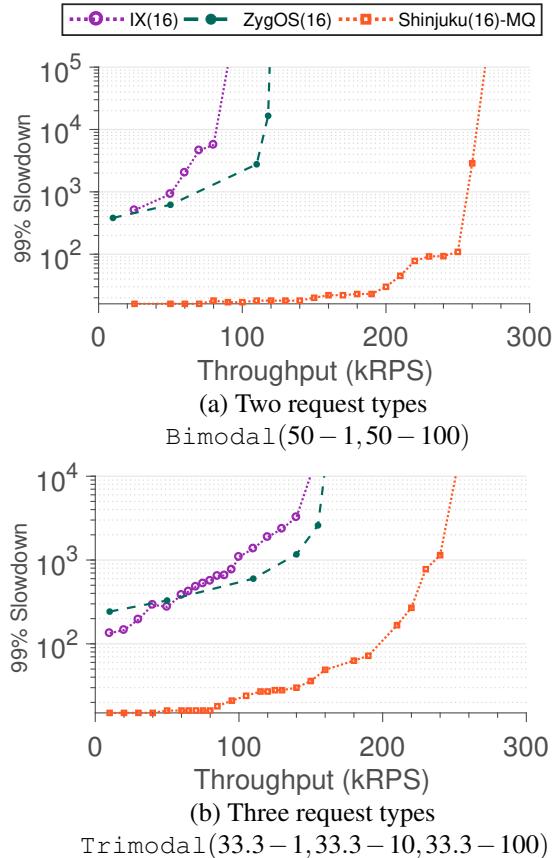


Figure 6: Systems comparison with multi-modal synthetic workloads. Shinjuku uses the multi-queue policy.

tailed workloads only a few requests will be preempted allowing Shinjuku to outperform ZygOS for both scenarios. ZygOS also has a high stealing rate (60%) even for homogeneous workloads which exacerbates its stealing overheads. A similar performance drop was also observed in the original ZygOS paper [46].

Figure 5c uses a Bimodal(99.5 – 0.5, 0.5 – 500) service time distribution where 99.5% of the requests have a $0.5\mu\text{sec}$ service time and 0.5% $500\mu\text{sec}$. Shinjuku with the single queue policy is vastly better than both IX and ZygOS, achieving up to **50% lower tail latency at low load** and **5x better throughput** for a given $300\mu\text{sec}$ tail latency target. IX and ZygOS lack pre-emption, hence the 0.5% of long requests determine the overall 99th percentile tail latency as short requests are frequently blocked behind them. The task stealing in ZygOS improves upon IX but is not sufficient to deal with the high dispersion in service times. In contrast, Shinjuku preempts long requests and places them at the tail of the single queue to allow short requests to complete quickly.

Figure 6 evaluates the three systems with multiple request types, a key experiment that was missing from the original IX and ZygOS papers. We use Shinjuku’s multi-queue policy which assumes knowledge of the request types (e.g., from packet inspection). Figure 6a uses a Bimodal(50 – 1, 50 – 100) workload, while Figure 6b uses a Trimodal(33 – 1, 33 – 10, 33 – 100) workload. In all cases, Shinjuku places preempted requests to the head of their corresponding queues. Both figures show the 99th percentile of request slowdown (overall request latency / service time) with a logarithmic y-axis. The preemptive, multi-queue policy allows Shinjuku to outperform IX and ZygOS by having **94% lower slowdown at low load and over 2x higher throughput (RPS)**. In addition to the frequent preemption that avoids head-of-line blocking, Shinjuku benefits from its ability to select the type of requests (long vs. short) to serve next based on their ratio of queuing time to target latency.

4.3 Shinjuku Analysis

How important is frequent preemption? Figure 7a

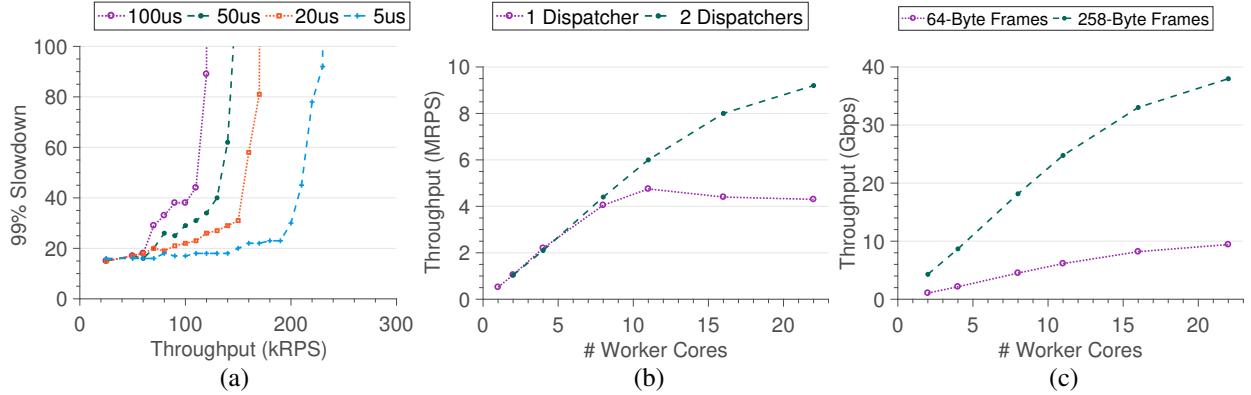


Figure 7: (a) Two request types Bimodal(50 – 1, 50 – 100) with varying preemption time slice. (b) Shinjuku throughput (Million RPS) as we scale the worker cores. (c) Shinjuku throughput (Gbps) as we scale the worker cores.

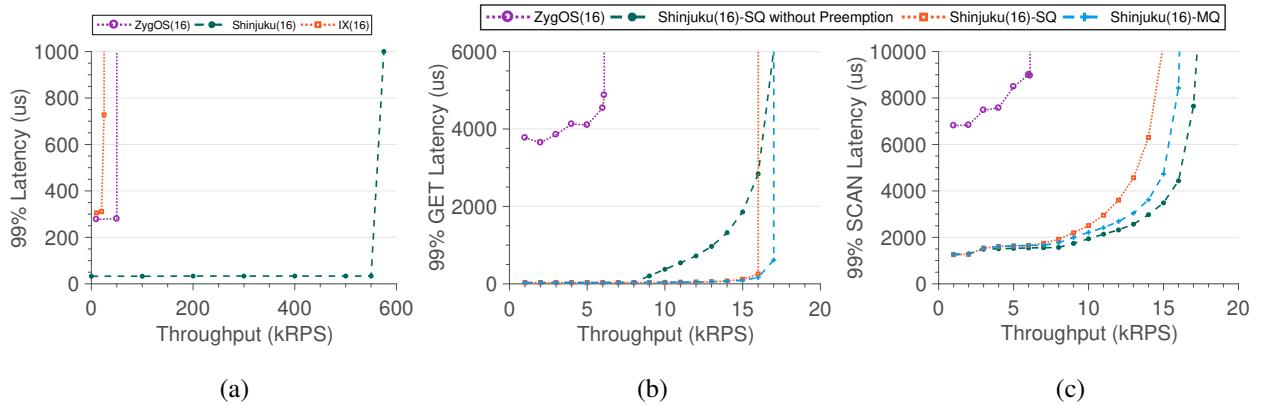


Figure 8: RocksDB (a) Shinjuku, IX, and ZygOS 99.5% GET 0.5% SCAN(1000). (b + c) Shinjuku Performance - 50% GET(b) 50% SCAN(5000)(c).

varies the preemption interval for a Bimodal(50 – 1, 50 – 100) synthetic workload. Shinjuku uses the multi-queue policy. The shorter the preemption interval, the better Shinjuku performs as the impact of 100 μ sec requests on 1 μ sec is reduced. Shinjuku performs well even at the very frequent 5 μ sec preemption interval.

How does Shinjuku scale? Figures 7{b,c} examine how Shinjuku scales with more workers. We issue short requests with 1 μ sec fixed service time to stress the dispatcher. We also use the Intel XL710-QDA2 40Gb NICs so that networking is not a bottleneck. Since each worker thread can saturate its core, we turn off hyper-threading and pin each worker thread to a physical core. We use the two hyperthreads in the 12th physical core for the dispatcher and the networking threads. Figure 7b shows that a single dispatcher thread scales almost linearly to 11 worker cores, which is the socket size in our server. A second dispatcher thread allows Shinjuku to schedule across the 22 worker cores on both sockets for a single application. Shinjuku can schedule 5M and 9.5M RPS with 1 and 2 dispatchers respectively. Figure 7c measures the outgoing network throughput of

Shinjuku using two dispatchers. Shinjuku saturates the 40Gb NIC when reply frames are as short as 258 bytes.

These two figures validate that a single Shinjuku application can scale to high core counts and high line rates even with short 1 μ sec service times.

4.4 RocksDB Comparison

We use RocksDB with a simple server we ported to IX, ZygOS, and Shinjuku. Client requests are looked up in a RocksDB database created on an in-memory file system (/tmpfs/) with random key-value pairs. We use two request types: GET requests for a single key-value pair that execute within 6 μ sec; SCAN requests that scan 1,000 or 5,000 key-value pairs and require 240 μ sec or 1,200 μ sec respectively. We use memory-mapped plain tables as the backing files to avoid memory copies and access to block devices. Shinjuku uses a preemption time slice of 15 μ sec and places preempted requests at the head of their corresponding queues for the multi-queue policy and at the tail for the single-queue policy.

Figure 8a compares IX, ZygOS, and Shinjuku with the single queue policy for a 99.5-0.5 mix of GET and

SCAN(1000) requests. Shinjuku provides a vast improvement over ZygOS in tail latency (88% decrease) and throughput (6.6x improvement). Frequent preemption in Shinjuku allows GET requests to avoid long queuing times due to SCAN requests. IX performs even worse due to the combination of highly imbalanced request service times and d-FCFS scheduling.

Preemption and queue selection policy matter: Figures 8b and 8c use a 50-50 workload between GET and SCAN(5000) requests. In addition to comparing with ZygOS, we modified the Shinjuku dispatcher to show the impact of using Shinjuku without preemption, the single-queue preemptive policy, and the multi-queue preemptive policy. IX is omitted because its latency is outside the range of our plot. The results show that Shinjuku without preemption (SQ without preemption) favors the longer SCAN requests over the shorter GET requests. The addition of preemption (SQ) fixes this problem and allows both request types to achieve fair throughput and low tail latency. The multi-queue policy (MQ) improves SCAN requests as it avoids excessive queuing for them as well. ZygOS performs significantly worse even than Shinjuku without preemption. ZygOS uses distributed queuing and is susceptible to head-of-line blocking for requests within the same connection. This supports our decision to decouple network processing and request scheduling in Shinjuku.

5 Related Work

Optimized network stacks: There is significant work in optimizing network stacks, including polling based processing (DPDK [3]), multi-core scalability (mTCP [31]), modularity and specialization (Sandstorm [40]), and OS bypass (Andromeda [22]). Shinjuku is orthogonal to this work as it optimizes request scheduling after network protocol processing.

Dataplane operating systems: Several recent systems optimize for throughput and tail latency by separating the OS dataplane from the OS control plane, an idea originating in Exokernel [25]. IX [16], Arrakis [45], MICA [39], Chronos [32], and ZygOS [46] fall in this category. Shinjuku improves on these systems by introducing preemptive scheduling that allows short requests to avoid excessive queuing.

Task scheduling: Li *et al.* [38] control tail latency by reducing the amount of resources dedicated to long-running requests that violate the SLO. Haque *et al.* [29] take the opposite approach and devote more resources to stragglers so that they finish faster. Interestingly, both approaches work well. However, these approaches are applicable to millisecond-scale workloads and require workloads that are dynamically parallelizable. Shinjuku

allows the development of efficient scheduling policies for requests 3 orders of magnitude shorter than what this line of work can handle.

Flow scheduling: PIAS [12] is a network flow scheduling mechanism that uses hardware priority queues available in switches to approximate the Shortest Job First (SJF) scheduling policy and prioritize short flows over long ones. We do not follow a similar approach in Shinjuku as SJF is optimal in terms of minimizing average but not tail latency [57]. Moreover, in order to be effective, PIAS requires some form of congestion control to keep the queue length short. This is not practical in non-networked settings where the runtime does not control the application.

Exit-less interrupts: The idea of safe, low-overhead interrupts was introduced in ELI for fast delivery of interrupts to VMs [10]. ZygOS [46] uses inter-processor interrupts for work stealing but does not implement preemptive scheduling. Shinjuku uses Dune [15] to optimize processor-to-processor interrupts.

User-space thread management: Starting with scheduler activations [11], there have been several efforts to implement efficient, user-space thread libraries [8, 56, 48, 1]. They all focus on cooperative scheduling. Shinjuku shows that preemptive scheduling is practical at microsecond-scales and leads to low tail latency and high throughput.

6 Conclusion

Shinjuku uses hardware support for virtualization to make frequent preemption practical at the microsecond scale. Hence, its scheduling policies can avoid the common pitfall of non-preemptive policies where short requests are blocked behind long requests. Shinjuku provides low tail latency and high throughput for a wide range of distributions and request service times regardless of the number of client connections. For the RocksDB KVS, we show that Shinjuku improves upon the recently published ZygOS system by 6.6x in throughput and 88% in tail latency.

Acknowledgements

We thank our shepherd, Irene Zhang, and the anonymous NSDI reviewers for their helpful feedback. We also thank John Ousterhout, Adam Wierman, and Ana Klimovic for providing feedback on early versions of this paper. This work was supported by the Stanford Platform Lab and by gifts from Google, Huawei, and Samsung.

References

- [1] libfiber: A user space threading library supporting multi-core systems. <https://github.com/brianwatling/libfiber>, 2015.
- [2] ebpf - extended berkeley packet filter. <http://prototype-kernel.readthedocs.io/en/latest/bpf/>, 2016.
- [3] Data plan development kit. <http://www.dpdk.org/>, 2018.
- [4] Ix-project: Protected dataplane for low latency and high performance. <https://github.com/ix-project/>, 2018.
- [5] libuv: Cross-platform asynchronous i/o. <https://libuv.org/>, 2018.
- [6] Memcached. <https://memcached.org/>, 2018.
- [7] Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.
- [8] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, ATC '02*, pages 289–302. USENIX Association, 2002.
- [9] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sen-gupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10*, pages 63–74, New Delhi, India, 2010. ACM.
- [10] Nadav Amit, Abel Gordon, Nadav Har’El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafir. Bare-metal performance for virtual machines with exitless interrupts. *Commun. ACM*, 59(1):108–116, December 2015.
- [11] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, SOSP '91*, pages 95–109, Pacific Grove, California, USA, 1991. ACM.
- [12] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, Oakland, CA, 2015. USENIX Association.
- [13] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003.
- [14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.
- [15] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, pages 335–348, Hollywood, CA, USA, 2012. USENIX Association.
- [16] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14*, pages 49–65, Broomfield, CO, 2014. USENIX Association.
- [17] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the “micro” back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, 2018. USENIX Association.
- [18] Broadcom. Ps225. <https://www.broadcom.com/products/ethernet-connectivity/network-adapters/ps225>, 2017.
- [19] Cavium. Liquidio smartnic. <https://www.cavium.com/product-liquidio-adapters.html>, 2018.
- [20] Intel Corp. Intel virtualization technology. <https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html>, 2018.
- [21] Microsoft Corp. Receive side scaling. <http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx>, 2018.
- [22] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Are-fin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermenio, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, 2018. USENIX Association.
- [23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.
- [24] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP '99*, pages 261–276, Charleston, South Carolina, USA, 1999. ACM.
- [25] D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95*, pages 251–266, Copper Mountain, Colorado, USA, 1995. ACM.
- [26] Facebook. Rocksdb. <http://rocksdb.org/>, 2018.
- [27] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10*, pages 341–342, Bangalore, India, 2010. ACM.

- [28] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.
- [29] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 161–175, Istanbul, Turkey, 2015. ACM.
- [30] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 625–638, Cambridge, Massachusetts, 2017. ACM.
- [31] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sungewan Ihm, Dongsu Han, and KyongSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, 2014. USENIX Association.
- [32] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14, San Jose, California, 2012. ACM.
- [33] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, Atlanta, Georgia, USA, 2016. ACM.
- [34] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, San Diego, California, USA, 2007. ACM.
- [35] Redis Labs. Redis. <https://redis.io/>, 2018.
- [36] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, Amsterdam, The Netherlands, 2014. ACM.
- [37] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, Seattle, WA, USA, 2014. ACM.
- [38] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 14:1–14:13, Barcelona, Spain, 2016. ACM.
- [39] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Seattle, WA, 2014. USENIX Association.
- [40] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, Chicago, Illinois, USA, 2014. ACM.
- [41] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blel, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zets. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, London, United Kingdom, 2015. ACM.
- [42] Jayakrishnan Nair, Adam Wierman, and Bert Zwart. The fundamentals of heavy-tails: Properties, emergence, and identification. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 387–388, Pittsburgh, PA, USA, 2013. ACM.
- [43] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.
- [44] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.
- [45] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, 2014. USENIX Association.
- [46] George Prekas, Marios Kogias, and Edouard Bugnion. Zygote: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, Shanghai, China, 2017. ACM.
- [47] N. Provos and N. Mathewson. libevent: An event notification library. <http://libevent.org>, 2018.
- [48] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, 2018. USENIX Association.
- [49] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

- [50] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, pages 342–358, Shanghai, China, 2017. ACM.
- [51] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, Pittsburgh, Pennsylvania, USA, 2010. ACM.
- [52] Mellanox Technologies. Rdma over converged ethernet. http://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf , 2014.
- [53] Mellanox Technologies. Bluefield multicore system on chip. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf , 2017.
- [54] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, pages 18–32, Farmington, Pennsylvania, 2013. ACM.
- [55] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, Savannah, GA, 2016. USENIX Association.
- [56] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, pages 268–281, Bolton Landing, NY, USA, 2003. ACM.
- [57] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Oper. Res.*, 60(5):1249–1257, September 2012.

Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads

Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, Hari Balakrishnan
MIT CSAIL

Abstract

Datacenter applications demand microsecond-scale tail latencies and high request rates from operating systems, and most applications handle loads that have high variance over multiple timescales. Achieving these goals in a CPU-efficient way is an open problem. Because of the high overheads of today’s kernels, the best available solution to achieve microsecond-scale latencies is kernel-bypass networking, which dedicates CPU cores to applications for spin-polling the network card. But this approach wastes CPU: even at modest average loads, one must dedicate enough cores for the *peak* expected load.

Shenango achieves comparable latencies but at far greater CPU efficiency. It reallocates cores across applications at very fine granularity—every 5 μ s—enabling cycles unused by latency-sensitive applications to be used productively by batch processing applications. It achieves such fast reallocation rates with (1) an efficient algorithm that detects when applications would benefit from more cores, and (2) a privileged component called the IOKernel that runs on a dedicated core, steering packets from the NIC and orchestrating core reallocations. When handling latency-sensitive applications, such as memcached, we found that Shenango achieves tail latency and throughput comparable to ZygOS, a state-of-the-art, kernel-bypass network stack, but can linearly trade latency-sensitive application throughput for batch processing application throughput, vastly increasing CPU efficiency.

1 Introduction

In many datacenter applications, responding to a single user request requires responses from thousands of software services. To deliver fast responses to users, it is necessary to support high request rates and microsecond-scale tail latencies (e.g., 99.9th percentile) [10, 24, 28, 56, 67]. This is particularly important for requests with service times of only a couple of microseconds (e.g., memcached [43] or RAMCloud [57]). Networking hardware has risen to the occasion; high-speed networks today provide round-trip times (RTTs) on the order of a few μ s [54, 55]. However, when applications run atop current operating systems and network stacks, latencies are in the *milliseconds*.

At the same time, as Moore’s law slows and network rates rise [26], CPU efficiency becomes paramount. In large-scale datacenters, even small improvements in

CPU efficiency (the fraction of CPU cycles spent performing useful work) can save millions of dollars [72]. As a result, datacenter operators commonly fill any cores left unused by latency-sensitive tasks with batch-processing applications so they can keep CPU utilization high as load varies over time [16]. For example, Microsoft Bing collocates latency-sensitive and batch jobs on over 90,000 servers [34], and the median machine in a Google compute cluster runs eight applications [76].

Unfortunately, existing systems do a poor job of achieving high CPU efficiency when they are also required to maintain microsecond-scale tail latency. Linux can only support microsecond latency when CPU utilization is kept low, leaving enough idle cores available to quickly handle incoming requests [41, 43, 76]. Alternatively, kernel-bypass network stacks such as ZygOS are able to support microsecond latency at higher throughput by circumventing the kernel scheduler [2, 18, 50, 57, 59, 61]. However, these systems still waste significant CPU cycles; instead of interrupts, they rely on spin-polling the network interface card (NIC) to detect packet arrivals, so the CPU is always in use even when there are no packets to process. Moreover, they lack mechanisms to quickly reallocate cores across applications, so they must be provisioned with enough cores to handle peak load.

This tension between low tail latency and high CPU efficiency is exacerbated by the bursty arrival patterns of today’s datacenter workloads. Offered load varies not only over long timescales of minutes to hours, but also over timescales as short as a few microseconds. For example, micro bursts in Google’s Gmail servers cause sudden 50% increases in CPU usage [12], and, in Microsoft’s Bing service, 15 threads can become runnable in just 5 μ s [34]. This variability requires that servers leave extra cores idle at all times so that they can keep tail latency low during bursts [16, 34, 41].

Why do today’s systems force us to waste cores to maintain microsecond-scale latency? A recent paper from Google argues that poor tail latency and efficiency are the result of system software that has been tuned for millisecond-scale I/O (e.g., disks) [15]. Indeed, today’s schedulers only make thread balancing and core allocation decisions at coarse granularities (every four milliseconds for Linux and 50–100 milliseconds for Arachne [63] and IX [62]), preventing quick reactions to load imbalances.

This paper presents **Shenango**, a system that focuses

on achieving three goals: (1) microsecond-scale end-to-end tail latencies and high throughput for datacenter applications; (2) CPU-efficient packing of applications on multi-core machines; and (3) high application developer productivity, thanks to synchronous I/O and standard programming abstractions such as lightweight threads and blocking TCP network sockets.

To achieve its goals, Shenango solves the hard problem of reallocating cores across applications at very fine time scales; it reallocates cores every 5 microseconds, orders of magnitude faster than any system we are aware of. Shenango proposes two key ideas. First, Shenango introduces an efficient algorithm that accurately determines when applications would benefit from additional cores based on runnable threads and incoming packets. Second, Shenango dedicates a single busy-spinning core per machine to a centralized software entity called the *IOKernel*, which steers packets to applications and allocates cores across them. Applications run in user-level *runtimes*, which provide efficient, high-level programming abstractions and communicate with the *IOKernel* to facilitate core allocations.

Our implementation of Shenango uses existing Linux facilities, and we have made it available at <https://github.com/shenango>. We found that Shenango achieves similar throughput and latency to ZygOS [61], a state-of-the-art kernel-bypass network stack, but with much higher CPU efficiency. For example, Shenango can achieve over five million requests per second of memcached throughput while maintaining 99.9th percentile latency below 100 μ s (one million more than ZygOS). However, unlike ZygOS, Shenango can linearly trade memcached throughput for batch application throughput when request rates are lower than peak load. To our knowledge, Shenango is the first system that can both multiplex cores and maintain low tail latency during microsecond-scale bursts in load. For example, Shenango’s core allocator reacts quickly enough to keep 99.9th percentile latency below 125 μ s even during an extreme shift in load from one hundred thousand to five million requests per second.

2 The Case Against Slow Core Allocators

In this section, we explain why millisecond-scale core allocators are unable to maintain high CPU efficiency when handling microsecond-scale requests. We define *CPU efficiency* as the fraction of cycles spent doing application-level work, as opposed to busy-spinning, context switching, packet processing, or other systems software overhead.

Modern datacenter applications experience request rate and service time variability over multiple

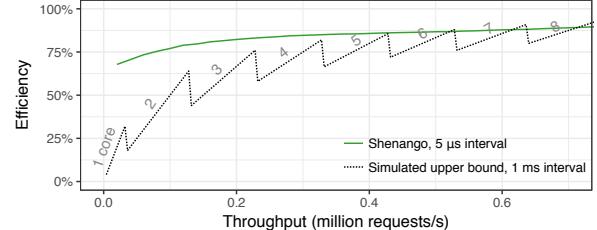


Figure 1: With 5 μ s intervals between core reallocations, a Shenango runtime achieves higher CPU efficiency than an optimal simulation of a 1 ms core allocator.

timescales [16]. To provide low latency in the face of these fluctuations, most kernel bypass network stacks, including ZygOS [61], statically provision cores for peak load, wasting significant cycles on busy polling. Recently, efforts such as IX [62] and Arachne [63] introduced user-level core allocators that adjust core allocations at 50–100 millisecond intervals. Similarly, Linux rebalances tasks across cores primarily in response to millisecond-scale timer ticks. Unfortunately, all of these systems adjust cores too slowly to handle microsecond-scale requests efficiently.

To show why, we built a simulator that determines a conservative upper-bound on the CPU efficiency of a core allocator that adjusts cores at one millisecond intervals. The simulator models an M/M/n/FCFS queuing system and determines through trial and error the minimum number of cores needed to maintain a tail latency limit for a given level of offered load. We assume a Poisson arrival process (empirically shown to be representative of Google’s datacenters [53]), exponentially distributed service times with a mean of 10 μ s, and a latency limit of 100 μ s at the 99.9th percentile. To eliminate any time dependence on past load, we also assume that the arrival queue starts out empty at the beginning of each one millisecond interval and that all pending requests can be processed immediately at the end of each millisecond interval. Together, these assumptions allow us to calculate the best case CPU efficiency regardless of the core allocation algorithm used.

Figure 1 shows the relationship between offered load and CPU efficiency (cycles used divided by cycles allocated) for our simulation. It also shows the efficiency of a Shenango runtime running the same workload locally by spawning a thread to perform synthetic work for the duration of each request. For the simulated results, we label each line segment with the number of cores assigned by the simulator; the sawtooth pattern occurs because it is only possible to assign an integer number of cores. Even with zero network or systems software overhead, mostly idle cores must be reserved to absorb bursts in load, resulting in a loss in CPU efficiency. This loss is especially

severe between one and four cores, and as load varies over time, applications are likely to spend a significant amount of time in this low-efficiency region. The ideal system would spin up a core for exactly the duration of each request and achieve perfect efficiency, as application-level work would correspond one-to-one with CPU cycles. Shenango comes close to this ideal, yielding significant efficiency improvements over the theoretical upper bound for a slow allocator, despite incurring real-world overheads for context switching, synchronization, etc.

On the other hand, a slow core allocator is likely to perform worse than its theoretical upper bound in practice. First, CPU efficiency would be even lower if there were more service time variability or tighter tail-latency requirements. Second, if the average request rate were to change during the adjustment interval, latency would spike until more cores could be added; in Arachne, load changes result in latency spikes lasting a few hundred milliseconds (§7.2) and in IX they last 1-2 seconds [62]. Finally, accurately predicting the relationship between number of cores and performance over millisecond intervals is extremely difficult; both IX and Arachne rely on load estimation parameters that may need to be hand tuned for different applications [62, 63]. If the estimate is too conservative, latency will suffer, and, if it is too liberal, unnecessary cores will be wasted. We now discuss how Shenango’s fast core allocation rate allows it to overcome these problems.

3 Challenges and Approach

Shenango’s goal is to optimize CPU efficiency by granting each application as few cores as possible while avoiding a condition we call *compute congestion*, in which failing to grant an additional core to an application would cause work to be delayed by more than a few microseconds. This objective frees up underused cores for use by other applications, while still keeping tail latency in check.

Modern services often experience very high request rates (millions of packets per second on a single server), and core allocation overheads make it infeasible to scale to per-request core reallocations. Instead, Shenango closely approximates this ideal, detecting load changes every five microseconds and adjusting core allocations over 60,000 times per second. Such a short adjustment interval requires new approaches to estimating load. We now discuss these challenges in more detail.

Core allocations impose overhead. The speed at which cores can be reallocated is ultimately limited by reallocation overheads: determining that a core should be reallocated, instructing an application to

yield a core, etc. Existing systems impose too much overhead for microsecond-scale core reallocations to be practical: Arachne requires 29 microseconds of latency to reallocate a core [63], and IX requires hundreds of microseconds because it must update NIC rules for steering packets to cores [62].

Estimating required cores is difficult. Previous systems have used application-level metrics such as latency, throughput, or core utilization to estimate core requirements over long time scales [22, 34, 48, 63]. However, these metrics cannot be applied over microsecond-scale intervals. Instead, Shenango aims to estimate instantaneous load, but this is non-trivial. While requests arriving over the network provide one source of load, applications themselves can independently spawn threads.

3.1 Shenango’s Approach

Shenango addresses these challenges with two key ideas. First, Shenango considers both thread and packet queuing delays as signals of compute congestion, and it introduces an efficient *congestion detection algorithm* that leverages these signals to decide if an application would benefit from more cores. This algorithm requires fine-grained, high-frequency visibility into each application’s thread and packet queues. Thus, Shenango’s second key idea is to dedicate a single, busy-spinning core to a centralized software entity called the *IOKernel* (§4). The IOKernel process runs with root privileges, serving as an intermediary between applications and NIC hardware queues. By busy-spinning, the IOKernel can examine thread and packet queues at microsecond-scale to orchestrate core allocations. Moreover, it can provide low-latency access to networking and enable steering of packets to cores in software, allowing packet steering rules to be quickly reconfigured when cores are reallocated. The result is that core reallocations complete in only 5.9 μ s and require less than two microseconds of IOKernel compute time to orchestrate. These overheads support a core allocation rate that is fast enough to both adapt to shifts in load and quickly correct any mispredictions in our congestion detection algorithm.

Application logic runs in per-application *runtimes* (§5), which communicate with the IOKernel via shared memory (Figure 2). Each runtime is untrusted and is responsible for providing useful programming abstractions, including threads, mutexes, condition variables, and network sockets. Applications link with the Shenango runtime as a library, allowing kernel-like functions to run within their address spaces.

At start-up, the runtime creates multiple kernel threads (i.e., pthreads), each with a local runqueue, up to the maximum number of cores the runtime may use.

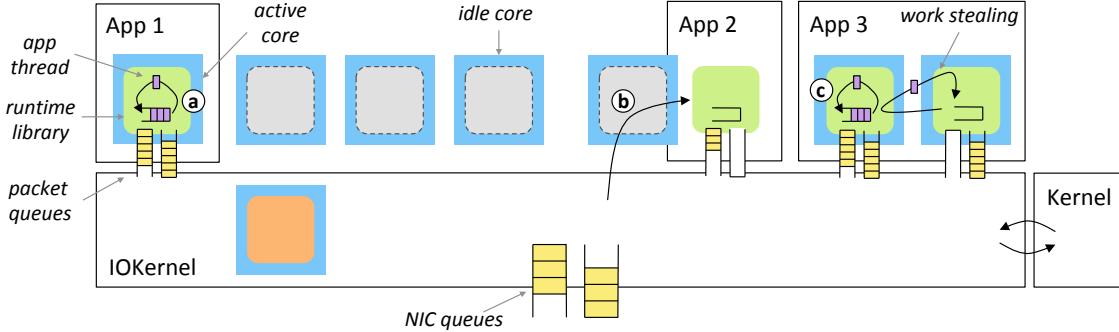


Figure 2: Shenango architecture. (a) User applications run as separate processes and link with our kernel-bypass runtime. (b) The IOKernel runs on a dedicated core, forwarding packets and allocating cores to runtimes. (c) The runtime schedules lightweight application threads on each core and uses work stealing to balance load.

Application logic runs in lightweight user-level threads that are placed into these queues; work is balanced across cores via work stealing. We refer to each per-core kernel thread created by the runtime as a *kthread* and to the user-level threads as *uthreads*. Shenango is designed to coexist inside an unmodified Linux environment; the IOKernel can be configured to manage a subset of cores while the Linux scheduler manages others.

4 IOKernel

The IOKernel runs on a dedicated core and performs two main functions:

1. At any given time, it decides how many cores to allocate to each application (§4.1.1) and which cores to allocate to each application (§4.1.2).
2. It handles all network I/O, bypassing the kernel. On the receive path, it directly polls the NIC receive queue and places each incoming packet onto a shared memory queue for one of the application’s cores. On the transmission path, it polls each runtime’s packet egress queues and forwards packets to the NIC (§4.2).

4.1 Core Allocation

The IOKernel must make core allocation decisions quickly because any time it spends on core allocations cannot be spent forwarding packets, thereby decreasing throughput. For simplicity, the IOKernel decouples its two decisions; in most cases, it first decides if an application should be granted an additional core, and then decides which core to grant.

4.1.1 Number of cores per application

Each application’s runtime is provisioned with a number of *guaranteed cores* and a number of *burstable cores*. A runtime is always entitled to use its guaranteed cores without risk of preemption (oversubscription is not allowed), but it may use fewer (even zero) cores if it

does not have enough work to occupy them. When extra cores are available, the IOKernel may allocate them as *burstable cores*, allowing busy runtimes to temporarily exceed their guaranteed core limit.

When deciding how many cores to grant a runtime, the IOKernel’s objective is to minimize the number of cores allocated to each runtime, while still avoiding compute congestion (§3). To determine when a runtime has more cores than necessary, the IOKernel relies on runtime kthreads to voluntarily yield cores when they are unneeded. When a kthread cannot find any work to do, meaning its local runqueue is empty and it did not find stealable work from other active kthreads, it cedes its core and notifies the IOKernel (we refer to this as *parking*). The IOKernel may also preempt burstable cores at any time, forcing them to park immediately.

The IOKernel leverages its unique vantage point to detect incipient compute congestion by monitoring the queue occupancies of active kthreads. When a packet arrives for a runtime that has no allocated cores, the IOKernel immediately grants it a core. To monitor active runtimes for congestion, the IOKernel invokes the *congestion detection algorithm* at 5 μ s intervals (Algorithm 1).

The congestion detection algorithm determines whether a runtime is overloaded or not based on two sources of load: queued threads and queued ingress packets. If any item is found to be present in a queue for two consecutive runs of the detection algorithm, it indicates that a packet or thread queued for at least 5 μ s. Because queued packets or threads represent work that could be handled in parallel on another core, the runtime is deemed to be “congested,” and the IOKernel grants it one additional core. We found that the duration of queuing is a more robust signal than the length of a queue, because using queue length requires carefully tuning a threshold parameter for different durations of requests [63, 74].

Implementing the queues as ring buffers enables a

Algorithm 1 Congestion Detection Algorithm

```
1: for each application app do
2:   for each active kthread k of app do
3:     rung  $\leftarrow$  k's runqueue
4:     prev_rung  $\leftarrow$  k's rung last iteration
5:     inq  $\leftarrow$  k's ingress packet queue
6:     prev_inq  $\leftarrow$  k's inq last iteration
7:     if rung contains threads in prev_rung or
8:       inq contains packets in prev_inq then
9:         try to allocate a core to app
10:        break     $\triangleright$  go to next app in outer loop
```

simple and efficient detection mechanism. Detecting that an item is present in a queue for two consecutive intervals is simply a matter of comparing the current head pointer with the tail pointer from the previous iteration. Runtimes expose this state to the IOKernel in a single cache line of shared memory per kthread.

Intuitively, core allocation is capable of oscillatory behavior, potentially adding and parking a core every iteration. This is by design because slower adjustments would either sacrifice tail latency or prevent us from multiplexing cores over short timescales. Indeed, modern CPUs are capable of efficient enough context switching; Process Context Identifiers (PCIDs) allow page tables to be swapped without flushing the TLB. Linux takes about 600 nanoseconds to switch between processes, so it is fast enough to handle the core reallocation rates produced by the IOKernel. In §7.3 we evaluate the impact of different core allocation intervals on tail latency and CPU efficiency.

4.1.2 Which cores for each application

When deciding which core to grant to an application, the IOKernel considers three factors:

1. Hyper-threading efficiency. Intel's HyperThreads enable two hardware threads to run on the same physical core. These threads share processor resources such as the L1 and L2 caches and execution units, but are exposed as two separate logical cores [51]. If hyper-threads from the same application run on the same physical core, they benefit from cache locality; if hyper-threads from different applications share the same physical core, they can contend for cache space and degrade each others' performance. Thus, the IOKernel favors granting hyper-threads on the same physical core to the same application.
2. Cache locality. If an application's state is already present in the L1/L2 cache of a core it is newly

Algorithm 2 Core Selection Algorithm

```
1: function CANBEALLOCATED(core)
2:   if core is idle then return True
3:   app  $\leftarrow$  the app currently using core
4:   if n_idle_cores is 0 and app is bursting then
5:     return True
6:   return False
7:
8: function SELECTCORE(app)
9:   for each active core c of app do
10:    chyper  $\leftarrow$  the hyper-thread pair core of c
11:    if CANBEALLOCATED(chyper) then
12:      return chyper
13:    c_recent  $\leftarrow$  core most recently yielded by app
14:    if CANBEALLOCATED(c_recent) then
15:      return c_recent
16:    if n_idle_cores > 0 then return any idle core
17:    app_bursting  $\leftarrow$  random bursting app
18:    return any core in use by app_bursting
```

granted, it can avoid many time-consuming cache misses. Because hyperthreads share the same cache resources, granting an application a hyper-thread pair of an already-running core will yield good cache locality. In addition, an application may experience cache locality benefits by running on a core that it ran on recently.¹ Thus, the IOKernel tracks current and past core allocations for runtimes.

3. Latency. Preempting a core and waiting for it to become available takes time, and wastes cycles that could be spent doing useful work. Thus, the IOKernel always grants an idle core instead of preempting a busy core, if an idle core exists.

The IOKernel's core selection algorithm (Algorithm 2) considers the three factors described above. A *core* is only eligible for allocation (function CANBEALLOCATED) if it is idle (line 2), or if there are no idle cores and the application using *core* is bursting (using more than its guaranteed number of cores) (line 4). Amongst the eligible cores, the selection algorithm SELECTCORE first tries to allocate the hyper-thread pair of a core the application is currently using (lines 9–12). Next, it tries to allocate the core that this application most recently used, but is no longer using (lines 13–15). Finally, the algorithm chooses any idle core if one exists, or a random core from a bursting application.

¹This benefit is ephemeral; a core with a clock frequency of 2.2 GHz can completely overwrite a 3 MB L2 cache in as little as 60 μ s.

Once the IOKernel has chosen a core to grant to an application, it must also select one of its parked kthreads to wake up and run on that core. For cache locality, it first attempts to pick one that recently ran on that core. If such a kthread is not available, the IOKernel selects the kthread that has been parked the longest, leaving other kthreads parked in case a core they ran on recently becomes available.

The runtime for SELECTCORE(APP) is linear in the number of active cores for APP (it checks whether each active core has an available hyper-thread). The congestion detection algorithm may invoke SELECTCORE up to once per active application in one pass, and the sum of active cores across active applications never exceeds the number of cores in the system. Thus the total cost of invoking the detection algorithm is linear in the total number of cores.

4.2 Dataplane

The IOKernel busy loops, continuously polling the incoming NIC packet queue and the outgoing application packet queues.

Packet steering. Because the IOKernel tracks which cores belong to each runtime, it can deliver incoming packets directly to a core running the appropriate runtime. In Shenango, each runtime is configured with its own IP and MAC address. When a new packet arrives, the IOKernel identifies its runtime by looking up the MAC address in a hash table. The IOKernel then chooses a core within that runtime using an RSS hash [4], and enqueues the packet to that core’s ingress packet queue. Shenango may occasionally reorder packets (e.g., when the number of cores allocated to a runtime changes), but we found that packets in the same flow typically arrive in the same runtime ingress packet queue over short time intervals (§7.3). Our system could be extended to further optimize packet steering through techniques like Intel’s Flow Director [8] or FlexNIC [42].

Polling transmission queues. Polling many egress queues in order to find packets to transmit can incur high CPU overhead, particularly in systems with many queues [68]. Because the IOKernel tracks which kthreads are active, it is able to only poll the outgoing runtime packet queues that correspond to active kthreads. This allows the CPU overhead of polling egress queues to scale with the number of cores in the system.

5 Runtime

Shenango’s runtime is optimized for programmability, providing high-level abstractions like blocking TCP network sockets and lightweight threads. Our design scales

to thousands of uthreads, each capable of performing arbitrary computation interspersed with synchronous I/O operations. By contrast, many previous kernel-bypass network stacks trade functionality for performance, forcing developers to use restrictive, event-driven programming models with APIs that differ significantly from Berkeley Sockets [2, 18, 40, 61].

Similar to a library OS [37, 60], our runtime is linked within each application’s address space. After the runtime is initialized, applications should only interact with the Linux Kernel to allocate memory; other system calls remain available, but we discourage applications from performing any blocking kernel operations, as this could reduce CPU utilization. Instead, the runtime provides kernel-bypass alternatives to these system calls (in contrast to scheduler activations [11], which activates new threads to recover lost concurrency). As an additional benefit, memory and CPU usage, including for packet processing, can be perfectly accounted to each application because the kernel no longer performs these requests on their behalf.

Scheduling. The runtime performs scheduling within an application across the cores that are dynamically allocated to it by the IOKernel. During initialization, the runtime registers its kthreads (enough to handle the maximum provisioned number of cores) with the IOKernel and establishes a shared memory region for network packet queues. Each time the IOKernel assigns a core, it wakes one of the runtime’s kthreads and binds it to that specific core.

Our runtime is structured around per-kthread runqueues and work stealing, similar to Go [6] and in contrast with Arachne’s work sharing model [63]. Despite embracing this more traditional design, we found that it was possible to make our uthread handling extremely efficient. For example, because only the local kthread can append to its runqueue, uthread wakeups can be performed without locking. Inspired by ZygOS, we perform fine-grained work stealing of uthreads to reduce tail latency, which is particularly beneficial for workloads that have service time variability [61].

Our runtime also employs run-to-completion, allowing uthreads to run uninterrupted until they voluntarily yield, in most cases. This policy further reduces tail latency with light-tailed request patterns.² When a uthread yields, any necessary register state is saved on the stack, allowing execution to resume later. When the yield is cooperative, we can save less register state

²Preemption within an application, as in Shinjuku [38], could reduce tail latency for request patterns with high dispersion or a heavy tail; we leave this to future work.

because function call boundaries allow clobbering of some general purpose registers as well as all vector and floating point state [49]. However, any uthread may be preempted if the IOKernel reclaims a core; in this case all register state must be saved.

To find the next uthread to run after a yield, the scheduler first checks the local runqueue; if it is empty and there are no incoming packets or expired timers to process, it engages in work stealing. It first checks the core’s hyper-thread sibling to exploit cache locality. If that fails, the scheduler tries to steal from a random kthread. Finally, the scheduler iterates through all active kthreads. It repeats these steps for a couple of microseconds, and if all attempts fail, the scheduler parks the kthread, yielding its core back to the IOKernel.

Networking. Our runtime is responsible for providing all networking functionality to the application, including UDP and TCP protocol handling. After a uthread yields or whenever the local runqueue is empty, each kthread checks its ingress packet queue for new packets to handle. Unlike previous systems, kthreads can also steal packets from remote ingress packet queues. This contrasts with ZygOS, which can steal application-level work above the TCP socket layer but must maintain flow consistent hashing of packets. Thus this stealing, along with the packet steering adjustments made by the IOKernel, can cause packet reordering over short timescales.

A variety of efficient techniques have been proposed to resequence packets [29, 30, 33]. Where ordering is required, our runtime provides a similar low overhead mechanism to reassemble the packet sequence in the transport layer. This resequencing involves acquiring a per-socket lock, but because packets from the same flow typically arrive at the same core over short time scales, cache locality is preserved and the overhead of acquiring the lock is small.

On the other hand, we found that there were significant advantages to relaxing ordering requirements and violating flow consistent hashing. ZygOS must send and receive packets from a given flow on the same core, so it relies on expensive IPIs to ensure timely processing of pending ingress packets and to ensure egress handling happens on the same core. By contrast, Shenango’s approach enables more fine-grained load balancing of network flow processing, yielding better performance with imbalanced workloads (§7.3).

An earlier version of the runtime attempted to support zero-copy networking. However, we found this approach had serious drawbacks. First, it required API changes, breaking compatibility with Berkeley Sockets. Second, we were surprised to find it had a negative impact on

performance. Upon further investigation, we discovered that our IOKernel’s throughput was sensitive to the amount of resident buffering because DDIO (an Intel technology that pushes packet payloads directly into the LLC) places limits on the maximum number of cache lines that can be occupied by packet data. When that limit is exceeded, packet data is pushed to RAM, greatly increasing access latency. By copying payloads, we can encourage DDIO to reuse the same buffers, thus staying within its cache occupancy threshold. This bears similarity to the “leaky DMA” issue [70].

Because an application could potentially corrupt its runtime network stack, we assume security validation (e.g., bandwidth capping and network virtualization) will be efficiently handled out-of-band, in exactly the same manner as for virtual machine guest kernels [23, 27].

6 Implementation

Shenango’s implementation consists of the IOKernel (§6.1), which runs as a separate, privileged process, and the runtime (§6.2), which users link with their applications. Shenango is implemented in C and includes bindings for C++ and Rust. The IOKernel is implemented in 2,244 LOC and the runtime is implemented in 6,155 LOC. Both components depend on a 4,762 LOC collection of custom library routines. The implementation currently supports 64-bit x86, and adapting it to other platforms would not require many changes. The IOKernel uses Intel Data Plane Development Kit (DPDK) [2], version 18.11, for fast access to NIC queues from user space. Our entire system runs in an unmodified Linux environment.

6.1 IOKernel Implementation

Shenango relies on several Linux kernel mechanisms to pin threads to cores and for communication between the IOKernel and runtimes. The IOKernel passes data via System-V shared memory segments that are mapped into each runtime. The runtime sets up a series of descriptor ring queues (inspired by Barreelfish’s implementation of lightweight RPC [17]), including ingress packet queues, egress packet queues, and separate egress command queues (to prevent head-of-line blocking). It also designates a portion of the mapped-memory for outgoing network buffers. We currently place all ingress packet buffers in a single, read-only region shared with all runtimes. In the future, we plan to maintain separate buffers, using NIC HW filtering to segregate packets.

To assign a runtime kthread to a specific core, the IOKernel uses `sched_setaffinity`. The IOKernel maintains a shared `eventfd` file descriptor with each kthread. When a kthread cannot find more uthreads to

run, it notifies the IOKernel via a command queue message that it is parking and then parks itself by performing a blocking read on its `eventfd`. To unpark a kthread, the IOKernel simply writes a value into the `eventfd`. To preempt runtime kthreads when it needs to reassign a core, the IOKernel directs a `SIGUSR1` signal to the intended kthread using the `tgkill` system call. This prompts the kthread to park itself. A malicious kthread could refuse to park after a signal. While we have yet to implement mitigation strategies, the IOKernel could wait a few microseconds and then migrate an offending kthread to a shared core that is multiplexed by the Linux scheduler, so that other runtimes are not impacted.

6.2 Runtime Implementation

Our runtime includes support for lightweight threads, mutexes, condition variables, read-copy-update (RCU), high resolution timers, and synchronous TCP and UDP sockets. Like the IOKernel, the runtime makes use of a limited set of existing Linux primitives; it allocates memory with `mmap`, creates kthreads through calls to `pthread_create()`, and interacts with the IOKernel through shared memory, `eventfd` file descriptors, and signals. We implemented TCP from scratch according to the RFC [36]. Our TCP stack is interoperable with those of Linux and ZygOS and includes flow control and fast retransmit but omits congestion control.

To improve memory allocation performance, the runtime makes use of per-kthread caches [21], particularly when allocating thread stacks and network packet buffers. The runtime provides an RCU subsystem to support efficient access to read-mostly data structures [52]. The runtime detects a quiescent period after each kthread has rescheduled, allowing it to free any stale RCU objects. Internally, RCU is used for the ARP table and for the TCP and UDP socket tables.

Shenango provides bindings for both C++ and Rust with idiomatic interfaces (e.g., like `std::thread`) and support for lambdas and closures respectively. Most of the bindings are implemented as a thin wrapper around the underlying C library. However, our uthread support takes advantage of a unique optimization. We extended Shenango’s spawn function to reserve space at the base of each uthread’s stack for the trampoline data (captures, space for a return value, etc.), avoiding extra allocations.

Preemption. Upon receipt of a `SIGUSR1` sent by the IOKernel, the Linux kernel saves the CPU state into a trapframe on the thread stack and invokes the signal handler installed by the runtime. The signal handler immediately transfers to the scheduler context and parks, placing the preempted uthread back into the runqueue. The running uthread could eventually be stolen by

another kthread or resume on the same kthread if it is re-granted a core.

During certain critical sections of runtime execution, preemption signals are deferred by incrementing a thread-local counter. These sections include the entire scheduler context, RCU and spinlock critical sections, and code regions that access per-kthread state. Supporting preemption of active uthreads poses some challenges. Pointers to thread-local storage (TLS) may become stale if a thread context starts executing on a different kthread. Unfortunately, gcc does not provide a way to disable caching these addresses. To our knowledge, Microsoft’s C++ compiler is the only compiler to support this. As a workaround, we use our own TLS mechanisms for per-kthread data structures that are accessed outside of the scheduler context, and we currently require that applications disable preemption during accesses to thread-local variables (including glibc’s `malloc` and `free`). We are considering extending the runtime to support TLS for each uthread, alleviating this burden on developers. However, the TLS data section would have to be kept small to prevent higher initialization overheads when spawning uthreads.

7 Evaluation

In evaluating Shenango, we aim to answer the following questions:

1. How do latency and CPU efficiency compare for Shenango and other systems across different workloads and service-time distributions? (§7.1)
2. How well can Shenango respond to sudden bursts in load? (§7.2)
3. What is the contribution of the individual mechanisms in Shenango to its observed performance? (§7.3)

Experimental setup. We used one dual-socket server with 12-core Intel Xeon E5-2650v4 CPUs running at 2.20 GHz, 64 GB of RAM, and a 10 Gbits/s Intel 82599ES NIC. We enabled hyper-threads and evaluated only the first socket, steering NIC interrupts, memory allocations, and threads. To reduce jitter, we disabled TurboBoost, C-states, and CPU frequency scaling. We generated load from six additional quad-core machines connected to the server through a Mellanox SX1024 switch and Mellanox ConnectX-3 Pro NICs. We used Ubuntu 18.04 with kernel version 4.15.0. We disabled kernel mitigations for Meltdown for consistency with prior results; future CPUs will support these mitigations in hardware [9].

Systems evaluated. We compare Shenango to Arachne, ZygOS, and Linux. Arachne is a state-of-the-art,

System	Kernel-bypass Net.	Lightweight Threading	Balancing Interval
Linux	✗	✗	4 ms
Arachne [63]	✗	✓	50 ms
ZygOS [61]	✓	✗	N/A
Shenango	✓	✓	5 μ s

Table 1: Features of the systems we evaluated.

user-level threading system [63]. It achieves better tail latency and CPU efficiency than Linux by introducing a user-level core allocator that adjusts the cores assigned to each application over millisecond timescales. However, Arachne provides no network stack integration and applications typically rely on Linux kernel system calls for network I/O. ZygOS is a state-of-the-art, kernel-bypass network stack [61] that builds upon IX [18] to achieve better tail latency, adding fine-grained load balancing of application-level work between cores. However, it does not support threads, instead requiring developers to adopt a restrictive, event-driven API, and it can only run on a fixed set of statically provisioned cores. Finally, *Linux* is the most widely deployed of these systems in practice, but its performance, as previously studied, is limited by kernel overheads [18, 35]. Table 1 summarizes the salient differences between Shenango and these three systems.

For Arachne, we used the latest available source code [1] as of mid January 2019. We found that the default load factor of 1.5, a tuning parameter for the core allocator, yielded the best results in our experiments. For ZygOS, we similarly used the latest available source code [7]. We found that ZygOS was unstable with recent kernels, so we instead used Ubuntu 16.04 with kernel version 4.11.0.

Finally, for Linux, we used prior work [43, 45] and invested substantial effort in finding the best possible configuration. In many cases, the performance of Linux was unstable, making it challenging to measure. For example, we noticed signs of performance hysteresis, where measurement runs converged to different values despite identical configuration [77]. Increasing the number of active flows resolved this issue by allowing for more uniform RSS hashing. We ran batch tasks using SCHED_IDLE (a Linux scheduling policy intended for very low priority background jobs), though we found this did not improve performance much over using the lowest normal scheduler priority (niceness 19).

Applications. We evaluate *memcached* (v1.5.6), a popular key-value store that is well supported by all four systems.³ We also wrote several new Shenango applications in Rust to measure different load patterns, taking advan-

³We don’t run LRU cache maintenance/eviction and slab rebalancing for Arachne because Arachne’s memcached implementation does not support them.

tage of language features like closures and move semantics. For example, we implemented a *spin-server* that emulates a compute-bound application by using the CPU for a specified duration before responding to each request. In addition, we implemented *loadgen*, a realistic load generator that can generate precisely-timed request patterns for our spin-server as well as for memcached. Combined, these two applications required 1,366 LOC. For comparing to other systems, we used variants of the ZygOS and Linux spin-servers in the ZygOS repository [7] and implemented our own spin-server for Arachne.

To support batch processing applications, we implemented a pthread shim layer for Shenango that enables it to run the entire PARSEC suite [19] without modifications. In our experiments, we use PARSEC’s *swaptions* benchmark for batch processing. It computes prices of a portfolio using Monte Carlo simulations; each thread computes the price of a swaption with no synchronization or data dependencies between threads. Finally, we ported the *gdnsd* (v2.4.0) [3] DNS server, to demonstrate Shenango’s UDP support. The source code for all of these applications is available on GitHub [5].

We used open-loop Poisson processes to model packet arrivals [69, 77]. Our experiments measure throughput and the 99.9th percentile tail response latency. All experiments use our Rust loadgen application to generate load over TCP, unless stated otherwise.

7.1 CPU Efficiency and Latency

In this section we evaluate the CPU efficiency and latency of memcached, the spin-server, and gdnsd. We use 6 client servers to generate load, enough to minimize client-side queuing delays. Each client uses 200 persistent connections (1200 total). We ramp up load gradually and measure each offered load over several seconds, so that bursts come only from the Poisson arrival process.

To ensure a fair comparison with ZygOS, which cannot support more than 16 hyperthreads with our NIC, we confine all systems to use 16 hyperthreads (8 cores) in total. Shenango must dedicate one core (2 hyperthreads) to running the IOKernel, so two fewer hyperthreads are available for applications; Arachne must dedicate one hyperthread to the core arbiter. For all but ZygOS, we also run *swaptions*, filling any unused cycles with lower-priority batch processing work. For ZygOS, we reserve all 16 hyperthreads for the latency-sensitive application, as required to achieve peak throughput.

Memcached. We use the USR workload from [13]: requests follow a Poisson arrival process and consist of 99.8% GET requests and 0.2% SET requests. For Shenango, we limit memcached to using at most 12 hyperthreads, because this yields the best performance

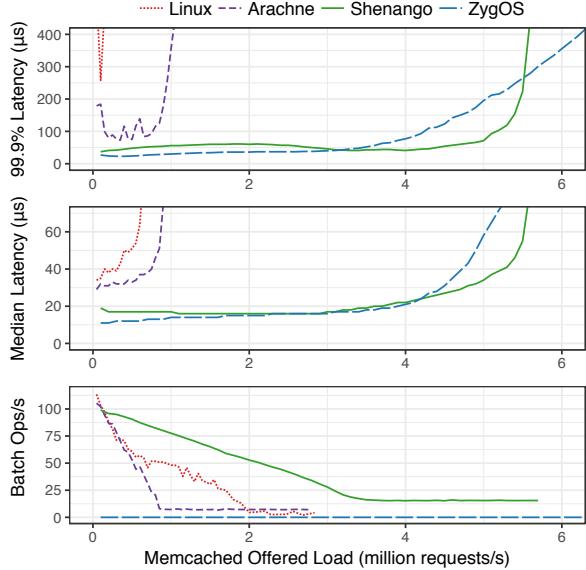


Figure 3: Shenango maintains consistently low median and 99.9% latency, comparable to those of ZygoOS, while allowing unused cycles to be used by a batch processing application.

for memcached. Figure 3 shows how 99.9th percentile latency for memcached, median latency for memcached, and throughput for the batch application (y-axes) change as we increase the load offered to memcached (x-axis). We only show data points for which achieved load is within 0.1% of offered load.

Shenango can handle over five million requests per second while maintaining a median response time of 37 μ s and 99.9th percentile response time of 93 μ s. Despite busy polling on all 16 hyperthreads, ZygoOS maintains similar response times only up to four million requests per second. ZygoOS does scale to support higher throughput than Shenango, though at a high latency penalty. Shenango achieves lower throughput because at the very low service times of memcached ($< 2 \mu$ s), the IOKernel becomes a bottleneck. We discuss options for scaling out the IOKernel further in Section 8. For all other systems, memcached is bottlenecked by CPU.

Similar to previous studies [18, 61], when there is no batch work running, we achieve about 800,000 requests per second with memcached in Linux before 99th percentile latency spikes (not shown). However, we found that Linux’s latency degrades significantly due to the presence of batch work, especially at the 99.9th percentile. For example, at 0.4 million requests per second, the 99.9th percentile latency without batch work is only 83 μ s compared to over 2 ms with batch work. Arachne improves upon Linux, maintaining 99.9th percentile latency below 200 μ s with batch work. However, even without batch work, both systems suffer

significantly from their use of the Linux network stack; kernel bypass enables both Shenango and ZygoOS to achieve much lower median latency and much higher peak throughput for memcached.

Shenango outperforms the other systems in terms of throughput for the batch application at all but the lowest loads. At very low load, Linux achieves the most batch throughput because it does not reserve any hyperthreads for the IOKernel or the core arbiter. As the load offered to memcached increases, Shenango’s batch throughput decreases linearly and then plateaus once the batch task is restricted to only the two remaining hyperthreads. Memcached throughput still increases beyond this point, however, because Shenango becomes more efficient near peak load, spending fewer cycles on core reallocations and work stealing.

In aggregate, our memcached results illustrate that Shenango has key advantages over previous systems. Shenango can achieve tail latencies similar to ZygoOS while at the same time sparing significantly more cycles for batch work than all three systems, despite reserving two hyperthreads for the IOKernel.

Spin-server. To evaluate Shenango’s ability to handle service-time variability in the presence of a batch processing application, we ran our spin-server with three service-time distributions, each with a mean of 10 μ s: *constant*, where all requests take equal time; *exponential*; and *bimodal*, where 90% of requests take 5 μ s and 10% take 55 μ s.

Figure 4 shows the resulting 99.9th percentile latency and batch throughput as we vary the load on the spin-server. All systems fall short of the theoretical maximum throughput achievable by an M/G/16/FCFS simulation, due to overheads such as packet processing. Compared to ZygoOS, Shenango achieves slightly higher throughput for the spin server, even though two out of Shenango’s 16 hyperthreads are dedicated to running the IOKernel. Shenango’s tail latency is similar to that of ZygoOS, but because ZygoOS must provision all cores for the spin server in order to achieve peak throughput, it does not achieve any batch throughput.

At the 99.9th percentile, Linux’s tail latency varies drastically, at times reaching several milliseconds, even at low load. Arachne achieves higher throughput than Linux for both applications, demonstrating the benefit of granting applications exclusive use of their cores. Surprisingly, we observe that Arachne’s tail latency is slightly higher at the lowest loads than at moderate load. We suspect that this is due to miscalculation of core requirements. Granting too few cores for up to 50 ms at a time can result in high latencies for many requests, par-

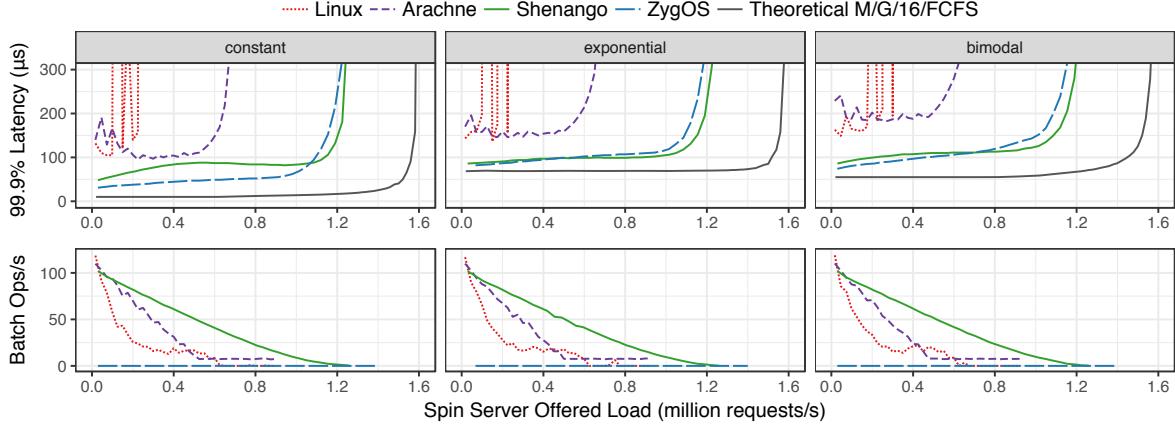


Figure 4: Shenango maintains low 99.9% latency across a variety of service time distributions (mean of $10\ \mu s$) and linearly trades off batch processing throughput for latency-sensitive throughput. Linux and Arachne suffer from poor latency and low throughput, while ZygoOS must dedicate all cores to the latency-sensitive spin server in order to achieve peak throughput, resulting in no batch throughput.

ticularly at low loads when there are few cores allocated to absorb the extra load. We also found that decreasing Arachne’s core allocation interval to 1 ms or $100\ \mu s$ yielded similar or worse performance for both the spin server and batch application, suggesting that Arachne’s load estimation mechanisms are not well-tuned for small core allocation intervals. In contrast, in this experiment Shenango reallocates cores up to 60,000 times per second, enabling it to adjust quickly to bursts in load and maintain much lower tail latency, while granting unused cycles to the batch application.

DNS. We evaluate UDP performance by running *gdnsd* and *swaptions* simultaneously for Linux and Shenango; we did not port *gdnsd* to ZygoOS or Arachne. Linux *gdnsd* can drive up to 900,000 requests per second with $41\ \mu s$ median latency and sub-millisecond 99.9th percentile latency before starting to drop packets. Shenango *gdnsd* is capable of scaling to 5.7 million requests per second (a $6.33\times$ improvement) with $36\ \mu s$ median latency and $73\ \mu s$ 99.9th percentile latency. We omit a graph due to space constraints.

7.2 Resilience to Bursts in Load

In this experiment, we generate TCP requests with $1\ \mu s$ of fake work, and measure the impact of sudden load increases on tail latency. We offer a baseline load of 100,000 requests per second for one second, followed by an instantaneous increase to an elevated rate. After an additional second at the new rate, the load drops back to the baseline rate. Any unused cores are allocated to batch processing, keeping overall CPU utilization at 100%.

Figure 5 shows the 99.9th percentile tail latency and throughput for Arachne and Shenango (computed over 10 ms windows). We exclude Linux because, under these conditions, it has milliseconds of tail latency even at the

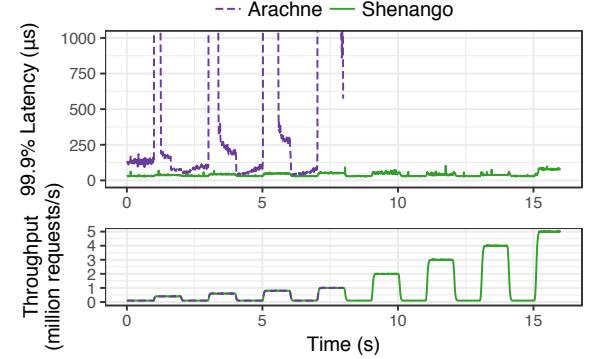


Figure 5: Under sudden changes in load, low tail latency is only possible with a short core allocation interval.

lowest offered load, and we exclude ZygoOS because it cannot adjust core allocations. By contrast, Arachne can eventually meet the loads offered in the experiment, up to 1 million requests per second. However, because of its slow core allocation speed, it can take over 500 milliseconds to add enough cores to adapt after a load transition, causing it to accumulate a backlog of pending requests. As a result, Arachne experiences milliseconds of tail latency, even after relatively modest shifts in load. By contrast, Shenango reacts so quickly that it incurs almost no additional tail latency, even when handling an extreme load shift from 100,000 to 5 million requests per second.

7.3 Microbenchmarks

We now evaluate the individual components of Shenango with microbenchmarks.

Thread library. Shenango depends on efficient thread scheduling to support high-level programming abstractions at low cost. Here we compare Shenango’s latency for common threading operations to Linux pthreads and to Go and Arachne’s optimized user space threading implementations (Table 2). These benchmarks are

	pthreads	Go	Arachne	Shenango
Uncontented Mutex	30	24	55	37
Yield Ping Pong	593	109	79	52
Condvar Ping Pong	1,900	281	203	100
Spawn-Join	12,996	462	595	148

Table 2: Nanoseconds to perform common threading operations (fastest highlighted in green). Shenango performs best for all but mutexes.

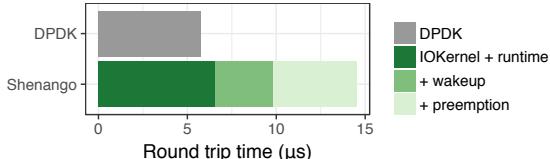


Figure 6: Traversing the network stack, waking a kthread, and preempting a kthread each add only a few μs of overhead to a packet’s RTT in Shenango.

written in C++ and configure each system to use a single core. Shenango outperforms all three systems in all but one benchmark because of its preallocated stacks, atomic-free wakeups, and care to avoid saving registers that can safely be clobbered. In Go, mutexes are slightly faster because its compiler can inline them.

Network stack and core allocation overheads. We evaluate the baseline latency of our network stack and the overhead of waking and preempting cores with a simple C/C++ UDP echo benchmark. The client is a minimal DPDK client. On the server side, we compare a minimal DPDK server to three variants of Shenango which are configured so that: (1) the runtime core busy-spins, (2) the runtime core does not busy-spin and must be reallocated on every packet arrival, and (3) a batch application fills all cores and must be preempted on every packet arrival. Figure 6 shows that the runtime and the IOKernel add little latency over using raw packets in DPDK. Waking sleeping kthreads and preempting running kthreads, however, do incur some overhead, due to the use of Linux system calls (§6.1). While we were pleasantly surprised to find that the overhead of these Linux mechanisms is acceptable, we believe they can be reduced in the future.

Packet load balancing. Shenango allows packet handling to be performed on any core; here we evaluate this approach. To challenge our system’s load balancing, we replicate the central graph of Figure 4 but vary the number of client connections used. With only 24 connections, RSS distributes flows unevenly across cores. Figure 7 shows that by allowing cores to steal packet processing work, including TCP protocol handling, Shenango is able to maintain good performance even with an unbalanced workload. In contrast, ZygOS’s latency degrades significantly because it only allows work stealing at the application layer and performs all packet processing on the core

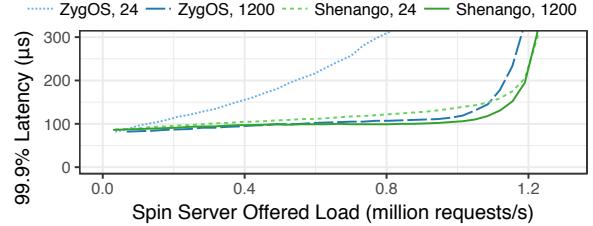


Figure 7: By work stealing packet handling, Shenango can load balance more effectively than ZygOS and maintain almost as good performance with 24 client connections as with 1200.

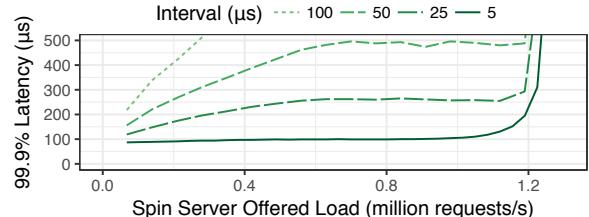


Figure 8: Shenango’s tail latency degrades with larger core allocation intervals.

on which a packet arrives. At the same time, the costs of Shenango’s fine-grained work stealing remain quite low. With 1200 connections, less than 0.07% of packets arrive at Shenango’s ingress network stack out of order. With 24 connections, this percentage increases at moderate loads but remains below 3%. The result is that the application spends less than 0.5% of its cycles resequencing packets.

Core allocation interval. A major strength of Shenango is its ability to make μs -scale adjustments to the allocation of cores to runtimes. To illustrate the impact of core allocation speed on Shenango’s performance, we replicate the central graph of Figure 4 but vary the interval between core allocations. Figure 8 demonstrates that a short interval between adjustments is required to maintain low tail latency. Such frequent reallocations do impact CPU efficiency; the batch application performs up to 6% fewer operations per second (of the max possible) with a 5 μs interval than with a 25, 50, or 100 μs interval. However, we do not think these efficiency savings are worth the tail latency increase of at least 150 μs . We did not use a smaller interval because, at faster rates, latency is only marginally improved but more cycles are wasted parking threads.

8 Discussion

We found, in practice, that the IOKernel can support packet rates of up to 6.5 million incoming and outgoing packets per second. This is sufficient to saturate a 10 Gbits/s NIC with 114 byte TCP packets or a 40 Gbits/s NIC with typical Ethernet MTU-sized packets. We note our evaluation of Shenango does not consider multisocket, NUMA machines. One option may be to run multiple instances of the IOKernel, one per socket.

Each IOKernel instance could exchange messages with the others, perhaps enabling coarse-grained load balancing between sockets. Such a design would enable our IOKernel to scale out further. We observed that the majority of IOKernel overhead was in forwarding packets rather than in orchestrating core allocations. Therefore, we also plan to explore hardware offloads, such as new NIC designs that can efficiently expose information about queuing buildups to the IOKernel.

9 Related Work

Two-level scheduling: In two-level scheduling (first proposed in [71]), a first-level spatial scheduler allocates cores to applications and a second-level scheduler handles threads on top of the allocated cores. Scheduler activations [11] provide a kernel mechanism to enable two-level scheduling; this work inspired recent systems such as Tessellation [22, 47], Akaros [65], and Calisto [32]. All of these systems decouple core allocation from thread scheduling. Shenango introduces a new approach to two-level scheduling by combining the first scheduler level directly with the NIC.

User-level threading: Several systems have multiplexed user space threads across one or more cores. Examples include Capriccio [73], Lithe [58], Intel’s TBB [64], μ Threads [14], Arachne [63], and the Go runtime [6]. Shenango’s runtime borrows many techniques from these prior works, including work stealing [20]. However, to our knowledge, no prior system is designed to tolerate core allocations and revocations at the granularity of μ s.

Dynamic resource allocation: When deciding how to allocate threads or cores across applications, previous systems have employed resource controllers that monitor performance metrics, utilization, or internal queue lengths (e.g., Tessellation [22], PerfIso [34], Arachne [63], SEDA [74], and IX [62]). However, because these metrics are gathered over several milliseconds or even seconds, they are too coarse-grained to manage tail latency. Furthermore, using core utilization to estimate core requirements is only possible in systems in which cores remain allocated to applications even while they are idle or busy-spinning [34, 63]; this approach wastes CPU cycles.

Several scheduling optimizations have been proposed to reduce tail latency. For example, Heracles [48] adjusts CPU isolation mechanisms (e.g., cache partitioning), Elfen Scheduling [75] strategically disables hyper-threading lanes, and Tail Control [44] improves upon work stealing. We are interested in exploring ways of integrating these techniques with Shenango in the future.

Kernel-bypass networking: Many systems bypass the kernel to achieve low-latency networking by using RDMA, SR-IOV, or libraries such as DPDK [2] or netmap [66]. Examples include MICA [46], IX [18], Arrakis [59], mTCP [35], Sandstorm [50], FaRM [25], HERD [39], RAMCloud [57], SoftNIC [31], ZygOS [61], Shinjuku [38], and eRPC [40]. IX and eRPC process packets in batches and may provide higher throughput than Shenango for workloads with short, uniform service times and many connections to balance load across cores. ZygOS is most similar to Shenango; it builds on IX by adding work stealing to improve load balancing within an application. However, none of these systems can dynamically reallocate cores across applications at a fine granularity. Instead, they statically partition cores across applications, or else use an external control plane to reconfigure core assignments over large timescales.

10 Conclusion

This paper presented Shenango, a system that can simultaneously maintain CPU efficiency, low tail latency, and high network throughput on machines handling multiple latency-sensitive and batch processing applications. Shenango achieves these benefits through its IOKernel, a dedicated core that integrates with networking to drive fine-grained core allocation adjustments between applications. The IOKernel makes use of a congestion detection algorithm that can react to application overload in μ s timescales by tracking queuing backlog information for both packets and application threads. This design allows Shenango to significantly improve upon previous kernel bypass network stacks by recovering cycles wasted on busy spinning because of the provisioning gap between minimum and peak load. Finally, our per-application runtime makes these benefits more accessible to developers by providing high-level programming abstractions (e.g., lightweight threads and synchronous network sockets) at low overhead.

11 Acknowledgments

We thank our shepherd KyoungSoo Park, the anonymous reviewers, John Ousterhout, Tom Anderson, Frans Kaashoek, Nickolai Zeldovich, and other members of PDS for their useful feedback. We thank Henry Qin for helping us evaluate Arachne. Amy Ousterhout was supported by an NSF Fellowship and a Hertz Foundation Fellowship. This work was funded in part by a Google Faculty Award and by NSF Grants CNS-1407470, CNS-1526791, and CNS-1563826.

References

- [1] Arachne: Towards Core-Aware Scheduling. <https://github.com/PlatformLab/Arachne>.
- [2] DPDK Boosts Packet Processing, Performance, and Throughput. <http://www.intel.com/go/dpdk>.
- [3] gdnsd – an authoritative-only dns server. <http://gdnsd.org/>.
- [4] Introduction to Receive Side Scaling. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>.
- [5] Shenango. <https://github.com/shenango>.
- [6] The Go Programming Language. <https://golang.org/>.
- [7] ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. <https://github.com/ix-project/zygos>.
- [8] Intel 82599 10 GbE Controller Datasheet. <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf>, 2016.
- [9] Intel Analysis of Speculative Execution Side Channels. Technical report, January 2018.
- [10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.
- [11] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *TOCS*, 1992.
- [12] D. Ardelean, A. Diwan, and C. Erdman. Performance Analysis of Cloud Applications. In *NSDI*, 2018.
- [13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*, 2012.
- [14] S. Barghi. uThreads: Concurrent User Threads in C++(and C). <https://github.com/samanbarghi/uThreads>.
- [15] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 2017.
- [16] L. A. Barroso, J. Clidaras, and U. Hözlé. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2013.
- [17] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.
- [18] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *TOCS*, 2017.
- [19] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [20] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *JACM*, 1999.
- [21] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX ATC*, 2001.
- [22] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, et al. Tessellation: Refactoring the OS around Explicit Resource Containers with Continuous Adaptation. In *DAC*, 2013.
- [23] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermenio, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooster, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*, 2018.
- [24] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.
- [25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

- [26] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.
- [27] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.
- [28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.
- [29] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters. In *EuroSys*, 2016.
- [30] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM*, 2017.
- [31] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, Univ. California, Berkeley, 2015.
- [32] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. In *EuroSys*, 2014.
- [33] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.
- [34] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. R. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *USENIX ATC*, 2018.
- [35] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.
- [36] P. Jon. Transmission Control Protocol: DARPA Internet Program Protocol Specification. Technical report, RFC-793, DARPA, 1981.
- [37] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *SOSP*, 1997.
- [38] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for μ second-scale Tail Latency. In *NSDI*, 2019.
- [39] A. Kalia, M. Kaminsky, and D. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014.
- [40] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.
- [41] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC*, 2012.
- [42] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *ASPLOS*, 2016.
- [43] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *EuroSys*, 2014.
- [44] J. Li, K. Agrawal, S. Elnikety, Y. He, I. A. Lee, C. Lu, and K. S. McKinley. Work Stealing for Interactive Services to Meet Target Latency. In *PPoPP*, 2016.
- [45] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SoCC*, 2014.
- [46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*, 2014.
- [47] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar*, 2009.
- [48] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

- [49] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement*, 2018.
- [50] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *SIGCOMM*, 2014.
- [51] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 2002.
- [52] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU Usage in the Linux Kernel: One Decade Later. Technical report, 2013.
- [53] D. Meisner, C. M. Sadler, L. A. Barroso, W. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *ISCA*, 2011.
- [54] Mellanox Technologies. HP and Mellanox Benchmarking Report for Ultra Low Latency 10 and 40Gb/s Ethernet Interconnect. http://www.mellanox.com/related-docs/whitepapers/HP_Mellanox_FSI%20Benchmarking%20Report%20for%202010%20%26%2040GbE.pdf, 2012.
- [55] Mellanox Technologies. RoCE vs. iWARP Competitive Analysis. http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf, 2017.
- [56] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.
- [57] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *TOCS*, 2015.
- [58] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. *PLDI*, 2010.
- [59] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. *OSDI*, 2014.
- [60] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olin-sky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *ASPLOS*, 2011.
- [61] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *SOSP*, 2017.
- [62] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *SoCC*, 2015.
- [63] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-Aware Thread Management. In *OSDI*, 2018.
- [64] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 2007.
- [65] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. In *SoCC*, 2011.
- [66] L. Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX ATC*, 2012.
- [67] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *HotOS*, 2011.
- [68] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*, 2017.
- [69] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *NSDI*, 2006.
- [70] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraiki, S. Ratnasamy, and S. Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *NSDI*, 2018.
- [71] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *SOSP*, 1989.
- [72] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.
- [73] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *SOSP*, 2003.

- [74] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.
- [75] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *USENIX ATC*, 2016.
- [76] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI²: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.
- [77] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *ISCA*, 2016.

End-to-end I/O Monitoring on a Leading Supercomputer

Bin Yang^{1,3}, Xu Ji^{2,3}, Xiaosong Ma⁴, Xiyang Wang³, Tianyu Zhang^{1,3}, Xiupeng Zhu^{1,3},
Nosayba El-Sayed^{*5}, Haidong Lan¹, Yibo Yang¹, Jidong Zhai², Weiguo Liu^{1,3}, and Wei Xue^{†2,3}

¹Shandong University, ²Tsinghua University, ³National Supercomputer Center in Wuxi, ⁴Qatar
Computing Research institute, HBKU, ⁵Emory University

Abstract

This paper presents an effort to overcome the complexities of production system I/O performance monitoring. We design Beacon, an end-to-end I/O resource monitoring and diagnosis system, for the 40960-node Sunway TaihuLight supercomputer, current ranked world No.3. Beacon simultaneously collects and correlates I/O tracing/profiling data from all the compute nodes, forwarding nodes, storage nodes and metadata servers. With mechanisms such as aggressive online+offline trace compression and distributed caching/storage, it delivers scalable, low-overhead, and sustainable I/O diagnosis under production use. Higher-level per-application I/O performance behaviors are reconstructed from system-level monitoring data to reveal correlations between system performance bottlenecks, utilization symptoms, and application behaviors. Beacon further provides query, statistics, and visualization utilities to users and administrators, allowing comprehensive and in-depth analysis without requiring any code/script modification.

With its deployment on TaihuLight for around 18 months, we demonstrate Beacon’s effectiveness with real-world use cases for I/O performance issue identification and diagnosis. It has successfully helped center administrators identify obscure design or configuration flaws, system anomaly occurrences, I/O performance interference, and resource under- or over-provisioning problems. Several of the exposed problems have already been fixed, with others being currently addressed. In addition, we demonstrate Beacon’s generality by its recent extension to monitor interconnection networks, another contention point on supercomputers. Both Beacon codes and part of collected monitoring data are released.¹

1 Introduction

Modern supercomputers are networked systems with increasingly deep storage hierarchy, serving applications with growing scale and complexity. The long I/O path from storage media to application, combined with complex software

stacks and hardware configurations, makes I/O optimizations increasingly challenging, both for application developers and supercomputer administrators. In addition, since I/O utilizes heavily shared system components (unlike computation or memory accesses), it usually suffers substantial inter-workload interference, causing high performance variance [37, 44, 47, 55, 60, 71].

Online tools that can capture/analyze I/O activities and guide optimization are highly needed. They also need to provide I/O usage information and performance records to guide future systems’ design, configuration, and deployment. To this end, several profiling/tracing tools and frameworks have been developed, including application-side (e.g., Darshan [31] and ScalableIOTrace [77]), backend-side (e.g., LustreDU [29], IOSI [50] and LIOProf [85]), and multi-layer tools (e.g., Modular Darshan [70] and GUIDE [91]).

These proposed tools, however, suffer one or more of the following limitations. Application-oriented tools often require developers to instrument their source code or link extra libraries. They also do not offer intuitive ways to analyze inter-application I/O performance behaviors such as interference issues. Backend-oriented tools can collect system-level performance data and monitor cross-application interactions, but have difficulty in identifying performance issues for specific applications and in finding their root causes. Finally, problematic applications issuing *inefficient I/O requests* escape the radar of backend-side analytical methods [50, 52] relying on high-bandwidth applications.

This paper reports our design, implementation, and deployment of a light-weight, end-to-end I/O resource monitoring and diagnosis system, Beacon, for TaihuLight, currently the world’s No.3 supercomputer [75]. It works with TaihuLight’s 40,960 compute nodes (over ten-million cores in total), 288 forwarding nodes, 288 storage nodes, and 2 metadata nodes. Beacon integrates frontend tracing and backend profiling into a seamless framework, enabling tasks such as automatic per-application I/O behavior profiling, I/O bottleneck/interference analysis, and system anomaly detection.

To our best knowledge, this is the first system-level, multi-layer monitoring and real-time diagnosis framework deployed on ultra scale supercomputers. Beacon collects per-

^{*}Most work conducted at Qatar Computing Research Institute.

[†]Wei Xue is the corresponding author. Email: xuewei@tsinghua.edu.cn

¹Github link: <https://github.com/Beaconsys/Beacon>

formance data simultaneously from different types of nodes (including compute, I/O forwarding, storage, and metadata nodes) and analyzes them collaboratively, *without requiring any involvement of application developers*. Its elaborated collection scheme and aggressive compression minimize the system cost: only 85 *part-time* servers to monitor the entire 40960-node system, with < 1% performance overhead on user applications.

We have deployed Beacon for production use since April, 2017. It has helped the TaihuLight system administration and I/O performance team to identify several performance degradation problems. With its rich I/O performance data collection and real-time system monitoring, Beacon successfully exposes the mismatch between application I/O patterns and widely-adopted underlying storage design/configurations. To help application developers and users, it enables detailed per-application I/O behavior study, with novel inter-application interference identification and analysis. Beacon also performs automatic anomaly detection. Finally, we have recently started to expand Beacon beyond I/O, to network switch monitoring.

Based on our design and deployment experience, we argue that having such end-to-end detailed I/O monitoring framework is a highly rewarding practice. Beacon’s all-system-level monitoring decouples it from language, library, or compiler constraints, enabling monitoring data collection and analysis for all applications and users. Much of its infrastructure reuses existing server/network/storage resources, and it has proven to bring with it negligible overhead. In exchange, users and administrators harvest deep insights into the complex I/O system components’ operation and interaction, and save both human resources and machine core-hours wasted on unnecessarily slow/jittery I/O, or system anomalies.

2 Background: TaihuLight Network Storage

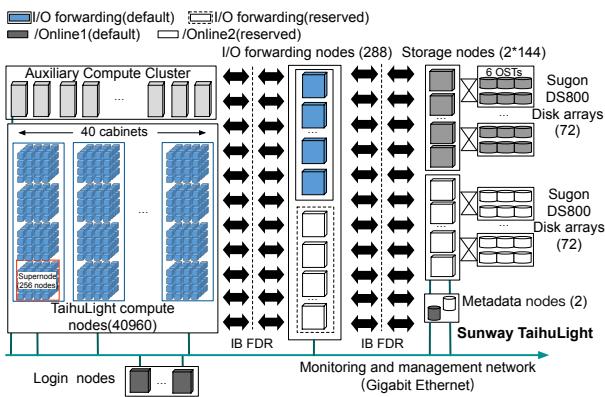


Figure 1: TaihuLight and its Icefish storage system architecture overview. Beacon uses the separate monitoring and management Ethernet network shown at the bottom.

We first introduce the TaihuLight supercomputer (and its Icefish I/O subsystem), where we performed our implementation and deployment. Though the rest of our discussion

will be based on this specific platform, many aspects of Beacon’s design and operation can be applied to other large-scale supercomputers or clusters.

TaihuLight is currently the world No.3 supercomputer, a many-core accelerated 125-petaflop system [36]. Figure 1 illustrates its architecture, highlighting the Icefish storage subsystem. The 40,960 260-core compute nodes are organized in 40 cabinets, each containing 4 supernodes. Through dual-rail FDR InfiniBand, all the 256 compute nodes in one supernode are fully connected and then connected to Icefish via a Fat-tree network. In addition, Icefish serves an Auxiliary Compute Cluster (ACC) with Intel Xeon processors, mainly used for data pre- and post-processing.

The Icefish backend employs the Lustre parallel file system [26], with an aggregate capacity of 10 PB on top of 288 storage nodes and 144 Sugon DS800 disk enclosures. An enclosure contains 60 1.2 TB SAS HDD drives, composing 6 OSTs, each an 8+2 RAID6 array. The controller within each enclosure connects to two storage nodes, via 2 fiber channels for path redundancy. Therefore every storage node manages 3 OSTs, while the two adjacent storage nodes sharing a controller form a failover pair.

Between the compute nodes and the Lustre backend is a layer of 288 *I/O forwarding nodes*. Each plays a dual role, both as a LWFS (Lightweight File System) based on Gluster [6] server to the compute nodes and client to the Lustre backend. This I/O forwarding practice is adopted by multiple other platforms that operate at such scale [15, 28, 54, 78, 90].

A forwarding node provides a bandwidth of 2.5 GB/s, aggregating to over 720 GB/s for the entire forwarding system. Each backend controller provides about 1.8 GB/s, amounting to a file system bandwidth of around 260 GB/s. Overall Icefish delivers 240 GB/s and 220 GB/s aggregate bandwidths for reads and writes, respectively.

TaihuLight debuted on the Top500 list in June 2016. At the time of this study, Icefish was equally partitioned into two namespaces: Online1 (for everyday workloads) and Online2 (reserved for ultra-scale jobs that occupy the majority of the compute nodes), with disjoint sets of forwarding nodes. A batch job can only use either namespace. I/O requests from a compute node are served by a specified forwarding node using a static mapping strategy for easy maintenance (48 fixed forwarding nodes for ACC and 80 fixed forwarding nodes for Sunway compute nodes).

Therefore the two namespaces, along with statically partitioned backend resources, are currently utilized separately by routine jobs and “VIP” jobs. One motivation for deploying an end-to-end monitoring system is to analyze the I/O behavior of the entire supercomputer’s workloads and to design more flexible I/O resource allocation/scheduling mechanisms. For example, motivated by the findings of our monitoring system, a dynamic forwarding allocation system [43] for better forwarding resource utilization has been developed and test deployed.

3 Beacon Design and Implementation

3.1 Beacon Architecture Overview

Figure 2 outlines the working of Beacon on top of the Icefish I/O architecture, designed to operate at different levels of distribution for both scalability and ease of management.

Beacon performs I/O monitoring at all five Icefish components: the LWFS client (on compute nodes), the LWFS server and Lustre client (both on forwarding nodes), the Lustre server (on storage nodes), and the Lustre metadata server (on metadata nodes). At each of these *monitoring points*, Beacon deploys lightweight monitoring daemons that collect I/O-relevant events, status, and performance data locally, then transmit data for aggregation. Aggressive first-pass compression is conducted on all compute nodes for efficient per-application I/O trace collection/storage.

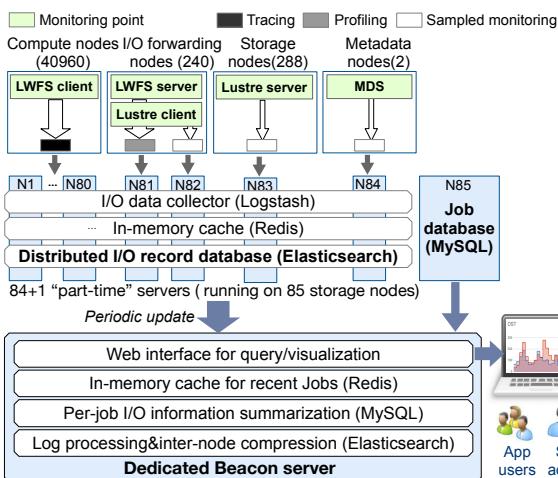


Figure 2: Beacon’s main components: daemons at monitoring points, distributed I/O record database, job database, plus a dedicated Beacon server. The different width of arrows into and out from a module indicates *data compression*.

Beacon has its major backend processing and storage workflow distributed to part of the storage nodes on their node-local disks, utilizing hardware resources and parallelism. To this end, Beacon divides the 40,960 compute nodes into 80 groups and enlists 80 of the 288 storage nodes to communicate with one group each. Two more storage nodes are used to collect data from the forwarding nodes, plus another for storage nodes and one last for MDS. Together, these 84 “part-time” servers (shown as “N1” to “N84” in Figure 2) are called *log servers*, which host a *distributed I/O record database* of Beacon. The numbers of such servers were selected empirically considering Icefish’s peak monitoring data processing workload.

These log servers adopt a layered software architecture built upon mature open-source frameworks. They collect I/O-relevant events, status and performance data through Logstash [9], a server-side log processing pipeline for simultaneously ingesting data from multiple sources. The data are then imported to Redis [16], a widely-used in-memory data

store, acting as a cache to quickly absorb monitoring output. Persistent data storage and subsequent analysis are done via Elasticsearch [5], a distributed, lightweight search and analytics engine supporting a NoSQL database. It also supports efficient Beacon queries, for real-time and offline analysis.

One more storage node (N85 in Figure 2) is used to host Beacon’s *job database* (implemented using MySQL [11]), which interacts with the job queuing system and keeps track of per-job information obtained by Beacon.

Finally, Beacon processes and presents its monitoring results to users (either system administrators or application users) using a *dedicated Beacon server*. There it performs two kinds of offline data analysis periodically: (1) second-pass, inter-node compression to further remove data redundancy by comparing and combining logs from compute nodes running the same job, and (2) extracting and caching in MySQL using SQL views the per-job statistic summary, while generating and caching in Redis common performance visualization results, to facilitate speedy user response. Log and monitoring data, after the two-pass compression, are permanently stored using Elasticsearch on this dedicated Beacon server.² Considering the typical daily data collection size of 10-100 GB, its 120 TB RAID5 capacity far exceeds the system’s lifetime storage space needs.

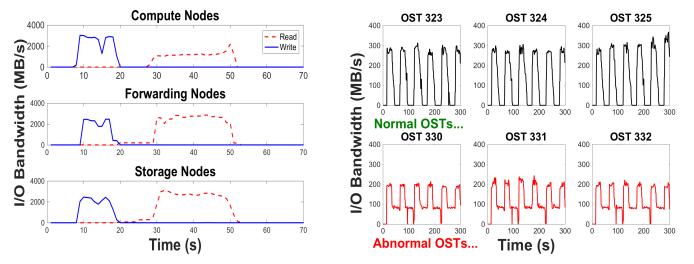


Figure 3: Sample display from Beacon’s web interface: (a) cross-layer read/write bandwidth of one user job, (b) bandwidth of three OSTs identified as undergoing anomaly.

On top of its Elasticsearch-MySQL-Redis stack, Beacon’s web interface provides users with a friendly GUI for I/O-related job/system information query processing and visualization. For instance, application users could query a summary of their programs’ I/O behavior based on job ID, along the entire I/O path, to help diagnosing I/O performance problems; system administrators can monitor real-time load levels on all forwarding, storage nodes and metadata servers, facilitating future job scheduling optimizations and center-level resource allocation policies. Figure 3 shows corresponding screenshots. Section 4 provides more details with concrete case studies.

All communication among Beacon entities uses a low-cost, easy-to-maintain Ethernet connection (marked in green in Figure 1), separate from *both* the main computation and

²Data in the *distributed I/O record database* are kept for 6 months.

the storage interconnects.

3.2 Multi-layer I/O Monitoring

Compute nodes On each of the 40,960 compute nodes, Beacon collects LWFS client trace logs. Each log entry contains the node’s IP, I/O operation type, file descriptor, offset, request size, and timestamp.

On a typical day, such raw trace data alone amounts to over 100 GB, making their collection/processing a non-trivial task on Beacon’s I/O record database, which takes away resources from the storage nodes. However, there exists abundant redundancy in HPC workloads’ I/O operations. For example, since each compute node is usually dedicated to one job at a time, the job IDs are identical among many trace entries. Similarly, due to the regular, tightly coupled nature of many parallel applications, adjacent I/O operations likely have common components such as target file, operation type, and request size. Recognizing this, Beacon performs aggressive online compression on each compute node to dramatically reduce the I/O trace size. This is done by a simple, linear algorithm comparing adjacent log items and combining those with identical operation type, file descriptor, and request size, and accessing *contiguous* areas. These log items are replaced with a single item plus a counter. Considering the low computing overhead, we perform such parallel first-pass compression on compute nodes.

Beacon conducts offline log processing and second-pass compression on the dedicated server. Here it extracts the feature vector $\langle \text{time}, \text{operation}, \text{file descriptor}, \text{size}, \text{offset} \rangle$ from the original log records, and performs inter-node compression by comparing feature vector lists from all nodes and merging identical vectors, using a similar approach as in block trace modeling [74] or ScalaTrace [61].

The compute-node-side first-pass compression reduces the raw trace size by a factor of **5.4** to **34.6** across 8 real-world, large-scale applications, where the gain relies on the amount of “immediate” redundancy in an application’s I/O operations. The second-pass compression on the dedicated Beacon server further delivers a 2- to 4-fold reduction. Detailed results are given in Appendix A.

Forwarding nodes On each forwarding node, Beacon profiles both the LWFS server and Lustre client. It collects the latency and processing time for each LWFS server request, and the request queue length for each LWFS server (by sampling the queue status once per 1000 requests). Rather than saving the per-request traces, the Beacon daemon periodically processes new traces and only saves I/O request statistics such as latency and queue length distribution.

For the Lustre client, Beacon collects request statistics by sampling the status of all outstanding RPC requests, once every second. Each sample contains the forwarding ID and RPC request size to the Lustre server.

Storage nodes and MDS On the storage nodes, Beacon daemons periodically sample the Lustre OST status table,

record data items such as the OST ID and OST total data size, and further send high-level statistics such as count of RPC requests and average per-RPC data size in the past time window. On the Lustre MDS, Beacon also periodically collects and records statistics on active metadata operations (such as *open* and *lookup*) at 1-second intervals, while storing a summary of the periodic statistics in its database.

3.3 Multi-layer I/O Profiling Data Analysis

All the aforementioned monitoring data are transmitted for long-term storage and processing at the Elasticsearch-based database on the dedicated Beacon server as JSON objects, on top of which Beacon builds I/O monitoring/profiling services. These include automatic anomaly detection that runs periodically, as well as query and visualization tools that supercomputer users and administrators could use interactively. Below we give more detailed descriptions.

Automatic anomaly detection Outright failures are relatively easy to detect in a large system, commonly handled by tools such as heartbeat detection [67, 72], and is beyond the scope of this work. However, alive yet very slow components, such as forwarding nodes and OSTs under performance degradation, may continue to serve requests, but at a much lower pace that drags down entire applications’ performance and reduces overall system utilization. With a busy storage system serving multiple platforms and on each many concurrent applications, such stragglers are rather hard to be identified. Assisted by its continuous, end-to-end I/O monitoring, Beacon enables automatic I/O system anomaly detection, identifying system components processing I/O workload at a significantly slower pace than their peers.

This is done by processing I/O monitoring data from the current and historical execution(s) of the same application, using clustering to detect apparent performance degradation on forwarding nodes and OSTs. The frequency of running such detection processing is configurable and is currently set at once every hour. Upon the identification of a serious system anomaly, an alarm email will be automatically generated and sent to TaihuLight administrators. We give a use case study in Section 4.2, plus detailed workflow description in Appendix B.

Per-job I/O performance analysis Upon a job’s completion, Beacon performs automatic analysis of its I/O monitoring data collected from all layers. It performs inter-layer correlation by first identifying jobs from the job database that run on given compute node(s) at the log entry collection time. The involved forwarding nodes, leading to relevant forwarding monitoring data, are then located via the compute-to-forwarding node mapping using a system-wide mapping table lookup. Finally, relevant OSTs and corresponding storage node monitoring data entries are found by file system lookup using the Lustre command `lfs`.

From the above data, Beacon derives and stores coarse-grained information for quick query, including average and

peak I/O bandwidth, average IOPS, runtime, number of processes (and compute nodes) performing I/O, I/O mode, total count of metadata operations, and average metadata operations per second during I/O phases. Among them, the *I/O mode* indicates the parallel file sharing mode among processes, where common modes include “N-N” (each compute process accesses a separate file), “N-1” (all processes share one file), “N-M” (N processes perform I/O aggregation to access M files, M<N), and “1-1” (only one of all processes performs sequential I/O on a single file).

To help users understand/debug their applications’ I/O performance, Beacon provides web-based I/O data visualization. This diagnosis system can be queried using a job ID, after appropriate authentication, and allows visualizing the I/O statistics of the job, both real-time and post-mortem. It reports the measured I/O metrics (such as bandwidth and IOPS) and inferred characteristics (such as the number of I/O processes and I/O mode). Users are also presented with user-configurable visualization tools, showing time-series measurement in I/O metrics, statistics information such as request type/size distribution, and performance variances. Our powerful I/O monitoring database allows further user-initiated navigation such as per-compute-node traffic history, and zooming control to examine data at different granularity. For security/privacy, users are only allowed to view I/O data from compute, forwarding, and storage nodes involved in and for the duration of their jobs’ execution.

I/O subsystem monitoring for administrators Beacon also provides administrators with the capability of monitoring the I/O status for any time period, on any node.

Besides all the user-visible information and facilities mentioned above, the administrators can further obtain and visualize: (1) detailed I/O bandwidth and IOPS for each compute node, forwarding node, and storage node, (2) resource utilization status of forwarding nodes, storage nodes and the MDS, including detailed request queue length statistics, and (3) I/O request latency distribution on forwarding nodes. Additionally, Beacon authorizes administrators with direct I/O record database access, to facilitate in-depth analysis.

Combining such facilities, administrators could perform powerful and thorough I/O traffic and performance analysis, e.g., by checking multi-level traffic, latency, and throughput monitoring information regarding a certain job execution.

3.4 Generality and Limitations

Beacon can be adopted by other platforms. The I/O forwarding architecture is widely used, by 9 out of the current Top 20 machines (listed in Table 1). It is also targeted by the DAOS Exascale storage design [54] and the TOKIO I/O monitoring framework [24].

Beacon’s building blocks, such as operation log collection and compression, scheduler-assisted per-application data correlation and analysis, history-based anomaly identification, automatic I/O mode detection, and built-in interfer-

Rank	Machine	Vendor	File system
3	Taihulight [19]	NRCPC	Lustre
4	Tianhe-2A [87]	NUDT	Lustre+H2FS
5	Piz Daint [15]	Cray	Lustre+GPFS
6	Trinity [21]	Cray	Lustre
9	Titan [20]	Cray	Lustre
10	Sequoia [17]	IBM	Lustre
12	Cori [1]	Cray	Lustre+GPFS
14	Oakforest-PACS [12]	Fujitsu	Lustre
18	K computer [8]	Fujitsu	FEFS [65]

Table 1: I/O forwarding adopters among top-20 supercomputers, as of November 2018

ence analysis, can all be performed on other supercomputers. Its data management components, such as Logstash, Redis, and ElasticSearch, are open-source software that will run on these machines as well. Our forwarding layer design validation and load analysis could also help recent platforms with a layer of burst buffer nodes, such as NERSC’s Cori [2].

Meanwhile, the current Beacon system has limitations that can be addressed in future work or application to other platforms. For example, it currently performs data analysis and detects anomalies by bringing unusual patterns to the attention of system administrators. Additional historical data collection to correlate symptoms and solutions would make the process more intelligent and reduce human labor requirement [86]. Similarly, application users who are not parallel I/O experts could benefit from system-generated direct suggestions (such as for I/O mode or request size change, and against using the parallel file system for metadata-heavy interactive tasks), beyond performance data visualization.

4 Beacon Use Cases

Beacon has been deployed on TaihuLight for around 18 months and has gathered massive I/O information. So far it has accumulated around 10 TB of trace data (after two passes of compression). This history contains 116,765 jobs that used at least 32 compute nodes, consuming 323,951,208 core-hours in total. 28,330 (24%) of these jobs featured non-trivial I/O, with per-job I/O volume over 200 MB.

The insights and issues revealed by Beacon’s monitoring and diagnosis have already helped TaihuLight administrators fix several design flaws, develop a dynamic and automatic forwarding node allocation tool, and improve system reliability/consistency plus application efficiency. Due to space limit, we focus on three types of use cases: (1) Performance issue diagnosis, (2) Automatic I/O anomaly diagnosis, and (3) Application and user behavior analysis.

4.1 Performance Issue Diagnosis

Forwarding node cache thrashing Beacon’s end-to-end monitoring facilitates *cross-layer correlation of I/O profiling data*, at different temporal or spatial granularity. By comparing the total request volume at each layer, Beacon has helped TaihuLight’s infrastructure management team in identifying a previously unknown performance issue, as detailed next.

A major reason driving the adoption of I/O forwarding or burst buffer layer is the opportunity to perform prefetching, caching, and buffering, to reduce the pressure on slower disk storage. Figure 4 shows the read volume on compute and forwarding node layers, during two sampled 70-hour periods in August 2017. Figure 4(a) shows a case with expected behavior, where the total volume requested by the compute nodes is significantly higher than that requested by the forwarding nodes, signaling good access locality and effective caching. Figure 4(b), however, tells the opposite story (that surprised system administrators): the forwarding layer could incur much higher read traffic from the backend than requested by user applications, reading much more data from the storage nodes than returning to compute nodes. Such situation does not apply to *writes*, where Beacon always shows matching aggregate bandwidth across the two levels.

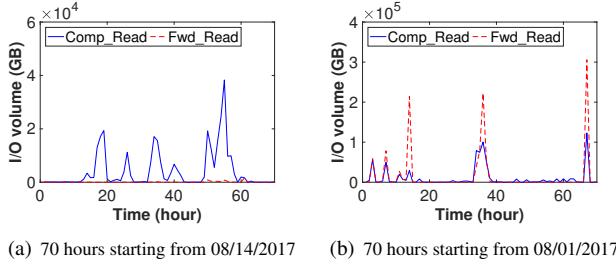


Figure 4: Sample segments of TaihuLight read volume history, each collected at two layers

Further analysis of the applications executed and their assigned forwarding nodes during the problem period in Figure 4(b) reveals an unknown *cache thrashing* problem, caused by the N-N sequential data access behavior. By default, the Lustre client has a 40 MB read-ahead cache for each file. Under the N-N sequential read scenarios, such aggressive prefetching causes severe memory contention, with data repeatedly read from the backend (and evicted on forwarding nodes). E.g., an 1024-process Shentu [25] execution has each I/O process read an 1-GB single file, incurring a $3.5 \times$ I/O amplification at the Lustre backend of Icefish. This echos previous finding on the existence of I/O self-contention within a single application [55].

Solution This problem can be addressed by adjusting the Lustre prefetching cache size per file. For example, changing it from 40MB per file to 2MB is shown to remove the thrashing. Automatic, per-job forwarding node cache reconfiguration, which leverages real-time Beacon monitoring results, is currently under development for TaihuLight. Alternatively, switching the application from an N-N to N-M mode (performing *I/O aggregation*, by having each set of N/M compute processes group their I/O to one file) also eliminates cache thrashing, and brings $3 \times$ I/O performance improvement. Given the close collaboration between application teams and machine administrators, making performance-critical program changes as suggested by monitoring data

analysis is an accepted practice on leading supercomputers.

Bursty forwarding node utilization Beacon’s continuous end-to-end I/O monitoring gives center management a global picture on system resource utilization. While such systems were often built and configured using rough estimates based on past experience, Beacon collects detailed resource usage history to help both in improving the current system’s efficiency and in assisting future system upgrade and design.

Figure 5 gives one example, again on forwarding load distribution, by showing two one-day samples from July 2017. Each row portraits the by-hour peak load on one of the same 40 forwarding nodes randomly sampled from the 80 active ones. The darkness reflects the maximum bandwidth reached within that hour. The labels “high”, “mid”, “low”, and “idle” correspond to that maximum residing in the >90%, 50-90%, 10-50%, or 0-10% interval (relative to the benchmarked per-forwarding-node peak bandwidth), respectively.

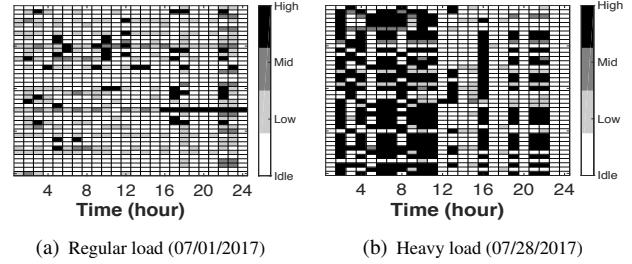


Figure 5: Sample TaihuLight one-day load summary, showing peak load level by hour, across 40 randomly sampled forwarding nodes

Figure 5(a) shows the more typical load distribution, where the majority of forwarding nodes stay lightly used for the vast majority of time (90.7% of cells show maximum load being under 50% of peak bandwidth). Figure 5(b) gives a very different picture, with a significant set of sampled forwarding nodes serving I/O-intensive large jobs for a good part of the day. 35.7% of the cells actually see a maximum load of over 99% of peak forwarding node bandwidth.

These results indicate that (1) overall there is forwarding resource overprovisioning (confirming prior findings [41, 52, 57, 64]), (2) even with the more representative low-load scenarios, it is not rare for forwarding node bandwidth to be saturated by application I/O, and (3) load imbalance across forwarding nodes exists regardless of load level, presenting idle resources potentially helpful to I/O-intensive applications.

Solution Recognizing the above, recently TaihuLight has enlisted more of its “backup forwarding nodes” into regular service. Meanwhile, a dynamic, application-aware forwarding node allocation scheme is designed and partially deployed (turned on for a subset of applications) [43]. Leveraging application-specific job history information, such an allocation scheme is intended to replace the default, static mapping between compute and forwarding nodes.

MDS request priority setting While overall we found that most TaihuLight jobs are rather metadata-light, Beacon

does observe a small fraction of parallel jobs (0.69%) with high metadata request rate (more than 300 metadata operations/s on average during I/O phases). Beacon found that these metadata-heavy (“high-MDOPS”) applications tend to cause significant I/O performance interference. Among jobs with Beacon-detected I/O performance anomaly, those sharing forwarding nodes with high-MDOPS jobs experienced an average $13.6 \times$ increase in read/write request latency during affected time periods.

Such severe delay and corresponding Beacon forwarding node queue status history prompted us to examine the TaihuLight LWFS server policy. We found that metadata requests were given priority over file I/O, based on the single-MDS design and the need to provide fast response to interactive user operations such as 1s. Here, as neither disk bandwidth nor metadata server capacity was saturated, such interference could easily remain undetected using existing approaches that focus on I/O-intensive workloads only [37, 52].

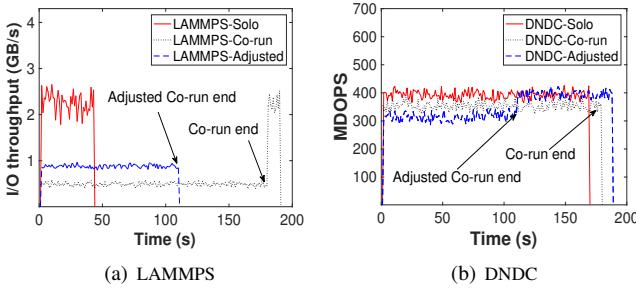


Figure 6: Impact of metadata operations’ priority adjustment

Solution As a temporary solution, we added probabilistic processing across priority classes to the TaihuLight LWFS scheduling. Instead of always giving metadata requests high priority, an LWFS server thread now follows a $P : (1 - P)$ split (P configurable) between picking the next request from the separate queues hosting metadata and non-metadata requests. Figure 6 shows the “before” and “after” pictures, with LAMMPS [34] (a classical molecular dynamics simulator with middle scale 256 compute nodes) running against the high-MDOPS DNDC [39] (a bio-geochemistry application for agro-ecosystems simulation). Throughput of their solo-runs, where each application runs by itself on an isolated testbed, is given as reference. With a simple equal probability split, LAMMPS co-run throughput doubles, while DNDC only perceives a 10% slowdown. For a long-term solution, we plan to leverage Beacon to automatically adapt the LWFS scheduling policies, considering operation types, the MDS load level, and application request scheduling fairness.

4.2 Automatic I/O Anomaly Diagnosis

In extreme-scale supercomputers, users typically accept jittery application performance, recognizing wide-spread resource sharing among jobs. System administrators, meanwhile, see different behaviors among system components

with homogeneous configuration, but cannot tell how much of that difference comes from these components’ functioning, and how much from the diversity of tasks they perform.

Beacon’s multi-layer monitoring capacity, therefore, presents a new window for supercomputer administrators to examine system health, by connecting statistics on application-issued I/O requests all the way to that of individual OST’s bandwidth measurement. Such connection guides Beacon to deduce what is considered the norm and what an exception. Leveraging this capability, we design and implement a lightweight, automatic anomaly detection tool to identify such *apparent exceptions* that signal significant performance degradation or faulty system components.

Application-driven anomaly detection Most I/O-intensive applications have distinct I/O phases, i.e., episodes in their execution where they perform I/O continuously, such as those to read input files during initialization or to write intermediate results or checkpoints. For a given application, such I/O phase behavior is often quite consistent. Taking advantage of such repeated I/O operations and its multi-layer I/O information collection, Beacon performs automatic I/O phase recognition, on top of which it conducts I/O-related anomaly detection. More specifically, larger applications (such as those using 1024 compute nodes or more) are spreading their I/O load to multiple forwarding nodes and backend nodes, giving us opportunities to directly compare the behavior of these servers processing requests *known to Beacon* as homogeneous or highly similar.

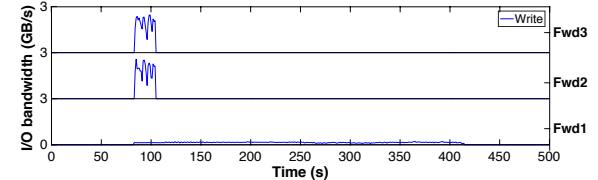


Figure 7: Forwarding bandwidth in a 6000-process LAMMPS run

Figure 7 gives an example of a 6000-process LAMMPS run with checkpointing. The 1500 compute nodes are assigned to 3 forwarding nodes, whose bandwidth and I/O time are reflected in the time-series data from Beacon. We can clearly see here the Fwd1 node is a straggler in this case, serving at a bandwidth much slower than its peak (without answering to other applications). As a result, there is a $20 \times$ increase in the application-visible checkpoint operation time, estimated using the other two forwarding nodes’ I/O phase duration.

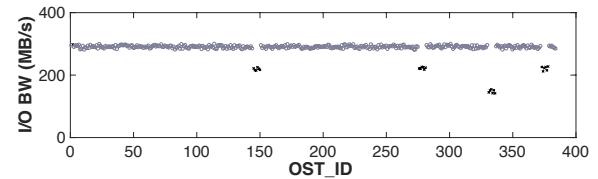


Figure 8: Per-OST bandwidth during a Shentu execution

Anomaly alert and node screening Such continuous, online application performance anomaly detection could iden-

tify forwarding nodes or backend units with deviant performance metrics, which in turn will trigger Beacon’s more detailed monitoring and analysis. If it finds such a system component to consistently under-perform relative to peers serving similar workloads, with configurable thresholds in monitoring window and degree of behavior deviation, it reports this as an automatically detected system anomaly. By generating and sending an alarm email to the system administration team, Beacon prompts system administrators to do a thorough examination, where its detailed performance history information and visualization tools are also helpful.

Such anomaly screening is especially important for expensive, large-scale executions. For example, among all applications running on TaihuLight so far, the parallel graph engine Shentu [49] has the most intensive I/O load. It scales well to the entire supercomputer in both computation and I/O, with 160,000 processes and large input graphs distributed evenly to nearly 400 Lustre OSTs. During test runs preparing for its Gordon Bell bid in April 2018, Beacon’s monitoring discovered a few OSTs significantly lagging behind in the parallel read, slowing down the initialization as a result (Figure 8). By removing them temporarily from service and relocating their data to other OSTs, Shentu cut its production run initialization time by 60%, saving expensive dedicated system allocation and power consumption. In this particular case, further manual examination attributed the problem to these OSTs’ RAID controllers, which were later fixed.

Had it not been for Beacon’s backend monitoring, applications like Shentu would have accepted whatever bandwidth they got, without suspecting I/O performance being abnormal. Similarly, had it not been for Beacon’s routine frontend tracing, profiling, and per-application performance anomaly detection, it would not have noticed the backend outliers. As full-system benchmarking requires taking the supercomputer offline and cannot be regularly attempted, Beacon provides a much more affordable way for continuous system health monitoring and diagnosis, by coupling application-side and server-side tracing/profiling information.

Duration (hours)	Location of anomaly	
	Forwarding node (times)	OSS+OST (times)
(0,1]	23	31
[1,4]	14	17
[4,12]	5	9
[12,96)	3	6
≥ 96 , manually verified	6	8

Table 2: Duration of Beacon-identified system anomalies

Beacon’s deployment on TaihuLight started around April 2017, with features and tools incrementally developed and added to production use. Table 2 summarizes the automatically identified I/O system anomaly occurrences at the two service layers, from Apr 2017 to Aug 2018. Such identification adopted a minimum threshold of measured maximum bandwidth under 30% of the known peak value, as well as a minimum duration of 60 minutes. Such parameters can be configured to adjust the anomaly detection system sensitiv-

ity. Most performance anomaly occurrences are found to be transient, lasting under 4 hours.

There are a total of 14 occasions of performance anomaly over 4 hours on the forwarding layer, and 23 on the backend layer, confirming the existence of fail-slow situations found common with data centers [42]. Reasons for such relatively long yet “self-healed” anomalies include service migration and RAID reconstruction. With our rather conservative setting during such initial deployment period, Beacon was set to send the aforementioned alert email when a detected anomaly situation lasted beyond 96 hours (except for large-scale production runs as in the Shentu example above, where the faulty units were immediately reported). With all these occasions, the Beacon detected anomaly was confirmed by human examination.

4.3 Application and User Behavior Analysis

With its powerful information collection and multi-layer I/O activity correlation, Beacon provides new capability to perform detailed application or user behavior analysis. Results of such analysis assist in performance optimization, resource provisioning, and future system design. Here we showcase several application/user behavior studies, some of which have brought corresponding optimizations or design changes to the TaihuLight system.

Type	$(0, 1K]$	$(1K, 10K]$	$(10K, 100K]$	$(100K, 1000K]$	$(1000K, \infty)$
Read	8.1 GB	101.0 GB	166.9 GB	1172.9 GB	2010.6 GB
Write	18.2 GB	83.9 GB	426.6 GB	615.9 GB	41458.8 GB

Table 3: Avg. per-job I/O volume by core-hour consumption

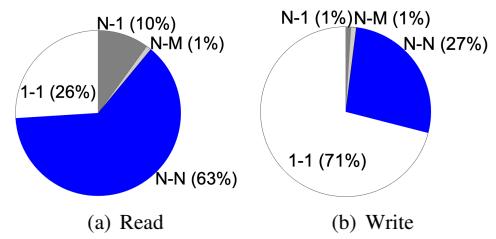


Figure 9: Distribution of file access modes, in access volume

I/O mode	Avg. read volume	Avg. write volume	Job count
N-N	96.8 GB	120.1 GB	11073
N-M	36.2 GB	63.2 GB	324
N-1	19.6 GB	19.3 GB	2382
1-1	33.0 GB	142.3 GB	16251

Table 4: Avg. I/O volume and job count by I/O mode

Application I/O mode analysis First, Table 3 gives an overview of I/O volume across all profiled jobs with non-trivial I/O, categorized by per-job core-hour consumption. Here, 1000K core-hours correspond to a 10-hour run using 100,000 cores on 25,000 compute nodes, and jobs with such consumption or higher write more than 40 TB of data on average.³ Overall, the amount of data read/written grows as

³Further examination reveals that in each core-hour category, average read/write volumes are influenced by a minority group of heavy consumers.

the jobs consume more compute node resources. The less resource-intensive applications tend to perform more reads, while the larger consumers are more write-intensive.

Figure 9 shows the breakdown of I/O-mode adoption among all TaihuLight jobs performing non-trivial I/O, by total read/write volume. The first impression one gets from these results is that the rather “extreme” cases, such as N-N and 1-1, form the dominant choices, especially in the case of writes. We suspected that this distribution could be skewed by a large number of small jobs doing very limited I/O, and calculated the average per-job read/write volume for each I/O mode. The results (Table 4) show that this is not the case. Actually, applications that choose to use 1-1 mode for writes actually have a much higher overall write volume.

The 1-1 mode is the closest to sequential processing behavior and is conceptually simple. However, it obviously lacks scalability and fails to utilize the abundant hardware parallelism in the TaihuLight I/O system. The wide presentation of this I/O mode might help explain the overall under-utilization of forwarding resources, discussed earlier in Section 4.1. Echoing similar findings (though not so extreme) on other supercomputers [57] (including Intrepid [7], Mira [10] and Edison [4]), effective user education on I/O performance and scalability would both help improve storage system utilization and reduce wasted compute resources.

The N-1 mode has a different story. It is an intuitive parallel I/O solution that allows compute processes to directly read to or write from their local memory without gather-scatter operations, while retaining the convenience of having a single input/output file. However, our detailed monitoring finds it a quite damaging I/O mode that users should steer away from, as explained below.

First, our monitoring results confirm findings by existing research [23, 56] that the N-1 mode offers low application I/O performance (by reading/writing to a shared file). Even with a large N, such applications receive no more than 250 MB/s I/O aggregate throughput, despite the peak TaihuLight backend combined bandwidth of 260 GB/s. For read operations, users here also rarely modify the default Lustre stripe width, confirming behavior reported in a recent ORNL study [48]. The problem is much worse with writes, as performance severely degrades due to file system locking.

This study, however, finds N-1 applications to be extraordinarily *disruptive* as they harm all kinds of neighbor applications that share forwarding nodes with them, particularly when N is large (e.g., over 32 compute nodes).

The reason is that each forwarding node operates an LWFS server thread pool (currently sized at 16), providing forwarding service to assigned compute nodes. Applications using the N-1 mode tend to flood this thread pool with requests in bursts. Unlike with the N-N or N-M modes, N-1 suffers from the aforementioned poor backend performance by using a single shared file. This, in turn, makes N-1 requests slow to process, further exacerbating their conges-

tion in the queue and delaying requests from other applications, even when those victims are accessing disjoint backend servers and OSTs.

Here we give a concrete example of I/O mode-induced performance interference, featuring the earthquake simulation AWP [35] (2017 Gordon Bell Prize winner) that started with N-1 mode. In this sample execution, it co-runs with the weather forecast application WRF [69] using the 1-1 mode, each having 1024 processes on 256 compute nodes. Under the “solo” mode, we assign each application a dedicated forwarding node in a small testbed partition of TaihuLight. In the “co-run” mode, we let them share one forwarding node (as the default compute-to-forwarding mapping is 512-to-1).

Operation	Avg. wait time	Avg. proc. time	Avg. queue length
WRF write (solo)	2.73 ms	0.052 ms	0.22
WRF write (co-run)	30.06 ms	0.054 ms	208.51
AWP read (solo)	58.17 ms	3.44 ms	226.37
AWP read (co-run)	58.18 ms	3.44 ms	208.51

Table 5: Performance interference during WRF and AWP co-run sharing a forwarding node

Table 5 lists the two applications’ average request wait/processing time and forwarding node queue length during these runs. Note that with “co-run”, the queue is shared by both applications. We find that the average wait time of WRF has been increased by 11× when co-running, but AWP is not affected. This result reveals the profound malpractice of the N-1 file sharing mode and confirms prior finding that I/O interference is access-pattern-dependent [47, 53].

Solution Our tests confirm that increasing the LWFS thread pool size does not help in this case, as the bottleneck lies on the OSTs. Meanwhile, avoiding the N-1 mode has been advised in prior work [23, 84], as well as numerous parallel I/O tutorials. Considering our new inter-application study results, it is rather an obvious “win-win” strategy that simultaneously improves large applications’ I/O performance and reduces their disruption to concurrent workloads. However, based on our experience with real applications, this message needs to be better promoted.

In our case, the Beacon developers worked with the AWP team to replace its original N-1 file read (for initialization/restart) with the N-M mode, during the 2017 Gordon Bell Prize final submission phase. This change produced an over 400% enhancement in I/O performance. Note that the GB Prize submission does not report I/O time; we found that AWP’s 130,000-process production runs spend the bulk of their execution time reading around 100 TB of input or checkpoint data. Significant reduction in this time greatly facilitated AWP’s development/testing and saved non-trivial supercomputer resources.

Metadata Server Usage Unlike forwarding nodes utilization (discussed earlier), the Lustre MDS is found with rather evenly distributed load levels by Beacon’s continuous load monitoring (Figure 10(a)). In particular, in 26.8% of the time, the MDS experiences a load level (in requests per second) above 75% of its peak processing throughput.

Beacon allows us to further split the requests between systems sharing the MDS, including the TaihuLight forwarding nodes, login nodes, and the ACC. TaihuLight administrators were surprised to find that over 80% of the metadata access workload actually comes from the ACC (Figure 10(b)).

Note that the login node and ACC have their own local file systems, ext4 and GPFS [66], respectively, which users are encouraged to use for purposes such as application compilation and data post-processing/visualization. However, since the users are likely TaihuLight users too, we found most of them prefer to directly use the main Lustre scratch file system intended for TaihuLight jobs, for convenience. While the I/O bandwidth/IOPS resources consumed by such tasks are negligible, user interactive activities (such as compiling or post-processing) turn out to be metadata heavy.

Large waves of unintended user activities correspond to the most heavy-load periods at the tail end in Figure 10(a), and have led to MDS crashes that directly affected applications running on TaihuLight. According to our survey, many other machines, including 2 out of the top 10 supercomputers (Sequoia [17] and Sierra [18]), also have a single MDS, assuming that their users follow similar usage guidelines.

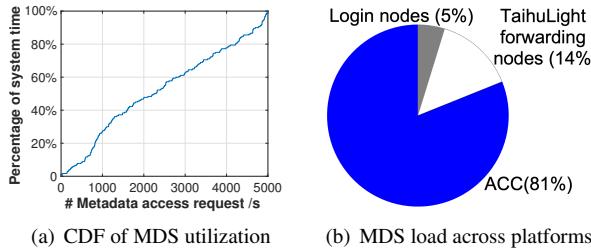


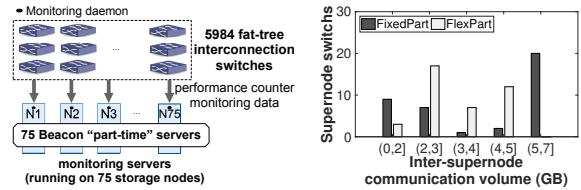
Figure 10: TaihuLight Lustre MDS load statistics

Solution There are several potential solutions to this problem. With the help of Beacon, we can identify and remind users performing metadata-heavy activities to avoid using the PFS directly. Or we can support more scalable Lustre metadata processing with an MDS cluster. A third approach is to facilitate intelligent workflow support that automatically performs data transfer, based on users' needs. This third approach is the one we are currently developing.

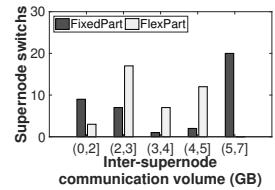
4.4 Extension to network monitoring

Encouraged by Beacon's success in I/O monitoring, in summer 2018 we began to design and test its extension to monitor and analyze network problems, motivated by the network performance debugging needs of ultra large-scale applications. Figure 11(a) shows the architecture of this new module. Beacon samples performance counters on the 5984 Mellanox InfiniBand network switches, such as port sent and received volumes. Again the data collected are passed to low-overhead daemons on Beacon log servers, more specifically, 75 of its 85 part-time servers, each assigned 80 switches. Similar processing and compression are conducted, with result data persisted in Beacon's distributed

database, then periodically relocated to its dedicated server for user queries and permanent storage.



(a) Overview of Beacon's network monitoring module



(b) Distribution of communication volume inter-supernode

This Beacon network monitoring prototype was tested in time to help in the aforementioned Shentu [49] production runs, for its final submission to Supercomputing'18 as an ACM Gordon Bell Award finalist. Beacon was called upon to identify the reason of aggregate network bandwidth significantly lower than theoretical peak. Figure 11(b) illustrates this with a 3-supernode Shentu test run. The dark bars (FixedPart) form a histogram of communication volumes measured on 40 switches connecting these 256-node supernodes for inter-supernode communication, reporting the count of switches within 5 volume brackets. There is a clear bipolar distribution, showing severe load imbalance and more than expected inter-supernode communication. This monitoring result led to discovery that due to the existence of faulty compute nodes within each supernode, the fixed partitioning relay strategy adopted by Shentu led to a subset of relay nodes receiving twice the "normal" load. Note that Shentu's own application-level profiling found communication volume across compute nodes very well balanced, hence the problem was not obvious to application developers until Beacon provided such switch-level traffic data.

Solution This finding prompted Shentu designers to optimize their relay strategy, using a topology-aware scholastic assignment algorithm to uniformly partition source nodes to relay nodes [49], whose results are shown by gray bars (Flex-Part) in Figure 11(b). The peak per-switch communication volume is reduced by 27.0% (from 6.3 GB to 4.6 GB), with significantly improved load balance, bringing a total communication performance enhancement of around 30%.

5 Beacon Framework Evaluation

We now evaluate Beacon's per-application profiling accuracy, as well as its performance overhead.

5.1 Accuracy Verification

Beacon collects full traces from the compute node side, thus has access to complete application-level I/O operation information. However, since the LWFS client trace interface provides only coarse timestamp data (at per-second granularity), and due to the clock drift across compute nodes, it is possible that the I/O patterns recovered from Beacon logs deviate from the application-level captured records.

To evaluate the degree of such errors, we compare the I/O throughput statistics reported by MPI-IO Test [40] to those by Beacon. In the experiments, we use MPI-IO Test to test different parallel I/O modes, including N-N and N-1 independent operations, plus MPI-IO library collective calls. 10 experiments were repeated at each execution scale.

The accuracy evaluation results are shown in Figure 12. We plot the average error in Beacon, measured as the percentage of deviation of the recorded aggregate compute node-side I/O throughput from the application-level throughput reported by the MPI-IO library.

We find Beacon able to accurately capture application performance, even for applications with non-trivial parallel I/O activities. More precisely, Beacon’s recorded throughput deviates from MPI-IO Test reported values by only 0.78–3.39% (1.84% on average) for the read test and 0.81–3.31% (2.03% on average) for write, respectively. Results are similar with high-IOPS applications, omitted due to space limit.

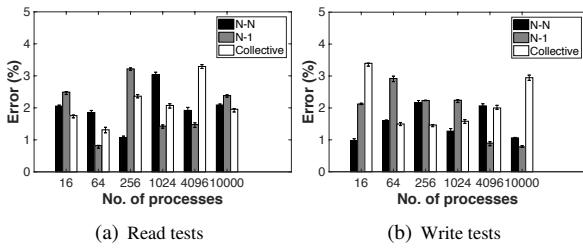


Figure 12: Average error rate of Beacon reported bandwidth (error bars show 95% confidence intervals.)

Beacon’s accuracy can be attributed to that it records *all* compute node-side trace logs, facilitated by its efficient and nearly lossless compression method (described in Section 3.2). We find that even though individual trace items may be off in timestamps, data-intensive applications on supercomputers seldom perform isolated, fast I/O operations (which are not of interest for profiling purposes); instead, they exhibit I/O phases with sustained high I/O intensity. By collecting a large set of per-application I/O trace entries, Beacon is able to paint an accurate picture of an application’s I/O behavior and performance.

5.2 Monitoring and Query Overhead

We now evaluate Beacon’s monitoring overhead in a production environment. We compare the performance of important I/O-intensive real-world applications and the MPI-IO Test benchmark discussed earlier, with and without Beacon turned on (T_w and $T_{w/o}$, respectively). We report the overall run time of each program and calculate the slowdown introduced by turning on Beacon. Table 6 shows the results, listing the average slowdown measured from at least 5 runs for each program (variance of slowdown across runs very low: under 2%). Note that for the largest applications, such testing was piggybacked on actual production runs of stable codes, with Beacon turned on during certain allocation time

frames. Applications like AWP often break their executions to run a certain number of simulation timesteps at a time.

Application	#Process	$T_{w/o}$ (s)	T_w (s)	%Slowdown
MPI-IO _N	64	26.6	26.8	0.79%
MPI-IO _N	128	31.5	31.6	0.25%
MPI-IO _N	256	41.6	41.9	0.72%
MPI-IO _N	512	57.9	58.4	0.86%
MPI-IO _N	1024	123.1	123.5	0.36%
WRF ₁	1024	2813.3	2819.1	0.21%
DNDC	2048	1041.2	1045.5	0.53%
XCFD	4000	2642.1	2644.6	0.09%
GKUA	16384	297.5	299.9	0.82%
GKUA	32768	182.8	184.1	0.66%
AWP	130000	3233.5	3241.5	0.25%
Shentu	160000	5468.2	5476.3	0.15%

Table 6: Avg. Beacon monitoring overhead on applications

These results show that the Beacon tool introduces very low overhead: under 1% across all test cases. Also, the overhead does not grow with application execution scale, and actually appears smaller (below 0.25%) for the two largest jobs, which use 130K processes or more. Such cost is particularly negligible considering the significant I/O performance enhancement and run time saving produced by optimizations or problem diagnosis from Beacon-supplied information.

Table 7 lists the CPU and memory usage of Beacon’s data collection daemon. In addition, the storage overhead from Beacon’s deployment on TaihuLight since April 2017 is around 10 TB. Such low operational overhead and scalable operation attest to Beacon’s lightweight design, with background trace-collection and compression generating negligible additional resource consumption. Also, having separate monitoring network and storage avoids potential disturbance to application execution.

Level	CPU usage	Memory usage (MB)
Compute node	0.0%	10
Forwarding node	0.1%	6
Storage node	0.1%	5

Table 7: System overhead of Beacon

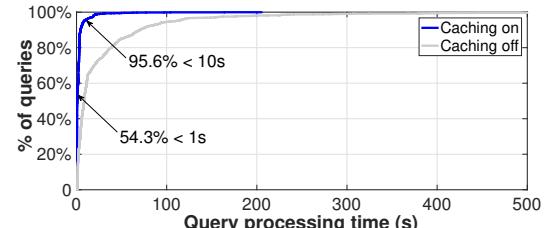


Figure 13: CDF of Beacon query processing time

Finally we assess Beacon’s query processing performance. We measured the query processing time of 2000 Beacon queries in September 2018, including both application users accessing job performance analysis and system administrators checking forwarding/storage nodes performance. In particular, we examined the impact of Beacon’s in-memory cache system between the web interface and Elasticsearch, as shown in Figure 2. Figure 13 gives the CDF of queries

in processing time and demonstrates that (1) the majority of Beacon user queries can be processed within 1 second and 95.6% of them under 10 seconds (visualization queries take longer), and (2) Beacon’s in-memory caching significantly improves user experience. Additional checking reveals that about 95% of these queries can be served from data cached.

6 Related Work

Several I/O tracing and profiling tools have been proposed for HPC systems, which can be divided into two categories: application-oriented tools and backend-oriented tools.

Application-oriented tools can provide detailed information about a particular execution on a function-by-function basis. Work at this front includes Darshan [31], IPM [76], and RIOT [81], all aiming at building an accurate picture of application I/O behavior by capturing key characteristics of the mainstream I/O stack on compute nodes. Carns et al. evaluated performance and runtime overheads of Darshan [30]. Wu et al. proposed a scalable methodology for MPI and I/O event tracing [58, 82, 83]. Recorder [56] focused on collecting additional HDF5 trace data.

Tools like Darshan provide user-transparent monitoring via automatic environment configuration. Still, instrumentation based tools have restrictions on programming languages or libraries/linkers. In contrast, Beacon is designed to be a non-stop, full-system I/O monitoring system capturing I/O activities at the system level.

Backend-oriented tools collect system-level I/O performance data across applications and provide summary statistics (e.g. LIOProf [85], LustreDU [29, 48, 62] and LMT [38]). However, identifying application performance issues and finding the cause of application performance degradation are difficult with these tools. While backend analytical methods [50, 52] made progress in identifying high-throughput applications using backend logs only, they lack application-side information. Beacon, on the other hand, holds complete cross-layer monitoring data to afford such tasks.

Along this line, there exist tools collecting multi-layer data. Static instrumentation was used to trace parallel I/O calls from MPI to PVFS servers [46]. SIOX [80] and IOPin [45] characterize HPC I/O workloads across the I/O stack. These projects extended the application-level I/O instrumentation approach that Darshan [31] used, to other system layers. However, their overhead hinders its deployment on large-scale production environments [70].

Regarding end-to-end frameworks, the TOKIO [24] architecture combined frontend tools (Darshan, Recorder) and backend ones (LMT). E.g., the UMAMI monitoring interface [53] provided cross-layer I/O performance analysis and visualization. In addition, OVIS [27] used the Cray specific tool LDMS [22] to provide scalable failure and anomaly detection. GUIDE [91] performed center-wide and multi-source log collection and motivated further analysis and optimizations. Beacon differs by its aggressive real-time per-

formance and utilization monitoring, automatic anomaly detection, and continuous per-application I/O pattern profiling.

I/O interference is identified as an important cause for performance variability in HPC systems [52, 63]. Kuo et al. [47] focused on interference from different file access patterns with synchronized time-slice profiles. Yildiz et al. [88] studied root causes of cross-application I/O interference across software and hardware configurations. To our knowledge, Beacon is the first monitoring framework with built-in features for inter-application interference analysis. Our study confirms findings on large-scale HPC applications’ adoption of poor I/O design choices [57]. It further suggests that aside from workload-dependent, I/O-aware scheduling [33, 52], interference should be countered with application I/O mode optimization and adaptive I/O resource allocation.

Finally, on network monitoring, there are dedicated tools [51, 59, 68] for monitoring switch performance, anomaly detection, and resource utilization optimization. There are also tools specializing in network monitoring/debugging for data centers [14, 73, 89]. However, these tools/systems typically do not target InfiniBand interconnections commonly used on supercomputers. To this end, Beacon adopts the open-source OFED stack [13, 32] to retrieve relevant information from IB network. More importantly, it leverages its scalable and efficient monitoring infrastructure, originally designed for I/O, for network problems.

7 Conclusion

We present Beacon, an end-to-end I/O resource monitoring and diagnosis system for the leading supercomputer TaihuLight. It facilitates comprehensive I/O behavior analysis along the long I/O path and has identified hidden performance and user I/O behavior issues, as well as system anomalies. Enhancement enabled by Beacon in the past 18 months has significantly improved ultra large-scale applications’ I/O performance and the overall TaihuLight I/O resource utilization. More generally, our results and experience indicate that this type of detailed multi-layer I/O monitoring/profiling is affordable at state-of-the-art supercomputers, offering valuable insights while incurring very low cost.

Acknowledgement

We appreciate the thorough and constructive comments from all reviewers. Particularly, we thank our shepherd, Haryadi Gunawi, for his responsiveness and detailed guidance. We also thank Prof. Zheng Weimin for his valuable guidance and advice, and colleagues Xiongchao Tang and Haojie Wang for their input. We thank NSCC-Wuxi for supporting our development, data collection, and deployment. This work is partially supported by the National Key R&D Program of China (Grant No. 2017YFA0604500 and 2016YFA0602100) and the National Natural Science Foundation of China (Grant No. 61722208, 41776010, and U1806205).

References

- [1] Cori supercomputer. <http://www.nersc.gov/users/computational-systems/cori/>.
- [2] Cray burst buffer in Cori. <http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/>.
- [3] DBSCAN. <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html>.
- [4] Edison supercomputer. <http://www.nersc.gov/users/computational-systems/edison/>.
- [5] Elasticsearch. <https://www.elastic.co/products/elasticsearch>.
- [6] GlusterFS. <https://www.gluster.org/>.
- [7] Intrepid. <https://www.alcf.anl.gov/intrepid>.
- [8] K supercomputer. <http://www.aics.riken.jp/en/>.
- [9] Logstash. <https://www.elastic.co/products/logstash>.
- [10] Mira supercomputer. <https://www.alcf.anl.gov/mira>.
- [11] MySQL database. <https://www.mysql.com>.
- [12] Oakforest-PACS supercomputer. http://jcahpc.jp/eng/ofp_intro.html.
- [13] Open Fabrics Alliance. <http://www.openfabrics.org/>.
- [14] PathDump. <https://github.com/PathDump>.
- [15] Piz Daint supercomputer. <https://www.cscs.ch/computers/dismissed/piz-daint-piz-dora/>.
- [16] Redis. <http://redis.io/>.
- [17] Sequoia supercomputer. <https://computation.llnl.gov/computers/sequoia>.
- [18] Sierra supercomputer. <https://hpc.llnl.gov/hardware/platforms/sierra>.
- [19] Sunway TaihuLight supercomputer. <https://http://www.nsccwx.cn/>.
- [20] Titan supercomputer. <https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/>.
- [21] Trinity supercomputer. <http://www.lanl.gov/projects/trinity/>.
- [22] AGELASTOS, A., ALLAN, B., BRANDT, J., CASSELLA, P., ENOS, J., FULLOP, J., GENTILE, A., MONK, S., NAKSINEHABOON, N., OGDEN, J., RAJAN, M., SHOWERMAN, M., STEVENSON, J., TAERAT, N., AND TUCKER, T. The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2014).
- [23] BENT, J., GIBSON, G., GRIDER, G., MCCLELLAND, B., NOWOCZYNKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. PLFS: A Checkpoint filesystem for parallel applications. In *Proceedings of Supercomputing* (2009).
- [24] BERKELEY, L., AND ANL. TOKIO:Total knowledge of I/O, 2017. <http://www.nersc.gov/research-and-development/tokio>.
- [25] BOWEN YU, YOUWEI ZHOU, H. L. X. T. W. C. J. Z. W. Y., AND ZHENG, W. Scalable graph traversal on Sunway TaihuLight with ten million cores. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2017).
- [26] BRAAM, P. J., AND ZAHIR, R. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc* (2002).
- [27] BRANDT, J., GENTILE, A., MAYO, J., PEBAY, P., ROE, D., THOMPSON, D., AND WONG, M. Resource monitoring and management with OVIS to enable HPC in cloud computing environments. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2009).
- [28] BUDNIK, T., KNUDSON, B., MEGERIAN, M., MILLER, S., MUNDY, M., AND STOCKDELL, W. Blue Gene/Q resource management architecture. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)* (2010).
- [29] CARLYLE, A. G., MILLER, R. G., LEVERMAN, D. B., RENAUD, W. A., AND MAXWELL, D. E. Practical support solutions for a workflow-oriented Cray environment. In *Proceedings of Cray User Group Conference (CUG)* (2012).
- [30] CARNS, P., HARMS, K., LATHAM, R., AND ROSS, R. Performance analysis of Darshan 2.2.3 on the Cray XE6 platform. *Argonne National Laboratory (ANL)* (2012).
- [31] CARNS, P. H., LATHAM, R., ROSS, R. B., ISKRA, K., LANG, S., AND RILEY, K. 24/7 characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS)* (2009).
- [32] DANDAPANTHULA, N., SUBRAMONI, H., VIENNE, J., KANDALLA, K., SUR, S., PANDA, D. K., AND BRIGHTWELL, R. INAM-a scalable infiniband network analysis and monitoring tool. In *European Conference on Parallel Processing (Euro-Par)* (2011).
- [33] DORIER, M., ANTONIU, G., ROSS, R., KIMPE, D., AND IBRAHIM, S. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2014).
- [34] DUAN, X., CHEN, D., MENG, X., YANG, G., GAO, P., ZHANG, T., ZHANG, M., LIU, W., ZHANG, W., AND XUE, W. Redesigning LAMMPS for petascale and hundred-billion-atom simulation on Sunway TaihuLight. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2018).
- [35] FU, H., HE, C., CHEN, B., YIN, Z., ZHANG, Z., ZHANG, W., ZHANG, T., XUE, W., LIU, W., YIN, W., YANG, G., AND CHEN, X. 18.9-Pflops nonlinear earthquake simulation on Sunway TaihuLight: Enabling depiction of 18-Hz and 8-meter scenarios. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2017).
- [36] FU, H., LIAO, J., YANG, J., WANG, L., SONG, Z., HUANG, X., YANG, C., XUE, W., LIU, F., QIAO, F., ZHAO, W., YIN, X., HOU, C., ZHANG, C., GE, W., ZHANG, J., WANG, Y., ZHOU, C., AND YANG, G. The Sunway TaihuLight supercomputer: System and applications. *Science China Information Sciences* (2016).
- [37] GAINARU, A., AUPY, G., BENOIT, A., CAPPELLO, F., ROBERT, Y., AND SNIR, M. Scheduling the I/O of HPC applications under congestion. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2015).
- [38] GARLICK, J. Lustre monitoring tool, 2010. <https://github.com/LNL/lmt>.
- [39] GILTRAP, D. L., LI, C., AND SAGGAR, S. DNDC: A process-based model of greenhouse gas fluxes from agricultural soils. *Agriculture, Ecosystems & Environment* (2010).
- [40] GRIDER, G., NUNEZ, J., AND BENT, J. LANL MPI-IO test, 2008. <http://freshmeat.sourceforge.net/projects/mpiotest>.
- [41] GUNASEKARAN, R., ORAL, S., HILL, J., MILLER, R., WANG, F., AND LEVERMAN, D. Comparative I/O workload characterization of two leadership class storage clusters. In *Proceedings of the 10th Parallel Data Storage Workshop* (2015).
- [42] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARAMAN, S., LIN, X., EMAMI, T., SHENG, W.,

- BIDOKHTI, N., MCCAFFREY, C., SRINIVASAN, D., PANDA, B., BAPTIST, A., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNE-SHA, H. B., HAO, M., AND LI, H. Fail-slow at scale: evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST)* (2018).
- [43] JI, X., YANG, B., ZHANG, T., MA, X., ZHU, X., WANG, X., EI-SAYED, N., ZHAI, J., LIU, W., AND XUE, W. Automatic Application-Aware I/O Forwarding Resource Allocation for High-end System. In *17th USENIX Conference on File and Storage Technologies (FAST)* (2019).
- [44] JOKANOVIC, A., SANCHO, J. C., RODRIGUEZ, G., LUCERO, A., MINKENBERG, C., AND LABARTA, J. Quiet neighborhoods: Key to protect job performance predictability. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2015).
- [45] KIM, S. J., SON, S. W., LIAO, W.-K., KANDEMIR, M., THAKUR, R., AND CHOUDHARY, A. IOPin: Runtime profiling of parallel I/O in HPC systems. In *High Performance Computing, Networking, Storage and Analysis (SCC)* (2012).
- [46] KIM, S. J., ZHANG, Y., SON, S. W., PRABHAKAR, R., KANDEMIR, M., PATRICK, C., LIAO, W.-K., AND CHOUDHARY, A. Automated tracing of I/O stack. In *European MPI Users' Group Meeting* (2010).
- [47] KUO, C.-S., SHAH, A., NOMURA, A., MATSUOKA, S., AND WOLF, F. How file access patterns influence interference among cluster applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2014).
- [48] LIM, SEUNG-HWAN AND SIM, HYOGI AND GUNASEKARAN, RAGHUL AND VAZHAKUDAI, SUDHARSHAN S. Scientific user behavior and data-sharing trends in a petascale file system. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2017).
- [49] LIN, H., ZHU, X., YU, B., TANG, X., XUE, W., CHEN, W., ZHANG, L., HOEFLER, T., MA, X., LIU, X., ZHENG, W., AND XU, J. Shentu: Processing multi-trillion edge graphs on millions of cores in seconds. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2018).
- [50] LIU, Y., GUNASEKARAN, R., MA, X., AND VAZHAKUDAI, S. S. Automatic identification of application I/O signatures from noisy server-side traces. In *12th USENIX Conference on File and Storage Technologies (FAST)* (2014).
- [51] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016).
- [52] LIU, YANG AND GUNASEKARAN, RAGHUL AND MA, XIAOSONG AND VAZHAKUDAI, SUDHARSHAN S. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2016).
- [53] LOCKWOOD, G. K., YOO, W., BYNA, S., WRIGHT, N. J., SNYDER, S., HARMS, K., NAULT, Z., AND CARNS, P. UMAMI: A recipe for generating meaningful metrics through holistic I/O performance analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems* (2017).
- [54] LOFSTEAD, J., JIMENEZ, I., MALTZAHN, C., KOZIOL, Q., BENT, J., AND BARTON, E. Daos and friends: A proposal for an exascale storage system. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2016).
- [55] LOFSTEAD, J., ZHENG, F., LIU, Q., KLASKY, S., OLDFIELD, R., KORDENBROCK, T., SCHWAN, K., AND WOLF, M. Managing variability in the I/O performance of petascale storage systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).
- [56] LUU, H., BEHZAD, B., AYDT, R., AND WINSLETT, M. A multi-level approach for understanding I/O activity in HPC applications. In *IEEE International Conference on Cluster Computing (CLUSTER)* (2013).
- [57] LUU, H., WINSLETT, M., GROPP, W., ROSS, R., CARNS, P., HARMS, K., PRABHAT, M., BYNA, S., AND YAO, Y. A multi-platform study of I/O behavior on petascale supercomputers. In *International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2015).
- [58] MUELLER, F., WU, X., SCHULZ, M., DE SUPINSKI, B. R., AND GAMBLIN, T. ScalaTrace: tracing, analysis and modeling of HPC codes at scale. In *International Workshop on Applied Parallel Computing* (2010).
- [59] NATHAN, V., NARAYANA, S., SIVARAMAN, A., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Demonstration of the marble system for network performance monitoring. In *Proceedings of the SIGCOMM Posters and Demos* (2017).
- [60] NEUWIRTH, S., WANG, F., ORAL, S., VAZHAKUDAI, S., ROGERS, J., AND BRUENING, U. Using balanced data placement to address I/O contention in production environments. In *International Symposium on Computer Architecture and High PERFORMANCE Computing (SBAC-PAD)* (2016).
- [61] NOETH, M., RATN, P., MUELLER, F., SCHULZ, M., AND DE SUPINSKI, B. R. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing* (2009).
- [62] ORAL, S., SIMMONS, J., HILL, J., LEVERMAN, D., WANG, F., EZELL, M., MILLER, R., FULLER, D., GUNASEKARAN, R., KIM, Y., GUPTA, S., VAZHAKUDAI, D. T. S. S., ROGERS, J. H., DILLOW, D., SHIPMAN, G. M., AND BLAND, A. S. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2014).
- [63] OUYANG, J., KOCOLOSKI, B., LANGE, J. R., AND PEDRETTI, K. Achieving performance isolation with lightweight co-kernels. In *International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2015).
- [64] PAUL, A. K., GOYAL, A., WANG, F., ORAL, S., BUTT, A. R., BRIM, M. J., AND SRINIVASA, S. B. I/O load balancing for big data HPC applications. In *IEEE International Conference on Big Data (Big Data)* (2017).
- [65] SAKAI, K., SUMIMOTO, S., AND KUROKAWA, M. High-performance and highly reliable file system for the K computer. *Fujitsu Scientific & Technical Journal* (2012).
- [66] SCHMUCK, F. B., AND HASKIN, R. L. Gpf: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST)* (2002).
- [67] SERGENT, N., DÉFAGO, X., AND SCHIPER, A. Impact of a failure detection mechanism on the performance of consensus. In *Proceedings 2001 Pacific Rim International Symposium on Dependable Computing* (2001).
- [68] SHEN, S.-H., AND AKELLA, A. Decor: A distributed coordinated resource monitoring system. In *IEEE International Workshop on Quality of Service (IWQoS)* (2012).
- [69] SKAMAROCK, W. C., KLEMP, J. B., DUDHIA, J., GILL, D. O., BARKER, D. M., WANG, W., AND POWERS, J. G. A description of the advanced research wrf version 2. *National Center For Atmospheric Research Boulder Co Mesoscale and Microscale Meteorology Div* (2005).
- [70] SNYDER, S., CARNS, P., HARMS, K., ROSS, R., LOCKWOOD, G. K., AND WRIGHT, N. J. Modular HPC I/O characterization with Darshan. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools* (2016).

- [71] SONG, H., YIN, Y., SUN, X. H., THAKUR, R., AND LANG, S. Server-side I/O coordination for parallel file systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).
- [72] TAI, A. T., TSO, K. S., AND SANDERS, W. H. Cluster-based failure detection service for large-scale ad hoc wireless network applications. In *International Conference on Dependable Systems and Networks (DSN)* (2004).
- [73] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2018).
- [74] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *10th USENIX Conference on File and Storage Technologies (FAST)* (2012).
- [75] Top 500 list. <https://www.top500.org/resources/top-systems/>.
- [76] USELTON, A., HOWISON, M., WRIGHT, N., SKINNER, D., KEEN, N., SHALF, J., KARAVANIC, K., AND OLICKER, L. Parallel I/O performance: From events to ensembles. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS)* (2010).
- [77] VIJAYAKUMAR, K., MUELLER, F., MA, X., AND ROTH, P. C. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW)* (2009).
- [78] VISHWANATH, V., HERELD, M., ISKRA, K., KIMPE, D., MOROZOV, V., PAPKA, M. E., ROSS, R., AND YOSHII, K. Accelerating I/O forwarding in ibm blue gene/p systems. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).
- [79] WANG, Y., LIU, J., QIN, H., YU, Z., AND YAO, Y. The accurate particle tracer code. *Computer Physics Communications* (2017).
- [80] WIEDEMANN, M. C., KUNKEL, J. M., ZIMMER, M., LUDWIG, T., RESCH, M., BÖNISCH, T., WANG, X., CHUT, A., AGUILERA, A., NAGEL, W. E., KLUGE, M., AND MICKLER, H. Towards I/O analysis of HPC systems and a generic architecture to collect access patterns. *Computer Science-Research and Development* (2013).
- [81] WRIGHT, S. A., HAMMOND, S. D., PENNYCOOK, S. J., BIRD, R. F., HERDMAN, J., MILLER, I., VADGAMA, A., BHALLERAO, A., AND JARVIS, S. A. Parallel file system analysis through application I/O tracing. *The Computer Journal* (2013).
- [82] WU, X., AND MUELLER, F. Elastic and scalable tracing and accurate replay of non-deterministic events. In *International Conference on Supercomputing (ICS)* (2013).
- [83] WU, X., VIJAYAKUMAR, K., MUELLER, F., MA, X., AND ROTH, P. C. Probabilistic communication and I/O tracing with deterministic replay at scale. In *International Conference on Parallel Processing (ICPP)* (2011).
- [84] XIE, B., CHASE, J., DILLOW, D., DROKIN, O., KLASKY, S., ORAL, S., AND PODHORSZKI, N. Characterizing output bottlenecks in a supercomputer. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2012).
- [85] XU, C., BYNA, S., VENKATESAN, V., SISNEROS, R., KULKARNI, O., CHAARAWI, M., AND CHADALAVADA, K. LIOProf: Exposing Lustre file system behavior for I/O middleware. In *Proceedings of Cray User Group Conference (CUG)* (2016).
- [86] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).
- [87] XU, W., LU, Y., LI, Q., ZHOU, E., SONG, Z., DONG, Y., ZHANG, W., WEI, D., ZHANG, X., CHEN, H., XING, J., AND YUAN, Y. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. *Frontiers of Computer Science* (2014).
- [88] YILDIZ, O., DORIER, M., IBRAHIM, S., ROSS, R., AND ANTONIU, G. On the root causes of cross-application I/O interference in HPC storage systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016).
- [89] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2011).
- [90] YU, W., VETTER, J. S., AND ORAL, H. S. Performance characterization and optimization of parallel I/O on the Cray XT. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2008).
- [91] ZIMMER, CHRISTOPHER J. GUIDE: A scalable information directory service to collect, federate, and analyze logs for operational insights into a leadership HPC facility. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2017).

Appendix A Evaluation of Beacon Data Compression

App.	1st-pass	2nd-pass (lossless)	2nd-pass (lossy)
APT	5.4	2.1	2.3
WRF	14.2	3.8	5.5
DNDC	10.1	3.4	5.3
XCFD	12.2	3.8	6.2
GKUA	34.6	3.6	5.1
CAM	9.2	4.4	5.4
AWP	15.1	3.2	11.3
Shentu	22.2	2.6	5.7

Table 8: Compression ratio of sample applications

Table 8 summarizes the effectiveness of Beacon’s monitoring data compression. It gives the compression ratio under three kinds of methods of 8 applications, 5 of which are Shentu, LAMMPS, DNDC, WRF and AWP, discussed in more details previously. The other three are APT [79] (particle dynamics simulation), plus GKUA and XCFD (both closed-source computational fluid dynamics simulators).

We report the compression ratio of the 1st-pass compression (intra-node compression during monitoring data collection) and 2nd-pass compression (inter-node compression during offline log processing on dedicated Beacon server). We experimented with two compression techniques for the latter, one lossless and one lossy (with reduced data precision in file descriptor and offset).

Results in Table 8 indicate significant data size reduction by the 1st-pass compression, with a factor of 5.4 to 34.6 right at the source of monitoring. The second pass, on the other hand, achieves less impressive reduction, partly due to that data have already undergone one pass of compression. Here, though the compute nodes are performing similar I/O operations, different values in parameters such as file offset make it harder to combine data entries. In particular, lossy compression may bring an additional $2.3 \times$ - $11.3 \times$ after 1st-pass compression improvement in compression ratio, however trading the capability of performing certain analysis tasks. Considering our dedicated Beacon server’s storage capacity (120 TB) and Beacon’s data collection rate (10 TB in 18 months), we elect to use a lossless algorithm for our 2nd-pass compression.

Appendix B Anomaly Detection

Beacon performs two types of automatic anomaly detection, to identify *job I/O performance anomaly* and *node anomaly*, respectively.

Beacon detects job I/O performance anomaly by checking newly measured I/O performance results against historical records, based on the assumption that most data-intensive applications have rather consistent I/O behavior. First, it adopts the automatic I/O phase identification technique as in the IOSI system [50] developed on the Oak Ridge Na-

tional Laboratory Titan supercomputer, which uses Discrete Wavelet Transform (DWT) to find distinct “I/O bursts” from continuous I/O bandwidth time-series data. It then deploys DBSCAN algorithm [3], also used in IOSI, to check whether I/O phases from the new job execution conform to known clusters of the same application’s past executions at the same scale. More specifically, it performs 2-D clustering in terms of the I/O phases’ time duration and total I/O volume. When outliers are found, Beacon further utilizes its rich monitoring data to examine neighbor jobs that share forwarding node(s) with the job in question. In particular, it determines whether such neighbors have interference-prone features, such as high MDOPS, high I/O bandwidth, high IOPS, or N-1 I/O mode. Such findings are saved in the Beacon database and provided to users via the Beacon web-based application I/O query tool. Applications of course will need to accumulate at least several executions for such detection to take effect.

Beacon’s node anomaly detection relies on the execution of large-scale jobs (those using 1024 or more compute nodes in our current implementation), where it leverages the common homogeneity in I/O behavior across compute and server nodes to spot outliers. Its multi-level monitoring allows the correlation of I/O activities or loads back to actual client side issued requests. Again by using clustering algorithms like DBSCAN and configurable thresholds, Beacon performs outlier detection across forwarding nodes and OSTs involved in a single job, where the vast majority of entities report highly similar performance while a few members produce contrasting readings. Figure 8 in Section 4.2 gives an example of per-OST bandwidth data within the same execution.

Zeno: Diagnosing Performance Problems with Temporal Provenance

Yang Wu
Facebook

Ang Chen
Rice University

Linh Thi Xuan Phan
University of Pennsylvania

Abstract

When diagnosing a problem in a distributed system, it is sometimes necessary to explain the *timing* of an event – for instance, why a response has been delayed, or why the network latency is high. Existing tools offer some support for this, typically by tracing the problem to a bottleneck or to an overloaded server. However, locating the bottleneck is merely the first step: the real problem may be some other service that is sending traffic over the bottleneck link, or a machine that is overloading the server with requests. These off-path causes do not appear in a conventional trace and will thus be missed by most existing diagnostic tools.

In this paper, we introduce a new concept we call *temporal provenance* that can help with diagnosing timing-related problems. Temporal provenance is inspired by earlier work on provenance-based network debugging; however, in addition to the functional problems that can already be handled with classical provenance, it can also diagnose problems that are related to timing. We present an algorithm for generating temporal provenance and an experimental debugger called Zeno; our experimental evaluation shows that Zeno can successfully diagnose several realistic performance bugs.

1 Introduction

Debugging networked systems is already difficult for functional problems, such as requests that are processed incorrectly, and this has given rise to a rich literature on sophisticated debugging tools. Diagnosing timing-related problems, such as requests that incur a high delay, adds another layer of complexity: delays are often nondeterministic and can arise from subtle interactions between different components.

Performance debugging has already been explored in prior work. For instance, distributed tracing systems [55, 48, 58, 21, 33, 28, 37, 34, 64, 18, 44] can record and analyze executions of a request. These systems offer operators a lot of help with debugging performance problems; for instance, Dapper [55] produces a “trace tree” – a directed graph whose vertices represent execution stages and whose edges represent causal relationships. If the operator observes that a request is taking unusually long, she can inspect its trace tree and look for bottlenecks, such as the RPCs to an overloaded server. Similarly, network provenance systems [69, 67, 61, 30, 60], such as DTaP [68], can be used to generate a causal explanation of an observed event, and the operator can then inspect this explanation for possible bottlenecks.

However, in practice, locating a bottleneck is only the first step. The operator must then find the *causes* of the bottleneck

in order to fix the problem. Existing tools offer far less help with this step. For instance, suppose a misconfigured machine is sending a large number of RPCs to a storage backend, which becomes overloaded and delays requests from other clients. When the operator receives complaints from one of the clients about the delayed requests, she can inspect the trace tree or the provenance and identify the bottleneck (in this case, the storage backend). However, neither of these data structures explains *why* the bottleneck exists – in fact, the actual cause (in this case, the misconfigured machine) would not even appear in either of them!

The reason why existing approaches fall short in this scenario is that they focus exclusively on *functional causality* – they explain why a given computation had some particular result. This kind of explanation looks only at the direct inputs of the computation: for instance, if we want to explain the *existence* of a cup of coffee, we can focus on the source of the coffee beans, the source of the cup, and the barista’s actions. In contrast, *temporal causality* may also involve other, seemingly unrelated computations: for instance, the reason *why it took too long* to get the cup of coffee might be the many customers that were waiting in front of us, which in turn might be the result of a traffic jam elsewhere that caused an unusually large number of customers to pass by the local store. At the same time, some functional dependencies may turn out to be irrelevant when explaining delays: for instance, even though the coffee beans were needed to make the coffee, they may not have contributed to the delay because they were already available in the store.

The above example illustrates that reasoning about temporal causality is very different from reasoning about functional causality. This is not a superficial difference: as we will show, temporal causality requires additional information (about event ordering) that existing tracing systems do not capture. Thus, although systems like Dapper or DTaP do record timestamps and thus may appear to be capable of reasoning about time, they are in fact limited to functional causality and use the timestamps merely as an annotation.

In this paper, we propose a way to reason about temporal causality, and we show how it can be integrated with an existing diagnostic technique – specifically, network provenance. The result is a technique we call *temporal provenance* that can reason about *both* functional and temporal causality. We present a concrete algorithm that can generate temporal provenance for distributed systems, and we describe Zeno, a prototype debugger that implements this algorithm. We have applied Zeno to seven scenarios with high delay that are based on real incident reports from Google Cloud En-

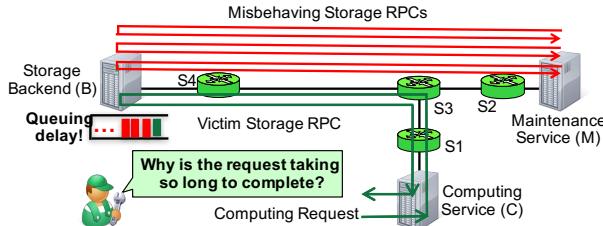


Figure 1: Scenario: The misconfigured maintenance service is overloading the storage backend and is causing requests from the computing service to be delayed.

gine. Our evaluation shows that, in each case, the resulting temporal provenance clearly identifies the cause of the delay. We also show that the runtime overhead is comparable to that of existing tools, such as Zipkin [1], which is based on Google Dapper [55]. In summary, our contributions are:

- The concept of temporal provenance (Section 2);
- an algorithm that generates temporal provenance (Section 4);
- a post-processing technique that improves the readability of timing provenance graphs (Section 5);
- Zeno, a prototype debugger that records and displays temporal provenance (Section 6); and
- an experimental evaluation of Zeno (Section 7).

In the following two sections, we begin with an overview of timing diagnostics and its key challenges.

2 Overview

Figure 1 illustrates the example scenario we have already sketched above. In this scenario, an operator manages a small network that connects a maintenance service M, a computing service C, and a storage backend B. Both M and C communicate with the backend using RPCs. A job on M is misconfigured and is sending an excessive number of RPCs (red) to the storage backend. This is causing queuing at the backend, which is delaying RPCs from the computing service (green). The operator notices the delays on C, but is unaware of the misconfiguration on M.

We refer to this situation as a *timing fault*: the RPCs from C are being handled correctly, but not as quickly as the operator expects. A particularly challenging aspect of this scenario is that the cause of the delays that C's requests are experiencing (the misconfiguration on M) is not on the path from C to B; we call this an *off-path cause*.

Timing faults are quite common in practice. To illustrate this, we surveyed incidents disclosed by Google Cloud Platform [2], which occur across a variety of different cloud services and directly impact cloud tenants. To obtain a good sample size, we examined all incidents that happened from

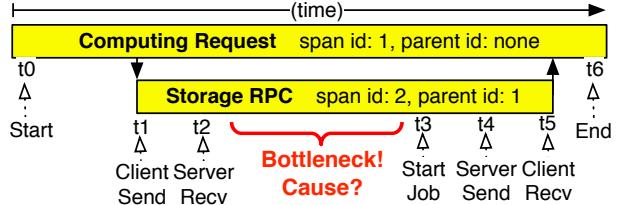


Figure 2: A trace tree for the slow computing requests in Figure 1. B received the storage RPC at t_2 but only started processing it at t_3 , after a long queuing delay.

January 2014 until May 2016, and we selected all 95 incident reports that describe both the symptoms and the causes. We found that more than a third (34.7%) of these incidents were timing faults.

2.1 Prior work: Trace trees

Today, a common way to diagnose such a situation is to track the execution of the request and to identify the bottleneck – that is, components that are contributing unusual delays. For instance, a distributed tracing system would produce a “trace tree” [55]. Figure 2 shows an example tree for one of the delayed responses from the computing service C in Figure 1. The yellow bars represent basic units of work, which are usually referred to as *spans*, and the up/down arrows indicate causal relationships between a span and its parent span. A span is also associated with a simple log of timestamped records that encode events within the span.

Trace trees are helpful because they show the steps that were involved in executing the request: the computation was started at t_0 and issued an RPC to the storage backend at t_1 ; the backend received the RPC at t_2 , started processing it at t_3 , and sent a response at t_4 , which the client received at t_5 ; finally, the computation ended at t_6 . This data structure helps the operator to find abnormal delays: for instance, the operator will notice that the RPC waited unusually long ($t_2 \dots t_3$) before it was processed by B.

However, the operator also must understand what *caused* the unusual delay, and trace trees offer far less help with this step. In our scenario, the cause – the misbehaving maintenance service – never even appears in any span! The reason is that *trace trees include only the spans that are on the execution path* of the request that is being examined. In practice, off-path causes are very common: when we further investigated the 33 timing faults in our survey from above, we found that, in over 60% of the cases, the real problem was not on the execution path of the original request, so it would not have appeared in the corresponding trace tree.

2.2 Prior work: Provenance

Another approach that has been explored recently [61, 30, 60, 68, 69, 67] is to use *provenance* [26] as a diagnostic tool. Provenance is a way to obtain causal explanations of

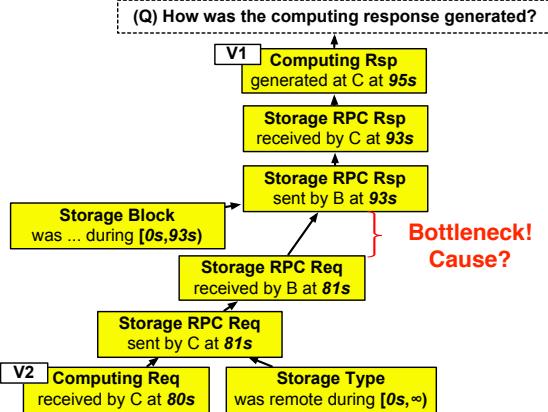


Figure 3: Time-aware provenance, as in DTaP [68], for the example scenario from Figure 1.

an event; a provenance system maintains, for each (recent) event in the network, a bit of metadata that keeps track of the event’s direct causes. Thus, when the operator requests an explanation for some event of interest (say, the arrival of a packet), the system can produce a recursive explanation that links the event to a set of causes (such as the original transmission of the packet and the relevant routing state). Such a representation is useful because the diagnostician often finds herself swimming in a sea of possibilities: at any given moment, there are millions of events happening in the data center, and many of them *could* hypothetically be related to the observed symptom. Moreover, a substantial fraction of these events tend to be unusual in some way or another, which is why the use of anomaly detection often yields many false positives. In contrast, provenance is a way to quickly and reliably identify the (few) events that actually *were* causally related, which can be an enormous time saver.

Provenance can be represented as a DAG, whose vertices represent events and whose edges represent direct causality. Figure 3 shows the DAG that a provenance system like DTaP [68] would generate for our example scenario. (We picked DTaP because it proposed a “time-aware” variant of provenance, which already considers a notion of time.) This data structure is considerably more detailed than a trace tree; for instance, it not only shows the path from the original request (V2) to the final response (V1), but also the data and the configuration state that were involved in processing the request along the way. However, the actual cause from the scenario (the misconfigured maintenance service) is *still* absent from the data structure. The reason is that DTaP’s provenance is “time-aware” only in the sense that it can remember the provenance of past system states. It does annotate each event with a timestamp, as shown in the figure, but it does not reason about temporal causality. Thus, it actually does not offer very much extra help compared to trace trees: like the latter, it can be used to *find* bottlenecks, such as the high response times in the backend, but it is not able to *explain*

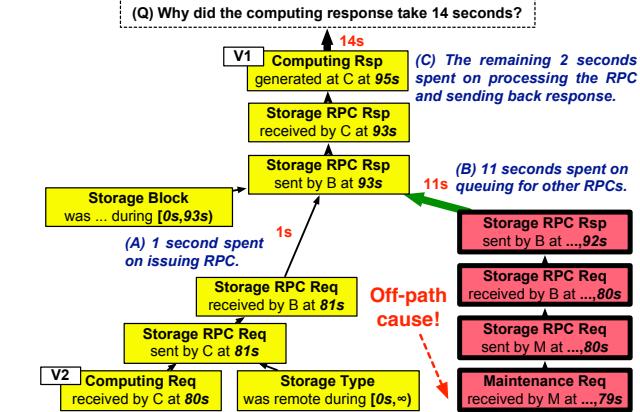


Figure 4: Temporal provenance, as proposed in this paper, for the example scenario from Figure 1.

them by identifying causally related events, such as the RPCs from the maintenance service. Tracking such dependencies between functional and temporal behavior, potentially across several components, is the problem we focus on.

2.3 Our approach

We propose to solve this problem with a combination of three insights. The first is that temporal causality critically depends on a type of information that existing tracing techniques tend not to capture: the *sequence* in which the system has processed requests, *whether the requests are related or not*. By looking only at functional dependencies, these techniques simply consider each request in isolation, and thus cannot make the connection between the slow storage RPC and the requests from the maintenance service that are delaying it. With provenance, we can fix this by including a second kind of edge $e_1 \rightarrow e_2$ that connects each event e_1 to the event e_2 that was processed on the same node and immediately after e_1 . We refer to such an edge as a *sequencing edge* (Section 4.1). Notice that these edges essentially capture the well-known happens-before relation [45].

Our next insight is a connection between temporal reasoning and the critical path analysis from scheduling theory. When scheduling a set of parallel tasks with dependencies, the critical path is the longest dependency chain, and it determines the overall completion time. This concept is not directly applicable to off-path causes, but we have found a way to generalize it (Section 4.3). The result is a method that recursively allocates delay to the branches of a provenance tree, which yields a data structure that we call *temporal provenance*.

Our third insight has to do with readability. At first glance, temporal provenance is considerably richer than classical provenance because it considers not only functionally related events, but also events that could have contributed only delay (of which there can be many). However, in practice, many of these events do not actually contribute to the end-to-end de-

lay, and the ones that do are often structurally similar – such as the maintenance requests in our example scenario – and can be aggregated. Thus, it is usually possible to extract a compact representation that can be easily understood by the human operator (Section 5).

Figure 4 shows the temporal provenance for a random computing request in our example scenario. Starting at the root, the provenance forks into two branches; the left branch (A) shows that one second was spent on issuing the RPC itself; and the right branch (B) shows that the majority of the delay (11 seconds) was caused by RPCs from the maintenance service (M). This tree has all the properties we motivated earlier: it provides a quantitative explanation of the delay, and it includes the actual cause (the maintenance service), even though it is an off-path cause and does not appear on the path the request has taken.

3 Background

Since temporal provenance is a generalization of network provenance, we begin with a brief description of the latter, and refer interested readers to [67] for more detail.

3.1 Network Datalog

For ease of exposition, we will present our approach in the context of *network datalog (NDlog)* [47]. The approach itself is not specific to either NDlog or to declarative languages; provenance has been applied to imperative systems that were written in a variety of languages [49, 22, 38, 35, 39, 60], and our own evaluation in Section 7 includes experiments with Google Dapper [55]. However, NDlog’s declarative syntax makes the provenance very easy to see.

In NDlog, the state of a node is modeled as *tables*, each of which contains a number of *tuples*. For example, an RPC server might have a table called `RPC` that contains the RPC calls it has received from clients. Tuples can be manually inserted, or they can be programmatically derived from other tuples. The former represent external inputs and are called *base tuples*, whereas the latter represent computations in the system itself and are called *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from one another. For instance, the rule $A(@L, P) : -B(@L, Q), Q=3*P$ says that a tuple $A(@L, P)$ should be derived on node L whenever a $B(@L, Q)$ tuple is also on that node, and $Q=3*P$. Here, P and Q are variables that are instantiated with values when the rule is applied; for instance, a $B(@L, 6)$ tuple would create an $A(@L, 2)$ tuple. The $@$ operator specifies the location of the tuple.

Note that, in this declarative formulation, the direct causes of a tuple’s existence are simply the preconditions of the rule that was used to derive it. For instance, if $A(@L, 2)$ was derived using the rule above, then the direct causes were the existence of $B(@L, 6)$ and the fact that $6=3*2$.

3.2 System model

If our goal was classical data provenance, the declarative description above would already be sufficient. However, since we are particularly interested in *timing*, we need to consider some more details of how the system works. For concreteness, we use an event-driven model: the system reacts to events such as packet arrivals and configuration changes; each node has a queue of events that it processes in a FIFO order; and each event can trigger one or more additional events, either locally or on another node. (Note that the “nodes” here do not necessarily have to correspond to physical machines; they could be different CPU cores, or line cards in a switch.) This model captures how pipelined semi-naïve evaluation [47] works in NDlog: the events are tuple insertions and deletions, and the processing corresponds to tuple derivations. However, more importantly, it is also a good description of networks and services with FIFO queues.

3.3 Classical provenance

In order to be able to answer provenance queries, a system must collect some additional metadata at runtime. Conceptually, this can be done by maintaining a large DAG, the *provenance graph*, that contains a vertex for every event that has occurred in the system, and in which there is an edge (a, b) between two vertices if event a was a direct cause of event b . (A practical implementation would typically not maintain this graph explicitly, but instead collect only enough information to reconstruct a recent subgraph when necessary; however, we will use this concept for now because it is easier to explain.) If the system later receives a provenance query $\text{QUERY}(e)$ for some event e , it can find the answer by locating the vertex that corresponds to e and then projecting out the subgraph that is rooted at e . This subgraph will be the *provenance of e* .

For concreteness, we will use a provenance graph with six types of vertices, which is loosely based on [68]:

- $\text{INS}([t_s, t_e], N, \tau)$, $\text{DEL}([t_s, t_e], N, \tau)$: Base tuple τ was inserted (deleted) on node N during $[t_s, t_e]$;
- $\text{DRV}([t_s, t_e], N, \tau)$, $\text{UDRV}([t_s, t_e], N, \tau)$: Derived tuple τ acquired (lost) support on N during $[t_s, t_e]$;
- $\text{SND}([t_s, t_e], N \rightarrow N', \pm\tau)$, $\text{RCV}([t_s, t_e], N \leftarrow N', \pm\tau)$: $\pm\tau$ was sent (received) by N to (from) N' during $[t_s, t_e]$.

Note that each vertex is annotated with the node on which it occurred, as well as with a time interval that indicates when the node processed that event. For instance, when a switch makes a forwarding decision for a packet, it derives a new tuple that specifies the next hop, and the time $[t_s, t_e]$ that was spent on this decision is indicated in the corresponding DRV vertex. This will be useful (but not yet sufficient) for temporal provenance later on.

The edges between the vertices represent their causal relationships. A SND vertex has an edge from an INS or a DRV that produced the tuple that is being sent; a RCV has an edge from the SND vertex for the received message; and a DRV vertex for a rule $A : -B, C, D$ has an edge from each precondition (B, C, and D) that leads to the vertex that produced the corresponding tuple. An INS vertex corresponds to an event that cannot be explained further (the insertion of a base tuple); thus, it has no incoming edges. The edges for the negative “twins” of these vertices – UDRV and DEL – are analogous.

The above definition has two useful properties. First, is recursive: the provenance of an event e includes, as subgraphs, the provenances of all the events that contributed to e . This is useful to an operator because she can start at the root and “drill down” into the explanation until she identifies a root cause. Second, there is a single data structure – the provenance graph – that can be maintained at runtime, without knowing a priori what kinds of queries will be asked later.

4 Temporal provenance

In this section, we generalize the basic provenance model from Section 3 to reason about the timing of events.

4.1 Sequencing edges

The provenance model we have introduced so far would produce provenance that looks like the tree in Figure 3: it would explain why the event at the top occurred, but it would not explain why the event occurred *at that particular time*. The fact that the vertices are annotated with timestamps, as in prior work [68], does not change this: the operator would be able to see, for instance, that the storage service took a long time to respond to a request, but the underlying *reason* (that requests from another node were queued in front of it) is not shown; in fact, it does not even appear in the graph!

To rectify this, we need to capture some additional information – namely, the *sequence* in which events were processed by a given node. Thus, we introduce a second type of edge that we call *sequencing edge*. A sequencing edge (v_1, v_2) exists between two vertices a and b iff either a) the corresponding events happened on the same node, and a was the event that immediately preceded b , or b) a is an SND vertex and b is the corresponding RCV vertex. We refer to the first type of edge as a *local* sequencing edge, and to the second type as a *remote* sequencing edge. In the illustrations, we will render the sequencing edges with green, dotted lines to distinguish them from the causal edges that are already part of classical provenance.

Although causal edges and sequencing edges often coincide, they are in fact orthogonal. For instance, consider the scenario in Figure 5(a). Here, a node X has two rules, $B : -A$ and $C : -A$; in the concrete execution (shown on the timeline),

A is inserted at time 0, which triggers both rules, but B is derived first, and then C. In the provenance graph (shown at the bottom), $\text{INS}(A)$ is connected to $\text{DRV}(B)$ by both a causal and a sequencing edge, since the two events happened back-to-back and B’s derivation was directly caused by A’s insertion. But $\text{DRV}(B)$ is connected to $\text{DRV}(C)$ only by a sequencing edge, since the former did precede the latter but was not a direct cause; in contrast, $\text{INS}(A)$ is connected to $\text{DRV}(C)$ only by a causal edge, since A’s insertion did cause C’s derivation, but the latter was directly delayed by another event.

4.2 Queries

Next, we turn to the question what a query for temporal provenance should look like, and what it should return. Unlike a classical provenance query $\text{QUERY}(e)$, which aims to explain a specific event e , a temporal provenance query aims to explain a *delay* between *a pair of events* e_1 and e_2 . For instance, in the scenario from Figure 1, the operator wanted to know why his request had taken so long to complete, which is, in essence, a question about the delay between the request itself (e_1) and the resulting response (e_2). Hence, we aim to answer queries of the form $\text{T-QUERY}(e_1, e_2)$, which ask about delay between two events e_1 and e_2 . Our only requirement is that the events are causally related – i.e., that there is a causal path from e_1 to e_2 .

As a first approximation, we can answer $\text{T-QUERY}(e_1, e_2)$ as follows. We first query the classical provenance $P := \text{QUERY}(e_2)$. Since we require that e_1 and e_2 are causally related, P will include a vertex for e_1 . We then identify all pairs of vertices (v_1, v_2) in P that are connected by a causal edge but not by a sequencing edge. We note that, a) in each such pair, v_2 must have been delayed by some other intervening event, and b) v_1 is nevertheless connected to v_2 via a multi-hop path along the sequencing edges. (The reason is simply that v_1 was one of v_2 ’s causes and must therefore have happened before it.) Thus, we can augment the causal provenance by adding these sequencing paths, as well as the provenance of any events along such a path. The resulting provenance P' contains all the events that have somehow contributed to the delay between e_1 and e_2 . We can then return P' as the answer to $\text{T-QUERY}(e_1, e_2)$.

4.3 Delay annotations

As defined so far, the temporal provenance still lacks a way for the operator to tell *how much* each subtree has contributed to the overall delay. This is important for usability: the operator should have a way to “drill down” into the graph to look for the most important causes of delay. To facilitate this, we additionally annotate the vertices with the delay that they (and the subtrees below them) have contributed.

Computing these annotations is surprisingly nontrivial and involves some interesting design decisions. Our algorithm is shown in Figure 6; we explain it below in several re-

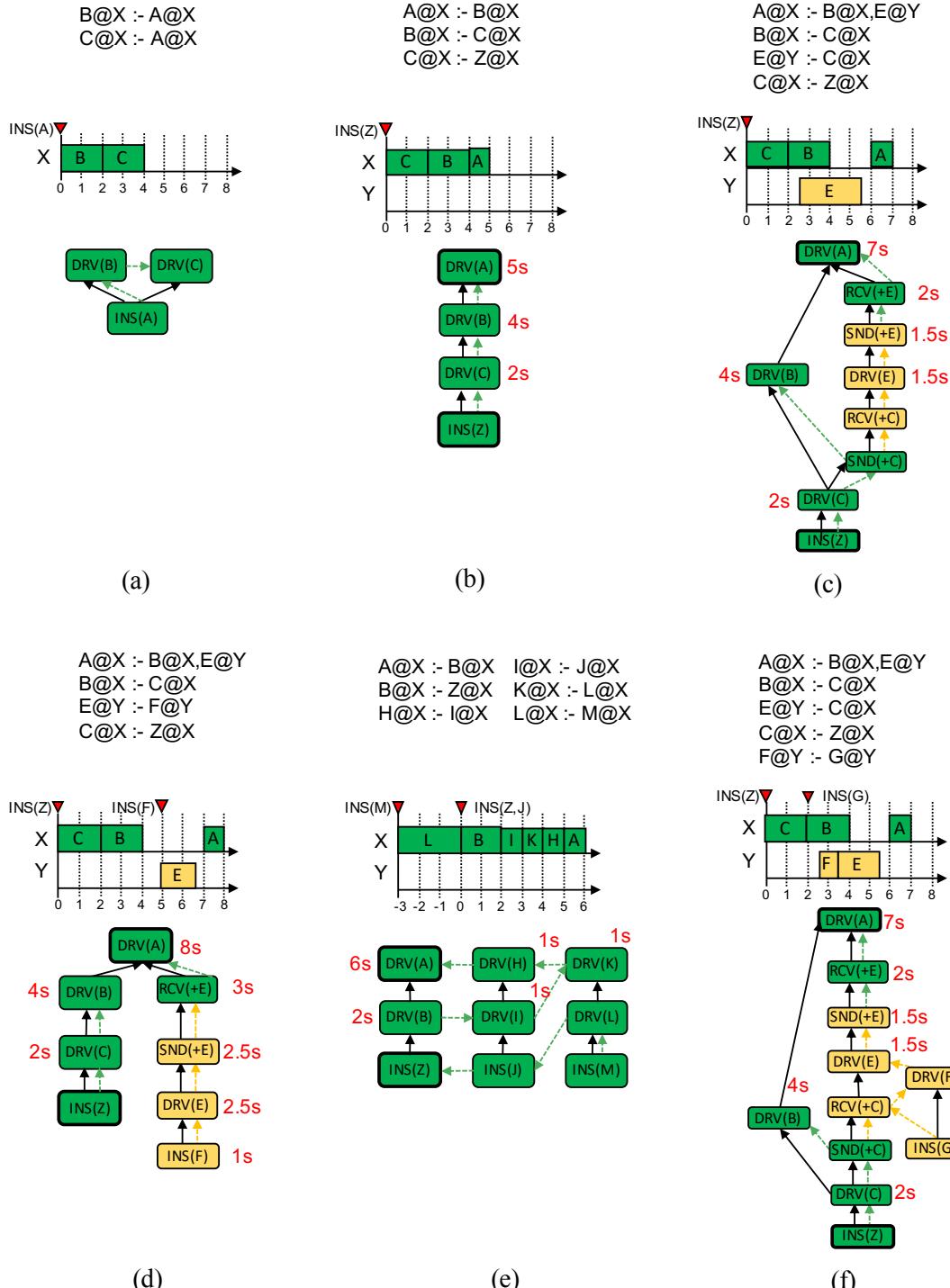


Figure 5: Example scenarios, with NDlog rules at the top, the timing of a concrete execution in the middle, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all scenarios; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity.

```

1: // the subtree rooted at  $v$  is responsible for the delay during  $[t_s, t_e]$ 
2: function ANNOTATE( $v, [t_s, t_e]$ )
3:   ASSERT( $t_e == t_{end}(v)$ )
4:   if  $[t_s, t_e] = \emptyset$  then
5:     RETURN
6:   // weight  $v$  by the amount of delay it contributes
7:   SET-WEIGHT( $v, t_e - t_s$ )
8:   // recursive calls for functional children in order of appearance
9:    $C \leftarrow$  FUNCTIONAL-CHILDREN( $v$ )
10:   $T \leftarrow t_s$ 
11:  while  $C \neq \emptyset$  do
12:     $v' \leftarrow c \in C$  WITH MIN  $t_{end}(c)$ 
13:     $C \leftarrow C \setminus \{v'\}$ 
14:    if  $t_{end}(v') \geq T$  then
15:      ANNOTATE( $v', [T, t_{end}(v')]$ )
16:       $T \leftarrow t_{end}(v')$ 
17:    // recursive calls for sequencing children
18:     $s \leftarrow$  SEQUENCING-CHILD( $v$ )
19:     $E \leftarrow t_{start}(v)$ 
20:    while  $T < E$  do
21:      ANNOTATE( $s, [\text{MAX}(T, t_{start}(s)), E]$ )
22:       $E \leftarrow t_{start}(s)$ 
23:       $s \leftarrow$  SEQUENCING-CHILD( $s$ )

```

Figure 6: Algorithm for computing delay annotations (explained in Sections 4.3–4.5).

finements, using the simple examples in Figures 5(b)–(f). The examples are shown in the same format as in Figure 5(a): each shows a set of simple NDlog rules, the timing of events during the actual execution, and the resulting temporal provenance, with the delay annotations in red. The query is always the same: T-QUERY(INS(Z), DRV(A)); that is, we want to explain the delay between the insertion of Z and the derivation of A. One difference to Figure 5(a) is that some of the examples require two nodes, X and Y. To make the connections more visible, we show the vertices that belong to Y in orange, and the ones that belong to X in green, as in Figure 5(a). If a vertex did not contribute to the delay, we omit its annotation.

Our algorithm computes the delay annotations recursively. A function ANNOTATE is called on the root of the provenance; the function then invokes itself on (some of) the children to compute the annotations on the subgraphs. As a first approximation, this works as follows:

Rule #1: Annotate the top vertex v with the overall delay T , then subtract the execution time t_v of v , and repeat with v 's child vertex, using delay $T - t_v$.

In our algorithm, this corresponds to line 7, which sets the weight for the current vertex, and the recursive call in line 15; lines 4–5 contain the base case, where the delay is zero.

4.4 Handling multiple preconditions

This approach works well for linear provenance, such as the one in Figure 5(b): deriving A from Z took 5s because it took 1s to compute A itself, and 4s to derive A's precondition, B; deriving B from Z took 4s because 2s were spent

on B itself and another 2s on C. However, it does *not* work well for rules with multiple preconditions. Consider the scenario in Figure 5(c): A now has two preconditions, B and E, so the question is how much of the overall delay should be attributed to each.

Two answers immediately suggest themselves: 1) since B completed after 4s, we can attribute 4s to B and the remaining 2s to E, which finished later, or 2) we can attribute the entire 6s to E, because it was the last to finish. The latter is somewhat similar to the choice made in critical path analysis [34, 62]; however, the theorems in Section 4.6 actually require the former: if we find a way to speed up E (or cause F to be inserted sooner), this can only reduce the end-to-end delay by 3s. Any further reductions would have to come from speeding up B. This leads to the following refinement:

Refinement #2: Like rule #1, except that the remaining delay is allocated among the preconditions in the order in which they were satisfied.

This refinement is implemented in lines 11–13 and 16 of our algorithm, which iterate through the preconditions in the order of their appearance (that is, local derivation or arrival from another node) and allocate to each the interval between its own appearance and the appearance of its predecessor.

Notice that this approach deviates from critical-path analysis in an interesting way. Consider the scenario in Figure 5(d): here, the provenance has two “branches”, one connected to the insertion of Z and the other to the insertion of F, but there is no causal path from Z to F. (We call such a branch an *off-path branch*.) This raises the question whether any delay should be apportioned to off-path branches, and if so, how much. Critical path analysis has no guidance to offer for this case because it only considers tasks that are transitively connected to the start task.

At first glance, it may seem that F's branch should not get any delay at all; for instance, F could be a configuration entry that is causally unrelated to Z and thus did not obviously contribute to a delay from Z to A. However, notice that all the “on-path” derivations (in Z's branch) finished at $t = 4s$, but A's derivation was nevertheless delayed until $t = 7s$ because E was not yet available. Thus, it seems appropriate that the other branch gets the remaining 3s.

4.5 Handling sequencing delays

The one question that remains is what to do if there is further delay after the last precondition is satisfied. This occurs in the scenario in Figure 5(e): although B is derived immediately after Z is inserted at $t = 0$, A's derivation is delayed by another 3s due to some causally unrelated derivations (I, K, and H). Here, the sequencing edges come into play: we can attribute the remaining delay to the predecessor along the *local* sequencing edge (here, DRV(H), which will subtract its own computation time and pass on any remaining delay to its own predecessor, etc. This brings us to the final rule:

Final rule: Like #2, except that, if any delay remains after the last precondition, that delay is attributed to the predecessors along the local sequencing edge.

This is implemented in lines 17–23.

So far, we have focused on the rule for DRV vertices, which is the most complex one. SND vertices are easy because they only have one (causal) child; RCV vertices are even easier because they cannot be delayed by sequencing; and INS vertices are trivial – they have no causal children.

4.6 Correctness

We have formally modeled the properties of temporal provenance, and we have proven that our algorithm achieves them. Due to lack of space, we cannot include the full model or the proofs here (they are available in Appendix A); however, we informally describe the properties below.

The properties we considered fall into two categories. The first category consists of basic properties that provenance is generally expected to have; for instance, the provenance should describe a correct execution of the system (validity), it should respect happens-before relationships (soundness), it should be self-contained and fully explain the relevant event (completeness), and it should only contain events that are actually necessary for the explanation (minimality). We have formalized these properties using existing definitions from [68]. Since these definitions are for provenance in general and do not consider the temporal aspect at all, our proofs basically indicate that we did not “break” anything.

The second category contains the properties of the delay annotations that our algorithm creates. Since this is a key contribution of this paper, we briefly sketch our approach. We carefully define what it means for a derivation $\tau : -c_1, c_2, \dots$ to be directly “delayed” by one of its preconditions, and we then recursively extend this definition to transitive delays (that is, one of the c_i was itself delayed by one of its own preconditions, etc.). Our first theorem (Section A.5) states that each vertex is labeled with the amount of (direct or transitive) delay that is contributed by the subtree that is rooted at that vertex. Our second theorem (Section A.6) essentially says that, if there is a vertex v in a temporal provenance tree that is annotated with T and the sum of the annotations on its children and immediate predecessors is $S < T$, then it is possible to construct another valid (but hypothetical) execution in which v ’s execution time is reduced by $(T - S)$ and in which the derivation finishes $(T - S)$ units of time earlier. This shows that the annotations really do correspond to the “potential for speedup” that we intuitively associate with the concept of delay.

4.7 Limitations

Temporal provenance is not magic: when the real reasons for a delay are complex – e.g., many small but unrelated factors

that simply add up – the temporal provenance will likewise be complex and will not show a single root cause. Even in cases where there really is a single unusual factor that causes a lot of delay, temporal provenance does not always single it out, since it has no notion of what is unusual, or which delays are avoidable; instead, it will simply identify *all* causes, annotate each with the delay it caused, and leave the decision to the operator. (However, it could be combined with an additional root-cause analysis, e.g., the one from [30].) Finally, unlike functional provenance, temporal provenance might experience a “Heisenberg effect” in certain cases – that is, collecting the necessary information could subtly alter the timing of the system and prevent the very bugs from appearing that the operator wishes to diagnose (or trigger new, different ones).

5 Improving readability

As defined above, temporal provenance is already useful for diagnostics because it can explain the reasons for a delay between two events. However, the provenance may not be as succinct as a human operator would prefer due to two reasons. First, the temporal provenance for $[e_1, e_2]$ contains the entire classical provenance of e_2 as a subgraph, even though some of the functional causes did not actually contribute to the delay. Second, sequencing delay is often the result of many similar events that each contribute a relatively small amount of delay. To declutter the graph, we perform two post-processing steps.

5.1 Pruning zero-delay subgraphs

Our first post-processing step hides any vertices that are annotated with zero (or not annotated at all) by the ANNOTATE function. The only exception is that we keep vertices that are connected via a causal path (i.e., a path with only causal edges) to a vertex that is annotated with a positive delay. For instance, in Figure 5, the original INS(z) vertex – the starting point of the interval – would be preserved, even though the insertion itself did not contribute any delay.

To illustrate the effect of this step we consider the example in Figure 5(f), which is almost identical to the one in Figure 5(c), except that an additional, unrelated derivation (F) occurred before the derivation of E. Here, the INS(G) and the DRV(F) would be hidden because they do not contribute to the overall delay.

5.2 Provenance aggregation

Our second post-processing step aggregates structurally similar subgraphs [54]. This helps with cases where there are many events that each contribute only a very small amount of delay. For instance, in our scenario from Figure 1, the delay is caused by a large number of RPCs from the maintenance

service that are queued in front of the RPC. The “raw” temporal provenance contains a subtree for each such RPC. During post-processing, these nearly identical subtrees would be aggregated into a single subtree whose weight is the sum of the weights of the individual trees, as shown in Figure 4.

There are two key challenges with this approach. The first is to decompose the temporal provenance into smaller subgraphs that can potentially be aggregated. At first glance, there are exponentially many decompositions, so the problem seems intractable. However, we observe that (1) aggregation is most likely to be possible for sequencing delays, which are often due to similar kinds of events (network packets, RPCs) that have a similar provenance; and that (2) the corresponding subtrees can easily be identified because they are laterally connected to the functional provenance through a chain of sequencing edges. Thus, we can extract candidates simply by following such lateral sequencing edges and by taking the subgraphs below any vertices we encounter.

The second challenge is deciding whether two subgraphs can be aggregated. As a first approximation, this is a graph isomorphism problem, and since our provenance graphs have a bounded chromatic index (which roughly corresponds to the number of preconditions in the largest rule), the classic algorithms – e.g., [23] – should work well. However, in our case the candidate subgraphs are often similar but rarely identical; thus, we need to define an equivalence relation that controls which vertices are safe to aggregate. We use a simple heuristic that considers two vertices to be similar if they share a tuple name and have been derived on the same node. To aggregate two subgraphs, we start at their root vertices; if the vertices are similar, we merge them, annotate them with the sum of their individual annotations, and recursively attempt to merge pairs of their children. If the vertices are not similar, we stop aggregation at that point and connect the two remaining subgraphs directly to the last vertex that was successfully aggregated.

The aggregation procedure is commutative and associative; thus, rather than attempting aggregation for all pairs of subgraphs, we can simply try to aggregate each new subgraph with all existing aggregates. In our experience, the $O(N^2)$ complexity is not a problem in practice because N is often relatively small and/or most of the subgraphs are similar, so there are very few aggregates.

6 The Zeno debugger

We have built a temporal provenance debugger called Zeno with five components in 23,603 lines of code.

Runtime: To demonstrate that Zeno can work with different languages and platforms, we built three different front-ends. The first is integrated with RapidNet [3] and enables Zeno to generate temporal provenance for NDlog programs. The second is integrated with the Zipkin [1] framework – a cross-language distributed tracing library that is based on Google

Dapper [55] and can run a network of microservices written in Node.js [4] (JavaScript), Pyramid [5] (Python), and WEBrick [6] (Ruby). The third is integrated with Mininet [7], which we use to emulate a network environment with P4 switches [17]. All front-ends share the same back-end for reasoning about temporal provenance. In our evaluation, we use the first and the third front-end for SDN applications, and the second one for native Zipkin services.

Provenance recorder: At runtime, our debugger must record enough metadata to be able to answer provenance queries later on. Previous work [69, 46, 61] has already shown that provenance can be captured at scale; this is typically done either (1) by explicitly recording all events and their direct causes, or (2) by recording only nondeterministic inputs at runtime, and by later replaying the execution with additional instrumentation to extract events and causes if and when a query is actually asked [68]. The Zipkin front-end follows the first approach, because Zipkin already has well-defined interfaces to capture both base events and intermediate events (such as RPC invocations and completions), which yields a complete trace tree for each request. Therefore, Zeno merely adds a post-processing engine that converts the trace trees to functional provenance and that infers most of the sequencing edges from the recorded sequence of events across all trace trees. In addition, Zeno extends the Zipkin runtime with dtrace [27] to capture sequencing edges that cannot be directly inferred (e.g., edges representing lock contention). The NDlog front-end uses the second approach and is based on an existing record+replay engine from [68]. The Mininet platform leverages P4 switches to directly obtain ingress/egress timestamps. (More on this below.) In both approaches, we record timestamps at microsecond-level precision, which should be sufficient in practice [57].

Query processor: The third and final component accepts queries $T\text{-QUERY}(e_1, e_2)$ from the operator, as defined in Section 4.2, and it answers each query by first generating the raw temporal provenance and then applying the post-processing steps. The resulting graph is then displayed to the operator.

Retention policy: To prevent the storage requirements from growing indefinitely, our prototype maintains the provenance data only for a configurable retention time R , and prunes it afterwards. Because of this, the result of a $T\text{-QUERY}(e_1, e_2)$ can be incomplete: for instance, if a particular forwarding decision was made based on a routing table entry that is older than R , the corresponding branch of the provenance tree will be “pruned” at that point, since the entry’s provenance will already have been discarded. However, if e_1 is no older than R , all vertices that would be annotated with a nonzero delay will be included, so, if most queries are about recent events, this is not a big sacrifice to make. If desired, the retention policy could easily be refined or replaced.

P4 integration: Obtaining sequencing edges is not always straightforward, especially at the network switches. Fortunately, we can leverage the in-band network telemetry (INT)

capability [16] in emerging switches [24] to obtain sequencing edges. These switches can stamp into a packet’s header the ingress/egress timestamps at each queue, which can then be used to obtain sequencing edges. If two packets p_1 and p_2 traverse the same queue and their ingress/egress timestamps were t_{i1}/t_{e1} and t_{i2}/t_{e2} , with $t_{i1} < t_{i2} < t_{e1} < t_{e2}$, then we know that p_2 must have been queued after p_1 , and Zeno can add a sequencing edge to the provenance graph. We have implemented an extension in our prototype to approximate this capability; note, however, that modern switches can perform these operations directly in hardware at line speed.

7 Evaluation

In this section, we report results from an experimental evaluation of our debugger. We have designed our experiments to answer four high-level questions: 1) Is temporal provenance useful for debugging realistic timing faults? 2) What is the cost for maintaining temporal provenance? 3) How fast can our debugger process diagnostic queries? And 4) Does temporal provenance scale well with the network size?

We ran our experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM. The OS was Ubuntu 16.04, and the kernel version was 4.10.0. Parts of the Zipkin front-end ran on a MacBook Pro, which has a 4-core 2.40 GHz Intel i5-4258 CPU with 8GB of RAM. The OS was macOS 10.13.2, and the kernel version was 17.3.0.

7.1 Diagnostic scenarios

We reproduced seven representative scenarios that we sampled from incidents reported by Google Cloud Engine [2]:

- **R1, Z1: Misbehaving maintenance task [8].** Clients of the Compute Engine API experienced delays of up to two minutes because a scheduled maintenance task caused queuing within the compute service. This is the scenario from Section 2.
- **R2, Z2: Elevated API latency [9].** A failure caused the URL Fetch API infrastructure to migrate to a remote site. This increased the latency, which in turn caused clients to retry, worsening the latency. Latencies remained high for more than 3 hours.
- **R3: Slow deployments after release [10].** A new release of App Engine caused the underlying pub/sub infrastructure to send an update to each existing instance. This additional load slowed down the delivery of deployment messages; thus, the creation of new instances remained slow for more than an hour.
- **R4: Network traffic changes [11].** Rapid changes in external traffic patterns caused the networking infrastructure to reconfigure itself repeatedly, which created a substantial queue of modifications. Since the network

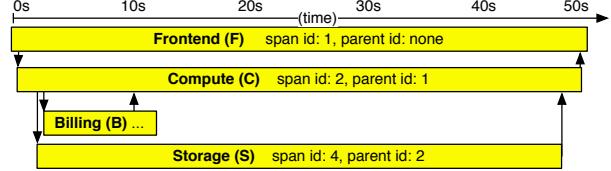


Figure 7: Zipkin trace tree for scenario Z1, which shows that the RPC to the storage service is the bottleneck, but the actual cause (the RPCs from the maintenance service) is off-path and thus is absent.

registration of new instances had to wait on events in this queue, the deployment of new instances was slowed down for 90 minutes.

- **Z3: Lock contention [12].** User-submitted jobs experience increased execution time for over 13 hours because lock contention in an underlying component slowed down query scheduling and execution.
- **Z4: Slow load jobs [13].** Load jobs to an analytics service experienced long latencies for 29 hours. The service received an elevated number of jobs that exceeded its ingestion capacity and caused new jobs to wait increasingly longer to be scheduled.
- **M1: Network congestion [14].** Two cloud services experienced high latency for over 6 hours due to network congestion.

We reproduced four scenarios in RapidNet (R1–R4) and four in the microservice environment (Z1–Z4), including two scenarios in both environments. The microservice scenarios used five to eight servers. (We do not model switches in the microservice scenarios.) We used single-core nodes for Z1 and Z2, but we used up to four cores for Z3 and Z4, to test Zeno’s ability to handle concurrency; in this case, we spread the workload equally across the available cores. The first two RapidNet scenarios use four switches, one controller, and three servers; for the remaining RapidNet scenarios, we used four switches and one controller but a larger number of servers (115 for R3, and 95 for R4). We reproduced the final scenario in Mininet (M1) with 20 P4 switches with 16 hosts organized in a three-tiered Fat-tree topology, where the sequencing edges were obtained using the ingress/egress timestamps exported by the P4 switches [17].

7.2 Identifying off-path causes

A key motivator for this work is the fact that off-path causes for a delay are often not even visible with an existing debugger. To test whether Zeno addresses this, we generated trace trees (using Zipkin) and classic provenance (using DTAP), and compared their ability to identify off-path causes.

Figure 7 shows a Zipkin trace tree for Z1. A human operator can clearly see that the API call to the frontend took 50 seconds, and that the compute RPC and the storage RPC

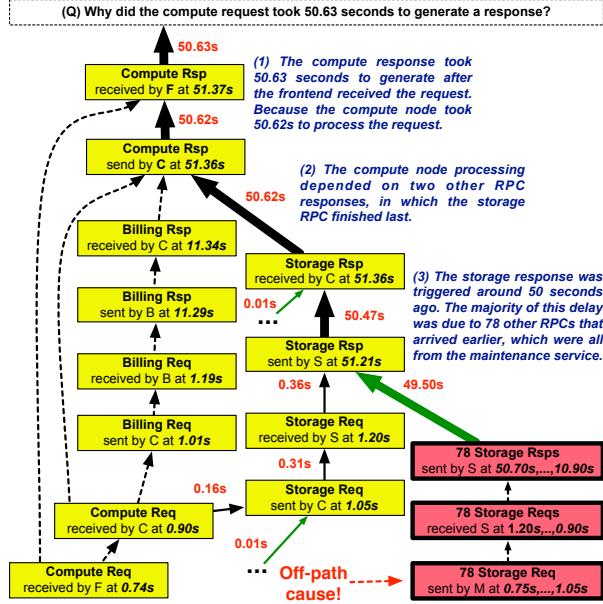


Figure 8: Temporal provenance for scenario Z1. In contrast to the trace tree in Figure 7, the off-path cause (the requests from the maintenance service) does appear and can easily be found by starting at the root and by following the chain of vertices with the highest delay.

both took almost as long. The latter may seem suspicious, but the trace tree contains no further explanation. Similarly, the classic provenance tree for Z1, which are essentially the yellow vertices in Figure 8, offers a more comprehensive explanation compared to the trace tree; however, like the trace tree, it also misses the actual off-path cause. This consistently happened in all experiments with Zipkin and DTaP: since these systems do not reason about temporal causality, the underlying cause was never included in any of their results. On the other hand, Z1’s temporal provenance (all vertices in Figure 8) not only captures the information from Zipkin or DTaP, but also explains that the requests from the off-path maintenance service are causing the delay.

7.3 Size of the provenance

The provenance has to be simple enough for the operator to make sense of it. Recall that, before showing the provenance to operators, our debugger (1) prunes vertices that do not contribute to the overall delay, and (2) aggregates subgraphs that are structurally similar. To quantify how well these techniques work, and whether they do indeed lead to a readable explanation, we re-ran the diagnostic queries in Section 7.1 with different subsets of these steps disabled, and we measured the size of the corresponding provenance graphs.

Figure 9 shows our results. The leftmost bars show the size of the raw temporal provenance, which ranged from 748 to 2,564 vertices. A graph of this size would be far too complex for most operators to interpret. However, not all of these

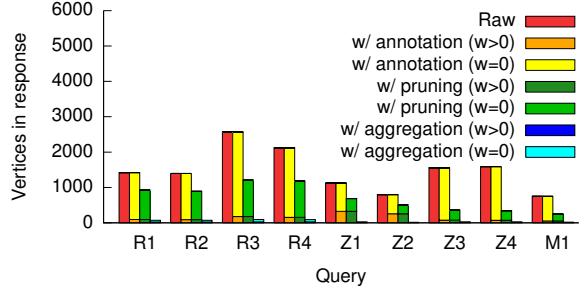


Figure 9: Size of the temporal provenance for all scenarios in Section 5, with different post-processing steps.

vertices actually contribute to the overall delay. The second set of bars shows the number of vertices that Zeno would annotate with a nonzero delay ($w > 0$) and a zero delay ($w = 0$), respectively: only 4.6–32.1% of all vertices actually contributed any delays. However, the subgraphs with nonzero delays nevertheless remain too large to read effectively.

Our first post-processing step prunes vertices and subtrees that are annotated with zero delay and that do not make a causal contribution. As the third set of bars shows, this reduces the size of the graph by more than 30% in all scenarios. The second and final post-processing step coalesces structurally similar subtrees and aggregates their delays. As the rightmost set of bars shows, this is extremely effective and shrinks the graph to between 13 and 93 vertices; the number of vertices that actually contribute delay is between 11 and 28. (Recall that vertices with a causal contribution are preserved even if they do not contribute delay.) A provenance graph of this size should be relatively easy to interpret.

To explain where the large reduction comes from, we sketch the raw provenance tree – without post-processing – for scenario Z1 in Figure 10. The structure of this tree is typical for the ones we have generated. First, there is a relatively small “backbone” (shown in yellow) that functionally explains the result and roughly corresponds to classical provenance. Second, there is a large number of small branches (shown in red) along long sequencing chains (shown in green) that describe the sources of any delay; these are collapsed into a much smaller number of branches, or even a single branch. Third, there are further branches (shown in white) that are connected via sequencing edges but do *not* contribute any delay; these are pruned entirely. The last two categories typically contain the vast majority of vertices, and our post-processing steps shrink them very effectively, which in this case yields the much simpler tree from Figure 8.

7.4 Runtime overhead

Next, we quantified the overhead of collecting the metadata for temporal provenance at runtime. We ran a fixed workload of 1,000 requests in all scenarios, and measured the overall

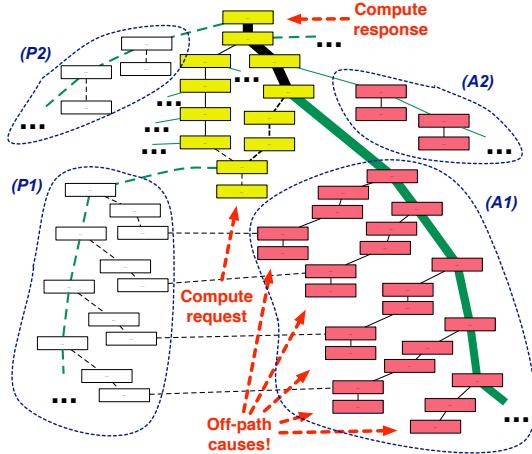


Figure 10: Sketch of the *raw* temporal provenance for scenario Z1. The post-processing steps from Section 5 reduce this to the provenance shown in Figure 8.

latency and the storage needed to maintain provenance. Zipkin is closely based on Dapper, which incurs low overhead in production systems [55]; for instance, instrumenting every request in a web search cluster increased latency by 16.3% and decreased throughput by 1.48% [55]. Temporal provenance mostly uses the data Zipkin collects, but does not modify its collection system; the dtrace [27] extension, which complements Zipkin traces, incurs an additional latency increase of 0.8% and a storage overhead of 270 bytes per *RPC*. In Mininet, each packet consumes 106 bytes, which includes the packet header and timestamps. In RapidNet, maintaining classical provenance alone resulted in a latency increase of 0.3–1.2% and a storage overhead of 145–168 bytes per *input event*. Maintaining temporal provenance causes an additional latency increase of 0.4–1.5% and a storage overhead of 49 bytes per *event*. Notice that, for temporal provenance, it is not enough to merely record input events, since this would not necessarily reproduce the timing or sequence of events.

The total storage consumption also depends on the retention time R . (Recall that Zeno prunes provenance data beyond R .) R needs to be large enough to cover the interval between the root cause and the time the query is issued. Intuitively, a small R should be sufficient because root causes of current issues are usually not in the distant past. To confirm this intuition, and to estimate a suitable value for R , we re-examined our survey of incidents disclosed by Google Cloud Platform [2]. We found 12 timing faults whose descriptions included timestamps for both the symptom and the root cause; in 11 of the 12 cases, the interval between the root cause and the symptom was less than 30 minutes.

7.5 Query processing speed

When the operator queries the temporal provenance, our debugger must execute the algorithm from Section 4 and apply

the post-processing steps from Section 5. Since debugging is an interactive process, a quick response is important. To see whether our debugger can meet this requirement, we measured the turnaround time for all queries, as well as the fraction of time consumed by each of the major components.

Figure 11(a) shows our results. We make two high-level observations. First, for scenarios where provenance is captured using deterministic replay (R1–R4), the turnaround time is dominated by the replay and by the storage lookups that would be needed even for classical provenance. This is expected because neither our annotation algorithm nor the post-processing steps are computationally expensive. Second, although the queries vary in complexity and thus their turnaround times are difficult to compare, we observe that none of them took more than 10 seconds, which should be fast enough for interactive use. Notice that this delay is incurred only once per query; the operator can then explore the resulting temporal provenance without further delays.

7.6 Scalability

To evaluate the scalability of Zeno with regard to the network size, we tested the turnaround time and provenance size of query R3 on larger networks with up to 700 nodes. We obtained these networks by adding more servers.

Turnaround time: As we can see from the left part of Figure 11(b), the turnaround time increased linearly with the network size, but it was within 65 seconds for all cases. As the breakdown shows, the increase mainly comes from the latency increase of the historical lookups and of the replay. This is because the additional nodes and traffic caused the size of the logs to increase. This in turn resulted in a longer time to replay the logs, and to search through the events. Profiling [15] shows that log replay is dominated by sending and receiving packets in RapidNet [3] (Recall from Section 6 that the replay engine is based on an existing one from [68].). Because the replay runs on a single machine, we can optimize turnaround time by reducing unnecessary I/O.

Size of the provenance: The right part of Figure 11(c) shows that the size of the raw provenance grew linearly to the network size – by 7x from 1,939 to 13,960 vertices – because traffic from additional servers caused additional delays, which required extra vertices to be represented in the provenance. With the annotation and aggregation heuristics applied, the number of vertices that actually contributed delay still grew, because more hops were congested due to busier networks. However, the increase – a factor of 1.5, from 28 to 40 – is much less than the increase in the network size (7x), which suggests that the heuristic scales well.

8 Related Work

Provenance: None of the provenance systems we are aware of [61, 30, 60, 68, 69, 67, 49, 22, 38, 35, 39] can reason about temporal causality, which is essential for diagnosing timing

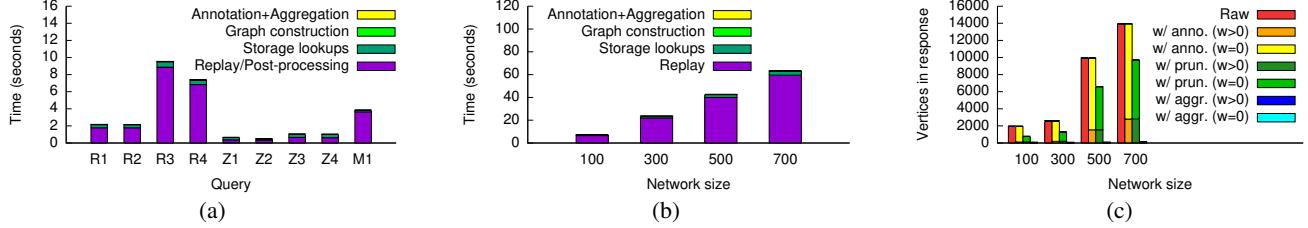


Figure 11: Turnaround for all queries in Section 7.1 (a). Scalability of turnaround (b) and provenance size (c) for R3.

faults. This is true even for DTaP [68] and its predecessor TAP [66], which are “time-aware” only in the very limited sense that they can remember the provenance of past system states. As our experiments confirm, these systems are not able to find off-path causes of timing faults.

Tracing: Tracing systems broadly fall into two classes. The first class of systems *infer* causality, e.g., using log messages [34], static analysis [64, 53], or external annotations [18, 44]; however, the inference is not always accurate, so such systems can have false positives and negatives. For example, Roy et al. [53] localizes network faults in real time by correlating end-host flow metrics with network paths that flows traverse; however, the technique relies on statistical analysis and applies best to huge data centers where the rich structure and massive volume of traffic information reduces imprecision. The second class of systems avoid this imprecision by *recording* causality, at the expense of instrumentation [55, 48, 58, 21, 33, 28, 37, 40, 57, 56]. For example, Canopy [40] annotates traces with performance data; SwitchPointer [57] divides time into epochs and records the epochs during which a packet was forwarded. To our knowledge, our approach is the first to explicitly record temporal *causality* using sequencing edges and thus also the first to offer precise reasoning about the causes of timing behavior.

Performance diagnosis: Existing systems have used machine learning or statistical analysis for performance diagnosis [63, 21, 19, 41, 20, 32] – they perform learning on the normal system behaviors and use learned models for diagnosis. This tends to work well when there is abundant training data, but its power is limited when diagnosing rare anomalies or occasional glitches, which are often the trickiest and most time-consuming problems to debug. Performance diagnosis can also be done by comparing “good” and “bad” instances [51, 52, 50, 54] and analyzing their differences, when both types of instances are available. Since these types of diagnosis do not use causality, the analysis is not always precise. DiffProv [30] does rely on causality, but it focuses exclusively on functional but not temporal causality. We believe that Zeno may be able to benefit from a similar differential diagnosis to narrow down the root causes further.

Timing faults: Our approach is potentially useful for diagnosing timing faults in real-time systems, where tasks have

deadlines [31]. Researchers have proposed solutions to control program timing, but they either require specialized hardware [36] or incur significant overhead [29]. Worst-case execution time analysis [59] can estimate an upper bound on the execution time of a program, but it does not reason about the causes of delays.

Queuing theory: Queuing theory [42, 43, 25] has been used to model, analyze, and optimize the performance of distributed systems. This approach, however, assumes a certain distribution of arrival patterns in the input workloads, which may not always hold in practice, and it does not automatically identify the causes of a performance violation. In contrast, temporal provenance can help diagnosing problems in practical systems without assumptions on the input model.

9 Conclusion

Diagnosing timing-related issues is a tricky business that requires expertise and a considerable amount of time. Hence, it seems useful to develop better tools that can at least partially automate this process. Existing tools work well for functional problems, but they fail to identify the root causes of temporal problems; this requires a very different approach that involves different information and a new way of reasoning about causality. We have proposed temporal provenance as a concrete solution to this problem. Although temporal provenance takes the concept of provenance in a somewhat different direction than the existing work on functional provenance, the two lines of work share the same starting point (classical provenance) and thus look very similar to the operator, which helps with usability. The experimental results from our prototype debugger suggest that temporal provenance can provide compact, readable explanations for temporal behavior, and that the necessary metadata can be collected at a reasonable cost.

Acknowledgments: We thank our shepherd Ariosto Panda and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CNS-1563873, CNS-1703936, CNS-1750158, CNS-1703936 and CNS-1801884, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

References

- [1] <http://zipkin.io/>.
- [2] <https://status.cloud.google.com/summary>.
- [3] <http://netdb.cis.upenn.edu/rapidnet/>.
- [4] <https://nodejs.org/>.
- [5] <https://trypyramid.com/>.
- [6] <https://github.com/ruby/webrick>.
- [7] <http://mininet.org/>.
- [8] <https://status.cloud.google.com/incident/compute/15039>.
- [9] <https://status.cloud.google.com/incident/appengine/14005>.
- [10] <https://status.cloud.google.com/incident/appengine/15005>.
- [11] <https://status.cloud.google.com/incident/compute/15057>.
- [12] <https://status.cloud.google.com/incident/bigquery/18003>.
- [13] <https://status.cloud.google.com/incident/bigquery/18007>.
- [14] <https://status.cloud.google.com/incident/appengine/15023>.
- [15] <https://github.com/gperftools/gperftools>.
- [16] In-band network telemetry. <http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf>.
- [17] The P4 language. <https://github.com/p4lang/>.
- [18] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP* (Oct. 2003).
- [19] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proc. OSDI* (Oct. 2012).
- [20] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. SIGCOMM* (2007).
- [21] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proc. OSDI* (Dec. 2004).
- [22] BATES, A., TIAN, D., BUTLER, K. R., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *Proc. USENIX Security* (Aug. 2015).
- [23] BODLAENDER, H. L. Polynomial algorithms for graph isomorphism and chromatic index on partial k-Trees. *Journal of Algorithms* (1990), 631–643.
- [24] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., McKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR* 44, 3 (2014).
- [25] BOUDEC, J.-Y. L., AND THIRAN, P. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, vol. LNCS 2050. Springer, 2001.
- [26] BUNEMAN, P., KHANNA, S., AND WANG-CHIEW, T. Why and where: A characterization of data provenance. In *Proc. ICDT* (Jan. 2001).
- [27] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proc. USENIX ATC* (2004).
- [28] CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Whodunit: Transactional profiling for multi-tier applications. In *Proc. SOSP* (Oct. 2007).
- [29] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI* (Oct. 2014).
- [30] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The Good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. SIGCOMM* (Aug. 2016).
- [31] CHEN, A., XIAO, H., PHAN, L. T. X., AND HAEBERLEN, A. Fault tolerance and the five-second rule. In *Proc. HotOS* (May 2015).
- [32] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. DSN* (2002).
- [33] CHEN, Y.-Y. M., ACCARDI, A., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. *Path-based failure and evolution management*. PhD thesis, University of California, Berkeley, 2004.
- [34] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The mystery machine: end-to-end performance analysis of large-scale Internet services. In *Proc. OSDI* (Oct. 2014).
- [35] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security* (Aug. 2011).
- [36] EDWARDS, S. A., AND LEE, E. A. The case for the precision timed (PRET) machine. In *Proc. DAC* (June 2007).
- [37] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proc. NSDI* (Apr. 2007).
- [38] GEHANI, A., AND TARIQ, D. Spade: Support for provenance auditing in distributed environments. In *Proc. Middleware* (Dec. 2012).
- [39] HASAN, R., SION, R., AND WINSLETT, M. The case of the fake picasso: Preventing history forgery with secure provenance. In *Proc. FAST* (2009).
- [40] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., ET AL. Canopy: An end-to-end performance tracing and analysis system. In *Proc. SOSP* (2017).
- [41] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *Proc. SIGCOMM* (Aug. 2009).
- [42] KLEINROCK, L. *Queueing Systems, Volume 1: Theory*. Wiley-Interscience, 1975.
- [43] KLEINROCK, L. *Queueing Systems, Volume 2: Computer Applications*. Wiley-Interscience, 1976.
- [44] KOSKINEN, E., AND JANNOTTI, J. Borderpatrol: Isolating events for black-box tracing. In *Proc. EuroSys* (Mar. 2008).
- [45] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (July 1978).
- [46] LOGOTHETIS, D., DE, S., AND YOCUM, K. Scalable lineage capture for debugging DISC analysis. Tech. Rep. CSE2012-0990, UCSD.
- [47] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking. *Communications of the ACM* 52, 11 (Nov. 2009), 87–95.
- [48] MILLER, B. P. Dpm: A measurement system for distributed programs. *IEEE Transactions on Computers* 37, 2 (1988), 243–248.
- [49] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *Proc. USENIX ATC* (May 2006).

- [50] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. NSDI* (Apr. 2012).
- [51] NOVAKOVIĆ, D., VASIĆ, N., NOVAKOVIĆ, S., KOSTIĆ, D., AND BIANCHINI, R. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. USENIX ATC* (June 2013).
- [52] REYNOLDS, P., KILLIAN, C. E., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI* (May 2006).
- [53] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive realtime datacenter fault detection and localization. In *Proc. NSDI* (2017).
- [54] SAMBASIVAN, R. R., ZHENG, A. X., DE ROSA, M., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *Proc. NSDI* (Apr. 2011).
- [55] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPER, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure.
- [56] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *Proc. OSDI* (2016).
- [57] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *Proc. NSDI* (2018).
- [58] TIERNEY, B., JOHNSTON, W., CROWLEY, B., HOO, G., BROOKS, C., AND GUNTER, D. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. HPDC* (July 1998).
- [59] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution-time problem—overview of methods and survey of tools. *ACM TECS* 7, 3 (May 2008), 36:1–36:53.
- [60] WU, Y., CHEN, A., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Automated bug removal for software-defined networks. In *Proc. NSDI* (Mar. 2017).
- [61] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing missing events in distributed systems negative provenance. In *Proc. SIGCOMM* (Aug. 2014).
- [62] YANG, C.-Q., AND MILLER, B. P. Critical path analysis for the execution of parallel and distributed programs. In *Proc. DCS* (1988).
- [63] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. CPI 2: CPU performance isolation for shared compute clusters. In *Proc. EuroSys* (Apr. 2013).
- [64] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. Iprof: A non-intrusive request flow profiler for distributed systems. In *Proc. OSDI* (Oct. 2014).
- [65] ZHOU, W. *Secure Time-Aware Provenance For Distributed Systems*. PhD thesis, University of Pennsylvania, 2012.
- [66] ZHOU, W., DING, L., HAEBERLEN, A., IVES, Z., AND LOO, B. T. TAP: Time-aware provenance for distributed systems. In *Proc. TaPP* (June 2011).
- [67] ZHOU, W., FEI, Q., NARAYAN, A., HAEBERLEN, A., LOO, B. T., AND SHERR, M. Secure network provenance. In *Proc. SOSP* (Oct. 2011).
- [68] ZHOU, W., MAPARA, S., REN, Y., LI, Y., HAEBERLEN, A., IVES, Z., LOO, B. T., AND SHERR, M. Distributed time-aware provenance. In *Proc. VLDB* (Aug. 2013).
- [69] ZHOU, W., SHERR, M., TAO, T., LI, X., LOO, B. T., AND MAO, Y. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD* (June 2010).

A Formal Model

Temporal provenance preserves all properties of classical provenance (validity, soundness, completeness, and minimality). We have obtained the corresponding proofs by extending the formal model from TAP/DTaP [65]. Although there are some parts of the proof from [65] that require few or no changes (e.g., because they only relate to functional provenance and not to sequencing), we present the full formal model here for completeness. Our extensions include the following:

- Temporal provenance has a different set of vertex types (Section 3.3) and contains sequencing edges (Section 4.1); consequently, temporal provenance graphs are constructed differently (Section A.2).
- The validity property, in addition to its prior requirements from TAP, requires that the temporal provenance include all the events necessary to reproduce the execution temporally (Section A.3).
- The proofs follow the same structure as in TAP, but are adjusted to handle the different graph structure and the stronger validity property of temporal provenance (Section A.4).

We have also formally modeled the properties of the delay annotations that our algorithm creates (and that were not part of [65]):

- Definitions of direct delay and transitive delay; and a theorem states that each vertex is labeled with the amount of delay that is contributed by the subtree that is rooted at that vertex (Section A.5).
- A theorem states that the annotations do correspond to the “potential for speedup” that we intuitively associate with the concept of delay (Section A.6).

A.1 Background: Execution Traces

To set stage for the discussion, we introduce some necessary concepts for reasoning about the execution of the system in our temporal provenance model.

An *execution trace* of an NDlog program can be characterized by a sequence of *events* that take place in the system, starting from the initial system state. Each event on a node captures the execution of a particular rule r that is triggered by a certain tuple τ , under the existence of some other tuples on the node, and that results in a new tuple being derived or an existing tuple being underived (i.e., lost). We formally define them below.

Definition (Event): An event $d@n$ on a node n is represented by $d@n = (\tau, r, t_s, t_e, c, \pm\tau')$, where

- τ is the tuple that triggers the event,
- r is the derivation rule that is being triggered,
- t_s is the time at which r is triggered (called start timestamp),
- t_e is the time at which r finishes its execution (called end timestamp),
- c is the set of tuples that are preconditions of the event, which must exist on n at time t_s , and
- τ' is the tuple that is derived (+) or underived (-) as a result of the derivation.

Definition (Trace): A trace \mathcal{E} is a sequence of events $\langle d_1@n_1, d_2@n_2, \dots, d_k@n_k \rangle$ that reflects an execution of the system from the initial state S_0 , i.e.,

$$S_0 \xrightarrow{d_1@n_1} S_1 \xrightarrow{d_2@n_2} \dots \xrightarrow{d_k@n_k} S_k.$$

To quantify the timing behaviors of the system, it is necessary to reason about the order among events. Ideally, we would like to have a total ordering among all events in all nodes in the system; however, due to the lack of fully synchronized clocks, this is difficult to achieve in distributed systems. To address this, we introduce the concept of trace *equivalence* that preserves the total ordering of events on each node, without imposing a total ordering among events across nodes. Intuitively, two traces \mathcal{E} and \mathcal{E}' are considered

equivalent if the subsequence of events that every node observes in \mathcal{E} is the same as that is observed in \mathcal{E}' .

Definition (Subtrace): \mathcal{E}' is a subtrace of \mathcal{E} (written as $\mathcal{E}' \subseteq \mathcal{E}$) iff \mathcal{E}' is a subsequence of \mathcal{E} . We denote by $\mathcal{E}|n$ the subtrace of \mathcal{E} that consists of all and only the events of \mathcal{E} that take place on node n .

Definition (Equivalence): Two traces \mathcal{E} and \mathcal{E}' are equivalent (written as $\mathcal{E} \sim \mathcal{E}'$) iff for all nodes n , $\mathcal{E}|n = \mathcal{E}'|n$.

By definition, the equivalence relation is transitive: if $\mathcal{E} \sim \mathcal{E}'$ and $\mathcal{E}' \sim \mathcal{E}''$, then $\mathcal{E} \sim \mathcal{E}''$.

Example: As an example, consider the following traces:

$$\begin{aligned}\mathcal{E}_1 &= \langle d_1@n_1, d_2@n_2, d_3@n_1, d_4@n_2 \rangle, \\ \mathcal{E}_2 &= \langle d_1@n_1, d_2@n_2, d_4@n_2, d_3@n_1 \rangle, \\ \mathcal{E}_3 &= \langle d_1@n_1, d_2@n_2, d_3@n_1 \rangle.\end{aligned}$$

It is easy to observe that \mathcal{E}_1 and \mathcal{E}_2 are equivalent, since $\mathcal{E}_1|n_1 = \mathcal{E}_2|n_1 = \langle d_1@n_1, d_3@n_1 \rangle$ and $\mathcal{E}_1|n_2 = \mathcal{E}_2|n_2 = \langle d_2@n_2, d_4@n_2 \rangle$. In contrast, \mathcal{E}_3 is a subtrace of \mathcal{E}_1 , but it is not equivalent to \mathcal{E}_1 (since $\mathcal{E}_3|n_2 \neq \mathcal{E}_1|n_2$).

A.2 Graph construction

We now describe our algorithm for constructing the temporal provenance that explains the reasons for a delay between two events. As discussed in Section 4.6, temporal provenance is *recursive* – the temporal provenance for $[e', e]$ includes, as subgraphs, the temporal provenances of all events that contributed to both e and the delay from e' to e . Leveraging this property, we can construct the temporal provenance of a pair of events “on demand” using a top-down procedure, without the need to materialize the entire provenance graph.

Towards this, we first define a function RAW-QUERY that, when called on a vertex v in the temporal provenance graph, returns two sets of immediate children of v : the first consists of vertices that are connected to v via causal edges, and the second consists of vertices that are connected to v via sequencing edges. Given an execution trace \mathcal{E} of the system, the temporal provenance for a diagnostic query T-QUERY(e', e) can be obtained by first constructing a vertex v_e that describes e (i.e., a DRV/UDRV/RCV vertex for e), and then calling RAW-QUERY recursively on the vertices starting from v_e until reaching the leaf vertices (lines 1-18); note that a vertex returned by a RAW-QUERY call is connected to its parent vertex via either a causal edge and/or a sequencing edge, depending on the set(s) it belongs to (lines 12-17). The resulting graph, denoted by $G(e', e, \mathcal{E})$, includes all necessary events to explain both e and the delay from e' to e . However, as it requires delay annotation (Sections 4.3–4.5) to be useful for diagnostics, we refer to it as the “raw” temporal provenance of T-QUERY(e', e).

The RAW-QUERY(v) procedures rely on a helper function called PREV-VERTEX to find vertices that are connected to v via sequencing edges. For ease of exposition, we first explain the pseudo-code of PREV-VERTEX in Figure 12: given an interval $[t', t]$ and a node N (supplied by RAW-QUERY calls), PREV-VERTEX finds the chain of preceding events that happened on N during $[t', t]$; it first locates the last event v whose execution ends at t and constructs a corresponding vertex (lines 51-60); it then shortens the interval to until the starting timestamp of v and recursively find prior events on N (line 61); it stops until the interval is exhausted or if no event can be found (line 67); finally, it recursively connects this chain of events using sequencing edges and returns the last event in the chain to its caller (line 65-66). For example, consider the provenance graph from Figure 13: a PREV-VERTEX($[2.5s, 3.5s], Y$) call will first find the DRV(F) event, which ends at exactly $t = 3.5s$; it constructs a vertex and shortens the interval to $[2.5s, 2.5s]$, by excluding the execution time of DRV(F); this interval is passed into a recursive call – PREV-VERTEX($[2.5s, 2.5s], Y$) – that finds the event of and constructs a vertex for INS(G); the recursion then stops because the interval becomes empty (because INS(G) takes a positive amount of time); the two constructed vertices are connected via sequencing edges and the last event in the chain – DRV(F) – is returned to the caller.

Figure 12 shows the pseudo-code for RAW-QUERY(v) depending on the type of v (DRV, UDRV, SND, RCV, INS and DEL). Note that each

```

1: function CONSTRUCT-GRAF( $v_e$ )
2:    $G \leftarrow \{v_e\}$  // the “raw” temporal provenance graph
3:   // a queue of vertices that need explanation
4:    $NodeToProcess \leftarrow \{v_e\}$ 
5:   while  $NodeToProcess \neq \emptyset$  do
6:      $v \leftarrow NodeToProcess.POP()$ 
7:      $S, S' \leftarrow RAW-QUERY(v)$ 
8:     for  $v' \in \{S \cup S'\}$  do
9:       if  $v' \notin G$  then
10:         $G \leftarrow G \cup v'$  // add vertices
11:         $NodeToProcess.PUSH(v')$ 
12:     for  $v' \in S$  do
13:       // add causal edges
14:        $G \leftarrow G \cup (v', v)_{causal}$ 
15:     for  $v' \in S'$  do
16:       // add sequencing edges
17:        $G \leftarrow G \cup (v', v)_{sequencing}$ 
18:   RETURN  $G$ 
19: function RAW-QUERY(DRV([ $t_s, t_e$ ], N,  $\tau, \tau_1, \tau_2, \dots, \tau_m$ ))
20:    $S \leftarrow \emptyset$ 
21:    $t_e^{max} \leftarrow 0$  // the last precondition was satisfied at  $t_e^{max}$ 
22:   for  $\tau_i \in \{\tau_1, \tau_2, \dots, \tau_m\}$  do
23:     Find  $d_i @ N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \dots, c_k\}, \pm \tau_i) \in \mathcal{E}$ :
24:      $t'_e \leq t_s$  and  $t'_e$  is maximized
25:      $t_e^{max} \leftarrow \text{MAX}(t_e^{max}, t'_e)$ 
26:     if  $r = r_{ins}$  then
27:        $S \leftarrow S \cup \text{INS}([t'_s, t'_e], N, \tau_i)$ 
28:     else if  $r = r_{rcv}$  then
29:        $S \leftarrow S \cup \text{RCV}([t'_s, t'_e], N \leftarrow r.N, \pm \tau_i)$ 
30:     else
31:        $S \leftarrow S \cup \text{DRV}([t'_s, t'_e], N, \tau_i, \tau_i : - \tau', c_1, c_2, \dots, c_k)$ 
32:     // include all preceding events that happened after the last
33:     // precondition was satisfied and before the derivation of  $\tau$ 
34:   RETURN ( $S; \text{PREV-VERTEX}([t_e^{max}, t_s], N)$ )
35: function RAW-QUERY(INS([ $t_s, t_e$ ], N,  $\tau$ ))
36:   RETURN ( $\emptyset; \emptyset$ )
37: function RAW-QUERY(SND([ $t_s, t_e$ ], N  $\rightarrow$   $N', \pm \tau$ ))
38:   Find  $d @ N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \dots, c_k\}, \pm \tau) \in \mathcal{E}$ :
39:    $t'_e \leq t_s$  and  $r \neq r_{rcv}$  and  $t'_e$  is maximized.
40:   if  $r = r_{ins/del}$  then
41:      $\text{RETURN } (\text{INS/DEL}([t'_s, t'_e], N, \tau);$ 
42:      $\quad \text{PREV-VERTEX}([t'_e, t_s], N))$ 
43:   else
44:      $\text{RETURN } (\text{DRV/UDRV}([t'_s, t'_e], N, \tau, \tau : - \tau', c_1, c_2, \dots);$ 
45:      $\quad \text{PREV-VERTEX}([t'_e, t_s], N))$ 
46: function RAW-QUERY(RCV([ $t_s, t_e$ ], N  $\leftarrow$   $N', \pm \tau$ ))
47:   Find  $d @ N' = (\tau', r, t'_s, t'_e, \pm \tau) \in \mathcal{E}$ :
48:    $t'_e \leq t_s$  and  $r = r_{snd}$  and  $t'_e$  is maximized
49:    $v \leftarrow \text{SND}([t'_s, t'_e], N' \rightarrow N, \pm \tau)$ 
50:   // a remote sequencing edge exists from the SND vertex
51:   RETURN ( $v; v$ )
52: function PREV-VERTEX([ $t', t$ ], N)
53:   if  $t' < t$  then
54:     // If an immediate preceding event exists, then add a
55:     // sequencing edge from the corresponding vertex.
56:     if  $\exists d @ N = (\tau', r, t_s, t_e, \{c_1, c_2, \dots\}, \pm \tau) : t_e = t$  then
57:        $v \leftarrow null$ 
58:       if  $r = r_{snd}$  then
59:          $v \leftarrow \text{SND}([t_s, t_e], N \rightarrow r.N, \pm \tau)$ 
60:       else if  $r = r_{rcv}$  then
61:          $v \leftarrow \text{RCV}([t_s, t_e], N \leftarrow r.N, \pm \tau)$ 
62:       else if  $r = r_{ins/del}$  then
63:          $v \leftarrow \text{INS/DEL}([t_s, t_e], N, \tau)$ 
64:       else
65:          $v \leftarrow \text{DRV/UDRV}([t_s, t_e], N, \tau, \tau : - \tau', c_1, c_2, \dots)$ 
66:     RETURN  $v$ 
67:   RETURN  $null$ 
68: function RAW-QUERY(UDRV([ $t_s, t_e$ ], N,  $\tau, \tau_1, \tau_2, \dots, \tau_m$ ))
69:    $S \leftarrow \emptyset$ 
70:    $t_e^{max} \leftarrow 0$  // the last precondition was satisfied at  $t_e^{max}$ 
71:   for  $\tau_i \in \{\tau_1, \tau_2, \dots, \tau_m\}$  do
72:     Find  $d_i @ N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \dots, c_k\}, \pm \tau_i) \in \mathcal{E}$ :
73:      $t'_e \leq t_s$  and  $t'_e$  is maximized
74:      $t_e^{max} \leftarrow \text{MAX}(t_e^{max}, t'_e)$ 
75:     if  $r = r_{ins/del}$  then
76:        $S \leftarrow S \cup \text{INS/DEL}([t'_s, t'_e], N, \tau_i)$ 
77:     else if  $r = r_{rcv}$  then
78:        $S \leftarrow S \cup \text{RCV}([t'_s, t'_e], N \leftarrow r.N, \pm \tau_i)$ 
79:     else
80:        $S \leftarrow S \cup \text{DRV/UDRV}([t'_s, t'_e], N, \tau_i, \tau_i : - \tau', c_1, c_2, \dots, c_k)$ 
81:     // includes all preceding events that happened after the last
82:     // precondition was satisfied and before the underivation of  $\tau$ 
83:   RETURN ( $S; \text{PREV-VERTEX}([t_e^{max}, t_s], N)$ )
84: function RAW-QUERY(DEL([ $t_s, t_e$ ], N,  $\tau$ ))
85:   RETURN ( $\emptyset; \emptyset$ )

```

Figure 12: Algorithm for constructing temporal provenance graph for a given execution trace \mathcal{E} . The trace \mathcal{E} consists of events (Definition A.1), which are recorded at runtime or reconstructed via replay. The function $\text{RAW-QUERY}(v)$, when called on a vertex v , returns two sets of immediate children of v , which are connected to v via causal edges and sequencing edges, respectively. It calls $\text{PREV-VERTEX}([t', t], N)$ as a subprocedure, which finds a chain of vertices connected via sequencing edges that immediately precedes v during $[t', t]$.

Algorithm 1 Extracting traces from provenance

```

1: // This algorithm extracts the trace  $\mathcal{A}(e', e, \mathcal{E})$  from the “raw” temporal provenance  $G(e', e, \mathcal{E})$ ; for ease of explanation, we rewrite  $G$  as  $(V, E)$ , where  $V$  represents vertices and  $E$  represent edges
2: proc EXTRACTTRACE( $G = (V, E)$ )
3: // Calculate the out-degree of every vertex in  $G$ 
4: for all  $v \in V$  do  $\text{degree}(v) \leftarrow 0$ 
5: for all  $e = (v, v') \in E$  do  $\text{degree}(v)++$ 
6: // Generate the race based on topological sort
7:  $trace \leftarrow \emptyset$ 
8:  $NodeToProcess \leftarrow V$ 
9: while  $NodeToProcess \neq \emptyset$  do
10:   // Select the next event based on topological ordering and timestamps
11:   select  $v \in NodeToProcess : \text{degree}(v) = 0$  and  $\exists v'$ 
        that is located on the same node and has a larger end timestamp
12:    $NodeToProcess.\text{REMOVE}(v)$ 
13:   if  $\text{type}(v) = \text{DRV}$  or  $\text{UDRV}$  or  $\text{SND}$  then
14:      $\text{preconditions} \leftarrow \emptyset$ 
15:     for  $\forall (v', v) \in E$  s.t.  $(v', v)$  is a causal edge do
16:        $\text{preconditions}.ADD(\text{tuple}(v'))$  //  $\text{tuple}(v')$  is a precondition
17:     // find the trigger from the preconditions
18:      $trigger \leftarrow \tau' \in \text{preconditions}:$ 
        (a)  $\tau'$  is a message, or (b)  $\tau'$  is a state and
         $\exists \tau'' \in \text{preconditions}$  that has a larger end timestamp
19:      $\text{preconditions}.DELETE(trigger)$ 
20:      $event \leftarrow (trigger, \text{rule}(v), \text{startTime}(v),$ 
         $\text{endTime}(v), \text{preconditions}, \text{tuple}(v))$ 
21:      $trace.push\_front(event)$ 
22:   if  $\text{type}(v) = \text{RCV}$  then
23:      $output \leftarrow \text{tuple}(v)$ 
24:      $trigger \leftarrow \text{tuple}(v'): (v', v) \in E$  s.t.  $\text{type}(v') = \text{SND}$ 
25:      $event \leftarrow (trigger, \text{rule}(v), \text{startTime}(v),$ 
         $\text{endTime}(v), \emptyset, \text{tuple}(v))$ 
26:      $trace.push\_front(event)$ 
27:   if  $\text{type}(v) = \text{INS}$  or  $\text{DEL}$  then
28:      $output \leftarrow \text{tuple}(v)$ 
29:      $event \leftarrow (\emptyset, \text{rule}(v), \text{startTime}(v),$ 
         $\text{endTime}(v), \emptyset, \text{tuple}(v))$ 
30:      $trace.push\_front(event)$ 
31:   for all  $(v', v) \in E$ ,  $\text{degree}(v') \leftarrow \text{degree}(v') - 1$ 
35: return  $trace$ 

```

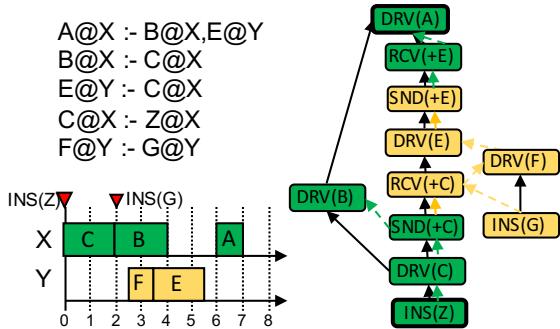


Figure 13: An example scenario, with NDlog rules at the top left, the timing of a concrete execution in the bottom left, and the temporal provenance graph at the right. The query is T-QUERY(INS(Z), DRV(A)); the start and end vertices are marked in bold. Vertex names are shortened and some fields are omitted for clarity.

DRV or UDRV vertex is also associated with the corresponding derivation rule. Next, we explained the pseudo-code of RAW-QUERY(v) for each vertex type in more detail. For ease of exposition, we use the provenance graph from Figure 13:

- To explain a SND vertex, we find the most recent event in the original trace that produced (or deleted) the tuple that is being sent (line 37),

construct an INS (or DEL) or a DRV (or UDRV) vertex for the found event, and add an incoming causal edge from the constructed vertex (lines 38-41); in addition, a SND vertex has an incoming sequencing edge from the chain of preceding events that happened after the message was produced or deleted (the PREV-VERTEX calls in lines 39 and line 41). For example, in the temporal provenance graph from Figure 13, the SND(+C) vertex has a causal edge from the DRV(C) vertex, because DRV(C) functionally triggered SND(+C); in addition, a PREV-VERTEX([2s, 2s], X) call adds a sequencing edge from DRV(C) to SND(+C), as the former directly precedes the latter.

- A RCV has an incoming causal edge and an incoming remote sequencing edge from the SND vertex for the received message (lines 42-46). This is the case for RCV(+E) and RCV(+C) in Figure 13.
- A DRV vertex for a rule $A : -B, C, D$ has an incoming causal edge for each precondition (B, C, and D) that leads to the vertex that produced the corresponding tuple (line 22); this can be an INS, a RCV, or another DRV (lines 25-30); in addition, a DRV vertex has an incoming sequencing edge from the chain of preceding events that happened after the last precondition was satisfied (the PREV-VERTEX call in line 33). For example, in the provenance from Figure 13, DRV(F) has a causal edge from its (only) precondition INS(G); in addition, a PREV-VERTEX([2.5s, 3.5s], Y) finds the preceding event RCV(+C) that occurred between INS(G) ended and DRV(F) started.
- An INS vertex corresponds to the insertion of a base tuple, which cannot be explained further; thus, it has no incoming edges (line 35). This is true for INS(Z) and INS(G) in Figure 13.
- The edges for the negative “twins” of these vertices – UDRV and DEL – are analogous.

A.3 Properties

Given the “raw” temporal provenance $G(e', e, \mathcal{E})$ of a diagnostic query T-QUERY(e', e) in an execution trace \mathcal{E} , we say that $G(e', e, \mathcal{E})$ is correct if it is possible to extract a subtrace from G that has the properties of validity, soundness, completeness, and minimality. We first describe our algorithm for extracting such a subtrace, and then formally define these properties and their proofs.

Definition (Trace Extraction): Given a temporal provenance $G(e', e, \mathcal{E})$, the trace $\mathcal{A}(e', e, \mathcal{E})$ is extracted by running Algorithm 1 based on topological sort.

Algorithm 1 converts the vertices in the provenance graph to events and then uses a topological ordering and timestamps to assemble the events into a trace. In particular, Line 13-33 implements the construction of one individual event, where the information of a rule evaluation (such as triggering event, conditions, and action) is extracted from vertices in $G(e', e, \mathcal{E})$: a DRV/UDRV/SND vertex and their children; a pair of RCV and SND vertices; or a INS/DEL vertex. In the algorithm, $\text{type}(v)$, $\text{tuple}(v)$, $\text{rule}(v)$, $\text{startTime}(v)$ and $\text{endTime}(v)$ denote the vertex type, the tuple, the derivation rule, the start timestamp, and the end timestamp of the vertex v , respectively. For example, Algorithm 1 extracts the following trace from the provenance graph in Figure 13: $\langle \text{INS}(Z)@X, \text{DRV}(C)@X, \text{SND}(+C)@X, \text{INS}(G)@Y, \text{DRV}(B)@X, \text{RCV}(+C)@Y, \text{DRV}(F)@Y, \text{DRV}(E)@Y, \text{SND}(+E)@Y, \text{RCV}(+E)@X, \text{DRV}(A)@X \rangle$.

We will show that the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ obtained from Algorithm 1 satisfies the following four properties.

Definition (Soundness): A subtrace \mathcal{A} extracted from $G(e', e, \mathcal{E})$ is sound if and only if it is a subtrace of some trace \mathcal{E}' that is equivalent to \mathcal{E} , i.e., $\mathcal{A} \subseteq \mathcal{E}' \sim \mathcal{E}$.

Intuitively, the soundness property means that $\mathcal{A}(e', e, \mathcal{E})$ must preserve all the happens-before relationships among events and the exact timestamps of

events in the original execution trace obtained from running the NDlog program. Ideally, we would like $\mathcal{A}(e', e, \mathcal{E})$ to be a subtrace of \mathcal{E} , but without synchronized clocks, we cannot always order concurrent events on different nodes. However, for practical purposes \mathcal{E} and \mathcal{E}_0 are indistinguishable: each node observes the same sequence of events in the same order and at the same times.

Definition (Completeness): A subtrace \mathcal{A} extracted from $G(e', e, \mathcal{E})$ is complete if and only if it ends with the event e and e happens at the same time as in \mathcal{E} .

Intuitively, completeness means that $\mathcal{A}(e', e, \mathcal{E})$ must include all events necessary to reproduce e both functionally and temporally. Note that the validity property already requires that any event that is needed for e be included in $\mathcal{A}(e', e, \mathcal{E})$; hence, we can simply verify the completeness property of a valid trace by checking whether it ends with e .

Definition (Validity): A subtrace \mathcal{A} extracted from $G(e', e, \mathcal{E})$ is valid if and only if, given the initial state S_0 , for every event $d_i @ N_i = (\tau_j, r_i, t_i, t'_i, c_i, \pm\tau'_i) \in \mathcal{A}$, the following holds:

- (a) there exists $d_j @ N_j = (\tau_j, r_j, t_j, t'_j, c_j, \pm\tau'_j)$ that precedes $d_i @ N_i$ in \mathcal{A} such that $\tau_i = \tau'_j$;
- (b) for all $\tau_k \in c_i$, we have $\tau_k \in S_{i-1}$, where $S_0 \xrightarrow{d_1 @ n_1} S_1 \xrightarrow{d_2 @ n_2} \dots \xrightarrow{d_{i-1} @ n_{i-1}} S_{i-1}$; and
- (c) based on the conditions (a) and (b), consider the set of all events P_i such that $d_k @ N_k \in P_i$ generates $\tau_k \in (c_i \cup \tau_i)$; denote $d_j @ N_j$ as the latest event in P_i ; if $N_j = N_i$ and $t'_j < t_i$, there must exist a set of events $\{d_p^1 @ N_i, \dots, d_p^m @ N_i\} \in \mathcal{A}$ such that: $t'_j = t_p^1$, $t_p^m = t_p^{m+1}$, $1 \leq m < n$; and $t_p^n = t_i$.

Intuitively, the validity property means that $\mathcal{A}(e', e, \mathcal{E})$ must correspond to a correct execution of the NDlog program both in terms of functionality and timing. Condition (a) states that any event that triggers a rule evaluation must be generated before the rule is evaluated. Condition (b) states that the preconditions of the rule evaluation must hold at the time of the rule evaluation. Finally, condition (c) requires that the evaluation is “work-conserving”: the node cannot be idle when it is ready to compute a derivation.

Definition (Minimality): A subtrace \mathcal{A} extracted from $G(e', e, \mathcal{E})$ is minimal iff there exists no trace \mathcal{E}' such that: (a) there $\exists d_i @ N_i$ where $d_i @ N_i \in \mathcal{A}$ and $d_i @ N_i \notin \mathcal{E}'$; (b) \mathcal{E}' is valid, sound, and complete.

Intuitively, minimality means that $\mathcal{A}(e', e, \mathcal{E})$ should not contain any events that are not necessary to reproduce e . If this property were omitted, $\mathcal{A}(e', e, \mathcal{E})$ could trivially output the complete trace \mathcal{E} .

A.4 Proofs

Lemma 1 For any execution \mathcal{E} , and a temporal provenance query T-QUERY(e', e), the provenance graph $G(e', e, \mathcal{E})$ is acyclic.

Proof. We first show that if there exists a cycle in $G(e', e, \mathcal{E})$, the cycle cannot include two vertices located on different nodes. Suppose there exists a cycle that contains two vertices v_1 and v_2 located on N_1 and N_2 respectively. Then the cycle must contain a least one pair of SND and RCV vertices in both the path from v_1 to v_2 , and the path from v_2 to v_1 . Each SND and RCV corresponds to a message communication which takes a positive amount of time. Therefore, traversing from v_1 along the cycle back to v_1 results in an absolute increment in the timestamp. This is a contradiction.

If all the vertices in the cycle are located on the same node, then we can order the vertices according to their associated timestamps (now all the timestamps are with respect to the same local clock). Such order corresponds to the precedence of events in the execution. As time always progresses forward, such cycle cannot exist in $G(e', e, \mathcal{E})$. \square

Theorem 2 $\mathcal{A}(e', e, \mathcal{E})$ is sound.

Proof. We need to show that a) all the events in $\mathcal{A}(e', e, \mathcal{E})$ also appear in \mathcal{E} at the same time (and thus in any $\mathcal{E}_0 \sim \mathcal{E}$), and b) the local event ordering pertains on each node, that is, for any two events $d_1 @ N_i$ and $d_2 @ N_i$ in $\mathcal{A}(e', e, \mathcal{E})$ that are located on the same node N_i , $d_1 @ N_i$ precedes $d_2 @ N_i$ in $\mathcal{A}(e', e, \mathcal{E})$ iff $d_1 @ N_i$ precedes $d_2 @ N_i$ in \mathcal{E} .

Condition a. We perform a case analysis by considering the type of the root vertex of $G(e', e, \mathcal{E})$:

- **DRV.** According to Algorithm 1 (lines 13-22), an event $d_i @ N_i$ is generated and included in $\mathcal{A}(e', e, \mathcal{E})$ for each DRV vertex (and its children) in the provenance graph $G(e', e, \mathcal{E})$. However, by construction, each DRV vertex v corresponds to a rule evaluation in \mathcal{E} . In our model, the rule evaluation is modeled as an event $d_j @ N_j = (\tau_j, r_j, t_j, t'_j, \{c_j^1, \dots, c_j^P\}, \pm\tau'_j)$, where τ_j is the trigger event, r_j and $[t_j, t'_j]$ are the rule used in and the time interval of the rule evaluation, c_j^k represents preconditions, and $\pm\tau'_j$ is the generated update. We need to show that $d_i @ N_i$ is identical to $d_j @ N_j$. This follows straightforwardly from the construction of $G(e', e, \mathcal{E})$: The RAW-QUERY(v) procedures generate a DRV vertex v by: (a) find a derivation event $d_j @ N_j$ from \mathcal{E} , (b) add incoming edges from the trigger event (a vertex for τ_j), and (c) add incoming edges from the preconditions (vertices for $\{c_j^1, \dots, c_j^P\}$). Algorithm 1 reverses this process and generates event $d_i @ N_i$ from these information, which is extracted from $d_j @ N_j$, and therefore $d_i @ N_i = d_j @ N_j$.
- **RCV/INS/DEL/UDRV/SND** Following the same argument for the DRV case above, we can prove that condition (a) holds.

Condition b. According to Algorithm 1 (specifically, Line 11), $d_1 @ N_i$ precedes $d_2 @ N_i$ in $\mathcal{A}(e', e, \mathcal{E})$, iff $d_2 @ N_i$ has a larger timestamp than $d_1 @ N_i$. However, $d_2 @ N_i$ is assigned a larger timestamp iff $d_1 @ N_i$ precedes $d_2 @ N_i$ in the actual execution \mathcal{E} . Note that events on different nodes may be reordered in $\mathcal{A}(e', e, \mathcal{E})$, but this is captured by the equivalence (\sim) relation. \square

Theorem 3 $\mathcal{A}(e', e, \mathcal{E})$ is complete.

Proof. We need to show that a) $\mathcal{A}(e', e, \mathcal{E})$ contains an event $d_i @ N_i$ that generates e at the same time as in \mathcal{E} , and b) $d_i @ N_i$ is the last event in $\mathcal{A}(e', e, \mathcal{E})$.

Condition a. By construction, the vertex for e has incoming edges from vertices representing the triggering event τ and all preconditions c_1, \dots, c_P (if any). Algorithm 1 (specifically, Lines 13-28) construct an event $(\tau, r, t, t', c, \pm\tau)$, where r and $[t, t']$ are the rule name and time interval encoded in the vertex. Note that the tuple τ as well as the timestamps t and t' are exactly the ones that are extracted from \mathcal{E} (Algorithm 12).

Condition b. Now we have proved that some event $d_i @ N_i$ that generates e must exist in $\mathcal{A}(e', e, \mathcal{E})$, we next show that $d_i @ N_i$ is the last event in $\mathcal{A}(e', e, \mathcal{E})$. The provenance graph $G(e', e, \mathcal{E})$ is rooted by a vertex that corresponds to e . Since all other vertices in $G(e', e, \mathcal{E})$ have a directed path to the root vertex, the corresponding events must all be ordered before $d_i @ N_i$, so $d_i @ N_i$ must necessarily be the last event in the subtrace. \square

Theorem 4 $\mathcal{A}(e', e, \mathcal{E})$ is valid.

Proof. Lemma 1 shows that any provenance graph $G(e', e, \mathcal{E})$ is acyclic, and thus $G(e', e, \mathcal{E})$ has a well-defined height: the length of the longest path from any leaf to e . We prove validity using structural induction on the height of the provenance graph $G(e', e, \mathcal{E})$.

Base case: The height of $G(e', e, \mathcal{E})$ is one. In this case, e must be an insertion or deletion of a base tuple; $G(e', e, \mathcal{E})$ contains a single INS (or DEL) vertex that corresponds to the update of the base tuple. Therefore, $\mathcal{A}(e', e, \mathcal{E})$ consists of a single event and is trivially valid, because the event has neither a trigger nor any precondition (Algorithm 1 lines 29-33).

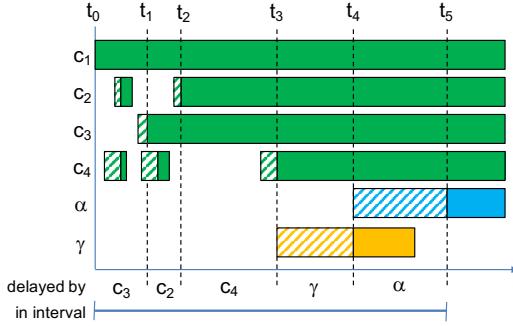


Figure 14: Illustration for the definition of direct and transitive delay. Shaded boxes represent intervals where a tuple was being derived, and solid boxes represent intervals where the tuple existed. The derivation is $\alpha : -c_1, c_2, c_3, c_4$, and the interval in question is $[t_0, t_5]$; γ is an unrelated tuple whose derivation just happened to be sequenced before that of α .

Induction case: Suppose the validity of the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ holds for any provenance graph with height less than k ($k \geq 1$). Consider a provenance graph $G(e', e, \mathcal{E})$ with height $k + 1$. We perform a case analysis by considering the type of the root vertex of $G(e', e, \mathcal{E})$. For every event $d_i @ N_i = (\tau_i, r_i, t_i, t'_i, c_i, \pm \tau'_i) \in \mathcal{A}(e', e, \mathcal{E})$, we prove that the three conditions in Definition A.3 hold.

- **DRV.** We know that, by construction, the DRV vertex has an incoming edge from vertices representing the triggering event τ and all preconditions c_1, \dots, c_p . By the induction hypothesis, Algorithm 1 outputs a valid trace $d_1 @ N_1, \dots, d_j @ N_j$ for the subgraph for the trigger event τ , where $d_j @ N_j$ corresponds to the generation of τ (following the completeness property proved in Theorem 3). Because of the nature of Algorithm 1 (which is based on topological sort), $d_j @ N_j$ must be ordered before $d_i @ N_i$, which satisfies condition (a) in the definition of validity. For example, in the provenance graph from Figure 13, the trigger event INS(G) must precede the derived event DRV(F) in the extracted trace, because a causal edge exists from the former to the latter. Similarly, valid traces are generated for the updates that support the preconditions c_1, \dots, c_p , which satisfies conditions (b). Condition (c) holds by construction: the original execution trace \mathcal{E} is valid and must include a set of events $\{d_p^1 @ N_i, \dots, d_p^n @ N_i\}$ that satisfies condition (c); the PREV-VERTEX call in Figure 12 finds all these events because the call recursively finds such events from \mathcal{E} until the interval between the end of $d_j @ N_j$ and the start of $d_i @ N_i$ is fully exhausted (line 48); therefore, all events in $\{d_p^1 @ N_i, \dots, d_p^n @ N_i\}$ will be represented by vertices in the temporal provenance; the extraction algorithm merely reverses this process and reconstructs each of $\{d_p^1 @ N_i, \dots, d_p^n @ N_i\}$, while preserving their ordering and timestamps (following the soundness property proved in Theorem 2). Therefore, the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ is valid. For example, consider the DRV(F) event in the provenance from Figure 13: there is a gap of [2.5s, 3.5s] between when its last precondition INS(G) completed and when its own derivation started; in the original execution, node Y must be busy during the gap, because it is work-conserving; in this case, Y was busy with handling RCV(+C); while constructing the vertex for DRV(F), the RAW-QUERY procedure calls PREV-VERTEX([2.5s, 3.5s], Y), which finds the RCV(+C) event from the original trace and added a vertex to G; Algorithm 1 extracts events from G based on topological ordering, therefore, RCV(+C) will present in \mathcal{A} , after INS(G) and before DRV(F).
- **RCV.** We know that, by construction, the RCV vertex has an incoming edge from a SND vertex with the same tuple τ .

By the induction hypothesis, Algorithm 1 outputs a valid trace $d_1 @ N_1, \dots, d_j @ N_j$ for the subgraph rooted at the SND vertex, where $d_j @ N_j$ corresponds to the generation of τ (following the completeness property proved in Theorem 3). Because of the nature of Algorithm 1 (which is based on topological sort), $d_j @ N_j$ must be ordered before $d_i @ N_i$, which satisfies condition (a) in the definition of validity. A SND vertex have no preconditions, consequently, conditions (b) holds trivially. $d_i @ N_i$ and $d_j @ N_j$ happened on different nodes, which satisfies condition (c) trivially. Therefore, the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ is valid.

- **UDRV/SND.** Following the same argument for the DRV case above, the extracted trace $\mathcal{A}(e', e, \mathcal{E})$ is valid.
- **INS/DEL.** This case cannot occur because INS and DEL have no preconditions, so the tree would have to have a height of one.

□

Theorem 5 $\mathcal{A}(e', e, \mathcal{E})$ is minimal.

Proof. We prove the minimality property by induction on the syntactic structure of $\mathcal{A}(e', e, \mathcal{E})$: we show that an event $d_i @ N_i \in \mathcal{A}(e', e, \mathcal{E})$ cannot be removed because it is necessary for some event $d_j @ N_j$ appeared later in the trace. Suppose that $\mathcal{A}(e', e, \mathcal{E}) = d_1 @ N_1, \dots, d_m @ N_m$.

Base case. According to the completeness property (Theorem 3), the last event $d_m @ N_m$ in $\mathcal{A}(e', e, \mathcal{E})$ generates e . Therefore the base case trivially holds, as the removal of $d_m @ N_m$ breaks the completeness property.

Induction case. Suppose the last k events $d_{m-k+1} @ N_{m-k+1}, \dots, d_m @ N_m$ ($K \geq 1$) cannot be removed. We show that event $d_{m-k} @ N_{m-k}$ cannot be removed as well: According to Algorithm 1, $d_{m-k} @ N_{m-k}$ is constructed from a vertex v . v must have an outgoing edge to some other vertex in $G(e', e, \mathcal{E})$. Otherwise, v would not be included in $G(e', e, \mathcal{E})$ which is a subgraph rooted by e . Consider u as the first vertex on the path from v to the root of $G(e', e, \mathcal{E})$. According to Algorithm 1, an event $d_j @ N_j$ is constructed from u and its children (if any). Given the edge from v to u , we know that $d_j @ N_j$ depends on $d_{m-k} @ N_{m-k}$, and that $d_{m-k} @ N_{m-k}$ precedes $d_j @ N_j$. By applying the induction hypothesis ($d_j @ N_j$ cannot be removed from $\mathcal{A}(e', e, \mathcal{E})$), we can conclude that $d_{m-k} @ N_{m-k}$ also cannot be removed. □

A.5 Delay annotations

In this section, we show that each vertex is annotated with the delay that it contributed. We first define what it means for a derivation to be directly “delayed” by one of its preconditions (Definition A.5), and then recursively extends this definition to transitive delays (Definition A.5). We continue by discussing several properties of the annotations computed by the algorithm from Figure 6 (Definition A.5, Lemmas 6–9). This allows us to further prove the first theorem which states that the algorithm from Figure 6 labels each vertex with the amount of (direct or transitive) delay that is contributed by the subtree that is rooted at that vertex (Theorem 10).

Definition (Direct delay): Consider a derivation rule $\alpha : -c_1, c_2, \dots, c_k$ and an interval $[t_0, t_5]$, such that α begins its derivation at $t_4 < t_5$ and finishes it at time t_5 . We say that a precondition c_i directly delays the derivation of α during an interval $[t_x, t_y]$, $t_0 \leq t_x, t_y \leq t_4$, iff

- (a) c_i became true at t_y and remain true until t_4 (and was false before t_y); and
- (b) there either was some c_j , $i \neq j$, that delayed the derivation of α during some interval $[x, t_x]$; or there was no such c_j , and $t_x = t_0$.

For convenience, we say that α itself delays its own derivation during $[t_4, t_5]$. Find the time $t_3 \leq t_4$ such that t_3 is the earliest time when all preconditions were true (and remained true until t_4). If a tuple γ resides on the same node as α and the derivation of γ happened during $[t_3, t_4] \subseteq [t_3, t_4]$, we also say that γ directly delays the derivation of α .

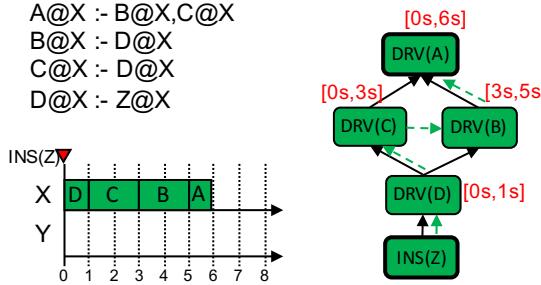


Figure 15: An example scenario, with NDlog rules at the top left, the timing of a concrete execution in the bottom left, and the resulting temporal provenance at the right. The query is T-QUERY(INS(Z), DRV(A)); the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is annotated with its annotation interval (Definition A.5).

Figure 14 contains a brief illustration. c_3 directly delays the derivation of α during the interval $[t_0, t_1]$, because: (a) c_3 became true at t_1 and remained true until t_4 ; (b) t_0 was the start of the interval in question (the second case of condition (b)). c_2 directly delays the derivation of α during the interval $[t_1, t_2]$, because: (a) c_2 became true at t_2 and remained true until t_4 ; (b) c_3 delayed the derivation of α during $[t_0, t_1]$ (the first case of condition (b)). Similarly, c_4 delays the derivation of α during the interval $[t_2, t_3]$. γ delays the derivation of α during the interval $[t_3, t_4]$ because, during that interval, all preconditions were true and γ was derived on the same node as α . Finally, α delays the derivation of itself during $[t_4, t_5]$. We can now expand this definition to other derivations:

Definition (Transitive delay): Consider two derivations $\alpha : -c_1, c_2, \dots, c_k$ and $\beta : -d_1, d_2, \dots, d_m$, and suppose β (directly or transitively) delays the derivation of α during an interval $[t_0, t_3]$. Then we say that a tuple d_i transitively delays the derivation of α during an interval $[t_1, t_2]$, $t_0 \leq t_1, t_2 \leq t_3$, iff d_i directly delays the derivation of β during $[t_1, t_2]$.

We can think of the definition of transitive delay as recursively partitioning the interval $[t_0, t_5]$ into smaller intervals that are each associated with some lower-level derivation that caused delay to the top-level derivation of α .

Definition (Annotation interval): We associate a vertex v in G with an annotation interval $I_v^\alpha = [t_s, t_e]$ for each call of the ANNOTATE($v, [t_s, t_e]$) procedure in the algorithm in Figure 6.

Figure 15 shows how the algorithm in Figure 6 would have assigned annotation intervals to an example temporal provenance graph. Before presenting the main theorem, we discuss a few properties of annotation intervals.

Lemma 6 In the algorithm in Figure 6, each invocation of the ANNOTATE($v, [t_s, t_e]$) procedure assigns a set of annotation intervals $\{I_{v^i}^\alpha\}$ to vertices $\{v^i\}$ such that $\bigcap I_{v^i}^\alpha = \emptyset$.

Proof. This holds by construction. When v has no child, $\{I_{v^i}^\alpha\} = \emptyset$ and the condition holds trivially. When v has children: the first WHILE loop in the ANNOTATE procedure subdivides the interval between t_s and the end timestamp of the last precondition into annotation intervals for functional children (in lines 8–16); the second WHILE loop subdivides the interval between the end timestamp of the last precondition and $t_s(v)$ into annotation intervals for sequencing vertices (in lines 17–23); note that if a functional precondition v' is also connected via a sequencing edge to v , it is only handled by the first while loop, because $T = t_{end}(v') = t_{start}(v) = E$ after the first while loop finishes and the second while loop will not execute; therefore, all the generated annotation intervals within an ANNOTATE call are non-overlapping. \square

This lemma states that the annotation intervals generated by recursive calls within the same ANNOTATE invocation do not overlap. For example, in Figure 15, the annotation intervals of the DRV(C) and DRV(B) vertices are both assigned by a recursive call on the DRV(A) vertex and thus do not overlap.

Lemma 7 An annotation interval I_v^α of vertex v always ends at $t_e(v)$, where $t_e(v)$ is when the execution of v finishes or the end timestamp of v (Section 3.3).

Proof. This holds by construction of the algorithm in Figure 6. In the first WHILE loop in the ANNOTATE procedure (in lines 8–16), the annotation interval associated with v always ends with $t_e(v)$. In the second WHILE loop (in lines 17–23), the annotation interval of the current vertex is $E = t_s(s)$, which is the start timestamp of the previous vertex connected via a sequencing edge; E is also the end timestamp of the current vertex, which follows from the construction of sequencing edges (PREV-VERTEX calls in the algorithm in Figure 12). \square

This lemma states that the annotation interval ends when the actual execution finishes. For example, this holds for all annotation intervals in Figure 15.

Lemma 8 Suppose a vertex v is associated with an annotation interval I_v^α , there exists a chain of ancestor vertices $v \rightarrow a_1 \rightarrow a_2 \rightarrow \dots \rightarrow e$ (\rightarrow represents an edge in G , and e is the root of G) such that for each a_i (including e) there exists an annotation interval $I_{a_i}^\alpha$ and $I_v^\alpha \subseteq I_{a_i}^\alpha$.

Proof. This holds by construction of the algorithm in Figure 6. It follows from the recursive nature of ANNOTATE calls that the annotation interval of each vertex v is a subinterval of one annotation interval of one of its parents: in the first while loop (in lines 8–16), the ANNOTATE is called with an interval of $[T, t_{end}(v')]$, $t_s \leq T$ and $t_{end}(v') \leq t_{end}(v)$ because v' is a child of v ; in the second while loop (in lines 17–23), the ANNOTATE is called with an interval of $[\text{MAX}(T, t_{start}(s)), E]$, $t_s \leq T \leq \text{MAX}(T, t_{start}(s))$ and $E \leq t_{start}(v) \leq t_e$ (Lemma 7). We can simply find the specified chain by following such parents recursively until reaching the root vertex e . As the annotation interval is initially I_v^α and is gradually extended as we climb the chain, $I_v^\alpha \subseteq I_{a_i}^\alpha$. \square

For instance, consider the provenance from Figure 15, suppose $[0s, 1s]@DRV(D)$ represents that the DRV(D) vertex is associated with an annotation interval of $[0s, 1s]$; the ancestor chain of $[0s, 1s]@DRV(D)$ would be $[0s, 1s]@DRV(D) \rightarrow [0s, 3s]@DRV(C) \rightarrow [0s, 6s]@DRV(A)$.

Lemma 9 Each vertex v in G is associated with at most one annotation interval I_v^α , that is, each vertex v is annotated at most once by the algorithm in Figure 6.

Proof. We prove by contradiction. Without loss of generality, suppose a vertex v is associated with two annotation intervals I_v^α and $I_v^{\alpha'}$. There must exist two corresponding ancestor chains (Lemma 8). We make two observations about the chains: (a) they cannot be identical, because an ancestor chain represents a unique stack of recursive ANNOTATE calls; by the nature of a single-rooted DAG, there cannot exist two stacks of recursive calls that visit the exact same sequence of vertices; (b) the two chains must share a common suffix, this holds trivially because both of the chains end at the root of G . Based on these observations, we can represent the two ancestor chains as $v \rightarrow \dots \rightarrow a_i \rightarrow a_j \rightarrow \dots$ and $v \rightarrow \dots \rightarrow a'_i \rightarrow a_j \rightarrow \dots$, where $a_i \neq a'_i$. It follows from Lemma 8 that $I_v^\alpha \subseteq I_{a_i}^\alpha$ and $I_v^{\alpha'} \subseteq I_{a'_i}^{\alpha'}$. It follows from Lemma 7 that $[t_e(v) - \epsilon, t_e(v)] \subseteq I_v^\alpha$ and $[t_e(v) - \epsilon, t_e(v)] \subseteq I_v^{\alpha'}$, where ϵ is a small value. Therefore, $I_{a_i}^\alpha$ and $I_{a'_i}^{\alpha'}$ overlap. This contradicts with Lemma 6, because $I_{a_i}^\alpha$ and $I_{a'_i}^{\alpha'}$ are divided from $I_{a_j}^\alpha$ in the same ANNOTATE call and cannot overlap. \square

These lemmas allow us to formulate our main claim:

Theorem 10 Suppose T-QUERY(e', e) returns $G(e', e, \mathcal{E})$ in some execution \mathcal{E} , and suppose a vertex v in G is annotated with a value T by the algorithm in Figure 6. Then $T > 0$ iff v directly or transitively delayed the derivation of e during an interval $[t_1, t_2] \subseteq [\text{START}(e'), \text{FINISH}(e)]$ and $T = t_2 - t_1$, and $T = 0$ otherwise.

Proof. We begin by observing that the algorithm in Figure 6 labels each vertex at most once (Lemma 9). Therefore, we only need to show that any single invocation of the ANNOTATE procedure in Figure 6 correctly labels vertices with respect to Definition A.5.

Next, we observe that the ANNOTATE procedure in Figure 6 partitions the interval to explain into annotation intervals of other vertices in exactly the same way that the definition requires. Therefore, I_v^α is exactly the direct or transitive delay of v . We discuss the partition logic of the ANNOTATE procedure in more detail below.

The children of a DRV vertex in the provenance graph would be DRV, INS, or RCV vertices for its preconditions, and lines 8–16 iterate over these vertices in the order of their end times. (The original trace only records the preconditions of an event at the point when its derivation starts; thus, if a precondition had temporarily become true and then false again, the corresponding DRV vertex would not appear as children here.) The loop calls ANNOTATE on vertex v' with a subinterval of $[t_s, t_e]$ that ends at the point where the precondition is fully derived, and starts either at t_s itself or the end of the previous interval. This subinterval is the annotation interval $I_{v'}^\alpha$ for v' (Definition A.5). Preconditions that were already true at t_s and remained true during the entire interval do not enter the IF block and thus do not generate a recursive call. The first WHILE loop exits with T set to the end time of the last precondition; the WHILE loop that follows it (in lines 17–23) subdivides any non-empty interval between the last satisfied precondition and the start of the derivation of v , just as the definition requires. Again, here each of the divided intervals is the annotation interval $I_{v'}^\alpha$ for another vertex v' (Definition A.5). In particular, noticed that recursive calls happen only for vertices that directly delayed v (and, hence, directly or transitively delayed the vertex in the original query).

Finally, we observe that each vertex v gets labeled with the length of I_v^α in line 7. The labeled value is also the amount of direct or transitive delay that v contributes, because we have proved above that the I_v^α is exactly the direct or transitive delay of v . For example, the intervals annotated beside vertices in Figure 15 are also their direct or transitive delay. \square

A.6 Semantics of delay annotations

Although the definitions from Section A.5 do capture the intuitive notion of “delay”, we want to reinforce this by formalizing another aspect of this concept: if a vertex v really did delay a derivation by some time T_v , then it should be possible to “speed up” the derivation by T (i.e., cause it to happen T_v units of time sooner) by reducing the duration of v by T_v . In other words, we should be able to construct a valid (hypothetical) trace that differs from the actual trace in that v takes less time, such that the hypothetical trace finishes T_v units of time earlier. (Note that the hypothetical trace might not be “realistic” in a practical sense because some of the events in it may take zero time, and thus be instantaneous; the goal is merely to demonstrate that v is really “responsible for” T_v units of delay.) For example, Figures 16 shows the steps of “speeding up” vertices based on their annotations ((a) → ... → (g)). This procedure shortens the overall (hypothetical) execution at each step and eventually eliminates any delay.

For this discussion, the annotation intervals that are computed by the algorithm in Figure 6 are not directly useful, because they describe the delay that was caused by an entire subgraph of the provenance. Hence, we first describe how we have derived a more fine-grain form of annotation, which describes the delay that is contributed by a vertex itself (Definition A.6). We then discuss two properties of the derived annotation (Lemmas 11–12). We continue by defining the procedure of “speeding up” an execution based on derived annotations (Definitions A.6–A.6). We conclude by presenting the main theorem which states that if there is a vertex v in a temporal provenance tree with a (derived) annotation of T , then it is possible to construct another valid (but hypothetical) execution in which v 's finished time is reduced by T and in which the derivation finishes T units of time earlier (Definition A.6 and Theorem 13).

Definition (Speedup interval): The speed interval $I_v^\delta = [t_s, t_e]$ of vertex v is the difference between v 's annotation interval, as computed by the algorithm from Figure 6, and the union of the annotation intervals of the vertices directly annotated by v (via the recursive calls in the ANNOTATE procedure).

Intuitively, I_v^δ represents the interval during which the execution of v itself delays e . For example, in the provenance from Figure 16(a), red intervals represent annotation intervals and blue intervals represent speedup intervals. The speed up interval of DRV(A) is $[6s, 7s]$, which is the difference from its annotation interval $[0s, 7s]$, and the union of the annotations intervals of DRV(B) and RCV(+E) ($[0s, 4s] \cup [4s, 6s]$). Speed up intervals have the following two properties:

Lemma 11 The speedup interval I_v^δ of vertex v always ends at $t_e(v)$, where $t_e(v)$ is when the execution of v finishes or the end timestamp of v (Section 3.3).

Proof. The annotation interval of v always ends at $t_e(v)$ (Lemma 7). We prove that that I_v^δ ends when I_v^α ends. If v has no child, $I_v^\delta = I_v^\alpha$; if v has children, the two WHILE loops in the algorithm in Figure 6 distribute the interval between t_s and $t_s(v)$ to other vertices via recursive ANNOTATE calls, and the remaining interval in $[t_s, t_e]$ is the speedup interval; in either case, I_v^δ ends when I_v^α ends, and therefore, I_v^δ ends at $t_e(v)$. \square

Lemma 12 Given the temporal provenance of T-QUERY(e', e), consider the set of all speedup intervals $\{I_{v_i}^\delta\}$: (a) $\bigcap_i I_{v_i}^\delta = \emptyset$; (b) $\bigcup_i I_{v_i}^\delta = I_e^\alpha$.

Proof. In a temporal provenance graph, vertices with annotation intervals form a tree, because vertex v is annotated at most once (Lemma 9) by a parent of v . Consequently, vertices with speedup intervals form a tree (Definition A.6). We prove by structural induction on the height of the tree.

Base case: The height of the tree is one. Denote the root vertex as v . The set of speedup intervals has one element ($\{I_{v_i}^\delta\} = I_v^\delta$), condition (a) holds. $I_v^\delta = I_v^\alpha$ because v has no child (Definition A.6), condition (b) holds.

Induction case: Suppose the conditions hold for trees (of vertices with annotation or speedup intervals) with height less than k ($k \geq 1$). Consider a tree with depth $k+1$, rooted at vertex v . Without loss of generality, denote $T(v_1)$ and $T(v_2)$ as subtrees of v that have annotation intervals. Note that the speedup intervals of vertices in $T(v)$ must be a subinterval of the annotation interval of v , because: the speedup interval is simply the difference of the annotation interval of v and the annotation intervals of the children of v (Definition A.6); the annotation interval of v is a subinterval of the annotation interval of its parent in the tree (Lemma 8).

It follows from Definition A.6 that the speedup interval of v and that of $T(v_1)$ (or $T(v_2)$) cannot overlap. It follows from Lemma 6 that the speedup intervals of $T(v_1)$ and $T(v_2)$ cannot overlap. It follows from the induction hypothesis that the speedup intervals within $T(v_1)$ (or $T(v_2)$) cannot overlap. Therefore, condition (a) holds. It follows from Definition A.6 that condition (b) holds. \square

Intuitively, the above two lemmas states that, given a temporal provenance that explains T-QUERY(e', e), the overall delayed interval – that is, $[t_s(e'), t_e(e)]$ – can be subdivided into a sequence of all speedup intervals $\{I_{v_i}^\delta\}$. In the example provenance from Figure 16(a), such a sequence of speedup intervals is $[0s, 2s]@DRV(C)$, $[2s, 4s]@DRV(B)$, $[4s, 5.5s]@DRV(E)$, $[5.5s, 6s]@RCV(+E)$, and $[6s, 7s]@DRV(A)$.

Definition (Terminal event): A vertex v is a terminal event if any of the following conditions holds:

- (a) if v is annotated, v ends at $t_e(e)$, and $I_v^\delta = \emptyset$ (a vertex is annotated if it is associated with an annotation interval in the original G);
- (b) if v is not annotated, on any path in G from v to e , select u as the first annotated vertex, $I_u^\alpha = \emptyset$.

$A@X :- B@X, E@Y$
 $B@X :- C@X$
 $E@Y :- C@X$
 $C@X :- Z@X$

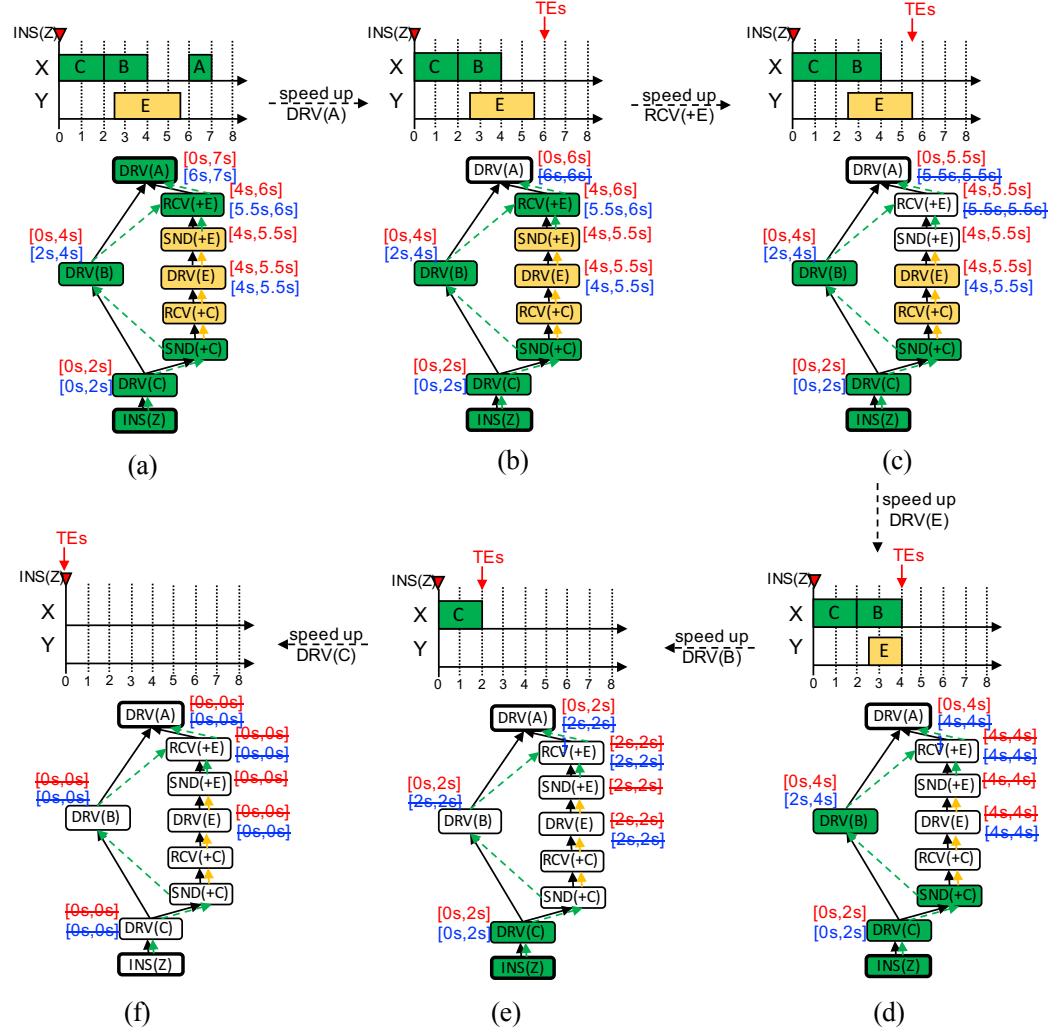


Figure 16: An example of “speeding up” temporal provenance using a series of transformations (Definition A.6). The NDlog rules are at the top left. Each sub-figure shows a step of the transformation ((a) → ... → (g)). In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is $\text{T-QUERY}(\text{INS}(Z), \text{DRV}(A))$ in all sub-figures; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition A.5) and speed up interval (blue, Definition A.6). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertices are terminal events.

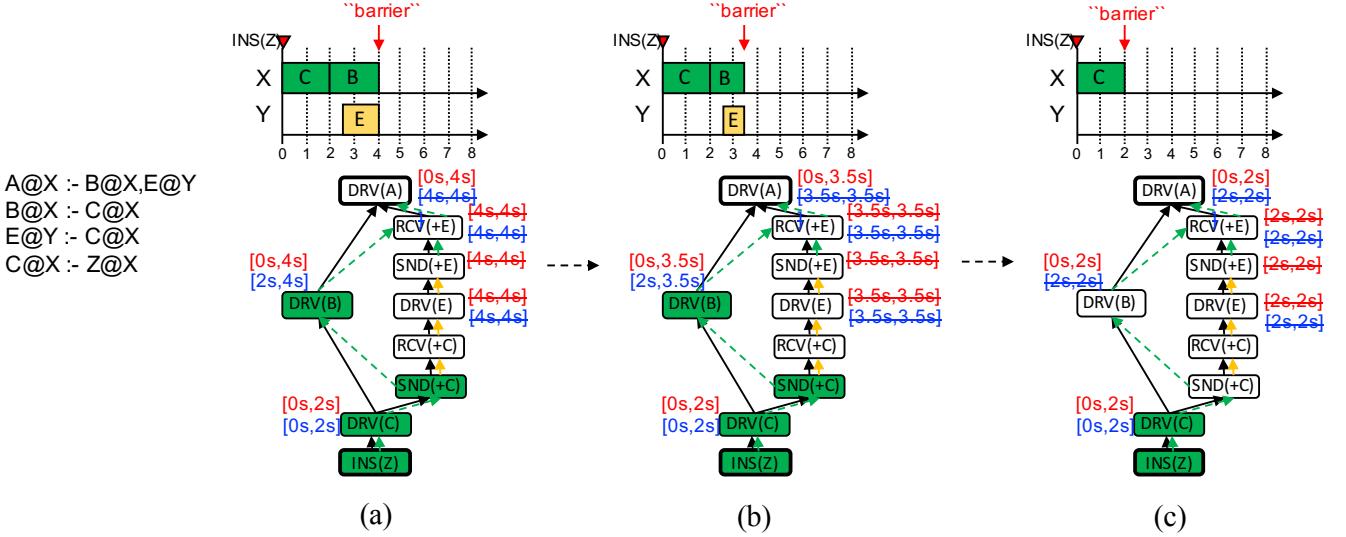


Figure 17: An example of “speeding up” temporal provenance using an annotated vertex $\text{DRV}(B)$ (Definition A.6). The NDlog rules are at the left. In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is $\text{T-QUERY}(\text{INS}(Z), \text{DRV}(A))$ in all sub-figures; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition A.5) and speed up interval (blue, Definition A.6). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertices are terminal events.

Intuitively, terminal events represent executions that no longer contribute any delay (in a hypothetical execution). Condition (a) describes an event that finishes at the end of the entire execution and that no longer contributes any delay. For example, $\text{RCV}(+E)$ in Figure 16(c) is a terminal event: it was annotated in the original provenance (Figure 16(a)); it ends at $t = 5.5s$, which is the end timestamp of $\text{DRV}(A)$; and its speedup interval is empty. Condition (b) describes an event that only belongs to subgraphs that no longer contribute any delay. For example, $\text{RCV}(C)$ in Figure 16(e) is a terminal event: it was not annotated in the original provenance (Figure 16(a)); on its (only) path to $\text{DRV}(A)$, the first annotated vertex is $\text{DRV}(E)$, whose annotation interval is already empty ($[4s, 4s]$). Next, we describe steps to transform the original execution to hypothetical executions.

Definition (Speed up): Given a vertex v in $G(e', e, \mathcal{E})$, where $t_e(v) = t_e(e)$ and $I_v^\delta > 0$, v speeds up G by I_v^δ using the following procedure. Consider a “barrier” t_b that moves on the timeline; it starts from the right boundary of I_v^δ and moves leftwards (and thus t_b becomes smaller); it stops when it reaches the left boundary of I_v^δ . For ease of exposition, we say that the “barrier” pushes a timestamp t when we set t to $\min(t, t_b)$. During its move, if the “barrier” encounters a vertex v_i that is either v or a terminal event, it transforms v_i by pushing these timestamps: (a) the starting timestamp (or the ending timestamp) of v_i , (b) the left boundary (or the right boundary) of $I_{v_i}^\alpha$ (if any); and (c) the left boundary (or the right boundary) of $I_{v_i}^\delta$ (if any).

Intuitively, the “speed up” operation represents a transformation step that essentially “squeezes” a set of vertices to the left. Note that, while v speeds up G , only v itself and terminal events – vertices that no longer contribute any delay – are pushed leftwards. For example, Figure 17 shows the process of speeding up the provenance using $\text{DRV}(B)$: the “barrier” starts from the right boundary of $I_{\text{DRV}(A)}^\delta$ (Figure 17(a)); while it moves, the “barrier” pushes $\text{DRV}(B)$ as well as terminal events $\text{DRV}(E)$ and $\text{RCV}(C)$ leftwards (Figure 17(b) shows the snapshot of $t_b = 3.5s$); the “barrier” stops at the left boundary of $I_{\text{DRV}(A)}^\delta$ (Figure 17(c)).

Figure 16 shows the process of speeding up an entire provenance graph until it becomes instantaneous. Next, we briefly show the effect of each “speed up” operation:

- (a) \rightarrow (b), $\text{DRV}(A)$ speeds up G by $I_{\text{DRV}(A)}^\delta = [6s, 7s]$: the execution is shortened to $[0s, 6s]$, $\text{DRV}(A)$ becomes a terminal event;
- (b) \rightarrow (c), $\text{RCV}(+E)$ speeds up G by $I_{\text{RCV}(+E)}^\delta = [5.5s, 6s]$: the execution is shortened to $[0s, 5.5s]$, $\text{RCV}(+E)$ and $\text{SND}(+E)$ become terminal events;
- (c) \rightarrow (d), $\text{DRV}(E)$ speeds up G by $I_{\text{DRV}(E)}^\delta = [4s, 5.5s]$: the execution is shortened to $[0s, 4s]$, $\text{DRV}(E)$ and $\text{RCV}(+C)$ become terminal events;
- (d) \rightarrow (e), $\text{DRV}(B)$ speeds up G by $I_{\text{DRV}(B)}^\delta = [2, 4s]$: the execution trace is shortened to $[0s, 2s]$, $\text{DRV}(B)$ becomes terminal events;
- (e) \rightarrow (f), $\text{DRV}(C)$ speeds up G by $I_{\text{DRV}(C)}^\delta = [0, 2s]$: the execution trace is shortened to $[0s, 0s]$, all events are now terminal events.

Definition (Well-annotated): Consider an annotated temporal provenance graph $G(e', e, \mathcal{E})$. G is well-annotated iff either (a) $t_s(e') = t_e(e)$, that is, the entire execution is instantaneous; (b) we can transform G into another valid and well-annotated temporal provenance graph G' by locating an unique vertex v , where $t_e(v) = t_e(e)$ and $I_v^\delta > 0$, and speeding up G by v (Definition A.6).

Theorem 13 Temporal provenance is well-annotated.

Proof. Consider the speedup intervals $\{I_{v_i}^\delta\}$ of G . It follows from Lemma 12 that $\{I_{v_i}^\delta\}$ do not overlap and unions to $[t_s(e'), t_e(e)]$. Therefore, we can sort intervals in $\{I_{v_i}^\delta\}$ by descending (ending) timestamp. At the i th step, we speed up G by v_i . We need to prove that: (a) each “speed up” operation pushes the timestamps of all events that ends during $I_{v_i}^\delta$; (b)

$A@X \Rightarrow B@X, E@Y$
 $B@X \Rightarrow C@X$
 $E@Y \Rightarrow C@X$
 $C@X \Rightarrow Z@X$

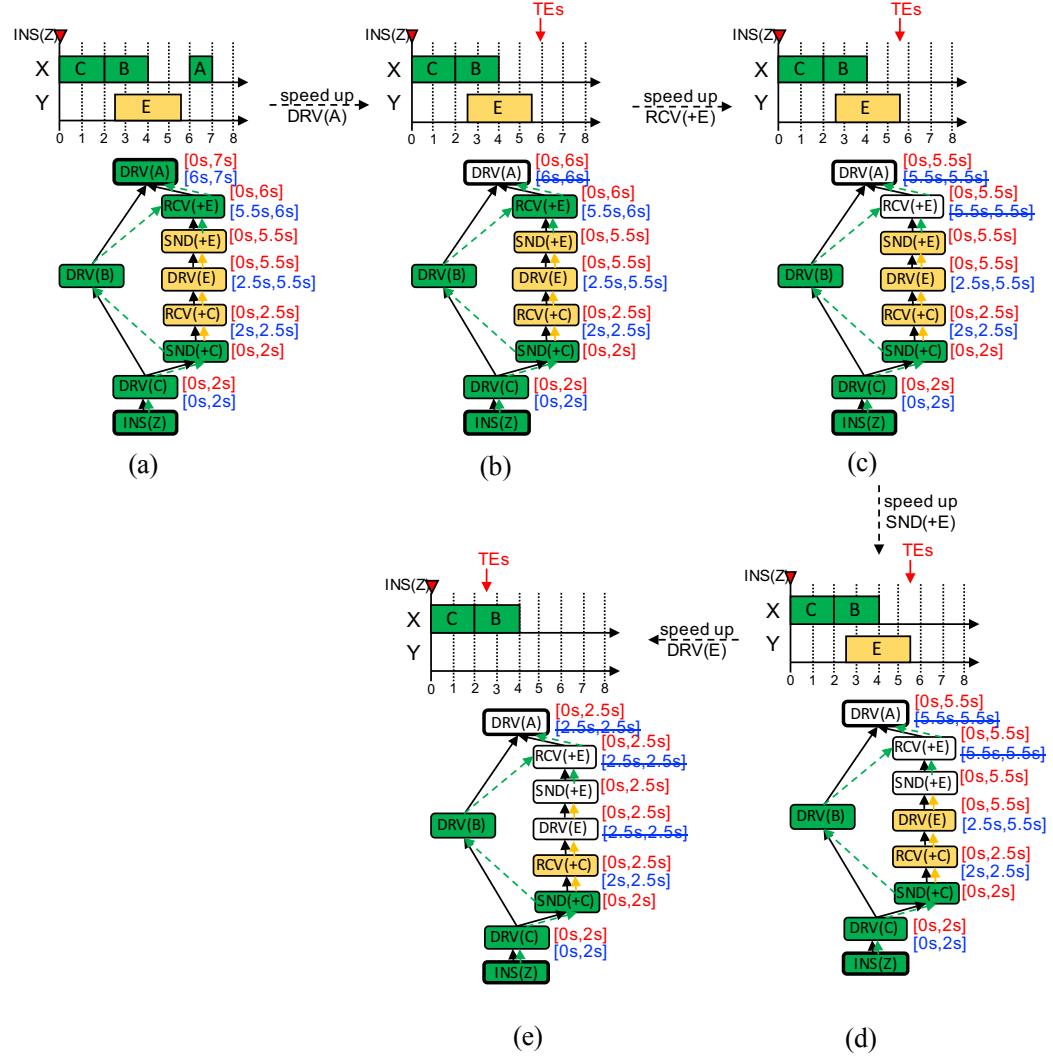


Figure 18: An example of “poorly annotated” temporal provenance. The NDlog rules are at the top left. Each sub-figure shows a step of the transformation ((a) → ... → (g)). In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is $T\text{-QUERY}(INS(Z), DRV(A))$ in all sub-figures; the start and end vertices are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition A.5) and speed up interval (blue, Definition A.6). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertices are terminal events.

the length of the execution $[t_s(e'), t_e(e)]$ is reduced by the length of $I_{v_i}^\delta$; (c) the temporal provenance remains valid.

To prove condition (a), given any vertex v'_i that ends during $I_{v_i}^\delta$, we perform a case analysis of v'_i :

- $v'_i = v_i$: the timestamps of $v'_i = v_i$ is pushed, by the construction of Definition A.6.
- $v'_i \neq v_i$ and v'_i is annotated in the original provenance: v'_i must be a terminal event, and therefore, its timestamps is pushed. Because $I_{v'_i}^\delta$ ends when v'_i ends (Lemma 11); consequently, $I_{v'_i}^\delta$ must end during $I_{v_i}^\delta$; if $I_{v'_i}^\delta$ is not empty, $I_{v'_i}^\delta$ will overlap with $I_{v_i}^\delta$, which contradicts with Lemma 12.
- $v'_i \neq v_i$ and v'_i is not annotated in the original provenance: v'_i must be a terminal event, and therefore, its timestamps is pushed. Because, given any path from v'_i to e , consider the first annotated ancestor u and its child on the path w ; if we assume that I_u^α is not empty when the “barrier” reaches the end of w , then I_u^α must start before the end of w ; by construction of the algorithm from Figure 6, w must be annotated by u , which contradicts the fact that w is not annotated.

Condition (b) follows directly from the statement above: the execution is shortened by the length of $I_{v_i}^\delta$, because all events that end during $I_{v_i}^\delta$ are pushed leftwards until the left boundary of $I_{v_i}^\delta$.

Condition (c) holds because the “speed up” operation does not invert causality: if an event a caused another event b , it does not alter the ordering of a and b ; nor does it delete any event. \square

Note that Definition A.6 weeds out some annotation approaches. For example, Figure 18 shows how a straw-man approach that associates the entire delay with the last precondition would have annotate the same provenance graph in Figure 16. The result is not well annotated: while $\text{DRV}(E)$ speeds up G ((d) \rightarrow (e)), another vertex $\text{DRV}(B)$ becomes the bottleneck; however, $\text{DRV}(B)$ cannot be pushed leftwards, because it is not a terminal event, that is, it has not been “sped up”.

Theorem 13 suggests that the annotations on temporal provenance do correspond to the “potential for speedup” that one may intuitively associate with the concept of delay. This is useful, because, while the temporal provenance maybe gigantic and complex, operators can focus on vertices with annotations and gain a comprehensive understanding of the end-to-end delay, including potential operations to speed up.

Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks

Anurag Khandelwal
UC Berkeley

Rachit Agarwal
Cornell University

Ion Stoica
UC Berkeley

Abstract

Confluo is an end-host stack that can be integrated with existing network management tools to enable monitoring and diagnosis of network-wide events using telemetry data distributed across end-hosts, even for high-speed networks. Confluo achieves these properties using a new data structure — Atomic MultiLog — that supports highly-concurrent read-write operations by exploiting two properties specific to telemetry data: (1) once processed by the stack, the data is neither updated nor deleted; and (2) each field in the data has a fixed pre-defined size. Our evaluation results show that, for packet sizes 128B or larger, Confluo executes thousands of triggers and tens of filters at line rate (for 10Gbps links) using a single core.

1 Introduction

Recent years have witnessed tremendous progress on (the notoriously hard problem of) network monitoring and diagnosis by exploiting programmable network hardware [1–18]. This progress has been along two complementary dimensions. First, elegant data structures and interfaces have been designed that enable capturing increasingly rich telemetry data at network switches [1–6, 10, 13–17]. On the other hand, recent work [6–12] has shown that capitalizing on the benefits of above data structures and interfaces does not need to be gated upon the availability of network switches with large data plane resources — switches can store a small amount of state to enable in-network visibility, and can embed rich telemetry data in the packet headers; individual end-hosts monitor local packet header logs for monitoring spurious network events. When a spurious network event is triggered, network operator can diagnose the root cause of the event using switch state along with packet header logs distributed across end-hosts [7–10].

Programmable switches have indeed been the enabling factor for this progress — on design and implementation of novel interfaces to collect increasingly rich telemetry data, and on flexible packet processing to embed this data into the packet headers. To collect these packet headers and to use

them for monitoring and diagnosis purposes, however, we need end-host stacks that can support:

- **monitoring of rich telemetry data** embedded in packet headers, *e.g.*, packet trajectory [7–11], queue lengths [1, 10], ingress and egress timestamps [10], etc. (§2.2);
- **low-overhead diagnosis** of network events by network operator, using header logs distributed across end-hosts;
- **highly-concurrent low-overhead read-write operations** for capturing headers, and for using the header logs for monitoring and diagnosis purposes using minimal CPU resources. The challenge here is that, depending on packet sizes, monitoring headers at line rate even for 10Gbps links requires 0.9–16 million operations per second!

Unfortunately, end-host monitoring and diagnosis stacks have not kept up with advances in programmable hardware and are unable to simultaneously support these three functionalities (§2.1, §6). Existing stacks that support monitoring of rich telemetry data (*e.g.*, OpenSOC [19], Tigon [20], Gigascope [21], Tribeca [22] and PathDump [8]) use general-purpose streaming and time-series data processing systems; we show in §2.1 that these systems are unable to sustain the target throughput even for 10Gbps links. This limitation has motivated design of stacks (*e.g.*, Trumpet [23]) that can monitor traffic at 10Gbps using a single core, but only by limiting the functionality — they do not support monitoring of even basic telemetry data like packet trajectory and queue lengths; we discuss in §2.1 that this is in fact a fundamental design constraint in these stacks.

Confluo is an end-host stack, designed and optimized for high-speed networks, that can be integrated with existing network management tools to enable monitoring and diagnosis of network-wide events using telemetry data distributed across end-hosts. Confluo simultaneously supports the above three functionalities by exploiting two properties specific to telemetry data and applications. First, telemetry data has a special structure: once headers are processed in the stack, these headers are not updated and are only aggregated over

long time scales. Second, unlike traditional databases where each record may have fields of arbitrary size, packet headers capture a precise protocol with fixed field sizes (*e.g.*, 32-bit IP addresses, 16-bit port numbers, 16-bit switchIDs [8–10], 16-bit queue lengths [1, 10], 32-bit timestamps [10], etc.)¹.

Confluo achieves its goals using a new data structure — Atomic MultiLog — that exploits the above two properties of telemetry data to trim down traditional lock-free concurrency mechanisms to a bare minimum without sacrificing correctness guarantees. A MultiLog, as the name suggests, generalizes traditional logs into a *collection* of lock-free logs. Atomic MultiLog uses a collection of such logs, one for each of the filters and aggregates (for monitoring purposes), one for each of the materialized views (for diagnosis purposes), and one for raw header logs. Atomic MultiLogs use the first property to efficiently maintain an updated view of these logs upon receiving new headers (each new header may incur multiple concurrent write operations on Atomic MultiLog for updating individual logs). Essentially, we show that the first property allows trimming down the traditional lock-free concurrency mechanisms to updating two integers per header (§3); using atomic hardware primitives readily available in commodity servers, Atomic MultiLog is able to ingest millions of headers per second using a single CPU core.

As headers are processed in the stack, Confluo also needs to simultaneously execute monitoring and diagnosis queries that, in turn, require executing multiple concurrent read operations on Atomic MultiLogs. We show that having fixed field sizes in packet headers makes it extremely simple to handle race conditions for concurrent reads and writes over individual logs within an Atomic MultiLog. Finally, we show that these two properties allow Atomic MultiLog to not only achieve highly-concurrent read and write operations but to also support two strong distributed systems properties. First, updates to all the individual logs within an Atomic MultiLog are visible to the monitoring and diagnosis application atomically (formal proofs in [24]); and second, atomic snapshots of telemetry data distributed across the end-hosts can be obtained using a simple distributed algorithm (§4).

Confluo implementation is now open-sourced [25], with an API that is expressive enough to integrate Confluo with most existing end-host based monitoring and diagnosis systems [8–11, 23]. We have compiled an exhaustive list of monitoring and diagnosis applications from these systems; we show, in [24], that our implementation already supports all these applications. Evaluation of Confluo using packet traces from standard generators [26, 27], and from real testbeds [8, 9] shows that, even for 128B packets, Confluo executes thousands of triggers and tens of filters at line rate (for 10Gbps links) using a single core. Moreover, for 40Gbps links and beyond, where multiple cores may be necessary, Confluo’s performance scales well with number of cores.

¹Packet headers can contain arbitrary number of fields, and the number of fields may vary across each packet; however, each field has a fixed size.

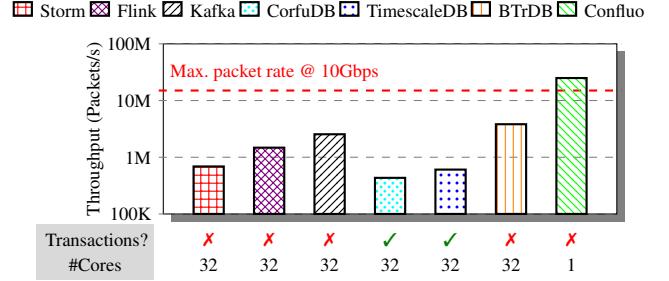


Figure 1: Header ingestion rates (no filters, aggregates, or indexes) for several open-sourced streaming and time-series data processing systems, and for Confluo, on a single end-host. The workload uses 64B TCP packets using DPDK’s pktgen tool [28]. Unfortunately, existing systems are unable to sustain write rates for 10Gbps links, even when using 32 cores. Note that: (1) CorfuDB and TimescaleDB tradeoff write rates for stronger semantics; (2) BTrDB results use 16B packet prefixes since it does not support larger entries; (3) Storm and Flink results use Kafka as a data sink since these systems do not store data. See §2.1 for discussion.

2 Confluo Overview

This section provides an overview of Confluo. We start by elaborating on the observation that end-host monitoring and diagnosis stacks have not kept up with increasing network bandwidths and with advances in programmable network hardware (§2.1). We then outline Confluo interface, along with an example on how a network operator can use this interface for monitoring and diagnosis (§2.2). We conclude the section with a high-level overview of Confluo design (§2.3).

2.1 Motivation

Existing end-host stacks fall short of simultaneously supporting the three functionalities outlined in the introduction either because they cannot scale to large network bandwidths (10Gbps and beyond), or do not support monitoring of rich telemetry data (*e.g.*, packet trajectory, queue lengths, ingress and egress timestamps, and many others outlined in [10]). We discuss these challenges next.

Challenges with larger network bandwidths. Existing end-host monitoring stacks that support rich telemetry data (*e.g.*, Time Machine [29], Gigascope [21], Tribeca [22]) were designed for 1Gbps links, with reported performance of 180–610 Mbit/sec [21] and 20–30k headers/sec [22]. While these systems are not available for evaluation, they are unlikely to scale to 10Gbps and higher link bandwidths since this would require processing 10–100× more headers. To overcome this limitation, recently developed stacks [8, 9, 19, 20] use open-source streaming and time-series data processing systems. However, as shown in Figure 1, these systems are unable to support write rates at 10Gbps even when using 32 cores. We believe that the fundamental reason behind this limitation is that these systems are targeting data types that are too general — supporting the three functionalities outlined in the introduction with minimal CPU resources requires exploiting the

Table 1: Confluo’s End-Host API. In addition, Confluo exposes certain API to the coordinator to facilitate distributed snapshot (§4). All supported operations are guaranteed to be atomic. See §2.2 for definitions and detailed discussion.

	API	Description
Monitoring	<code>setup_packet_capture(fExpression, sampleRatio)</code>	Capture packet headers matching filter <code>fExpression</code> at <code>sampleRatio</code> .
	<code>filterId = add_filter(fExpression)</code>	Add filter <code>fExpression</code> on incoming packet headers.
	<code>aggId = add_aggregate(filterId, aFunction)</code>	Add aggregate <code>aFunction</code> on headers filtered by <code>filterId</code> .
	<code>trigId = install_trigger(aggId, condition, period)</code> , <code>remove_filter(filterId), remove_aggregate(aggId),</code> <code>uninstall_trigger(trigId)</code>	Install trigger over aggregate <code>aggId</code> evaluating <code>condition</code> every <code>period</code> .
Diagnosis	<code>add_index(attribute)</code>	Add an index on a packet header <code>attribute</code> .
	<code>Iterator<Header> it = query(fExpression, tLo, tHi)</code>	Filter headers matching <code>fExpression</code> during time <code>(tLo, tHi)</code> .
	<code>agg = aggregate(fExpression, aFunction, tLo, tHi)</code>	Compute aggregate <code>aFunction</code> on headers matching <code>fExpression</code> during time <code>(tLo, tHi)</code> .
	<code>remove_index(attribute)</code>	Remove index for specified packet header <code>attribute</code> .

Table 2: Elements of Confluo filters, aggregates and triggers.

	Operator	Examples
Relational	Equality	<code>dstPort==80</code>
	Range	<code>ipTTL<3, srcIP in 10.1.3.0/24</code>
	Wildcard	<code>dstIP like 192.*.*.1</code>
Boolean	Conjunction	<code>srcIP=10.1.3.2 && pktSize<100B</code>
	Disjunction	<code>dstPort==80 dstPort==443</code>
	Negation	<code>protocol!=TCP</code>
Aggregate	AVG	<code>AVG(ipTTL)</code>
	COUNT, SUM	<code>COUNT(ecn), SUM(pktSize)</code>
	MAX, MIN	<code>MIN(ipTOS), MAX(tcpRxWin)</code>

specific structure in network packet headers, especially for 40-100Gbps links where multiple cores may be necessary to process packet headers at line rate.

Challenges with monitoring rich telemetry data. The aforementioned limitations of streaming and time-series data processing systems have motivated custom-designed end-host monitoring stacks [23, 30–34]. State-of-the-art among these stacks (*e.g.*, Trumpet [23] and FloSIS [34]) can operate at high link speeds — Trumpet enables monitoring at line rate for 10Gbps links using a single core; similarly, FloSIS can support offline diagnosis for up to 40Gbps links using multiple cores. However, these systems achieve such high performance either by giving up on online monitoring (*e.g.*, FloSIS) or by applying filters only on the first packet in the flow (*e.g.*, Trumpet). This is a rather fundamental limitation and severely limits how rich telemetry data embedded in the packet headers is utilized — for instance, since header state (*e.g.*, trajectories or timestamps) may vary across packets, monitoring and diagnosing network events requires applying filters to each packet [6, 8, 9, 18]. For instance, if a packet is rerouted due to failures or bugs, its trajectory in the header could be used to raise an alarm [8, 9, 18]; however, if this is not the first packet in the flow, optimizations like those in

Trumpet will fail to trigger this network event². On the other hand, if filters were applied to each and every packet, these systems will observe significantly worse performance.

2.2 Confluo Interface

We now describe Confluo interface. Confluo is designed to integrate with existing tools that require a high-performance end-host stack [8, 9, 11, 12, 23]. To that end, Confluo exposes an interface that is expressive enough to enable integration with most existing tools; we discuss, in [24], that Confluo interface already allows implementing all applications from recent end-host monitoring and diagnosis systems.

Confluo operates on packet headers, where each header is associated with a number of attributes that may be protocol-specific (*e.g.*, attributes in TCP header like `srcIP`, `dstIP`, `rwnd`, `ttl`, `seq`, `dup`) or custom-defined (*e.g.*, packet trajectories [8, 9, 11], or queue lengths [1, 10], timestamps [10], etc.). Confluo does not require packet headers to be fixed; each header can contain arbitrary number of fields, and the number of fields may vary across each packet.

API. Table 1 outlines Confluo’s end-host API. While Confluo captures headers for all incoming packets by default, it can be configured to only capture headers matching a filter `fExpression`, sampled at a specific `sampleRatio`.

Confluo uses a match-action language similar to [8, 23] with three elements: *filters*, *aggregates* and *triggers*. A filter is an expression `fExpression` comprising of relational and boolean operators (Table 2) over an arbitrary subset of header attributes, and identifies headers that match `fExpression`. An aggregate evaluates a computable function (Table 2) on an attribute for all headers that match a certain filter expression. Finally, a trigger is a boolean condition (*e.g.*, `<`, `>`, `=`, etc.) evaluated over an aggregate.

²For some applications, detecting such cases may be necessary due to privacy laws. The canonical example here is that of a bug leading to incorrect packet forwarding and violating isolation constraints in datacenters storing patient information — patient data from two healthcare providers must never share the same network element due to HIPAA laws [35, 36]

Scenarios:

- Scenario 1:** flow1 rate + flow2 rate > bandwidth, flow1 priority = flow2 priority. Packet drops for flow1, flow2 at S.
- Scenario 2:** flow1 rate + flow2 rate > bandwidth, flow1 priority < flow2 priority. Packet drops for flow1 at S.
- Scenario 3:** flow1 rate + flow2 rate < bandwidth, bug at S drops based on packet timing. Packet drops for flow1, flow2 at S.

Monitoring Code (C/C++):

```

Tracking retransmissions (rtms):
MAXSEQ((maxSeq, maxTs), pkt) {
    if (pkt.seqNo > maxSeq)
        return (pkt.seqNo, pkt.ts)
    else return (maxSeq, maxTs)
}

SEQ, TS=add_aggregate(flow, MAXSEQ)
cond = seqNo<SEQ && ts>TS+tdelay
rtms = add_filter(cond)
R = add_aggregate(rtms, COUNT)
T = add_trigger(R, R>T, 1ms)

```

Diagnosis Code (C/C++):

```

t = T.timestamp,
p1 = flow1 priority, p2 = flow2 priority
r1 = flow1 retransmits, r2 = flow2 retransmits,
c1 = aggregate(r1, COUNT, t-1ms, t),
c2 = aggregate(r2, COUNT, t-1ms, t),
check if c1 ≈ c2 > 0 && p1 = p2
t, r1, r2, c1, c2, p1, p2 → Same as above
check if c1 ≈ 0 && c2 > 0 && p1 < p2
or, c2 ≈ 0 && c1 > 0 && p2 ≤ p1
ti = Timestamp buckets of packets in rtms,
δi = ti - ti-1 and σδ = STDEV on δi
check if AVG(δi) ≈ 100ms && σδ < 1ms

```

Figure 2: Examples of monitoring and diagnosis of network events in Confluo. See §2.2 for details.

Confluo supports ad-hoc filter queries and aggregates via indexes on arbitrary packet header attributes. These indexes serve to speed up diagnostic queries when filters or aggregates have not been pre-defined. We describe the design and implementation of Confluo indexes, filters, aggregates and triggers in §3.2 and §3.3.

Examples. Figure 2 shows Confluo functionality using a simple example comprising three scenarios where switch S is dropping packets. This example assumes that the monitoring and diagnosis application employing Confluo uses TCP retransmissions as an indicator of packet loss. A network operator can use Confluo to maintain an aggregate to determine the latest TCP sequence number SEQ and the corresponding packet timestamp TS in a flow. The operator then filters out packets that have TCP sequence number smaller than SEQ and timestamp larger than TS by a delay threshold (t_{delay}) as probable retransmissions. Confluo can then be configured to trigger an alarm if estimated retransmission count exceeds a limit. Confluo also allows the operator to issue diagnostic queries to the relevant end-hosts to determine priorities of involved flows, their retransmission counts, and periodicity of retransmissions during the relevant time-period to distinguish between the three scenarios based on observed values.

2.3 Confluo Design Overview

We now provide an overview of Confluo design (Figure 3), that comprises a central coordinator interface and an end-host module at each end-host in the network.

Coordinator Interface. Confluo’s coordinator interface allows monitoring and diagnosing network-wide events by delegating monitoring responsibilities to Confluo’s individual end-host modules, and by providing the diagnostic information from individual modules to the network operator. An operator submits *control programs* composed of Confluo API calls to the coordinator, which in turn contacts relevant end-host modules and coordinates the execution of Confluo API calls via RPC. The coordinator API also allows obtaining distributed atomic snapshots of telemetry data distributed across the end-hosts (§4).

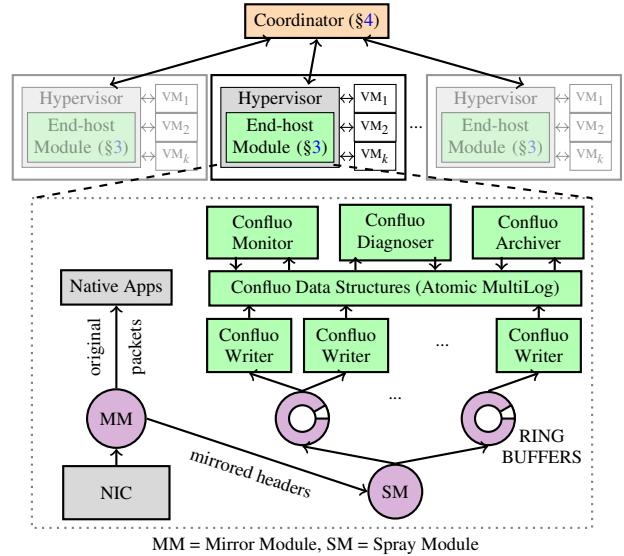


Figure 3: High-level Confluo Architecture (§2).

End-host Module. Confluo conducts bulk of monitoring and diagnosis operations at the end-hosts. Confluo captures and monitors packets in the hypervisor, where a software switch could deliver packets between NICs and VMs. A *mirroring module* mirrors packet headers to a *spray module*, that writes these headers to one of multiple ring buffers in a round-robin manner. Confluo currently uses DPDK [37] to bypass the kernel stack, and Open vSwitch [38] to implement the mirror and spray modules. This choice of implementation is merely to perform our prototype evaluation without the overheads of existing cloud frameworks (e.g., KVM or Xen); our implementation on OVS trivially allows us to integrate Confluo with these frameworks.

Confluo’s end-host module makes two important architectural choices. First, as outlined in §1, Confluo optimizes for highly-concurrent operations, potentially from multiple cores processing different packet streams, at the end-host. To that end, Confluo uses multiple ring buffers so that downstream modules can keep up with incoming headers. Multiple Confluo writers read headers from these ring buffers and write them to Confluo data structures. Achieving high throughput with multiple Confluo writers requires highly

concurrent write operations. This is where Confluo’s new data structure — Atomic MultiLog — makes its key contribution. Recall from §1 that Atomic MultiLog exploits two unique properties of network logs — append-only workload and fixed field sizes for each header attribute — to minimize the overheads of traditional lock-free concurrency mechanisms while providing atomicity guarantees. We describe the design and implementation of Atomic MultiLogs in §3.

The second architectural decision is to separate threads that “read” from, and that “write” to Atomic MultiLog. Specifically, read threads in Confluo implement monitoring functionality (that requires evaluating potentially thousands of triggers on each header) and on-the-fly diagnosis functionality (that requires evaluating ad-hoc filters and aggregates using header logs and materialized views). The write threads, on the other hand, are the Confluo writers described above. This architectural decision is motivated by two observations. First, while separating read and write threads in general leads to more concurrency issues, Atomic MultiLog provides low-overhead mechanisms to achieve highly concurrent reads and writes. Second, separating read and write threads also require slightly higher CPU overhead (less than 4% in our evaluation even for a thousand triggers per packet); however, this is a good tradeoff to achieve on-the-fly diagnosis, since interleaving reads and writes within a single thread may lead to packet drops when complex ad-hoc filters need to be executed (§3).

Atomic MultiLogs guarantee that all read/write operations corresponding to an individual header become visible to the application atomically. However, due to a number of reasons (*e.g.*, different queue lengths on the NICs during packet capturing, random CPU scheduling delays, etc.), the ordering of packets visible at an Atomic MultiLog may not necessarily be the same as ordering of packets received at the NIC. One easy way to overcome this problem, that Confluo naturally supports, is to use ingress/egress NIC timestamps to order the updates in Atomic MultiLog to reflect the ordering of packets received at the NIC; almost all current generation 10Gbps and above NICs support ingress and egress packet timestamps at line rate. Without exploiting such timestamps or any additional information about packet arrival ordering at the NIC, unfortunately, this is an issue with any end-host based monitoring and diagnosis stack.

Distributed Diagnosis. Confluo supports low-overhead diagnosis of spurious network events even when diagnosing the event requires telemetry data distributed across multiple end-hosts [8–11]. Diagnosis using telemetry data distributed across multiple end-hosts leads to the classical consistency problems from distributed systems — unless all records (packets in our case) go through a central sequencer, it is impossible to achieve an absolutely perfect view of the system state. Confluo does not attempt to solve this classical problem, but rather shows that by exploiting the properties

of telemetry data, it is possible to simplify the classical distributed atomic snapshot algorithm to a very low-overhead one (§4). This is indeed the strongest semantics possible without all packets going through a central sequencer.

3 Confluo Design

We now describe the design for Confluo end-host module (see Figure 3), that comprises of packet processing (mirror and spray) modules, multiple concurrent Confluo writers, the Atomic MultiLog, Confluo monitor, diagnoser and archival modules. We discussed the main design decisions made in the packet processing and writer modules in §2.3. We now focus on the Atomic MultiLog (§3.1, §3.2) and the remaining three modules (§3.3, §3.4).

3.1 Background

We briefly review two concepts from prior work that will be useful in succinctly describing the Atomic MultiLog.

Atomic Hardware Primitives. Most modern CPU architectures support a variety of atomic instructions. Confluo will use four such instructions: `AtomicLoad`, `AtomicStore`, `FetchAndAdd` and `CompareAndSwap`. All four instructions operate on 64 bit operands. The first two permit atomically reading from and writing to memory locations. `FetchAndAdd` atomically obtains the value at a memory location and increments it. Finally, `CompareAndSwap` atomically compares the value at a memory location to a given value, and only if they are equal, modifies the value at the memory location to a new specified value.

Concurrent Logs. There has been a lot of prior work on design of efficient, lock-free concurrent logs [39–42] that exploit the append-only nature in many applications to support high-throughput writes. Intuitively, each log maintains a “`writeTail`” that marks the end of the log. Every new append operation increments the `writeTail` by the number of bytes to be written, and then writes to the log. Using the above hardware primitives to atomically increment the `writeTail`, these log based data structure support extremely high write rates.

It is easy to show that by additionally maintaining a “`readTail`” that marks the end of completed append operations (and thus, always lags behind the `writeTail`) and by carefully updating the `readTail`, it is possible to guarantee *atomicity* for concurrent reads and writes on a single log (see [24] for a formal proof). Using atomic hardware primitives to update both `readTail` and `writeTail`, it is possible to achieve high throughput for concurrent reads *and* writes for such logs.

3.2 Atomic MultiLog

An Atomic MultiLog uses a collection of concurrent lock-free logs to store packet header data, packet attribute indexes, aggregates and filters defined in §2.2 (see Figure 4). As outlined earlier, Atomic MultiLog exploit two unique properties of network logs to facilitate this:

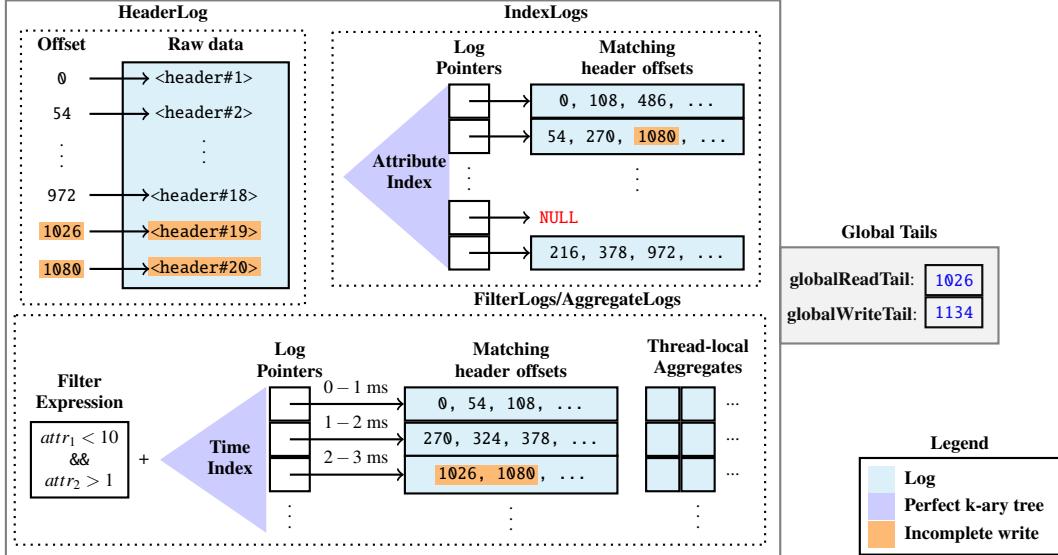


Figure 4: The Atomic MultiLog uses a collection of concurrent lock-free logs to store packet headers, indexes, aggregates and filters (as defined in §2.2) and efficiently updates these data structures as new packet headers arrive. See §3.2 for details.

- **Property 1:** Packet headers, once processed by the stack, are not updated and only aggregated over long time scales.
- **Property 2:** Each packet header attribute has a fixed size (number of bits used to represent the attribute)

HeaderLog. This concurrent append-only log stores the raw data for all captured packet headers in Confluo. Each packet header in the HeaderLog has an offset, which is used as a unique reference to the packet across all data structures within the Atomic MultiLog. We will discuss in §3.2.1 how this simplifies guaranteeing atomicity for operations that span multiple data structures within the Atomic MultiLog.

IndexLog. An Atomic MultiLog stores an IndexLog for each indexed packet attribute (e.g., `srcIP`, `dstPort`), that maps each unique attribute value (e.g., `srcIP=10.0.0.1` or `dstPort=80`) to corresponding packet headers in HeaderLog. IndexLogs efficiently support concurrent, lock-free insertions and lookups using two main ideas.

Protocol-defined fixed attribute widths in packet headers allow IndexLogs to use a perfect k-ary tree [43] (referred to as an attribute index in Figure 4) for high-throughput insertions upon new data arrival. Specifically, an n -bit attribute is indexed using a k-ary tree with a depth of $\lceil \frac{n}{\log_2 k} \rceil$ nodes, where each node indexes $\log_2 k$ bits of the attribute. For instance, Figure 5 shows an example of a 2^{16} -ary tree for IP addresses, where the root node has 2^{16} child pointers corresponding to all possible values of the 16-bit IP prefix, and each of its children have 2^{16} pointers for the 16-bit IP suffix.

The use of a perfect k-ary tree greatly simplifies the write path. All child pointers in a k-ary tree node initially point to `NUL`. When a new packet attribute value (e.g., `srcIP=10.0.0.1`) is indexed, all unallocated nodes along

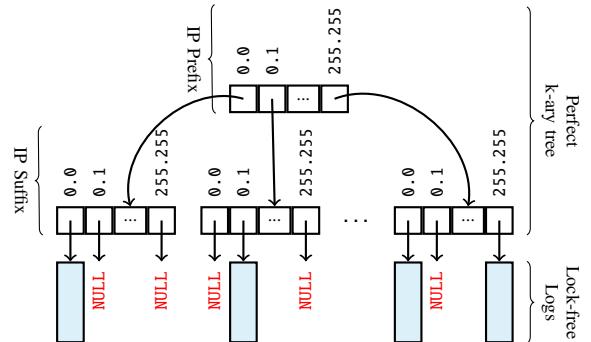


Figure 5: 2^{16} -ary IndexLog for 32-bit IP address. Each node in the tree (depth=2) has $k=2^{16}$ children and indexes 16 bits (2 bytes) of the IP address.

the path corresponding to the attribute value are allocated. This is where an IndexLog uses the second idea — since the workload is append-only, HeaderLog offsets for attribute value to packet header mapping are also append-only; thus, traditional lock-free concurrent logs can be used to store this mapping at the leaves of the k-ary tree.

Conflicts among concurrent attribute index nodes and log allocations are resolved using the `CompareAndSwap` instruction, thus alleviating the need for locks. Subsequent packet headers with the same attribute value are indexed by traversing the tree to the relevant leaf, and appending the header's offset to the log. To evaluate range queries on the index, Confluo identifies the sub-tree corresponding to the attribute range (e.g., `10.0.0.0/24`); the final result is then the union of header offsets across logs in the sub-tree leaves.

FilterLog. A FilterLog is simply a filter expression (e.g., `srcIP==10.0.0.1 && dstPort==80`), and a time-indexed

collection of logs that store references to headers that match the expression (bucketed along user-specified time intervals). The logs corresponding to different time-intervals are indexed using a perfect k-ary tree, similar to IndexLogs.

AggregateLog. Similar to FilterLogs, an AggregateLog employs a perfect k-ary tree to index aggregates (*e.g.*, $\text{SUM}(\text{pktSize})$) that match a filter expression across user-specified time buckets. However, atomic updates on aggregate values is slightly more challenging — it requires reading the most recent version, modifying it, and writing it back. Maintaining a single concurrent log for aggregates requires handling complex race conditions to guarantee atomicity.

Confluo instead maintains a collection of *thread-local* logs, with each writer thread executing read-modify-write operations on its own aggregate log. The latest version of an aggregate is obtained by combining the most recent thread-local aggregate values from individual logs. We note that the use of thread-local logs restricts aggregation to associative, commutative operations, that are sufficient to implement network monitoring and diagnosis functionalities.

3.2.1 Atomic Operations on Collection of Logs

End-to-end Atomic MultiLog operations may require updating multiple logs across HeaderLog, IndexLogs and FilterLogs. Even if individual logs support atomic operations, end-to-end Atomic MultiLog operations are not guaranteed to be atomic by default. Fortunately, it is possible to extend the readTail/writeTail mechanism for concurrent logs to guarantee atomicity for Atomic MultiLog operations; however, this requires resolving two challenges.

First, in order to guarantee total order for Atomic MultiLog operations, its component logs must agree on an ordering scheme. Confluo uses HeaderLog as single source of ground truth, and designates its readTail and writeTail as globalReadTail and globalWriteTail for the Atomic MultiLog. Before packet headers are written to different ring buffers, Confluo first atomically increments globalWriteTail by the size of the packet header using `FetchAndAdd`. This atomic instruction resolves potential write-write conflicts, since it assigns a unique HeaderLog offset to each header. When Confluo writers read headers from different ring buffers, they update all relevant logs in Atomic MultiLog, and finally update the globalReadTail to make the data available to subsequent queries.

The globalReadTail imposes a *total order* among Atomic MultiLog write operations based on HeaderLog offsets: Confluo only permits a write operation to update the globalReadTail after all write operations writing at smaller HeaderLog offsets have updated the globalReadTail, via repeated `CompareAndSwap` attempts. This ensures that there are no “holes” in the HeaderLog, and allows Confluo to ensure atomicity for queries via a simple globalReadTail check. In particular, queries first atomically obtain globalReadTail value using `AtomicLoad`, and only access headers and their

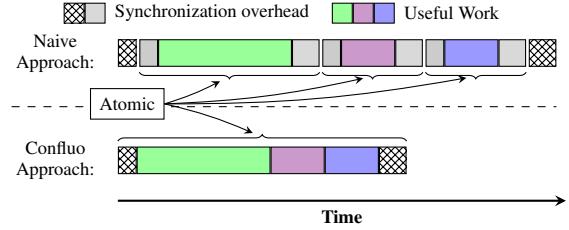


Figure 6: Confluo relaxes atomicity guarantees of individual logs, guaranteeing atomicity only for end-to-end Atomic MultiLog operations. Different colors correspond to operations on different logs.

references (across IndexLogs, FilterLogs and AggregateLogs) if the header lies within the globalReadTail in HeaderLog. Note that since queries do not modify globalReadTail, they cannot conflict with other queries or write operations.

The second challenge lies in preserving atomicity for operations on Confluo aggregates, since they are not associated with any single packet header that lies within or outside the globalReadTail. To this end, aggregate values in AggregateLogs are versioned with the HeaderLog offset of the write operation that updates it. To get the final aggregate value, Confluo obtains the aggregate with the largest version smaller than the current globalReadTail for each of the thread-local aggregates. Since each Confluo writer thread modifies its own local aggregate, and queries on aggregates only access versions smaller than the globalReadTail, operations on pre-defined aggregates are rendered atomic.

While the operations above enable end-to-end atomicity for Atomic MultiLog operations, we note that readTail updates for each individual log in the Atomic MultiLog may add up to a non-trivial amount of overhead (Figure 6). Confluo alleviates this overhead by observing that in any Atomic MultiLog operation, the globalReadTail is only updated after each of the individual log readTails are updated. Therefore, any query that passes the globalReadTail check trivially passes the individual readTail checks, obviating the need for maintaining individual readTails. Removing individual log readTails relaxes unnecessary ordering guarantees for them, while enforcing it only for end-to-end operations. This significantly reduces contention among concurrent operations.

3.3 Monitor & Diagnoser Modules

We now describe Confluo monitor and diagnoser modules.

Monitor Module. This module is responsible for online evaluation of Confluo triggers via a dedicated monitor thread. Confluo triggers operate on pre-defined aggregates (§2.2) in the Atomic MultiLog. Since the aggregates are updated for every packet, trigger evaluation itself involves little work. The monitor thread wakes up at periodic intervals, and first obtains relevant aggregates for intervals since the trigger was last evaluated, performing coarse aggregations over multiple stored aggregates over sliding windows. It then checks if the trigger predicate (*e.g.*, $\text{SUM}(\text{pktSize}) > 1\text{GB}$) is satisfied, and if so, generates an alert.

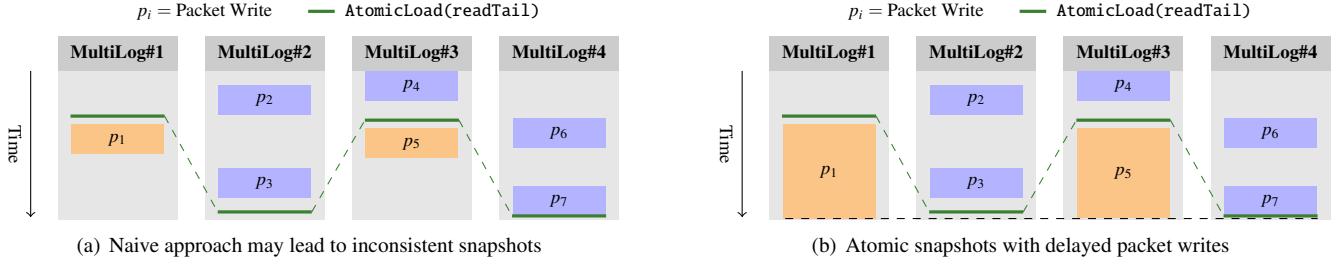


Figure 7: Simply obtaining (global) readTails for a collection Atomic MultiLogs can yield inconsistent snapshots, as shown in (a), where `AtomicLoad` on readTails at different Atomic MultiLogs are skewed in time, and packets p_1, p_5 appear to be written after p_3, p_7 (inconsistent). (b) We can render the same snapshot consistent by delaying completion of p_1, p_5 until *after* `AtomicLoad` on on Atomic MultiLog #4.

Diagnoser Module. Confluo’s diagnoser module serves ad-hoc queries on packet headers captured by the Atomic MultiLog. Recall from Table 1 that Confluo allows a diagnostic query to provide a filter expression `fExpression` as well as a time range. If there already exists a filter `fExpression`, query execution is fairly straightforward — since FilterLogs are time-indexed (Figure 4), Confluo simply looks up the FilterLog(s) to extract packet header offsets corresponding to the specified time interval, drops the offsets that are greater than the `globalReadTail` value, and returns packet headers corresponding to the remaining offsets. Confluo allows nested queries; Confluo can apply additional filters on these packet headers or obtain attribute aggregates for them.

If a filter for `fExpression` specified in the query does not already exist, Confluo first performs IndexLog lookups for individual packet attributes in the filter expression (§3.2), and then combines their results based on the boolean operators in the expression (Table 2). This can be an expensive operation; to that end, Confluo uses several optimizations. For instance, Confluo first converts the filter expression to its canonical disjunctive normal form (DNF) [44], where the resulting filter expression is a disjunction (OR) of conjunction (AND) clauses. The DNF form yields the most selective filter sub-expressions in its conjunction clauses. In order to minimize the number of packet references scanned for a specific conjunction clause, Confluo uses the tail value for individual attributes IndexLog as an estimate of their selectivity; Confluo then evaluates the conjunction clause by scanning through IndexLog entries for the most selective attribute, dropping all packet headers that occur after the `globalReadTail`, or do not satisfy the remaining predicates in the clause. The results for individual conjunction clauses are combined using a simple set union for the disjunction operator.

3.4 Archival Module

Confluo stores network logs with rich telemetry data, along with materialized views, pre-defined filters and aggregates to support low-overhead monitoring and diagnostic queries. Storing these logs and materialized views in their raw form over long time periods would lead to tremendous storage re-

quirements. Confluo overcomes this via periodic archival of Atomic MultiLog data. Our current implementation employs a basic approach — an archival thread periodically flushes packet header entries up to a certain offset in the HeaderLog to secondary storage, along with associated IndexLog, FilterLog and AggregateLog entries, and ensures that the in-memory footprint does not exceed a user-configured threshold. While Confluo data structures are amenable to several approaches that exist for log archival (e.g., periodically summarizing older data with aggregated statistics, log compression [45–47], compaction [48–50], etc.), a detailed treatment of the archival process is an interesting future work.

4 Distributed Diagnosis

Confluo Coordinator interface (Figure 3) facilitates monitoring and diagnosis of network-wide events. Recall from §2.3 that operators express monitoring and diagnosis tasks via *control programs* composed of Confluo API calls (Table 1). Based on the control program, the coordinator interface delegates tasks to individual end-host modules and collects diagnostic information from them. The coordinator interface facilitates consistent distributed analysis for high-speed networks via a *distributed atomic snapshot* algorithm.

Existing approaches for distributed snapshots either use a centralized sequencer to order all writes to the system (e.g., transaction managers [51–53], log sequencers [54–56]) simplifying global snapshots, or employ algorithms with weak consistency guarantees (e.g., causal consistency [57]). However, neither is acceptable for Confluo; the former is infeasible for high speed networks, while the latter provides weaker consistency semantics than Confluo end-host stack.

Confluo does not attempt to resolve complex distributed consistency issues, but instead strives for an efficient distributed *atomic* snapshot algorithm. We note that append-only semantics in Confluo greatly simplify snapshot for individual Atomic MultiLogs³. While naively reading readTails at individual Atomic MultiLogs across multiple end-hosts

³ Atomic snapshot of any Atomic MultiLog is trivially obtained by reading its `globalReadTail`.

Algorithm 1 Distributed Atomic Snapshot

Obtains the snapshot vector (Atomic MultiLog readTails).

At Coordinator:

- 1: `snapshotVector` $\leftarrow \emptyset$
- 2: Broadcast `FreezeReadTail` requests to all Atomic MultiLogs
- 3: **for each** `mLog` in `multiLogSet` **do**
- 4: Receive `readTail` from `mLog` & add to `snapshotVector`
- 5: Broadcast `UnfreezeReadTail` requests to all Atomic MultiLogs
- 6: **for each** Atomic MultiLog `mLog` **do**
- 7: Wait for ACK from `mLog`
- 8: **return** `snapshotVector`

At Each Atomic MultiLog:

On receiving FreezeReadTail request

- 1: Atomically read and freeze `readTail` using `CompareAndSwap`
- 2: Send `readTail` value to Coordinator

On receiving UnfreezeReadTail request

- 1: Atomically unfreeze `readTail` using `CompareAndSwap`
 - 2: Send ACK to Coordinator
-

may not produce an atomic snapshot (Figure 7(a)), it does hint towards a possible solution.

In particular, atomic distributed snapshot in Confluo reduces to the widely studied problem of obtaining a snapshot of n atomic registers in shared memory architectures [58–60]. These approaches, however, rely on multiple iterations of register reads with large theoretical bounds on iteration counts. While feasible in shared memory architectures where reads are cheap, they are impractical for distributed settings since reads over the network are expensive.

Confluo’s atomic distributed snapshot algorithm exploits the observation that any snapshot can be rendered atomic by *delaying* completion of certain writes that would otherwise break atomicity for the snapshot. For instance, in Figure 7(a), if we ensure that packet writes p_1 and p_5 do not complete until after the `globalReadTail` read on Atomic MultiLog #4 (dashed line in Figure 7(b)), the original snapshot becomes atomic since p_1 and p_5 now appear to be written after p_3 and p_7 , in line with the actual *order* of events.

Algorithm 1 outlines the steps involved in obtaining an atomic snapshot. The coordinator interface first sends out `FreezeReadTail` requests to all Atomic MultiLogs in parallel. The Atomic MultiLogs then freeze and return the value of their `readTail` atomically via `CompareAndSwap`. This temporarily prevents packet writes across the Atomic MultiLogs from completing since they are unable to update the corresponding `readTails`, but does not affect Confluo queries. Once the coordinator receives all the `readTails`, it issues `UnfreezeReadTail` requests to all the Atomic MultiLogs, causing them to unfreeze their `readTail` via `CompareAndSwap`. They then send an acknowledgement to the coordinator interface, allowing pending writes to complete at once. Since the first `UnfreezeReadTail` message is sent out only after the last Atomic MultiLog `readTail` has been read, all writes that would conflict with the snapshot are delayed until after the snapshot has been obtained.

The coordinator interface executes the snapshot algorithm

across arbitrary collections of end-hosts based on the provided control program, and generates a snapshot vector. Note that while the `readTails` remain frozen, write operations can still update `HeaderLog`, `IndexLogs`, `FilterLogs` and `AggregateLogs`, but wait for the `readTail` to unfreeze (up to one network round-trip time) in order to make their effects visible. As such, write throughput in Confluo is minimally impacted, but write latencies can increase for short durations. Moreover, since Confluo supports annotating packets with NIC timestamps to determine ordering (§2.3) before potentially delaying packet writes, Confluo’s atomic snapshot algorithm does not affect the accuracy of diagnostic queries.

5 Evaluation

Confluo prototype is implemented in $\sim 20K$ lines of C++. In this section, we evaluate Confluo to demonstrate:

- Confluo can capture packet headers at line rate (even for 10Gbps and higher bandwidth links) while evaluating thousands of triggers and tens of filters with minimal CPU utilization (§5.1);
- Confluo can exploit rich telemetry data embedded in packet headers to enable a large class of network monitoring and diagnosis applications (§5.2).

5.1 Confluo Performance

We now evaluate Confluo performance on servers with 2×12 -core 2.30GHz Xeon CPUs and 252GB RAM, connected via 10Gbps links. We used DPDK’s `pktgen` tool [28] to generate network traffic composed of TCP packets with 54 byte headers, IPs drawn from a /24 prefix and ports drawn from 10 common application port values. Our experiments used up to 5 attribute indexes, corresponding to the connection 4-tuple (source/destination IPs and ports) and the packet timestamp. We perform all our evaluations with Confluo running in the user space to avoid the performance bottlenecks out of Confluo implementation (*e.g.*, hypervisor overheads).

Packet Capture. Figure 8(a) shows Confluo peak packet capture rate as the number of attribute indexes and pre-defined filters are increased on a single core. Without any filters or indexes, the Atomic MultiLog is able to sustain ~ 25 million packets/s per core, with throughput degrading gracefully as more filters or indexes are added. The degradation is close to linear with the number of indexes, since each additional index incurs fixed indexing overhead for every packet. The degradation is sub-linear for filters, since additional filters incur negligible overheads for packets that do not match them. Interestingly, as we show in [24], monitoring and diagnosing even complex network issues only requires a few filters (often bounded by the number of active flows on a server) and 1-2 indexes in Confluo.

The packet capture performance indicates that, even when average packet size is 128B or larger, Confluo can sustain

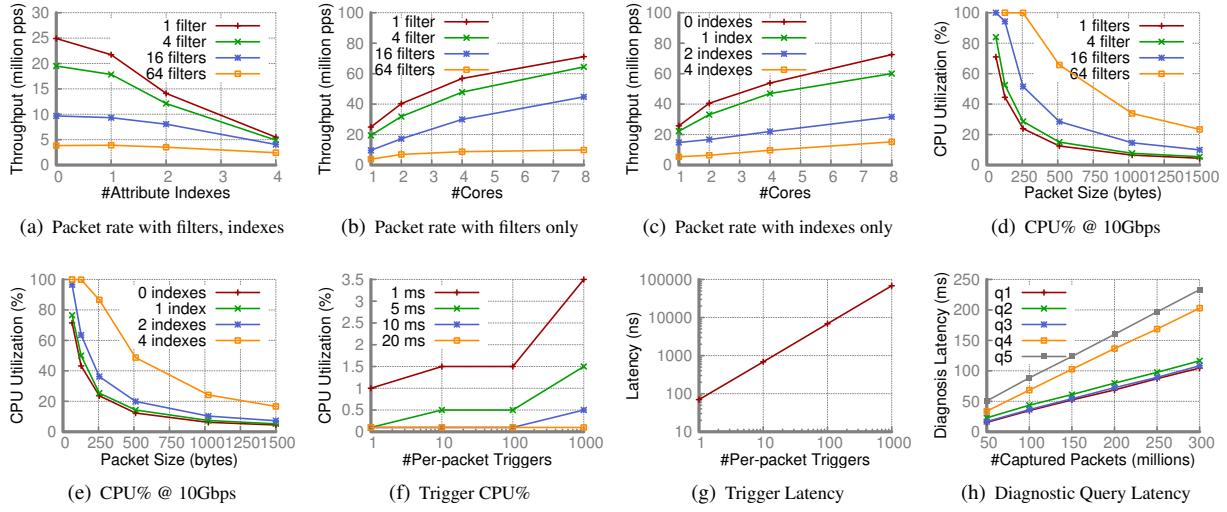


Figure 8: (a) Confluo’s peak packet capture throughput (measured in packets per second or pps) for 64B packets degrades gracefully on increasing the number of attribute indexes and the number of pre-defined filters; (b, c) the peak throughput scales well with the number of cores, even as the number of pre-defined filters and indexes are increased. (d, e) At line rate of 10Gbps, Confluo can handle average packet size as small as 128B with 16 filters and 2 indexes on a single core. (f, g) Confluo can evaluate 1000s of trigger queries with less than 4% CPU utilization at 1ms intervals, and with latency less than 70 μ s. (h) Diagnostic query latency in Confluo increases linearly with number of captured packets in Confluo, and varies across different queries due to differing intermediate result cardinalities and complexity for combining them. The filters in the figures use the following templates (varying value of A, B, IP, and port for various filters): (q1) packets from VM A to VM B; (q2) packets to VM A; (q3) packets from VM A on destination port P; (q4) packets between (IP₁, P₁) and (IP₂, P₂); and (q5) packets to or from VM A.

line rate for 10Gbps link using a single core! Real-world workloads [61] show that average packet size in datacenter networks is much larger. Confluo is able to ingest such workloads on a single core with each of 64 filters, 1000 triggers, and 5 indexes, updated for each packet. Figure 8(b) and 8(c) show packet capture scaling with number of cores. We note that, while packet capture scales well, it is not perfectly linear; this is due to stalling of globalReadTail updates for Confluo writers that attempt to update the Atomic MultiLog out-of-order (§3.2). However, the impact of stalling is mitigated to a great extent due to the use of lock-free primitives, and the use of a globalReadTail instead of separate readTails for each log in Atomic MultiLog.

CPU Utilization at 10Gbps. Figure 8(d) and 8(e) show CPU utilization for Confluo updating data structures, varying with the packet size for different number of filters and indexes. Observe that CPU utilization is higher for smaller packet sizes, since smaller packet sizes at line rate correspond to higher packet rates. For smaller packet sizes along with 4 indexes and 64 filters, CPU becomes a bottleneck; however, CPU utilization drops dramatically with fewer filters or indexes. Confluo can scale up its packet capture rate with more CPU cores, as discussed before.

Evaluating Triggers. Recall from §3.2 that Confluo evaluates triggers over pre-defined aggregates, making trigger evaluation extremely cheap. Figure 8(f) shows that even

when Confluo evaluates 1000 triggers at 1ms time intervals, the CPU utilization remains < 4% of a single core. This is because a single trigger evaluation incurs roughly 100ns latency, with latency increasing to 70 μ s for 1000 triggers⁴.

Diagnosis Latency. We evaluate Confluo’s diagnostic query performance using five queries (q1 to q5 outlined in Figure 8). Since these queries combine results from different Confluo IndexLogs, query latency depends on intermediate result cardinalities. Consequently, the query latency increases linearly with the number of captured packets, since cardinalities of intermediate results also grow linearly with the latter. As such, Confluo is able to perform complex diagnostic queries on-the-fly with sub-second latencies on 100s of millions of packets (Figure 8(h)).

Atomic Snapshots. To evaluate the overhead of atomic snapshots in Confluo, we measure percentage decrease in packet capture rate while periodically performing snapshots across 1 – 8 end-hosts (to emulate diagnostic queries). We found the impact of atomic snapshots on write rate to be insignificant — while performing snapshots every 1ms, packet rate at each end-host drops by < 2%, even as number of end-hosts in the snapshot is increased from 1 to 8. This result might be non-intuitive; the reason is that Confluo only

⁴A 70 μ s latency over 1ms period may result in as high as 7% CPU utilization; we believe the discrepancy is because of the reporting frequency for CPU utilization metrics from the OS.

blocks updates to the globalReadTail during the snapshot operation — bulk of the writes including those to HeaderLog, IndexLogs and FilterLogs can still proceed, with entire set of pending globalReadTail updates going through at once when the snapshot operation completes.

We note that a diagnostic query that spans multiple servers would incur the end-host query execution latency shown in Figure 8(h), as well as the atomic snapshot latency. Since the snapshot algorithm queries different Atomic MultiLogs across different end-hosts in parallel, the snapshot is obtained in roughly 1 network round-trip (about $\sim 180\mu\text{s}$ in our setup), with slightly higher latencies across larger number of end-hosts due to skew in queuing and scheduling delays (about 1.2ms for 128 end-hosts). Since network-wide diagnosis tasks often only involve a very small fraction of a data center’s end-hosts, Confluo can employ switch metadata to isolate the end-hosts it needs to query, similar to [9].

5.2 Confluo Applications

We now use Confluo to detect and debug a variety of network issues in modern data center networks. Our setup (comprising 96 virtual machines and Pica8 P-3297 switches), deployment and workloads are exactly the same as those in [8, 9], but with the end-host stack replaced with Confluo. Consequently, our setup inherits (1) in-network mechanisms that embed switch ID and timestamp at each switch traversed by a packet in its header, and (2) switch pointers to end hosts where the telemetry data for packets processed by the switch are stored. While we present only a subset of Confluo applications here for brevity, we discuss more applications in [24].

Path Conformance. We demonstrate Confluo’s ability to quickly monitor and debug path conformance violations by randomly routing a subset of the packets *within a flow* via a particular switch S . Each end-host is configured with a single filter that matches packets that pass through switch S . A companion trigger to the filter raises an alert if the count of packets satisfying the filter is non-zero. Confluo monitor evaluates the trigger at 1 ms intervals, and alerts the presence of path non-conformant packets within milliseconds of its incidence at the end-host.

Figure 9(a) shows the latency in Confluo with varying number of path conformance checks (filters). We note that while a single conformance check incurs average batch latency of $1\mu\text{s}$, 100 checks incur $11\mu\text{s}$ latency; this indicates sub-linear increase in latency with the number of checks. As such, Confluo is able to perform *per-packet* path conformance checks with minimal overheads.

Packet Losses at a Single Switch. In this application, we consider monitoring and diagnosis for generalized versions of the scenarios from Figure 2 (left), where k flows compete at a common output port at switch S and one or more of these flows experience packet losses. Confluo’s approach is outlined in Figure 2 (right). Confluo exploits network telemetry

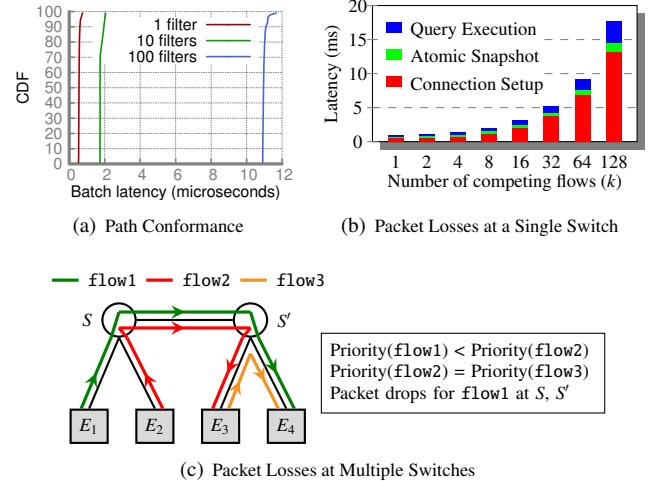


Figure 9: (a) Confluo can perform 100 path conformance checks and ingest packet headers in batches of size 32 in about $11\mu\text{s}$ per-batch ($\sim 350\text{ns}$ per-packet); (b) Diagnosis latency for packet losses due to traffic congestion; most of the time is spent in connection setup. Confluo takes $< 18\text{ms}$ for querying 128 hosts. (c) Setup used for monitoring and diagnosing packet losses at multiple switches.

data in packet headers (switch IDs and timestamps) to identify contending TCP flows and their destination end-points.

Confluo is able to detect the presence of packet loss due to TCP retransmissions in under 1ms (trigger periodicity), and the coordinator interface receives the alert within $\sim 250\mu\text{s}$. Figure 9(b) shows the diagnosis latency at the coordinator as the number of competing flows (k) at switch S increases. With more flows, Confluo has to contact more end-hosts to collect diagnostic information. Even while collecting diagnostic information across 128 end-hosts, the time taken to obtain the atomic snapshot and performing the diagnostic query at the coordinator are only 1.2 ms and 3 ms respectively. Most of the diagnosis time is spent in establishing connections to the relevant end-hosts, although this can be mitigated via connection pooling. Even so, Confluo is able to diagnose the issue across 128 hosts in under 18 ms.

Packet Losses at Multiple Switches. We now consider a scenario where a flow experiences packet losses at multiple switches, as outlined Figure 9(c). Again, we detect packet losses using TCP retransmissions, and employ telemetry data embedded in packet headers (switch IDs) to aid diagnosis. Using ideas discussed in [8, 9], we issue diagnostic queries to determine the flow information (IDs, traffic volume and priorities) that contended with flow1 at switches S and S' . By comparing the traffic volume and priorities of contending flows, Confluo concludes that the losses for flow1 are due to contention with higher priority flow2 and flow3 at switches S and S' . Confluo takes roughly 1.8ms for the end-to-end diagnosis: 1.15ms for connection setup, 180 μs for the snapshot algorithm and 350 μs for performing the actual query.

TCP Outcast. In TCP outcast problem [62], two sets of

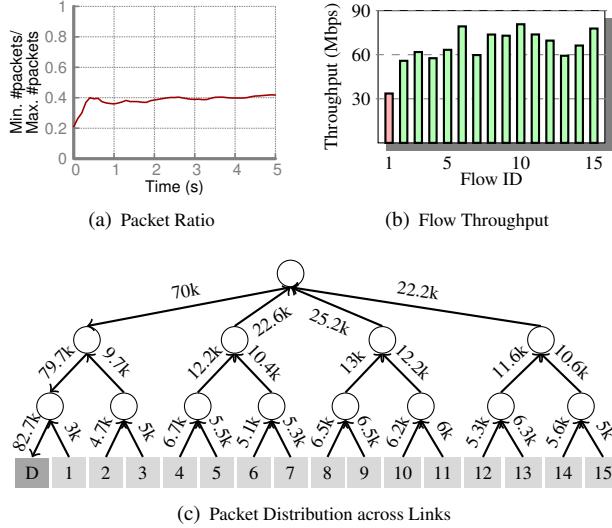


Figure 10: Diagnosing TCP Outcast. (a) Confluo measures the cumulative ratio of smallest and largest packet counts across all flows at 10ms intervals to diagnose outcast; smallest and largest packet counts correspond to flows with smallest and largest hop-counts respectively, with their ratio stabilizing to 0.4 in 1s after measurement starts. (b) Flow throughputs at $t = 1$ s. (c) Using [7–10], Confluo can obtain packet distribution across links (numbers along links) in a 1s window during outcast. Circles represent switches, 1–15 represent flowIDs, and D represents destination end-host.

flows (one with small number of flows, and one with large number of flows) from two different input ports of a switch compete for the same output port; it has been shown [62] that in such a scenario, TCP can result in severe throughput degradation for the small set of flows. This occurs due to *port blackout* in switches that employ tail-drop queuing, wherein a batch of consecutive packets are dropped from an input port. In TCP Outcast, this disproportionately affects the small set of flows, leading to TCP timeouts.

In our experiment, we recreate a setup similar to [62], where 15 TCP flows with different sources and the same destination (shown as D in the figure) compete for a single output port at the final-hop switch. One flow traverses a 1-hop path, two of them traverse a 3-hop path, and the remaining 12 traverse a 5-hop path. All links in the setup have 1Gbps bandwidth. To monitor the TCP outcast problem, Confluo first adds triggers to detect packet losses (Table 2(b)). Once the trigger raises an alarm, the coordinator interface issues diagnostic queries at 10ms intervals to obtain packet count for each flow in that window, and compute cumulatively (1) ratio of smallest to largest packet counts across all flows, and (2) individual flow throughputs (Figure 10). Each diagnostic query incurs an average latency of 250μ s.

Owing to port blackout, the flow with smallest hop-count observes the lowest throughput, while flows with larger hop-counts observe higher throughput (Figure 10(b)). By exploiting telemetry data embedded in packet headers, Confluo can

also obtain the number of packets transmitted through each link in the network over a 1s window (Figure 10(c)).

6 Related Work

We already discussed related work in network monitoring and diagnosis in §2.1. In this section, we focus on related work in the context of Atomic MultiLog.

There has been a lot of work on the design of efficient, concurrent logs [39–42, 54–56, 63–65]. Since log-based systems have been around for several decades, it would be impractical to attempt an exhaustive comparison. However, at a high-level, we note that traditional log-based systems focus on simple atomic operations on a *single* log; in contrast, Confluo combines a *collection of logs* in the Atomic MultiLog to support atomic filters, aggregates and triggers over packet headers. By relaxing the atomicity guarantees for its individual logs and guaranteeing atomicity only for end-to-end MultiLog operations, Confluo achieves high concurrency for these collection of logs. Figure 1 compares the performance of Confluo against the state-of-the-art log-based system [54].

Database Management Systems (DBMS) [66–68] use secondary indexes to support filters and aggregates on records. Unfortunately, atomically updating tree-based index structures such as B-Trees [69, 70] and Tries [71–74] incur high write overheads due to complex tree traversals and locking overheads, resulting in low write throughput. On the other hand, hash-based indexes [75–77] sustain high throughput, but do not support ordered access to data items. Confluo borrows heavily from these approaches, but makes design trade-offs to meet the high throughput and rich functionality requirements of network monitoring and diagnosis (§3.2).

7 Conclusion

Confluo is an end-host stack that can be integrated with existing network management tools to enable monitoring and diagnosis of network events. Confluo achieves this using Atomic MultiLog, a new data structure that exploits structure in network traffic to support highly concurrent read-write operations. Confluo executes 1000s of triggers and 10s of filters at line rate (for 10Gbps links) on a single core.

Acknowledgments

We would like to thank our shepherd, Cole Schlesinger, and anonymous NSDI reviewers for their insightful feedback. We are also grateful to Praveen Tammana for helping us in setting up experimental testbed, and for sharing packet traces from PathDump and SwitchPointer experiments. This research is supported in part by NSF CISE Expeditions Award CCF-1730628, NSF DGE-1106400, NSF CNS-1704742, and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

- [1] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, “Language-Directed Hardware Design for Network Performance Monitoring,” in *ACM SIGCOMM*, 2017.
- [2] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, “SketchVisor: Robust Network Measurement for Software Packet Processing,” in *ACM SIGCOMM*, 2017.
- [3] Y. Li, R. Miao, C. Kim, and M. Yu, “Flowradar: a better netflow for data centers,” in *USENIX NSDI*, 2016.
- [4] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One sketch to rule them all: Rethinking network flow monitoring with UnivMon,” in *ACM SIGCOMM*, 2016.
- [5] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” in *USENIX NSDI*, 2013.
- [6] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, “Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility,” in *ACM SIGCOMM*, 2014.
- [7] P. Tammana, R. Agarwal, and M. Lee, “CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks,” in *USENIX SOSR*, 2015.
- [8] P. Tammana, R. Agarwal, and M. Lee, “Simplifying Datacenter Network Debugging with PathDump,” in *USENIX OSDI*, 2016.
- [9] P. Tammana, R. Agarwal, and M. Lee, “Distributed Network Monitoring and Debugging with Switch-Pointer,” in *USENIX NSDI*, 2018.
- [10] “In-band Network Telemetry (INT).” <https://p4.org/assets/INT-current-spec.pdf>.
- [11] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, “Passive Realtime Datacenter Fault Detection and Localization,” in *USENIX NSDI*, 2017.
- [12] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang, “Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis,” in *USENIX SOSR*, 2016.
- [13] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, “Compiling Path Queries,” in *USENIX NSDI*, 2016.
- [14] A. Gupta, R. Harrison, A. Pawar, M. Canini, N. Feamster, J. Rexford, and W. Willinger, “Sonata: Query-Driven Streaming Network Telemetry,” in *ACM SIGCOMM*, 2018.
- [15] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks,” in *USENIX NSDI*, 2014.
- [16] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, “Planck: Millisecond-scale monitoring and control for commodity networks,” in *ACM SIGCOMM*, 2014.
- [17] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, “Packet-Level Telemetry in Large Datacenter Networks,” in *ACM SIGCOMM*, 2015.
- [18] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, “007: Democratically Finding the Cause of Packet Drops,” in *USENIX NSDI*, 2018.
- [19] “OpenSOC.” <http://opensoc.github.io/>.
- [20] “Tigon.” <http://tigon.io>.
- [21] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, “Gigascope: A Stream Database for Network Applications,” in *ACM SIGMOD*, 2003.
- [22] M. Sullivan, “Tribeca: A Stream Database Manager for Network Traffic Analysis,” in *VLDB*, 1996.
- [23] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “Trumpet: Timely and Precise Triggers in Data Centers,” in *ACM SIGCOMM*, 2016.
- [24] A. Khandelwal, R. Agarwal, and I. Stoica, “Confluo: Distributed Monitoring and Diagnosis Stack for High Speed Networks.” Technical Report, 2018.
- [25] “Confluo GitHub Repository.” <https://github.com/ucbrise/confluo>.
- [26] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, “NetBricks: Taking the V out of NFV,” in *USENIX OSDI*, 2016.
- [27] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, “SoftNIC: A Software NIC to Augment Hardware,” Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.
- [28] “The Pktgen Application.” <https://readthedocs.io/en/latest/>.
- [29] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, “Enriching Network Security Analysis with Time Travel,” in *ACM SIGCOMM*, 2008.

- [30] “Deepfield Defender.” <http://deepfield.com/products/deepfield-defender/>.
- [31] “Kentik Detect.” <https://www.kentik.com>.
- [32] F. Fusco, M. P. Stoecklin, and M. Vlachos, “NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic,” *VLDB*, 2010.
- [33] P. Giura and N. Memon, “NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring,” in *Springer-Verlag RAID*, 2010.
- [34] J. Lee, S. Lee, J. Lee, Y. Yi, and K. Park, “FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency,” in *USENIX ATC*, 2015.
- [35] “The health insurance portability and accountability act.” <http://www.hhs.gov/ocr/privacy/>.
- [36] “Cisco Compliance Solution for HIPAA Security Rule Design and Implementation Guide.” <https://tinyurl.com/y94u8sqq>.
- [37] “Intel Data Plane Development Kit (DPDK).” <http://dpdk.org>.
- [38] “Open vSwitch (OVS).” <http://openvswitch.org>.
- [39] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, “Scaling Concurrent Log-structured Data Stores,” in *ACM EuroSys*, 2015.
- [40] M. P. Herlihy and J. M. Wing, “Linearizability: A Correctness Condition for Concurrent Objects,” *ACM TOPLAS*, 1990.
- [41] “A Fast Lock-Free Queue for C++.” <http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++>.
- [42] P. Tsigas and Y. Zhang, “A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems,” in *ACM SPAA*, 2001.
- [43] P. E. Black, “perfect k-ary tree.” <https://www.nist.gov/dads/HTML/perfectKaryTree.html>.
- [44] “Disjunctive normal form.” https://en.wikipedia.org/wiki/Disjunctive_normal_form.
- [45] R. Agarwal, A. Khandelwal, and I. Stoica, “Succinct: Enabling Queries on Compressed Data,” in *USENIX NSDI*, 2015.
- [46] “Configuring compression in Cassandra.” https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_config_compress_t.html.
- [47] “RocksDB Tuning Guide.” <https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide>.
- [48] “Memtables in Cassandra.” <https://wiki.apache.org/cassandra/MemtableSSTable>.
- [49] “Configuring compaction in Cassandra.” https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_configure_compaction_t.html.
- [50] “SSTable and Log Structured Storage: LevelDB.” <https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveledb>.
- [51] “SQLServer: Distributed Transactions (Database Engine).” [https://technet.microsoft.com/en-us/library/ms191440\(v=sql.105\).aspx](https://technet.microsoft.com/en-us/library/ms191440(v=sql.105).aspx).
- [52] “Oracle: Distributed Transactions Concepts.” https://docs.oracle.com/cd/B10501_01/server.920/a96521/ds_txns.htm.
- [53] “Postgres: eXtensible Transaction Manager.” <https://wiki.postgresql.org/wiki/DTM>.
- [54] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, “CORFU: A Shared Log Design for Flash Clusters,” in *USENIX NSDI*, 2012.
- [55] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, “Tango: Distributed Data Structures over a Shared Log,” in *ACM SOSP*, 2013.
- [56] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, “vCorfu: A Cloud-Scale Object Store on a Shared Log,” in *USENIX NSDI*, 2017.
- [57] K. M. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM TOCS*, 1985.
- [58] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, “Atomic Snapshots of Shared Memory,” *JACM*, 1993.
- [59] H. Attiya and O. Rachman, “Atomic Snapshots in O (N Log N) Operations,” *SIAM Journal on Computing*, 1998.
- [60] H. Attiya, M. Herlihy, and O. Rachman, “Atomic Snapshots Using Lattice Agreement,” *Springer-Verlag Distributed Computing*, 1995.
- [61] T. A. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding Data Center Traffic Characteristics,” in *ACM SIGCOMM CCR*, 2009.

- [62] P. Prakash, A. Dixit, Y. C. Hu, and R. Komella, “The TCP Outcast Problem: Exposing Unfairness in Data Center Networks,” in *USENIX NSDI*, 2012.
- [63] B. Chandramouli, G. Prasaad, D. Kossmann, J. Lewandoski, J. Hunter, and M. Barnett, “FASTER: A Concurrent Key-Value Store with In-Place Updates,” in *ACM SIGMOD*, 2018.
- [64] “Lock-Free Programming.” https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf.
- [65] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, “Black-box Concurrent Data Structures for NUMA Architectures,” in *ACM ASPLOS*, 2017.
- [66] “Oracle Database.” <https://www.oracle.com/index.html>.
- [67] “MySQL.” <https://www.mysql.com>.
- [68] “Microsoft SQL Server.” <https://www.microsoft.com/en-us/sql-server/sql-server-2016>.
- [69] R. Bayer and E. McCreight, “Organization and Maintenance of Large Ordered Indices,” in *ACM SIGMOD*, 1970.
- [70] A. Braginsky and E. Petrank, “A Lock-free B+Tree,” in *ACM SPAA*, 2012.
- [71] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, “Concurrent Tries with Efficient Non-blocking Snapshots,” in *ACM SIGPLAN PPoPP*, 2012.
- [72] S. Heinz, J. Zobel, and H. E. Williams, “Burst tries: a fast, efficient data structure for string keys,” *ACM TOIS*, 2002.
- [73] N. Askitis and R. Sinha, “HAT-trie: A Cache-conscious Trie-based Data Structure for Strings,” in *ACSC*, 2007.
- [74] D. R. Morrison, “PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric,” *JACM*, 1968.
- [75] “MySQL: Comparison of B-Tree and Hash Indexes.” <https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html>.
- [76] “Oracle: About Hash Clusters.” https://docs.oracle.com/cd/B28359_01/server.111/b28310/hash001.htm.
- [77] “SQL Server: Hash Indexes.” <https://docs.microsoft.com/en-us/sql/database-engine/hash-indexes>.

DETER: Deterministic TCP Replay for Performance Diagnosis

Yuliang Li

Harvard University

Rui Miao

Alibaba Group

Mohammad Alizadeh

Massachusetts Institute of Technology

Minlan Yu

Harvard University

Abstract

TCP performance problems are notoriously difficult to diagnose because subtle differences in TCP parameters and features may lead to completely different performance. The gold standard for diagnosis is to collect packet traces and trace TCP executions. However, it is not easy to use these tools in large-scale data centers where many TCP connections interact with each other. In this paper, we introduce DETER, a deterministic TCP replay tool, which runs lightweight recording all the time at all the hosts and then replays selected collections where operators can collect packet traces and trace TCP executions for diagnosis. The key challenge for deterministic TCP replay is the butterfly effect—a small timing variation causes a chain reaction between TCP and the network that drives the system to a completely different state in the replay. To eliminate the butterfly effect, we propose to replay individual TCP connection separately and capture all the interactions between a connection with the applications and the network. We show that DETER has low recording overhead and can help diagnose many TCP performance problems such as long latency related to receive buffer shrinking, zero windows, late fast retransmission, frequent retransmission timeout, and problems related to the switch shared buffer.

1 Introduction

Modern data center applications increasingly rely on high throughput and low latency TCP performance. Yet, these applications often experience TCP performance problems that are hard to diagnose. This is because the TCP stack is a complex system that involves many heuristics to deal with network conditions and application behaviors, and it has many variations that optimize for different traffic scenarios and application objectives.

As a result, there is simply no single best setting for all scenarios. Researchers invent more than two TCP variations every year and there are already tens of congestion control algorithms to choose in Linux. TCP in Linux 4.4 has 63

parameters to configure, some of which are less known to normal application developers, such as early retransmission flag and TCP low latency flag which provides options for optimizing specific traffic settings. Other parameters are hard to configure even for TCP experts, as they have to run TCP multiple times to fully understand the influences of different parameter settings and the interactions of various TCP features. For example, thin-dupACK dynamically changes the threshold of the number of dupACK for fast retransmission based on the size of the current transfer. TSO window divisor affects the Nagle test for TSO, which decides how many packets to wait in order to form a larger packet.

Moreover, TCP is under continuous, error-prone development. There are 16 bugs identified in Linux TCP [25] in just July and August of 2018. As an example, one bug is related to DCTCP, where the DCTCP CC’s ACK generation conflicts with the basic TCP framework’s ACK generation, resulting in some packets never being acknowledged [19].

Many misconfigurations and bugs are hard to diagnose because they are sporadic and intermittent. However, they are still sufficient to degrade application performance, especially in data centers where large scale distributed systems often involve thousands of requests to fulfill a task [48, 39], because a single long latency may delay the entire task [32, 42].

Although diagnosing TCP performance problems is notoriously hard, the gold standard tools are still the same as what have been used for tens of years: capturing packet traces [18] and tracing TCP executions [13, 1]. While these tools are useful for diagnosing individual connections, using them in large-scale data center environments is hard, because there are millions of flows from hundreds of thousands of hosts interfering with each other constantly. Collecting packets and tracing TCP executions at all hosts and switches takes large quantities of storage, computing, and bandwidth resources. TCP counters [58, 28] are useful lightweight tools in production, but they are not detailed enough to diagnose the complex settings and interactions mentioned above (see more examples of complex TCP performance problems in §5).

A common way to debug complex large-scale systems is

deterministic replay [52, 40, 33, 49, 27, 37]. Deterministic replay is proven to be an effective tool for developers to recreate performance problems, identify their root causes, and uncover many long-standing bugs in popular software. It would be ideal if we can deterministically replay TCP (i.e., deterministically re-execute the TCP code).

However, deterministically replaying a large network of TCP connections is difficult because TCP is a tightly coupled system with multiple interacting parties: applications, the network, other TCP connections traversing through a common switch, and the kernel at hosts.

In particular, the closed-loop nature of TCP creates a *butterfly effect*, where even small timing variations (e.g., clock drifts) between the runtime and the replay can drive the system to an entirely different state. Better time synchronization cannot solve this problem: even a nanosecond of timing variation leads to completely different TCP behaviors (§2.2). This is because small timing variations at hosts can cause different packet arriving orders at switches and therefore different packet drops. The differences in packet drops cause different TCP behaviors (e.g., congestion control) in turn, leading to different traffic rates from TCP senders and causing more differences in switch behaviors such as packet drops. Such butterfly effect propagates to many flows in the entire network after many rounds.

To eliminate the butterfly effect, we propose DETER, a DEterministic TCP Replay system, which breaks the closed loop interactions by replaying each TCP connection separately. We identify the minimal set of signals that capture all the interactions between a TCP connection with the application and the network, and record these signals at hosts in a lightweight manner. Specifically, DETER captures application socket calls and any impact on packets (e.g., if they are dropped or marked ECN) in the runtime. In the replay, we no longer need switches because all their actions to packets have been recorded and can be simply replayed. Since all the switch actions are deterministically replayed, we break the butterfly effect. We also isolate the TCP connection with other connections in the network because they only interact through switch actions.

The next question is how to deterministically replay an individual TCP connection. Although we already capture the interactions with the application and the network, there are still non-determinisms in the kernel at hosts. We design a customized solution for TCP which captures TCP-kernel interactions such as the kernel calling TCP handler functions, TCP reading kernel variables, and locks in thread scheduling. Note that we do not need to capture every packet, as the sender and receiver can generate packets and ACK for each other. The size of our total recording is just 2.1~3.1% of the size of fully compressed packet traces.

Since the recording is lightweight, DETER can run at all times for every connection on each host. Upon observing a performance problem, we can use DETER to zoom into

any TCP connection, deterministically replay its exact same execution, capture packet traces, and examine TCP state during the execution—all after the fact. We can also iteratively debug the same performance problem instance multiple times to collect different levels of detail each time.

Once we have the packet traces for each connection using DETER, we can also replay network queues in a physical network, emulator, or simulator as long as the setup has the same topology, routing, buffer size, and switch queuing algorithms as the runtime. During the replay, we push all the outgoing packets for all the senders into the network based on their timestamps. We also introduce a heuristic that significantly improves the accuracy of replaying packet drops.

We demonstrate the benefits of DETER by showing how we diagnose TCP performance problems in a Spark application with 6.2K connections, tail latency problems in an empirical web search workload with incasts, and example performance problems in a local testbed. With DETER, we can also diagnose a wide range of performance problems that require tracing the TCP execution, such as long latency related to receive buffer shrinking, zero windows, late fast retransmission, frequent retransmission timeout, and problems related to the switch shared buffer. The main limitation of DETER is that it requires recording at both the sender and the receiver of a connection and therefore cannot work when we do not have access to both ends.

2 Diagnosis Example and Challenges

We use a diagnosis example to demonstrate the benefits of deterministic replay. We then use the example to show the key challenge to enable the deterministic replay—the butterfly effect. Even a nanosecond of sending timing variation leads to completely different TCP behaviors.

2.1 A Diagnosis Example

We use an example to show how DETER helps diagnose TCP performance problems. We run a network with two senders (A and B) and one receiver, which are connected to a single switch and 10 Gbps links between them. Each sender sends two long flows of 20 MB each. 30 ms after the long flow starts, sender A sends a short flow of 30 KB to the same receiver. In one run, the short flow takes 49 ms to complete, which is two orders of magnitude higher than its expected completion time. In comparison, the RTO is just 16 ms.

Usually, people diagnose a problem by reproducing it. However, this problem is very hard to reproduce (shown in §2.2). If we cannot reproduce a problem, we have to rely on the information captured online, such as the TCP counters that data centers usually continuously monitor [58, 28]. Unfortunately, TCP counters are not enough for diagnosing this problem. The counter for retransmission timeout is two, but twice the RTO (2×16 ms) is still much less than 49 ms.

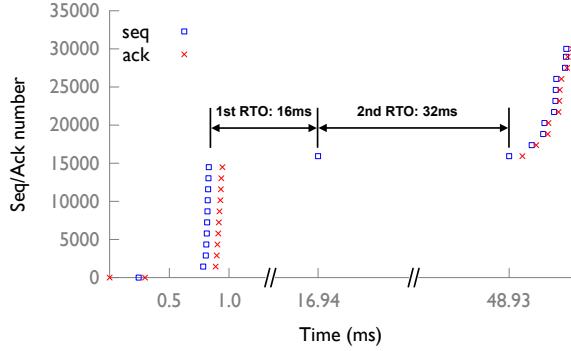


Figure 1: Receiver side Seq and Ack number of the short flow that experiences 49 ms FCT.

With DETER, we can deterministically replay the connections using the lightweight data recorded in the runtime (Table 1). During the replay, we capture the packet trace at the receiver side for the short flow (Figure 1). The trace shows that the second timeout is 32 ms. This is because the two timeouts are consecutive and thus trigger exponential backoff. The trace also shows the reason why the sender experiences the second timeout: the receiver receives the first retransmitted packet at 16.94 ms, but it does not send an ACK. Without the ACK, the sender has to retransmit again at 48.93 ms.

Why does the receiver not send an ACK for the first retransmitted packet? DETER allows us to replay multiple times, in order to collect more data and iteratively diagnose the problem. We replay again and use Ftrace [1] to get the function call graph on the processing of the first retransmitted packet. It shows that TCP enters the delayed ACK function, which means TCP decides to delay the ACK for the first retransmitted packet. The delayed ACK timeout is 40ms (which is a hardcoded value in the kernel and not configurable), which is longer than 2*RTO, so the second retransmission triggers first.

The root cause of this problem is that delayed ACK is very risky in the presence of RTO, because after RTO the sender can only send one packet. Ideally, the receiver needs a way to identify retransmissions (e.g., the sender marks the retransmitted packets), so it does not delay the ACK for them. As a workaround today, reducing the delayed ACK timeout can mitigate the problem.

2.2 Butterfly Effect

While deterministic TCP replay is a powerful tool for diagnosing TCP performance problems, it is not easy to ensure determinism. For the above example, if we simply replay with the same socket calls at the same times as the runtime, we cannot reproduce the problem.¹ Figure 2 shows that when we replay 100 times, the short flow always has way less than 49 ms flow completion times (FCT). In the production

¹We synchronize the clocks among the senders and receivers to 100s of nanoseconds precision by PTP (Precision Time Protocol [2]).



Figure 2: FCT of the short flow across 100 attempts of replay with socket calls. The blue dashed-line is 49 ms.

where there are more flows and more dynamic traffic than our testbed, it is more difficult to reproduce the same problem.

The key challenge for the deterministic replay is the butterfly effect. Packet sending times at hosts often have microsecond-level variation between the replay and the runtime. This is caused by the inherent host non-determinisms, such as the clock drift, context switching, kernel scheduling, and cache state [42].

The small variation gets amplified by the butterfly effect—the closed loop interactions between switches and TCP. A small packet sending time variation may change the order of packets from different hosts at a switch, which causes *switch action variations*—the switch may drop or mark ECN on a different set of packets. This starts the butterfly effect in the closed loop between switches and TCP: *Switch action variations* cause *TCP behavior variations* (e.g., TCP changing congestion window size differently). TCP behavior variations change its flow sending rates, which affect the queue lengths at all the switches the flow traverses ever since and lead to more switch action variations. Such a chain reaction between switches and TCP affects more and more flows all over the network in multiple rounds.

One may expect that reducing the sending time variation (e.g., better clock synchronization, more deterministic packet processing time) can improve the replay accuracy. However, our experiment shows that even a nanosecond of variation can lead to completely different packet-level behaviors.

We run an ns3 simulation [15] to control the sending time variation. We use the same topology and traffic as in §2.1. For the runtime, we set the host packet processing delay to 10 us, the same as what we measure in the testbed. The short flow incurs a long flow completion time because of the correlated RTO and delayed ACK. We then replay the experiment with the same socket calls and timings. To simulate different levels of sending time variation, we simulate a normal distribution of host packet processing delay with the same mean delay of 10 us but with a standard deviation ranging from 0 to 1000 ns. For each level, we replay 100 times.

Figure 3 shows the percentage of replays that reproduce the correlated RTO and ACK delay on the short flow. Once the sending time variation exceeds zero, even just 1 ns, the probability of reproducing the same problem suddenly drops.

This is because with a non-zero sending time variation, there is always a chance that a switch takes different actions on a packet between the runtime and the replay. Smaller timing variation can only delay the appearance of different ac-

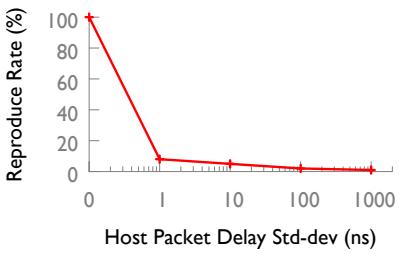


Figure 3: The rate of reproducing the correlated RTO and ACK delay.

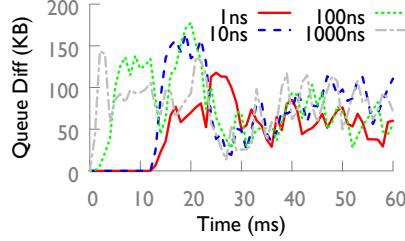


Figure 4: The time series of queue length difference.

tions, but cannot prevent it. Once the switch takes a different action, the butterfly effect starts, causing a chain reaction of changing sending rates and queue lengths. The chain reaction persists regardless of the level of the sending time variation.

Figure 4 illustrates this. We show the time series of queue length difference between runtime and replay experienced by each packet. For each level of sending time variation, we show a typical one of the 100 replays². For 1 ns variations, although the queue length difference starts later than with higher variations, once the difference starts at 12 ms, it never goes down to 0.

This result indicates that we cannot simply rely on reducing the sending time variation. This motivates our DETER design, which decouples the TCP and the network so that switch action variations cannot affect TCP.

3 DETER Design

In this section, we discuss DETER design with four key ideas: first, we break the butterfly effect by replaying individual TCP connections separately and record TCP’s interactions with the application and the network. Second, to deterministically replay each TCP connection, we record all the non-determinisms that happen in the interactions between TCP and the kernel. Third, we introduce a rate-based sampling solution to reduce the overhead of recording packet sending times. Finally, with the packet traces of all the connections, we show how to replay switch queuing behaviors.

3.1 Breaking the Butterfly Effect

We break the closed loop between TCP and the switches by replaying individual connections separately. We identify the minimal set of signals that capture all the interactions of a TCP connection with the application and the network.

TCP interacts with applications through socket calls. DETER captures all socket calls and its input arguments such as the number of read/recv bytes and socket flags.

²Although we cannot show all 100 replays here, we inspect each of them, and they have similar trend.

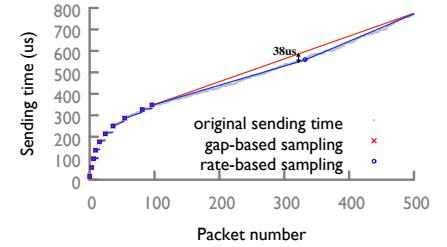


Figure 5: Inference error of sampling approaches. (The lines indicate the inferred sending time of each packet. The end points of the lines are sampled sending times.)

TCP interacts with the network through packets. TCP sends packets into the network and receives packets from the network. We do not need to record most incoming packets because we replay the sender and the receiver of a connection together and they can automatically generate packets for each other. We only need to record how the switches inside the network change the packet stream such as dropping packets or marking them with ECN bits. At the receiver, we detect packet drops by checking if the IP_ID fields are continuous and ECN by checking the ECN bits (see Section 4 for details) and record them there.

Note that for a TCP connection, it does not matter which switch drops or marks the packets. Only the final changes to the packets matter. So in the replay, we no longer need switches because their actions to packets have been recorded and we can just replay them. Since the switch actions are deterministically replayed, we break the butterfly effect.

A TCP connection interacts with other connections when they share switch resources in the network and cause different switch actions³. Since we recorded switch actions, we also isolate the interactions among TCP connections.

In summary, in the runtime, we record socket calls and switch changes to packets at all the hosts. Users can specify which connections to replay. To replay a connection, we set up a simple two-host testbed that runs as a sender and a receiver for every single connection without involving any switches. We run a socket call generator to generate socket calls at the right time and run a packet corrector to inject actions on packets before they arrive at the TCP sender and receiver. We can easily parallelize the replay of multiple connections because we replay each connection independently.

3.2 Handling Non-determinisms in the Kernel

The next question is how to deterministically replay a single TCP connection. It is complex to replay a general system [49, 27], which requires record and replay lots of non-determinisms. We use the knowledge of TCP to design a customized replay for TCP, which is lightweight. Specifically, besides the interaction with the application and the network,

³We discuss TCP connections on the same host in the next subsection.

TCP also has three non-determinisms from interacting with the kernel: the kernel may call TCP handler functions, the TCP may read kernel variables, and there is thread scheduling.

(1) TCP handler function calls from kernel: The kernel may call some TCP handler functions. For example, the OS timer may call TCP timeout handler. The kernel may also call resume transmission handler, which sends more packets in the send buffer. We need to record them.

(2) Reading kernel variables: TCP reads a few variables that are updated by other kernel programs (or hardware), such as memory pressure indicator, the jiffies (a low-resolution clock), the mstamp (a microsecond-resolution clock), and the send queue byte count. We should record the return value of each read.

(3) Thread scheduling: TCP works in a multi-threaded environment. Different threads, such as applications, NIC interrupts, and OS timers, access the shared socket variables by calling TCP handler functions. For example, an application thread calls a socket call handler to copy data into the socket send buffer; a NIC interrupt may call the TCP receive packet handler to frees up some space of the send buffer; OS timer may call the timeout handler to send a pending packet in the send buffer. It is important to ensure the order of different threads accessing the same variable. Fortunately, TCP uses a single socket lock to ensure that only one thread can access all the shared variables at a time. Thus, we just record the order of lock acquisition of different threads by giving a sequence number for each lock acquisition.

In the replay, we run the same TCP stack with the same TCP configuration as the runtime. In addition to the socket call generator and the packet corrector, we also generate handler calls from the kernel based on the recorded logs. We feed in the recorded kernel variables when TCP reads them. We also enforce the order of lock acquisition of different threads (see §4 for more details).

3.3 Sampling Packet Sending Times

So far we have ensured the ordering of TCP behaviors (e.g., the sequence of packets, state updates, loss detections, timeouts). One remaining question is how to replay packet sending times accurately. Recording the sending times for all the packets takes high storage overhead. To reduce the overhead, we choose to sample packets, record the sending times for sampled packets, and infer the times for the other packets. The question is how to select the samples in real-time while bounding the inference error within a given threshold th .

Strawman solution: gap-based sampling. TCP usually sends packets in bursts. So intuitively for each burst, we can keep the sending time of the first packet and the burst length. Assuming all the packets in the same burst follow the same sending rate, we can then infer the sending times of all the unsampled packets. We can identify packets in the same bursts if their interarrival time is below a threshold.

We perform a simple experiment to show that this approach has an unbounded error. We send two flows from two senders through a shared 10 Gbps link. The second flow starts 500us after the first flow. Figure 5 shows the packet sending time series of the first flow. All the packets from the 96-th to the 499-th are in the same burst (i.e., no gap of packet sending time), but the rate changes. As a result, the inferred sending time of the 323-th packet is 38 us later than the actual time.

Our solution: Rate-based sampling. Gap-based sampling fails to sample packets when the packet rate changes. Therefore, instead of recording the burst length, we propose to record the packet rate. When the inferred sending time based on the recorded packet rate is wrong (i.e., the difference with the actual time is above the threshold th), we sample a new packet. We set th to 5 us by default.

Specifically, in the runtime, we follow Algorithm 1. s is the previous sampled packet and p is the new packet. Given the sending time of s ($s.time$) and a packet rate r , we can infer the sending time of p ($p.time$). In reverse, to ensure that our inferred sending time of p falls in the range of $[p.time - th, p.time + th]$, we must ensure our recorded packet rate r falls in the range of $p_range = [\frac{p.index - s.index}{p.time + th - s.time}, \frac{p.index - s.index}{p.time - th - s.time}]$ (Line 2). Thus, we compare the recorded rate range rec_range and p_range . If they overlap, it means we can find a rate, in the intersection of rec_range and p_range , that can be used to infer a bounded sending time for both p and all the previous packets between s and p . Thus, we do not need to sample p (Line 4). Otherwise, if the two ranges do not overlap, we sample p , record a rate in rec_range , and reset rec_range (Line 6-7).

DETER can generate the full packet trace for each connection, by combining the recorded (inferred) sending times with the packets generated by the replay of TCP execution.

Algorithm 1 DETER Sampling sending time. $p.index$ is its index within its 5-tuple flow, and $p.time$ is its sending time.

```

1: procedure SAMPLE( $p$ : a new packet)
2:    $p\_range = [\frac{p.index - s.index}{p.time + th - s.time}, \frac{p.index - s.index}{p.time - th - s.time}]$ 
3:   if  $p\_range \cap rec\_range \neq \emptyset$  then
4:      $rec\_range = p\_range \cap rec\_range$ 
5:   else
6:     record( $s.index, s.time, rec\_range.mid$ )
7:      $s = p; rec\_range = [-\infty, \infty]$ 
```

3.4 Replaying Switch Queues

Because we can get all the packets, their sizes, and sending times for each connection in the network (§3.2 and §3.3), we can use them to replay switch queues in simulators (e.g., ns3 [15]) by pushing all the packets at the right time into the network. Replayng switch queues can help us understand the interaction between different connections at the switches (e.g., which flows contend for the queues).

The simulator needs the same topology and switch data plane (e.g., forwarding tables, buffer sharing policies, switching delay, and link propagation delay) as the runtime. Today, many vendors build high-fidelity simulators for their own devices [5, 21, 11]. One can also choose to replay switch queues in a physical network if available. Replayng switch queues also requires that the hosts during the runtime have microsecond-level synchronization, so that the relative packet sending time error across hosts are small. Clock synchronization in data centers is moving towards sub-microseconds level [2, 22, 34].

Replaying the exact queueing behavior is both impractical and unnecessary. It requires recording the exact order of enqueue and dequeue, which is too heavy for the runtime. On the other hand, it is often good enough to show the contending flows and their occupancies with high accuracy.

Thus, we opt for a simple design that can achieve high accuracy. We simply push all the packets into the network at the right time. It can achieve high accuracy because the switch queue occupancy is a continuous function with respect to packet sending times. Since the difference in packet sending times between the runtime and the replay is bounded, the difference of switch queue occupancy is also bounded. Specifically, suppose a packet’s arrival time at a port differs by k packets transmission time, and the fan-in of that port is f , the queue difference is at most $(f - 1)k$. k is small because our sampling bounds the sending time error to 5 us, and there are limited hops to amplify it. f is also small because the destinations of flows traversing a switch are random⁴. Even if f is large, such as during incast, the queue occupancy is also large, so the difference is a small fraction of the queue.

However, one exception is packet drops. Because dropping packets or not is a binary decision (not a continuous function), even if a microsecond level difference can cause different drops. Specifically, a runtime dropped packet may get through, which we call a *false-accept*. It also occupies some free space in the queue, leaving less space for later packets that should be in the queue, so one of the later packets may get mistakenly dropped, which we call a *false-drop*.

We propose to reduce the probability of false-accepts and false-drops by letting the hosts tag *should-be-dropped* packets. In this way, we ensure that the switches only drop packets with tags (for eliminating false-accept) and always deliver packets without tags (for eliminating false-drop).

The key challenge is how to know which switch to drop the tagged packets. Since the switch queue occupancy is a continuous function, it has bounded differences with respect to the sending time difference. We propose to decide whether to drop packets at a switch based on the switch’s queue occupancy upon packet arrivals. That is, when a *should-be-dropped* packet arrives at a switch, and the queue occupancy is above a threshold (e.g., $>$ queue max length - 5 MTU), the

⁴In theory, the fan-in is within 4 for 99.7% of the time for a 64-port switch with random traffic.

Type	Data recorded
Interaction w/ network	losses, ECN, reordering
Interaction w/ applications	socket calls
Handler call from kernel	Timeout handler, resume transmission handler, packet receive handler
Kernel variables	Infrequently updated variables, e.g., jiffies, memory pressure indicator
	Influence of frequently updated variables e.g., RACK loss detection
Order of lock acquisitions	sequence number of lock acquisitions for diff. threads
Timestamp samples	Sampled packet sending time (time, index, rate)

Table 1: Runtime recorded data.

switch drops the packet.

When a packet only experiences one congested switch on its path, which is the most common case, our solution works well. In the rare case when there are multiple congestion spots on the path, DETER may drop the packet at a wrong location. Our evaluation shows that this heuristic reduces the error of dropping packets⁵ from 58.3% to 2.87%.

4 Implementation

In this section, we discuss the implementation details of DETER. We just need 139 extra lines of code in the Linux kernel. Then we accomplish the record and replay with two kernel modules and two userspace programs (3000 lines of C and C++ code in total).

Runtime recording. For each connection, we first record its configurations, and then record the data listed in Table 1 during its runtime. We note that the configurations of connections on the same server are mostly the same, so we only record the parameters that differ from the default values. Our current prototype starts the recording after the connection is successfully built⁶. We now discuss the runtime recording.

Interaction with the network: This includes packet drops, ECN, and packet reordering. In our design, we use the IP_ID field to detect packet drops: Linux sends packets of each connection with consecutive IP_ID values, so the receiver can check if there are gaps in the series of incoming packets to detect drops (Similarly, the sender can detect drops in the incoming ACKs)⁷. On other platforms that do not have the consecutive IP_ID feature, we use LossRadar[44] to detect drops, which only takes O(#loss) space. The host also checks the ECN of the IP header of each incoming packet, and record

⁵Percentage of false-accept, false-drop, and drop at wrong location in all drops.

⁶Record and replay for connection setup is not very different. The only difference is detecting the drop of the first packet (SYN and SYN-ACK). This can be solved by recording the IP_ID of all SYN packets at both sender and receiver, which just adds 8 bytes for each connection.

⁷This is different from TCP’s drop detection: TCP sender does not distinguish drop of a data packet or its ACK. We must distinguish them because both the sender and the receiver must replay accurately.

1 bit (CE) for it. Sometimes there may be packet reordering, which we can detect also using the IP_ID field.

Recording the interactions with the network is lightweight. In data centers, the packet drop rate is just 10^{-5} to 10^{-4} [51, 36]. For ECN we just need 1 bit per packet. Reordering is rare, so it does not cost much. We instrument the TCP receive packet handler to record them.

Socket calls from the application: We hook the TCP socket call handler functions to record the #bytes and flag, so that we do not need to change the application.

We can reduce the storage overhead of socket calls a lot. We find that there are often identical socket calls. For example, distributed file systems break large files into fixed-size chunks, so most of the send and receive sizes are the same. Thus, we store all the common patterns of socket calls (the common #bytes and flag pairs) for different applications, and only record the pattern numbers in the runtime. DETER associates connections to applications via their TCP port numbers.

Other TCP handler calls from the kernel: We hook the timeout handlers and the resume transmission handler, and record them when they get called.

Kernel variables read by TCP: We record the memory pressure indicator and jiffies with low overhead because their values change infrequently. The memory pressure indicator is very rarely set, and the jiffies increments by 1 every 4 ms. So we just maintain the values of the last read and only record the reads that return a new value.

The mstamp and the send queue byte count are updated frequently. We reduce the overhead by recording their *influences* instead of their values. Specifically, the variables influence the TCP executions by serving as the metrics of if-conditions in TCP. For example, TCP uses the mstamp to detect losses (RACK [16]). We just need to record the loss detection result, rather than the actual value of the clock. We identify and record all the if-conditions they affect (1 bit for each), which relates to loss detection, cwnd reset, TCP segmentation offload, and TCP small queue. Moreover, most of the if-conditions have a dominant result (e.g., loss detection mostly return false), so we reduce the overhead further by only recording when they have the uncommon result.

We use a special reader function to record these values. For example, in the TCP code, we replace `a=jiffies` with `a=reader(jiffies)` to record the value of jiffies and replace `if (mstamp>b)` with `if (reader(mstamp>b))` to record the influence of mstamp. The reader function simply records the value passed to it and returns this value.

Lock acquisition: We instrument TCP’s lock acquisition function to record which thread calls this function, so we know the order of lock acquisition by different threads. We also optimize the overhead. Specifically, one thread may acquire the lock many times consecutively. For example, NIC interrupt acquires one lock for each incoming packet, so there are often tens of lock acquisition by NIC interrupt in a row. Therefore, we record the number of consecutive lock

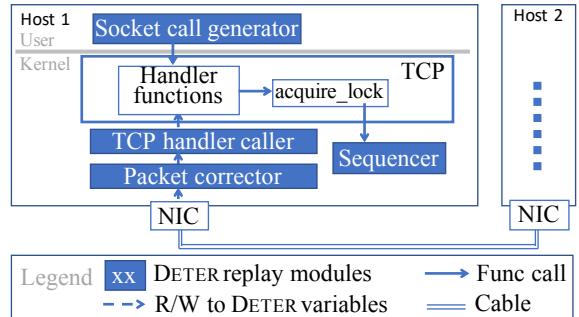


Figure 6: Replay implementation in DETER

acquisitions, instead of recording them individually.

Sampled sending times: To get the most accurate timestamps, we sample and record the sending times in the NIC driver, just before TCP pushes packets into the NIC ring buffer.

Replay. We now discuss the replay.

Replay TCP stacks. Figure 6 shows the replay setup. We implement the *packet corrector* with NetFilter [7]. It injects drops and CE bits to the incoming packets⁸. We also enforce the reordering here.

To replay the socket calls, we implement a *socket call generator* in the user space to inject socket calls from the applications according to the log.

We also implement a *TCP handler caller*, which is a kernel thread that calls TCP handler functions according to the log. The handler functions include the packet receive handler, the timeout handler, and the resume transmission handler. When calling the packet receive handler, it gets a packet from the packet corrector as an argument to the handler.

To enforce the order of different threads acquiring the lock, we implement a *sequencer*. It knows the order of different threads acquiring the lock based on the log. We instrument the lock acquisition function to check with the sequencer before it actually acquires the lock. If the current thread is not the next to acquire the lock, it waits for other threads until itself is the next to acquire the lock.

We reuse the reader function that we introduced before to feed the recorded kernel variables or their influences. During the replay, the reader function reads the log and return the corresponding value.

Replay sending and receiving timestamps. We only record packet sending times for replay. We then infer receiving times from sending times: for the received packet which triggers a new packet to send, we can estimate its receiving time as the sending time of the new packet minus the average packet processing time, which is measured separately. For the received packet that does not trigger a new packet, its gap with the previous received packet is close to their sending time gap,

⁸We require no packet drops before packets entering the packet corrector, so we must make sure no packets get unexpectedly dropped in the queues on the hosts (e.g., NIC ring buffer, softirq queue, qdisc) during the replay. We can set the sizes of these queues large enough to avoid unexpected drops.

because they experience similar network conditions. Note that only the sending times affect the switch queue replay, but not the receiving times.

Switch queue replay. We run Precision Time Protocol [2] in our testbed. We implement the switch queue replay in both testbed and simulation. For the testbed, we implement a DPDK packet generator that reads the packet trace, tags packets, and sends packets to the NIC at the right time. We use a NetFPGA-based switch to implement the drop accuracy improvement (§3.4). It is also implementable in P4 [23]. We also implement the replay in a packet-level simulation in ns3 [15], with the same topology, link delay and bandwidth, switch queueing algorithm, and routing state as the testbed.

5 Evaluation

In this section, we demonstrate the benefits of deterministic replay in DETER by showing how we diagnose TCP performance problems in a Spark application with 6.2K connections, the tail latency problem in an empirical web search workload with incasts, and example problems in a local testbed.

We also measure the CPU and storage overhead of DETER recording and the accuracy of DETER replay. Our evaluation shows that DETER only uses 2.1~3.1% compared to fully compressed packet traces and requires 0.094%-1.49% of CPU overhead. DETER also fully replays the sequences of packets at hosts and replays switch queues with lower than 1 MTU differences on average.

5.1 Diagnosis in Spark

Evaluation setting. We run a TeraSort job in Spark [24] that sorts 200 GB data on 20 servers connected with 10Gbps network in Amazon EC2 [20]. We use 4 executors (i.e., 4 cores) and 20GB memory on each server. The NIC MTU is 1500B. We enable TCP segmentation offload, and disable generic receive offload⁹. We run DETER on all servers to record data for all connections during the runtime and also run Tcpdump [18] to collect the packet traces as the groundtruth.

Replay accuracy. We use DETER to replay each connection and run Tcpdump during the replay. We compare the packet traces we collected during runtime and replay. The sequence of packets are exactly the same (we have a one-to-one mapping of TCP headers). The sending time differences between packets are lower than 5 us.

Diagnosis. We can use DETER to identify and diagnose tail latency problem in Spark. We define each flow as all the packets belonging to the same Spark message. Spark usually sends one large message with multiple socket calls. So if a socket call starts after all the previous packets are acknowledged, we treat the socket call as a new message. Otherwise, we treat it as part of the previous message.

⁹We have not implemented replay for it, but it is not hard (§7).

We find that the tail latency of flows from HDFS are mostly caused by receiver limit, because their receive windows frequently reach zero.

The 99.9 percentile latency for flows between Spark workers experience a variety of problems as summarized in Table 2. For flows shorter than 1MB, their tail latency are mostly caused by packet drops (RTO or fast retransmission (FR)). For flows longer than 10MB, their tail latency are mostly caused by receive window frequently reaching zero (Rwnd=0).

The flows in the range [100KB,1MB] are of particular interests, because most of their tail latencies (18 out of 24) are caused by multiple delayed ACKs. We show the sender side packet trace for one of them in Figure 7; others have similar patterns. The sender frequently gets blocked after sending a burst of packets, until around 40 ms later when the ACK comes back. Such burst-40ms-ACK pattern repeats multiple times and causes excessive delay. This is out of our expectation, because the receiver should acknowledge every two data packets.

So we use DETER to replay again, and use TCP Probe to print the variables that decide whether to delay the ACK. We find that TCP explicitly delays the ACK because the free space in the receive buffer is shrinking. This suggests that the root cause is the application not reading the data in the receive buffer in time. So we replay again and confirms that the receiver application is slow in issuing receive socket calls. Our guess is that the application is busy with processing data, so the CPU is the bottleneck in this case.

DETER helps us to effectively diagnose the problems caused by the network (e.g., RTO, fast retransmission). In addition, it also helps us identify problems caused by applications. This is helpful because in data centers it is often unclear where the performance bottleneck is, and blaming the network is often the first reaction [28]. Unlike previous systems that infer the bottleneck [58, 28], DETER helps us quantify the duration of different bottlenecks without instrumenting the applications.

Overhead. DETER records a total of 200.6 MB data in the runtime. For comparison, Tcpdump uses 22.4 GB to record only the IP and TCP headers and timestamps and 6.5 GB after applying the state-of-the-art compression solution [38]. DETER storage is only 3.1% of compressed packet traces.

If we keep using DETER to monitor a data center that continuously runs such Spark jobs, DETER storage overhead translates to 2.8 GB/host/day. We can delete the data every day if we do not see performance problems.

We also use Linux perf [8] to evaluate the CPU overhead of DETER recording. DETER uses 0.094% of total CPU time.

5.2 Diagnosis in Data Center Workload

Evaluation setting. We now generate TCP tail latency problems using empirical workloads modeled after traffic patterns that have been observed in production datacenters. We run

Flow size (MB)	<0.1	[0.1, 1]	[1, 10]	>10
RTO	8	3	4	0
FR	74	0	0	0
Delayed ACK	0	0	18	0
Rwnd=0	0	0	1	1
Slow start	0	0	1	0

Table 2: Reasons for 99.9-th percentile latency for flows of different sizes in Spark.

a client-server RPC call software [6] in the same 20-node Amazon EC2 testbed. The clients set up a persistent TCP connection to each server, and request flows according to Poisson process from a random server. We set the flow sizes following the distribution observed in a production data center running web search applications [26]. We also add incast traffic pattern, by having the client simultaneously request 10 random servers, so the 10 servers respond synchronously causing incast. We set the average request rate to have an 80% network load, and 20% of the load is incast traffic. We generate a total of 280K requests over 380 persistent connections. All 20 nodes run both client and server.

Similar to the Spark program, we use Tcpdump to collect traces at both the runtime and the replay and show that DETER can provide deterministic replay for all the connections.

Diagnosing tail latency. In Table 3, we classify the root causes into five categories: congestion (i.e., low throughput), the fast retransmission happens very late (late FR), ACK drops (so the sender gets stuck), tail drops (so the packets at the end of a flow get dropped), and RTO.

We analyze the short flows (100KB-1MB) with latency above 99.9-th percentile as an example. At the 99.9-th percentile, flows experience 173.8 slow down of completion time compared to the case of running the flow alone. We make the following interesting observations:

RTO is not the main root cause of tail latency. A widely discussed reason for tail latency is RTO [29, 54]. But actually RTO is rare in this experiment. The reason is that when there are multiple requests in the same connection, later requests can help recover the packet losses of previous requests, so TCP loss recovery is effective in this scenario.

Fast retransmission (FR) is delayed for 10s of milliseconds. When these flows experience loss, the senders start FR after 10s of duplicate ACKs (dupACKs). This is unexpected because the normal threshold for FR is 3 dupACKs. And this is bad because short flows usually do not have so many dupACKs. In fact, most (22 out of 27) of these flows do not have enough dupACKs on their own; their FR starts 10s of milliseconds later when another request in the same connection starts and triggers more dupACKs.

With DETER, we can replay repeatedly and gain more insight into the problem. To understand why it requires so many dupACKs for FR, we replay the connection of the flow that experiences late FR with the highest slow down. We use TCP Probe [13] to print out the threshold for dupACKs (tp->reordering) on every ACK’s arrival during the replay.

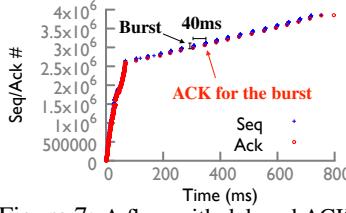


Figure 7: A flow with delayed ACKs.

Flow size (MB)	<0.1	[0.1, 1]	[1, 10]	>10
Congestion	149	35	25	2
Late FR	29	27	0	0
ACK drops	0	2	0	0
Tail drops	4	1	0	0
RTO	2	1	2	0

Table 3: Reasons for 99.9-th percentile latency for flows of different sizes in data center workload.

We find that this threshold starts at 3, but later increases (and never decreases), so when the flow that experience late FR arrive, the threshold is 45. We search in the TCP code, and find the threshold only increases when TCP detects reordering. So we replay again and print out the ACKs when the threshold increases, and find that they do reflect reordering.

A quick fix is to set the upper bound of this threshold (net.ipv4.tcp_max_reordering) lower, but it risks spurious retransmission in the presence of reordering. A potential optimization to TCP may be regularly reducing the threshold. **Overhead** DETER records a total of 103.8 MB, which is 2.1% of compressed packet traces. (Tcpdump records 16.8 GB, or 4.9 GB with compression.)

The CPU overhead is 1.49%. The overhead is higher than in Spark, because the client-server software only uses CPU to send and receive data, without any data processing. In fact, it spends 99.78% of its CPU time in the networking stack (including DETER). So 1.49% is very close to the lower bound of DETER CPU overhead.

5.3 Diagnosing RTO in a Testbed

RTO usually has a large impact on the latency. However, there are many different causes of RTO, and often involves different parameters. In §2.1 we have shown one case. Here we show two other causes for RTO that we see in our testbed. In all cases, TCP counters can only be the first step—knowing that timeouts and packet losses happen. But it is very hard to realize the relationship between the timeout and other events. With DETER, we can replay the connection to get the packet traces and trace the TCP execution to dig out the root cause.

Evaluation setting. We use 3 hosts connected through a single switch via 10 Gbps links. We pick two of the hosts as senders and the rest one as the receiver. Each of the senders sends one long flow (10 MB) to the receiver. One of the senders also sends a short flow (10 KB) to the receiver.

Root cause 1: Not enough dupACKs. In this case, the short flow experiences RTO. We use DETER to replay the connections and capture the packet trace. The trace shows that the short flow sends 7 packets in the first round, and the 5-th packet gets dropped. Thus, although the 6-th and the 7-th packets trigger dupACKs, the number of dupACKs is not enough to trigger fast retransmission.

Root cause 2: Setting large TCP receive buffer size. The receive buffer size is a frequently tuned parameter for networks with different bandwidth-delay products. For example,

an inter-data center connection with 100ms RTT and 1Gbps bandwidth need 12.5MB buffer size. Unfortunately, a large receive buffer can cause RTO issues. Here we show the diagnosis in an example with 10MB receive buffer.

We first replay and capture the packet trace. However, the time series of data packets and ACK packets shows a very different scenario. After a packet loss, there are more than 3 dupACKs, but the sender does not fast retransmits the lost packet. This is unexpected because just 3 dupACKs should trigger fast retransmission.

We first suspect that this may be the late FR case that we show in §5.2, so we replay again and print out dupACK threshold. But it shows that the threshold is 3.

To dig out the root cause, we replay again, and use Ftrace to get the function call graph of handling each ACK. Surprisingly, We find that TCP does not go to the dupACK branch. This means TCP even does not treat them as dupACKs. With the surprise in mind, we replay again and use TCP Probe to print the variables that are used to classify ACKs as duplicates. The `f`lag variable reveals the reason: TCP does not treat the ACKs as duplicate because the flag's WIN_UPDATE bit is set [9]. This means each of these ACKs carries a different window size. We confirm this in the packet trace: each ACK carries a larger window size.

The direct cause for this problem is that the receive buffer size is very large. The receive window starts with a small size, and increases two MSS per received data packet until reaching configured receive buffer size. Thus, the window size keeps growing throughout the lifetime of this connection. However, this also suggests a potential optimization to TCP that it should have a smarter classification for dupACKs.

5.4 Evaluating Switch Queue Replay

Now we evaluate the accuracy of replaying switch queues in our testbed and simulation. We first run traffic in our testbed, and replay the queue to evaluate the accuracy. Then to understand how the switch queue replay works under more switches and more congestions, we run empirical traffic in a large scale simulation, and replay the queues.

5.4.1 Evaluation with Testbed

Evaluation setting: The testbed comprises 3 hosts. To get the groundtruth of the queue content, we use a NetFPGA switch and program it to send out the queue content through the unused port. The switch has a total of 393 KB buffer shared across 3 ports¹⁰. The MTU is 1500 B. The host clocks are synchronized with 100s of nanoseconds precision by Precision Time Protocol [2].

Because congestion is the most challenging scenario to replay, we set traffic to have severe congestion. We use 2 hosts as senders and the rest one as a receiver. Each of the two senders generates 2 long flows (10 MB each) to the

receiver simultaneously. Each sender also generates 4 short flows (10 KB each) to the receiver, one every 5 ms. So there are a total of 4 long flows and 8 short flows.

During the runtime, we use DETER to collect data, and also collect the content of the congested queue. Then we first replay each connection to get the packet trace, and replay the queue. We replay the queue in both the original testbed, and in a simulation. The simulation has the same topology, and simulates the same link throughput, latency, and buffer setting as the NetFPGA switch.

Accuracy: The metric we use is *queue content difference*: the difference between the runtime queue q_{run} and the replay queue q_{rep} that each packet sees. Formally, we define $qdiff = \sum_{f \in q_{run} \cup q_{rep}} |f.size_{run} - f.size_{rep}|$, where $f.size_{run}$ means the bytes of flow f in the queue during the runtime and $f.size_{rep}$ is for the replay.

On average the queue content difference is 0.57 MTU in the testbed, and 1.0 MTU in the simulation. On the 99-th percentile, the difference is 4.83 MTU in the testbed, and 3.85 MTU in the simulation, both of which are very low compared to the buffer size. Replay in the testbed has a slightly higher tail difference because timing variations (e.g., thread scheduling) exist in the testbed, but not in the simulation.

5.4.2 Evaluation in Large Scale Simulation

Our testbed evaluation shows that the replay is effective for one switch. In production, there are more hosts, multiple layers of switches, and more congestions across the switches. So we use simulation to evaluate a larger scale network.

Evaluation setting: We run the simulation in ns3 [15], with 320 switches and 1024 hosts connected through a K=16 Fat-Tree with 10 Gbps links. Each switch has 2 MB buffer, shared by all its 16 ports¹¹. To simulate the clock synchronization error, we add a delta to each host's clock, with a uniform distribution between 0 and 5 us¹².

The traffic includes both empirical background traffic that follows the flow size distribution of a web search workload [26], and incast traffic. The source and the destination of each background flow are chosen uniformly random. The flow arrival rate follows a Poisson process, and we vary the flow arrival rate to achieve different levels of traffic load, from 10% to 80%. We also generate the incast traffic by having the client simultaneously requests 40 servers, each of which sends back 250 KB response (10 MB total response size). We generate 2400 incast per-second.

To understand how the sampling affects the accuracy, we sample the sending times with different threshold of error: 2 us, 5 us, and 10 us. We then replay the queues.

Accuracy: Figure 8 shows the queue content difference of all queues in the network. The difference increases mildly

¹¹For the buffer sharing policy, we use the commonly used dynamic threshold [31] with $\alpha = 4$.

¹²PTP in LAN can achieve sub-microsecond accuracy, and under 3.2 us in WAN most of the time [12]. More advanced clock synchronizations [41, 34] guarantee sub-microsecond accuracy. We choose 5 us to be conservative.

¹⁰We use the commonly used dynamic threshold [31] with $\alpha = 4$.

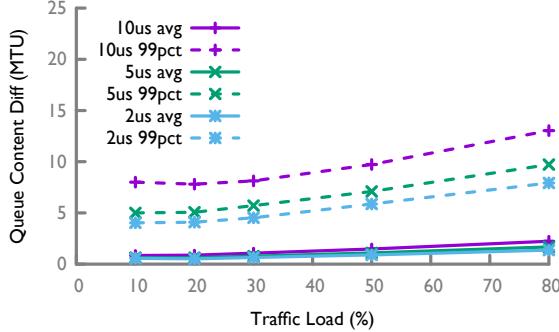


Figure 8: The queue content differences of replay in simulation.

with higher load, both on average and at the 99-th percentile. For example, with 5 us threshold of error, at 30% load, the maximum load of most data centers in practice [50, 14], the differences are 0.78 MTU on average and 5.7 MTU at 99 percentile. At 80% load, an extremely high load, the differences are 1.7 MTU on average and 9.7 MTU at the tail. It also shows that a 5 us threshold achieves relatively good accuracy: it only increases less than 0.3 MTU (on average) and less than 1.8 MTU (at tail) difference compared to 2 us.

We also compare the packet drop error with and without the drop accuracy improvement. The drop error is $\frac{\#false_accept + \#false_drop + \#drop_wrong_location}{\#packets_dropped_in_either_runtime_or_replay}$. Our evaluation shows that the drop error reduces significantly. For example, for 5 us sampling threshold at 30% load, the error reduces from 58.3% to 2.87%.

The drop error is low under various loads, from 2.52% at the 10% load, to 3.81% at the 80% load. There is no false-drop, as the simulation can avoid this (§3.4). Most errors are false-accepts. Only less than 0.37% of the drops show up at wrong locations, which means we can trust the drops in the replay with high confidence, because only 0.37% of them give wrong locations. Since 80% load is extremely high and we also added incast traffic, we believe most data centers would not stress the network at this level, so we believe the drop error rate is low in general.

5.4.3 Diagnosing RTO Using Queue Information

Sometimes RTO can be caused by the queuing mechanisms of switches. We run the traffic in a 4 host (A, B, C, D) testbed. B and C respectively send 5 long flows (500MB each) to A. In the middle of the long flow transmission, A, C and D respectively send 5 short flows (100KB each) to B simultaneously. Two of the long flows from C to A experience RTO. Using DETER to replay them, we find that they both drop a whole window of packets, at the same time. But this time we cannot find any problem in the TCP stack. So we use the packet traces for all the connections to replay switch queues in an ns3 simulator.

During the replay, we collect all the enqueue and drop events at the switch. The packets are dropped at queue 0 of the switch. Figure 9 shows the length and the cumulative drop count of queue 0. At around 10 ms, there is a sudden increase

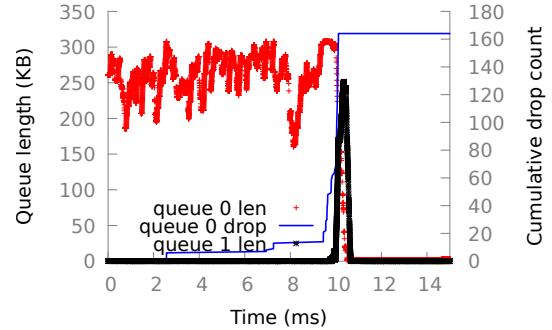


Figure 9: The lengths of two queues that share the buffer.

in drops. Unexpectedly, the queue length is decreasing at the same time. We suspect that the switch buffer sharing [31] causes this problem, because the threshold of a queue decreases when the total buffer utilization of the switch grows.

So we replay again and monitor other queues of the switch. We find that a burst of packets builds up queue 1 at the time of queue 0 drops packets. This confirms our hypothesis.

This problem could also happen in data centers because most data center switches use shared memory across different queues. The threshold of any queue is proportional to the total free buffer size. If the switch buffer utilization suddenly increases, the threshold shrinks, which causes temporary blackhole at the almost-full queues (e.g., queue 0 in Figure 9). The sudden increase in switch buffer utilization can happen because of incast, which is common in data centers.

6 Related Work

Replay systems. There are many replay systems for kernel, multicore applications and distributed systems [52, 40, 33, 49, 27, 37, 17]. They record the input and interaction of the target of replay (a subset of components of the entire system) with the rest of the system to isolate the target, and then make sure the target itself replays accurately. There are two ways to directly adopt such replay techniques for TCP: (1) Replay each host’s TCP stack separately. This means we should record every packet as they are the input to the stack, which is a significant overhead. (2) Replay the whole network altogether, including all connections and switches, which is very expensive and hard to get right as shown in §2.2. DETER customizes replay techniques for TCP: we replay each connection (a pair of TCP stacks), and only record the mutations to the packet stream in between (drop/ECN) to reduce the overhead of recording every packet, while avoiding replay the whole network together. We also introduce customized solutions to reduce the overhead of recording non-deterministic variables inside the TCP stack.

Monitoring tools in data centers. Per-packet monitoring tools [18, 10, 38] and TCP execution tracing tools [13] provide detailed information for diagnosis, but running them continuously is too expensive. To reduce overhead, people

collect coarser-grained information such as TCP counters [58, 28] or per-flow stats on the host [53] or switches [4, 43]. There are also query systems (e.g., Everflow [59], Trumpet [46], Marple [47]) that allow operators to specify the packets and events to capture in a network. DETER is complementary to these works in that it enables deterministic replay for debugging the same performance problem iteratively. DETER requires low recording overhead at runtime and allow operators to use all kinds of monitoring tools during the replay.

Other network-related replay. OFRewind [57] replays the switch control plane, while DETER replays TCP and the switch data plane. Monkey [30] and Swing [55] are tools that synthesize testing traffic based on the runtime recorded traffic pattern, while DETER focus on replay for diagnosis.

7 Discussion

Extension to other network transport features: Here are a few examples of transport features that may affect the replay. *Generic receive offload (GRO):* If GRO [3] is enabled, we also need to record the way it merges packets. It just requires recording the number of packets being merged into one segment, which is available in the skb metadata and just costs 6 bits per merge. Usually each merging contains 10s of packets, so the overhead is low. During the replay, the packet corrector should also merge the incoming packets as recorded.

Delay-based congestion control (CC): Our current prototype is based on loss-based CC. To extend our solution to delayed-based CC, we need to record the timestamps that used for updating CC states. We can compress them a lot, because consecutive timestamps differ by a few microseconds most of the time, so we just need a few bits to record the delta.

RED in switch: RED randomly drop packets. Replaying the queues and drops may have a large error in this case, but replaying TCP connections is not affected. This shows the benefit of our design decision: decoupling the replay of each individual connection, so that it does not depend on switches.

Use cases of DETER. DETER is designed for ease of use. The only requirement is that the user turns on DETER on both endpoints of the connection, which is often the case for network operators and cloud tenants. Internet application developers can also use DETER for performance testing. Data center network operators may also benefit from replaying the switch queues, because they may have the network topology and switch data plane simulators.

Host stack changes. If the host stack changes, DETER may need to change accordingly, but it is not hard. First, Linux already abstracts CC out of basic TCP framework, so changes to CC does not need to recode DETER in the basic framework, which contains most of the recording. Besides, we have principles for what to record and replay (Table 1 and §4), so it would be easy to identify the required changes to DETER.

We expect the recording overhead would not change much with stack changes, because most of the overhead comes

from socket calls and lock acquisitions, both of which are not sensitive to stack changes: socket call is determined by the applications, and most lock acquisitions are for receiving packets whose amount is determined by traffic volumes. The overhead associated with kernel variables is very small with our technique of recording their updates or influences, and we believe this benefit remains in the future.

Generality to other transport protocols. We believe the replay technique is general across different protocols. Basically, what other transport protocols do are not very different from TCP: reads from/writes to applications, sends/receives packets, and possibly controls sending rate based on packet measurement. Similar to TCP, we just need to record the interaction with the application and the network, and then make sure we handle the concurrency inside the protocol.

Network failures. Network failures (e.g., routing fluctuations or blackholes) do not affect DETER replaying the connections, but do affect DETER replaying the switch queues which assumes that the routing states are stable. However, network failures are themselves bigger problems than the problems related to switch queueing, and there are many other works focus on addressing such issues [59, 36, 45, 56, 43]. DETER is complementary to these works, because it helps to understand how TCP reacts to such conditions.

Storage overhead of socket calls. Usually the number of socket calls is much smaller than the number of packets. Production data center survey [26, 35] shows that most network bytes are from large flows (>1 MB), which usually mean large send/receive sizes. Moreover, even if an application has many short messages, the developers tend to batch them into a large one to reduce the CPU overhead. If some network does only have applications that generate small socket calls, recording every socket calls may be high overhead.

8 Conclusion

DETER enables deterministic TCP replay, which can reproduce performance problems, provide packet traces and support tracing of TCP executions. DETER eliminates the butterfly effect by replaying individual TCP connections separately and capture all the interactions between a TCP connection with the application and the network in a lightweight fashion. We demonstrate that DETER is effective in diagnosing a variety of TCP performance problems.

9 Acknowledgement

We thank our shepherd Alex C. Snoeren and NSDI reviewers for their helpful feedback. We thank Wei Bai for providing diagnosis cases. We also thank Danyang Zhuo, Yurong Jiang, Sivaramakrishnan Ramanathan and Bradley McDaniel for providing feedback. This paper is supported by the NSF grants CNS-1834263.

References

- [1] ftrace, 2008. <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>.
- [2] IEEE Standard 1588-2008, 2008. <http://ieeexplore.ieee.org/document/4579760/>.
- [3] Generic receive offload, 2009. <https://lwn.net/Articles/358910/>.
- [4] NetFlow, 2009. <http://www.cisco.com/go/netflow/>.
- [5] Broadcom moves from simulation to emulation with Mentor, 2014. <https://www.electronicsweekly.com/uncategorised/broadcom-moves-from-simulation-to-emulation-with-mentor>.
- [6] Empirical Traffic Generator, 2014. <https://github.com/datacenter/empirical-traffic-gen>.
- [7] NetFilter, 2014. <http://www.netfilter.org/>.
- [8] Linux perf, 2015. https://perf.wiki.kernel.org/index.php/Main_Page.
- [9] TCP window updates combined with dup acks sent in response to packet loss, 2015. <https://www.ietf.org/mail-archive/web/tcpcm/current/msg09480.html>.
- [10] In-band Network Telemetry, 2016. <http://p4.org/p4/inband-network-telemetry>.
- [11] Cisco Packet Tracer, 2016. <https://learningnetwork.cisco.com/docs/DOC-29644>.
- [12] IEEE 1588 PTP clock synchronization over a WAN backbone, 2016. <https://www.endace.com/ptp-timing-whitepaper.pdf>.
- [13] TCP Probe, 2016. <https://wiki.linuxfoundation.org/networking/tcpprobe>.
- [14] Microsoft Keynote at SIGCOMM 2017, 2017. <http://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf>.
- [15] Network Simulator 3, 2017. <https://www.nsnam.org/>.
- [16] RACK: a time-based fast loss detection algorithm for TCP, 2017. <https://tools.ietf.org/html/draft-ietf-tcpcm-rack-02>.
- [17] Mozilla RR, 2017. <https://rr-project.org/>.
- [18] Tcpdump, 2017. http://www.tcpdump.org/tcpdump_man.html.
- [19] DCTCP Bug, 2018. <https://github.com/torvalds/linux/commit/27cde44a259c380a3c0906fc4b42de7dde9b1ad>.
- [20] Amazon EC2, 2018. <https://aws.amazon.com/ec2/>.
- [21] Boson NetSim, 2018. <http://www.boson.com/netsim-cisco-network-simulator>.
- [22] Time Split to the Nanosecond Is Precisely What Wall Street Wants, 2018. <https://www.nytimes.com/2018/06/29/technology/computer-networks-speed-nasdaq.html>.
- [23] P4 language, 2018. <https://p4.org/>.
- [24] Spark TeraSort, 2018. <https://github.com/ehiggs/spark-terasort>.
- [25] Linux TCP Github, 2019. <https://github.com/torvalds/linux/tree/master/net/ipv4>.
- [26] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.
- [27] Gautam Altekar and Ion Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.
- [28] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.
- [29] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, 2009.
- [30] Yu-Chung Cheng, Urs Holzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *Usenix*, 2004.
- [31] Abhijit K. Choudhury and Ellen L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Trans. Netw.*, 1998.
- [32] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communication of the ACM*, 2013.
- [33] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, 2006.
- [34] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat.

- Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.
- [35] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. V12: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009.
- [36] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference*, 2015.
- [37] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.
- [38] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.
- [39] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.
- [40] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.
- [41] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.
- [42] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.
- [43] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.
- [44] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International Conference on Emerging Networking EXperiments and Technologies*, 2016.
- [45] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.
- [46] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.
- [47] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.
- [48] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.
- [49] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multi-processors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.
- [50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.
- [51] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagal, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hözlé, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM SIGCOMM Conference*, 2015.
- [52] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, 2004.

- [53] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with path-dump. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.
- [54] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009.
- [55] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2006.
- [56] Xin Wu, Daniel Turner, George Chen, Dave Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *Proceedings of the 2012 ACM SIGCOMM Conference*, 2012.
- [57] Andreas Wundsam, Dan Levin, Srinivas Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011.
- [58] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.
- [59] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, 2015.

JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs

Eunji Jeong
Seoul National University
Joo Seong Jeong
Seoul National University

Sungwoo Cho
Seoul National University
Dong-Jin Shin
Seoul National University

Gyeong-In Yu
Seoul National University
Byung-Gon Chun[†]
Seoul National University

Abstract

The rapid evolution of deep neural networks is demanding deep learning (DL) frameworks not only to satisfy the requirement of quickly executing large computations, but also to support straightforward programming models for quickly implementing and experimenting with complex network structures. However, existing frameworks fail to excel in both departments simultaneously, leading to diverged efforts for optimizing performance and improving usability.

This paper presents JANUS, a system that combines the advantages from both sides by transparently converting an imperative DL program written in Python, the de-facto scripting language for DL, into an efficiently executable symbolic dataflow graph. JANUS can convert various dynamic features of Python, including dynamic control flow, dynamic types, and impure functions, into elements of a symbolic dataflow graph. Our experiments show that JANUS can achieve fast DL training by exploiting the techniques imposed by symbolic graph-based DL frameworks, while maintaining the simple and flexible programmability of imperative DL frameworks at the same time.

1 Introduction

In recent years, deep neural networks have been widely used in various application domains such as computer vision, speech, and natural language processing for their powerful capabilities of extracting abstract features from data. Scientists have created deep learning (DL) frameworks – TensorFlow [1], PyTorch [31], Caffe2 [11], MXNet [6], and many more [2, 12, 27, 29, 40, 42, 46, 49] – to improve the performance of deep neural networks in various jobs and promote the use of deep neural networks in both production and research.

Such DL frameworks can be classified into two distinct families depending on their execution models. One

family comprises frameworks that base their execution on symbolic graphs constructed from DL programs. The other family consists of frameworks that directly execute DL programs in an imperative manner.

Symbolic graph execution. Frameworks such as TensorFlow [1], Caffe2 [11], and MXNet [6] formulate neural networks as symbolic dataflow graphs. Graph vertices denote the states and operations of a neural network, while graph edges indicate the flow of data between vertices. Operations in the graph are executed as their dependencies are solved, similar to how most dataflow systems process dataflow graphs [9, 17]. The graph representation allows the framework to identify which operations can be run in parallel, and apply various compiler optimization techniques such as common subexpression elimination or constant folding to generate optimized versions of graphs. Moreover, it is easy to process dataflow graphs on accelerator devices or deploy graphs across multiple machines by assigning an operation to the appropriate device or machine [24].

However, the separation of building a symbolic graph and executing it complicates user experience, because users are not actually running any numerical computations when defining neural networks through the framework interface. Rather, they are constructing graphs that will be executed later through separate functions.

Imperative program execution. In contrast, frameworks including PyTorch [31], TensorFlow Eager [40], and MXNet Imperative [26] have adopted the execution model of running operations imperatively, without going through a separate graph construction phase. Stemming from popular Python libraries for scientific, numerical computation such as NumPy [48] and Scikit-learn [4], this imperative approach is useful for rapidly experimenting and working with new neural network models, particularly those with complex structures. The native control flow statements of Python can be exploited to build mod-

[†] Corresponding author.

els of interest. Unfortunately, skipping the formation of a dataflow graph means that such frameworks lose the chance to apply the many optimizations that were possible in the symbolic graph execution model, leading to significant performance differences for certain models.

The different characteristics of DL frameworks suggest that we cannot achieve high performance and good usability at the same time. To reach high performance, we must sacrifice framework usability to a certain extent, and vice versa. Otherwise, users are forced to resort to an awkward approach of learning how to use several frameworks and switching between them according to the current task in hand.

From imperative programs to symbolic graphs. In this paper, we propose to transparently convert imperative Python DL programs into symbolic dataflow graphs before execution. By not altering the user-facing interface for building neural networks, we maintain the flexible programmability of frameworks with imperative execution models. At the same time, behind the scenes, we execute the symbolic graph versions of the imperative programs to enjoy the performance optimizations done by symbolic graph execution models.

However, this approach introduces a technical challenge of capturing the dynamic semantics of an imperative Python program in a symbolic dataflow graph. The dynamic aspects of Python, including dynamic control flow, dynamic typing, and impure functions, must be embedded in a symbolic graph correctly while providing the performance of symbolic graph execution frameworks.

To this end, we present JANUS, a DL framework that achieves the best of both worlds by receiving an imperative DL program as input and creating symbolic graphs of the program accordingly with speculative program context assumptions. JANUS makes environment assumptions on the program context (e.g., constant variables and branches) based on past iterations to simplify the dynamic nature of the program and transform the program into a symbolic graph. These assumptions are speculative, because the context may change during execution; an incorrect assumption results in an invalidation of a symbolic graph, in which case JANUS falls back to imperative execution to guarantee correctness. For design (Section 4.3.1) and implementation (Section 4.3.2) reasons, JANUS converts only the subset of Python programs into the efficient symbolic graphs, but the rest of them still can be executed imperatively, ensuring the full Python coverage.

We have implemented JANUS on TensorFlow 1.8.0 [1]. To demonstrate the performance of JANUS, we evaluated JANUS with 11 imperative DL programs in five categories: convolutional, recurrent, and recursive neural networks,

```

1 class RNNModel(object):
2     def __call__(self, sequence):
3         state = self.state
4         outputs = []
5         for item in sequence:
6             state = rnn_cell(state, item)
7             outputs += [state]
8         self.state = state
9         return compute_loss(outputs)
10
11 for sequence in sequences:
12     optimize(lambda: model(sequence))

```

Figure 1: A Python program that implements training process of a recurrent neural network (RNN) in an imperative manner. For each item in the sequence, `rnn_cell` function is called to produce the next state required for the next `rnn_cell` invocation. After finishing up processing the whole sequence, the model holds the final state by replacing `self.state` attribute for processing the next sequence.

generative adversarial networks, and deep reinforcement learning models that extensively use the dynamic features of Python. JANUS converted the programs into symbolic dataflow graphs successfully, trained the models to reach target accuracy with up to 18.7 times higher throughput compared to TensorFlow Eager, while executing the identical imperative programs.

2 Challenges and Proposed Solution

2.1 Challenges

Converting an imperative program written in Python into a DL dataflow graph brings on many challenges, because dataflow graphs consist of a restrictive set of operations, lacking the dynamic semantics of the programming language. More specifically, various characteristics of a Python program, such as the execution count and execution order of statements, the types of expressions, or the global program execution state, can only be determined after the program is actually executed. For the rest of this paper, we will refer to these characteristics as the *dynamic* features of Python. In contrast, DL dataflow graphs are expected to be defined before the computation starts, to apply aggressive graph optimizations and efficiently schedule the graph operations by viewing the entire graph. In this sense, DL dataflow graphs are usually considered to be *static* [23, 29, 31]. The difference in characteristics makes it difficult to embed dynamic Python features in static dataflow graphs.

Figure 1 depicts a DL program written in Python, of which semantics are difficult to be captured in a dataflow

Frameworks	Imp. pgm	Sym. exec	Correctness			Optimization w/ runtime info	Language
			DCF	DT	IF		
Symbolic: TensorFlow (TF), Caffe2, MXNet	✗	○	–	–	–	–	Python
Imperative: PyTorch (PTH), TF Eager, DyNet	○	✗	–	–	–	–	Python
Imperative to Symbolic							
Tracing: TF <code>defun</code> , PTH JIT <code>trace</code> , MXNet Gluon	○	○	✗	△	✗	○ (unsafe)	Python
JAX	○	○	○	△	✗	○ (unsafe)	Python subset
Swift for TensorFlow (S4TF)	○	○	○	–	○	✗	Swift
TF AutoGraph	○	○	△	△	△	○ (unsafe)	Python subset
PTH JIT script	○	○	○	△	△	✗	Python subset
JANUS	○	○	○	○	○	○	Python

Table 1: Comparison of DL frameworks with respect to correctly supported features for converting imperative programs into symbolic graphs ("Correctness") and the ability to optimize the generated graphs with the information given only at program runtime ("Optimization w/ runtime info"). Optimizations can be incorrect in some frameworks ("○ (unsafe)"), not preserving the original semantics of Python. The host language is also specified.

graph **correctly** due to the following representative dynamic features of Python.

- **Dynamic control flow (DCF)** Conditional branches and iterative loop constructs have different execution paths depending on intermediate values. Lines 5-7 of Figure 1 show an example of an iterative loop construct used in a DL program. Such control flow statements are intensively used in Python and must be correctly represented in the dataflow graph.
- **Dynamic types (DT)** Python is a dynamically-typed language, i.e., the type of a Python expression can only be determined at program execution time. The example program in Figure 1 does not have any type annotations (e.g. `int` or `float`), which makes it difficult to statically decide the type of target dataflow graph operations. Furthermore, various non-numerical types of Python, such as lists, dictionaries, and arbitrary class instances, are even harder to be converted into elements of a dataflow graph, of which vertices usually output numerical arrays.
- **Impure¹ functions (IF)** Another useful feature for using Python is the ease of accessing and mutating global states within functions. In Figure 1, the function `__call__` reads from and writes to an object attribute² at Lines 3 and 8, to pass the final state of a sequence to the next sequence. Since the modified global states can make the following function call behave differently, such reads and writes of global states must be handled correctly while generating dataflow graphs.

Moreover, correctness is not the only issue when converting an imperative program; achieving the high per-

formance of state-of-the-art symbolic graph execution DL frameworks is also a challenge on its own. State-of-the-art frameworks require additional information on dynamic types and control flow in order to optimize graph execution. However, a naïve, one-shot converter would be unable to extract this information from an imperative program before execution, and thus is incapable of supplying frameworks with such hints. For instance, if the input sequence at Line 2 in Figure 1 is expected to always have a fixed length, then that information can be exploited to unroll the following loop at Line 5 when generating the corresponding dataflow graph. It is unclear how a naïve converter would do this without actually executing the program to check the loop length.

2.2 Related Works

Previous works that try to translate a Python DL program into a dataflow graph either fail to capture the important dynamic semantics of Python, or run in slower performance due to the lack of sufficient information at graph build time. Table 1 summarizes state-of-the-art DL frameworks alongside their execution models and their status regarding the coverage and efficiency of graph conversion support.

Tracing-based graph generation approaches such as PyTorch’s JIT compiler (`torch.jit.trace`) [31], MXNet Gluon [27], and the `defun` [43] functionality of TensorFlow Eager [40] execute the imperative program once, and convert the single execution trace directly into a dataflow graph. Though this approach enables generating optimized symbolic graphs with sufficient information gathered from a specific execution trace, it fails to capture dynamic semantics of the Python interpreter correctly, leading to incorrect computation results for dynamically

¹A *pure function* is a function whose return value is determined only by its parameters, and has no side effects.

²"class members" in C++ terminology, except that the attributes are stored in dictionaries, without fixed data layout.

changing execution paths, dynamic types of non-tensor or non-input expressions, or impure functions of Python at runtime. Moreover, these approaches currently do not give any feedback about incorrectly-converted control flows to users, making the problem even worse.

On the other hand, there exist other approaches that select a less-dynamic host language and therefore succeed in capturing the wider semantics of source programs. JAX [12] limits the Python syntax and supports converting only pure-and-statically-composed functions. S4TF [42] supports Swift, losing the merit of supporting Python, the de-facto standard programming language for DL programming, and introduces new programming models that most DL researchers are unfamiliar with. Moreover, since the graph conversion occurs before actually executing the program, these approaches can miss the opportunity to further optimize the graph with the information only obtainable during the program execution. For example, always converting a Python loop into control flow operations can be sub-optimal if the loop iteration count is known to be fixed.

Concurrent works including AutoGraph-enabled TensorFlow `defun` functionality [45] and the "scripting" mode of PyTorch JIT (`torch.jit.script`) [31] also have limitations. AutoGraph makes users to explicitly provide the necessary information, or generates incorrect or sub-optimal graph in some cases, all of which could be avoided if sufficient information existed. For example, users must explicitly specify the types of Python lists, prohibiting the dynamic typed or heterogeneous elements. For another example, for dynamic control flow statements, the statements with non-tensor predicates are always unrolled, which is error-prone, and the statements with tensor-typed predicates are always converted to control flow operations, which can be sub-optimal. In the "scripting" mode of PyTorch JIT, users must use TorchScript, a subset of Python which does not allow variables to have dynamic types. Further graph optimizations based on the runtime information are also not possible.

2.3 Proposed Solution: Speculative Graph Generation and Execution

Existing optimizers and compilers for dynamic languages suggest a useful technique for performing such conversions from imperative programs to symbolic dataflow graphs: *speculative optimization*. Managed language runtimes have succeeded in exploiting the inherent static nature of dynamic programs which rarely changes during the execution to convert them into static, low-level representations while maintaining correctness. For example, JavaScript just-in-time (JIT) compilers convert dynamic

JavaScript programs into efficient machine code, and this conversion is done speculatively assuming that the program inherently maintains some statically fixed structures over repeated executions. In case this assumption breaks, the program falls back to the interpreter and attempts to compile the program again with different assumptions.

We propose to adopt this concept of speculative optimization when converting imperative DL programs into symbolic dataflow graphs. Converting various dynamic features like dynamic control flow and impure functions correctly may impose some inevitable overheads if we generate dataflow graphs in a conservative manner. To overcome this challenge, JANUS makes assumptions about the program's behavior based on the runtime profiling information, and generates a symbolic graph tailored for the assumptions. This speculatively constructed dataflow graph can show much better performance compared to the conservative counterpart due to specializations. If the assumptions do not hold, JANUS builds a new dataflow graph based on different assumptions. Since a DL program comprises a number of iterations of an optimization procedure, the speculative approach is a good fit since the interpreter is likely to execute specific code blocks of the program repeatedly.

Unlike the JIT compilers of managed language runtimes, however, the goal of JANUS is not to optimize the host language execution itself. In fact, when running imperative DL programs, the execution time of the language runtime is usually much shorter compared to the execution time of the mathematical operations for DL, such as convolution or matrix multiplication. However, since these mathematical operations are usually implemented in separate low-level language like C++, existing JIT compilers of managed language runtimes would execute them just as separated function invocations. Under such an execution model, it is impossible to see the multiple mathematical operations at once and apply compiler optimizations or execute them in parallel. On the other hand, JANUS understands the function invocations for such mathematical operations, and converts them into appropriate target graph operations, which can be optimized and be executed efficiently by symbolic graph executors.

3 JANUS System Design

In this section, we introduce JANUS, a DL framework that receives an imperative DL program and either executes it as is directly, or generates a symbolic graph version of the program and executes the graph instead.

The input program for JANUS is assumed to be written using the API and the programming model of existing imperative DL frameworks like TensorFlow Eager [40].

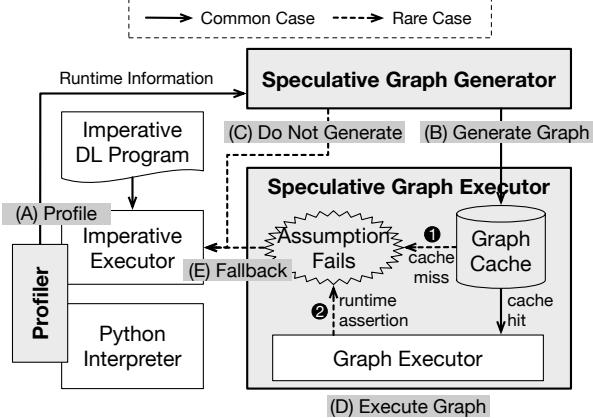


Figure 2: An illustration of the execution model of JANUS, showing how a DL program is processed by several components. *Profiler* observes imperative program execution and collects information to make the realistic assumptions. *Speculative Graph Generator* generates dataflow graphs from the program and hands the optimized graphs over to *Speculative Graph Executor*. The *Speculative Graph Executor* actually runs the generated graph and handles assumption failures.

Given an input program, JANUS extracts the main neural network computation part, over which the automatic differentiation is performed, and starts the speculative graph generation and execution process. From the user's point of view, the whole graph conversion and execution process is done transparently; in other words, the given DL program is automatically transformed into a corresponding graph representation without any interactions.

Figure 2 depicts the system components and the overall execution model of JANUS. The common case in which an efficient dataflow graph is utilized is depicted as solid lines in the figure, while the rare case where the graph representation is not available is depicted as dotted lines.

3.1 Fast Path for Common Cases

Runtime profiling. Once JANUS receives a DL program, the program is first executed imperatively, while the *Profiler* gathers runtime information required for making reasonable assumptions (Figure 2 (A)). Various information is collected, including control flow decisions on conditional branches, loop iteration counts for iterative loop constructs, variable type information, non-local variables, object attributes, and so on.

Symbolic graph generation. After a sufficient amount of information has been collected, the *Speculative Graph Generator* tries to convert the program into a symbolic dataflow graph with the assumptions based on the runtime information (Figure 2 (B)). To avoid making any

hasty generalizations, JANUS does not begin graph generation until the executor has profiled the program for a certain amount of iterations.³ First, JANUS traverses the abstract syntax tree (AST) of the DL program and generates the corresponding graph elements for each AST node, along with assertion operations that can validate the context assumption at runtime. Since JANUS targets DL programs, operations for automatic differentiation and model parameter updates are also automatically inserted if necessary. Next, the generated graph is further optimized by the post-processor, of which optimizations were not applicable to the original imperative DL program. Finally, the optimized graph and the assumption that were used to generate the graph are saved into the graph cache.

Graph execution. If a graph representation with correct assumptions regarding the program context is available, the *Speculative Graph Executor* executes the symbolic graph (Figure 2 (D)). Note that the same graph can be reused multiple times, given that the runtime context assumption holds for future invocations.

3.2 Accurate Path for Rare Cases

Assumption failure. Handling the assumptions is important to guarantee the correctness of the converted graph. If an assumption is proven to be wrong, the associated graph cannot be executed for the current runtime as it may produce incorrect results. Instead, JANUS falls back to the imperative executor (Figure 2 (E)) and resumes runtime profiling to make more relaxed assumptions for subsequent executions.

Assumptions that can be validated before actually executing the associated graph, such as type assumptions on input arguments, are checked when retrieving the graph from the graph cache (Figure 2 ①). In the unfortunate case where such an assumption is wrong, JANUS regards this as a cache miss and falls back to imperative execution.

On the other hand, for assumptions that can only be validated during graph execution (Figure 2 ②), it can be erroneous to simply abort the current execution to fall back to the imperative executor, because the global state may have been changed during the current execution. To solve this issue, JANUS defers state update operations until every assumption is validated (Section 4.2.3). This way, even if an assumption turns out to be wrong during computation, no state update operation has been triggered yet and thus no state has been mutated. Knowing this, the system can safely stop the current execution. In other words, states are updated in an all-or-nothing manner.

³We found that 3 iterations were enough to come up with a decent program context assumption, for our experimental workloads.

In order to validate an assumption, a runtime assertion is encoded into the symbolic graph as an operation called `AssertOp`. The `AssertOp` aborts the graph execution if the given condition fails. It also reports which assumption has been broken, and this information is used to give up further optimizations that rely on the assumptions that repeatedly break.

Imperatively executed programs. With Turing-complete graph representations, any Python program can be represented as a symbolic graph, in theory. However, the *Speculative Graph Generator* does not convert every single Python feature into a symbolic graph operation (Figure 2 (C)). For example, to ensure the all-or-nothing characteristic of state updates, programs that include invisible state mutations are not converted into symbolic graphs. Some complicated Python features such as *coroutines* and *generators* are also not converted, since they do not have any clear graph representations. Section 4.3 describes the design choices and current limitations of the *Speculative Graph Generator* in terms of Python coverage. In spite of such limitations of the *Speculative Graph Generator*, however, it is worth noting that JANUS users can still freely use the all features of Python on the imperative executor.

4 Symbolic Graph Generation

In this section, we describe in detail how JANUS converts an imperative DL program into a symbolic dataflow graph. We start the section by showing the conversion process of a basic DL program free of dynamic features (Section 4.1). Next, we explain how JANUS converts dynamic features of Python, including dynamic control flow, dynamic types, and impure functions, into symbolic graph operations (Section 4.2). JANUS uses the runtime information to simplify the dynamic program and treat it as a program of only static aspects, which is then easily transformed into a static graph. Finally, we discuss the Python coverage limitations of the *Symbolic Graph Generator* (Section 4.3). More thorough discussion about the Python coverage of JANUS is in Appendix A.

For simplicity, we describe our design using various operations of TensorFlow [1], a widely-used DL framework. However, our design is not necessarily coupled with TensorFlow and can be applied to other DL frameworks.

4.1 Graph Generation Basics

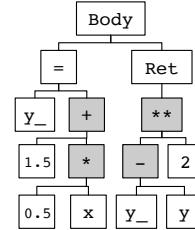
Figure 3(a) is a simple, imperative Python program that calculates a linear model, written as a pure function without any dynamic control flow or arbitrary Python objects. We use this program as an example to show the basic graph conversion process.

```

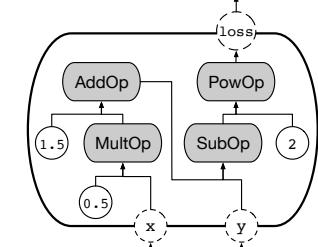
1 def loss_fn(x, y):
2     y_ = 0.5 * x + 1.5
3     return (y_ - y) ** 2

```

(a) Source code of a DL program calculating a linear model



(b) AST of `loss_fn`



(c) Generated graph from `loss_fn`

Figure 3: The Python source code, AST, and symbolic graph of a simple linear model that receives several external inputs. The static features of the program are represented as nodes in the AST, which in turn are converted to vertices of the symbolic graph.

Input parameters (x and y) are converted into graph input objects that require external inputs in order to execute the graph. In the case of TensorFlow, this corresponds to `PlaceholderOp`⁴s. At runtime, they are filled with the actual argument values. The return value of the `return` statement is marked as the computation target of the graph, so that we can retrieve the value after executing the graph.

Python literals such as `0.5`, `1.5` and `2` are simply converted into operations that output constant values – `ConstantOp` for TensorFlow. The conversion of mathematical operators is done by finding the corresponding mathematical graph operations and replacing them one-to-one. For standard Python operators such as `+` and `**`, JANUS places the appropriate primitive calculation operations in the graph, like `AddOp` and `PowOp` for TensorFlow.

An assignment to a Python local variable and a value retrieval from the same variable is converted into a connection between two operations, just as in Pydron [28]. Figures 3(b) and 3(c) illustrate how such a connection is made for the variable `y_` in Figure 3(a), along with the rest of the program.

4.2 Dynamic Features

In addition to the basic features, JANUS converts the dynamic features of Python into the elements of the symbolic DL graph as well to provide the performance of dataflow graphs while maintaining the same programmability of imperative DL frameworks. Moreover, JANUS

⁴PlaceholderOps are unique operations that generate errors unless they are provided with external inputs before graph execution. TensorFlow expects users to feed a dictionary `{ph1: v1, ph2: v2, ...}` to a `PlaceholderOp`.

exploits the fact that the dynamism in Python DL programs can often be simplified to static dataflow, treating a dynamic program as a program of only static aspects with appropriate program context assumptions. Context assumptions are generated based on the profile information JANUS gathers at runtime.

4.2.1 Dynamic Control Flow

Basic translation rules. Among various dynamic control flow statements, JANUS focuses on conditional branches, loop constructs, and function calls, similar to Pydron [28]. As shown in Pydron, these three constructs are enough to express most complex dynamic control flows in Python. Furthermore, they can all be expressed using special control flow graph operations proposed in recent works [19, 50] as follows.

Python’s conditional statement, the `if` statement, can be obtained by combining *switch* and *merge* primitives. The switch and merge primitives, originating from classic dataflow architectures [8, 10, 30], act as demultiplexers and multiplexers, respectively, selecting a single path to pass their inputs or outputs. In TensorFlow, the `SwitchOp` and `MergeOp` [50] serve as symbolic dataflow graph counterparts for these primitives, allowing JANUS to plant conditional branches in graphs.

The iterative statements of Python, `while` and `for`, are handled by using the switch and merge primitives together with loop context primitives that hold iteration *frames*. TensorFlow conveniently provides `EnterOp`, `ExitOp`, and `NextIterationOp` [50] for creating iteration frames and passing values over them.

Finally, for function calls, a separate graph is generated for the callee function, and a function invocation operation that points to the generated graph is inserted in the position of the function calls. Recent work proposes a TensorFlow implementation of this operation called `InvokeOp` [19], which can represent an invocation of a recursive function with automatic differentiation support.

Speculative graph generation: unrolling and inlining. If JANUS detects that only a single particular path is taken for a certain control flow statement during profiling, JANUS presumes that the control flow decision is actually fixed. The system replaces the control flow operation with an assertion operation that double-checks the assumption for this control flow decision, and proceeds with graph generation as if the control flow statement were unrolled. This allows JANUS to remove control flow operation overheads and apply graph optimizations such as common subexpression elimination or constant folding in broader portions of the graph. If the assertion operation fails, JANUS falls back to imperative execution.

To be more specific, for conditional branches, if the program takes only one side of the branch during profiling, JANUS generates that particular side of the branch in the final graph without any switch or merge primitives and adds an assertion operation that can detect a jump to the other side of the branch. For iterative statements, if the number of iterations of a loop is discovered to be fixed, JANUS unrolls the loop with this fixed iteration count, and adds an assertion operation to check that the number of iterations is indeed correct.

For function calls, if the callee is expected to be fixed for a function call at a certain position, JANUS inlines the callee function body inside the caller unless that function call is identified as a recursive one. In addition, for callee functions whose implementation is already known for JANUS, e.g., the functions provided by the framework such as `matmul()` or `conv2d()`, or Python built-in functions like `print()` or `len()`, JANUS adds the corresponding graph operations which behave the same as the original callee functions, based on the prior knowledge about their behaviors. Section 4.3.1 includes more details and limitations about such function calls.

4.2.2 Dynamic Type

Basic translation rules. The types of all expressions within a Python program must be known before JANUS can convert the program into a symbolic graph, because graph operations require operands to have fixed types. This is a challenging task for Python programs because we cannot determine the type of an arbitrary Python expression before actually executing the expression. Fortunately, it is possible to infer the types of some expressions, given the types of other expressions; for example, it is clear that the variable `c` in `c = a + b` is an integer if `a` and `b` are integers.

As a basic rule, JANUS converts numerical Python values such as scalars, list of numbers, and NumPy [48] arrays into corresponding tensors, and converts non-numerical values, including arbitrary class instances, into integer-typed scalar tensors which hold pointers to the corresponding Python values. Next, JANUS infers the types of other expressions that are derived from expressions covered by the basic rule.

Speculative graph generation: specialization. Expressions whose types cannot be inferred from other expressions require a different measure. For instance, it is impossible to identify the types of input parameters for functions, or Python object attribute accesses (`obj.attr`) without any external clues. Similarly, inferring the return types of recursive function calls is also challenging due to the circular dependencies. To make proper assumptions

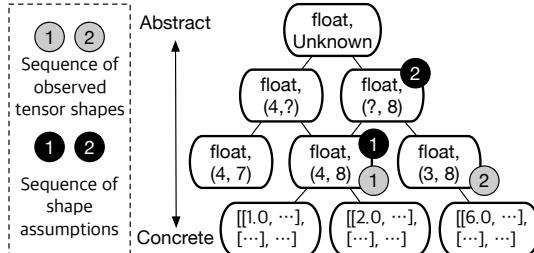


Figure 4: Type, shape, and value specialization hierarchy for an example tensor.

about the types of such expressions, *Profiler* observes the types of the expressions during imperative executions. Given these context assumptions, JANUS can finish inferring the types of remaining expressions, and construct a specialized dataflow graph accordingly.

In addition, JANUS makes further assumptions about the expressions to apply more aggressive optimizations. For numerical expressions, we can try to specialize the shape of tensors before constructing the graph. Furthermore, if a Python expression always evaluates to the same value while profiling, JANUS converts it into a constant node in the dataflow graph. With statically determined shapes or values, the graph can be further optimized, or even be compiled to the efficient machine code [44].

Figure 4 shows an example hierarchy of shapes and values that a certain tensor may have. After profiling the first few runs, JANUS finds out that even though the values of the tensor are different every time, they all have the same shape, for example (4, 8), as in the figure. JANUS exploits this information to generate a dataflow graph with an assumption that the shape of this tensor is (4, 8). When the assumption fails, JANUS tries to relax the assumption. For instance, in case the tensor has a shape (3, 8) for the next iteration to process a different size of mini-batch, JANUS modifies the assumption to suit both shapes (4, 8) and (3, 8), resulting in another dataflow graph with a shape assumption of (?, 8). The system does not have to repeat the graph generation process for a possible future case in which the example tensor has yet another unpredicted shape of (2, 8) or (6, 8).

4.2.3 Impure Functions

Naïve translation rules. It is common for a Python function to access global variables to calculate return values and have side-effects, mutating its enclosing Python context during execution. Likewise, it is common for a Python DL program to read from and write to global states such as global or nonlocal variables and heap objects. JANUS respects this characteristic and handles global state accesses alongside symbolic graph execution.

A trivial solution is to use TensorFlow’s PyFuncOps,

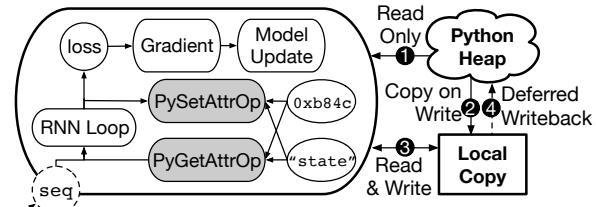


Figure 5: Symbolic dataflow graph generated graph from Figure 1 and the global states.

which can execute arbitrary Python functions as graph operations. A function for reading and updating a certain global state can be created and inserted in the appropriate position within the graph. However, this trivial approach has clear limitations. First, since only one Python function can be executed at a time due to the global interpreter lock (GIL), the overall performance can be reduced when multiple operations should be executed in parallel. It also complicates the fallback mechanism of JANUS. If a global state has already been mutated before the fallback occurs, instead of starting the imperative executor from the function entrance at fallback, execution must start from the middle of the function to be correct, by mapping the state update operation with corresponding Python bytecode.

Optimized graph generation: deferred state update. To make things simpler and also faster, JANUS does not mutate global states in place on the fly. JANUS instead creates local copies of global states, and mutates only the local copies during symbolic graph execution.

Figure 5 shows the symbolic dataflow graph version of the program in Figure 1, which includes the object attribute expressions (`self.state`) that access and mutate the global states. We add new graph operations `PyGetAttrOp` and `PySetAttrOp` to represent Python attribute read and write. Each of them receives an object pointer (`0xb84c`) and a name of the attribute ("state") as inputs, and behaves as follows: ① The `PyGetAttrOp` can access the Python heap to read the state unless a corresponding local copy exists. ② When the `PySetAttrOp` wants to update the attribute, a new value is inserted to the local copy instead of directly updating the Python heap. ③ Further read and write operations are redirected to the local copies. Note that JANUS inserts appropriate dependencies between `PyGetAttrOps` and `PySetAttrOps` if necessary to prevent any data hazards. ④ After the graph executor finishes this run, the local copies are written back to the Python heap. Global or nonlocal variables can also be regarded as the object attributes, where the global variables are the attributes of the global object, and the nonlocal variables are the attributes of the function’s closure objects. Subscript expressions (`obj[subscr]`) are

similarly implemented with equivalent custom operations, `PyGetSubscrOp` and `PySetSubscrOp`.

By not mutating the Python heap directly, JANUS can always bypass the Python GIL to execute more read and write operations in parallel. In addition, the fallback mechanism of JANUS can be simplified thanks to the all-or-nothing based state update mechanism.

4.3 Imperative-Only Features

Albeit being able to support a wide range of imperative DL programs, the current JANUS graph generator does not convert some particular features of Python into dataflow graph elements. Programs with such features are executed only on the imperative executor.

4.3.1 Coverage Limitations from Design

Alignment with the design principles. To be aligned with the design of JANUS in previous sections, the JANUS graph generator does not convert some features of Python. For example, to keep the implementation of local copies of global state simple (Section 4.2.3), Python objects with custom accessor functions (e.g., `__setattr__`) are not supported by the JANUS graph generator. Also, a function should always return the same type of value, to infer the type of call expressions (Section 4.2.2).

External function calls. JANUS must understand the behavior of the external functions, i.e., the framework-provided functions or foreign functions⁵, to convert them into corresponding graph operations. The JANUS graph generator converts the external functions into the graph operations based on a separate whitelist. Most of the framework-provided functions such as `matmul` or `conv2d`, and many commonly-used Python built-in functions such as `print` or `len` are included in this whitelist. We plan to cover more functions in the Python standard library.

JANUS handles such external functions with extra caution to ensure correctness. First, since the underlying assumption here is that the implementation of external functions never changes, JANUS prohibits the modification of the functions included in the whitelist. Also, if an external function includes state mutation (e.g., `assign()` in TensorFlow), the execution of the corresponding graph operation is deferred until all the other assumptions are validated, under the same principle about the deferred state update in Section 4.2.3.

4.3.2 Coverage Limitations from Implementation

Currently, JANUS does not cover a few features from Python that do not have clear graph representations. Such Python features include *coroutines*, *generators*, in-line

⁵functions written in the languages other than Python

class definitions and in-line import statements. We plan to support these features as future work.

5 Implementation

We implemented JANUS on top of TensorFlow [1] 1.8.0 and CPython [32] 3.5.2. JANUS exploits the existing TensorFlow graph executor and TensorFlow Eager imperative executor as its components. In this section, we explain the modifications to existing systems, and then describe how JANUS supports data-parallel training.

Modifications to existing systems. TensorFlow has been modified for several reasons. First, to transparently separate out the neural network computation from the rest of the Python program without extra user intervention, the automatic differentiation functionality of TensorFlow Eager is modified to trigger JANUS graph conversion. Second, to share the model parameters between eager mode and graph mode, JANUS slightly modifies the parameter storing mechanism of TensorFlow Eager. Third, several custom operations had been added, including the `InvokeOp` and `PyAttrOp` as described in earlier sections.

CPython has also been modified to have bytecode-level instrumentation functionality for non-intrusive profiling. Without modifying the interpreter, instrumentation for the profiling should exist at the Python source-code level, which would significantly affect the performance and the debuggability of the imperative execution.

Data-parallelization on JANUS. Using multiple machines equipped with multiple GPUs is a common approach for accelerating DL jobs. We integrate JANUS with Horovod [35], a distributed training module for TensorFlow that encapsulates the MPI collective communication [14] (e.g. AllReduce and AllGather) as an operation inside the symbolic graph. After converting an imperative program into a dataflow graph, JANUS inserts appropriate communication operations to the graph in order to get the average of gradients generated by multiple workers. Since the generated dataflow graph contains both communication and computation operations, we can parallelize their execution and therefore achieve higher throughput.

6 Evaluation

We present experimental results that show how imperative DL programs can be executed both correctly and efficiently when converted into symbolic graphs on JANUS.

6.1 Experimental Setup

Frameworks. As baseline frameworks representing symbolic graph execution frameworks and imperative execution frameworks respectively, we use TensorFlow [1] and TensorFlow Eager [40]. We could run the same DL

program on JANUS as on TensorFlow Eager, thanks to the transparent graph conversion feature of JANUS. In addition, to demonstrate the correctness of graph conversion of JANUS, we also compare JANUS with TensorFlow defun [43], which implements a trace-based graph conversion mechanism. TensorFlow-based frameworks have been chosen to avoid implementation-dependent performance differences.

Applications. We have evaluated JANUS with 11 models in five major neural network types, covering three convolutional neural networks (*CNN*; LeNet [21], ResNet50 [15], Inception-v3 [38]), two recurrent neural networks (*RNN*; LSTM [51], LM [20]), two recursive neural networks (*TreeNN*; TreeRNN [36], TreeLSTM [39]), two deep reinforcement learning models (*DRL*; A3C [25], PPO [34]), and two generative adversarial networks (*GAN*; AN [13], pix2pix [18]) as shown in Table 2. The datasets and the mini-batch sizes used for evaluation are also specified in the table.

These models are implemented in an imperative programming style, using a number of dynamic features in Python as shown in Table 2. First, large CNN models such as ResNet50 and Inception-v3 have conditional statements for handling batch normalization [16], which make them behave differently under particular conditions when training and evaluating the model. Next, RNNs include Python `for` loops, and they also include global state mutation statements to retain hidden states inside the models. Next, TreeNNs⁶ require all three kinds of dynamic features. They include recursive function calls, and conditional statements to separate recursion base cases and inductive cases. They also include values with undecided type; the return type of a recursive function is unknown until the function returns certain values. In addition, they include the Python object access to fetch the information of the current subtree. For DRL models⁷, Python `for` loops are used for handling an arbitrary length of the states of an episode, and global state mutation statements are used for storing the intermediate computation results to monitor the progress of the training. GAN models also use global state mutation statements for the same reason. All models use Python function calls, including Python class methods of high-level DL programming APIs such as Keras [7]. Training data instances fed into each neural network have different shapes over different training iterations, when the length of the dataset cannot be divided by the batch size.

⁶The implementation of TreeNN models on TensorFlow follows the recursion-based implementation with `InvokeOp` [19], and JANUS converts an imperative Python program into similar recursion-based graphs.

⁷The DL framework only handles model training and policy evaluation, and the environment simulation is handled by an external library [3].

Category	Model	DataSet	BS	DCF	DT	IF
CNN	LeNet	MNIST [22]	50	✗	○	✗
	ResNet50	ImageNet [33]	64	○	○	✗
	Inception-v3	ImageNet [33]	64	○	○	✗
RNN	LSTM	PTB [51]	20	○	○	○
	LM	1B [5]	256	○	○	○
TreeNN	TreeRNN	SST [37]	25	○	○	○
	TreeLSTM	SST [37]	25	○	○	○
DRL	A3C	CartPole [3]	20	○	○	○
	PPO	Pong [3]	256	✗	○	○
GAN	AN	MNIST [22]	128	✗	○	○
	pix2pix	Facades [47]	1	✗	○	○

Table 2: Categories, models, datasets, batch sizes ("BS"), and the dynamic features of the applications used for evaluation.

Environments. A homogeneous GPU cluster of 6 machines, connected via Mellanox ConnectX-4 cards with 100Gbps InfiniBand is used for evaluation. Each machine is equipped with two 18-core Intel Xeon E5-2695 @ 2.10 GHz, and 6 NVIDIA TITAN Xp GPU cards. Ubuntu 16.04, Horovod 0.12.1, CUDA 9.0, cuDNN 7, OpenMPI v3.0.0, and NCCL v2.1 are installed for each machine.

LeNet, LSTM, AN, and pix2pix models are evaluated on a single GPU, since these models and the datasets are regarded to be too small to amortize the communication cost of parallel execution. Similarly, TreeRNN, TreeLSTM, and A3C models are evaluated on CPUs on a single machine, since these models and datasets are regarded to be too small to amortize the communication between CPU and GPU. The other models are evaluated using multiple GPUs. ResNet50 and Inception-v3 models are evaluated using up to 36 GPUs, and LM is evaluated on up to 12 GPUs. The network bandwidth made the throughput of LM saturated on more than 2 machines with MPI collective communication, due to the huge parameter size of LM (0.83 billion parameters). Therefore, model convergence of LM is experimented with 6 GPUs. We evaluated the model convergence of PPO using 4 GPUs on a single machine, since the number of parallel actors used in the original paper was only 8.

6.2 Model Convergence

Figure 6 shows how the neural networks converge on various underlying frameworks, with ResNet50 with the ImageNet dataset, LM with the 1B dataset, TreeLSTM with the SST dataset, PPO with the Pong-v4 environment, and AN with the Facades dataset on four frameworks. For all evaluated models, JANUS, TensorFlow, and TensorFlow Eager succeeded to make the neural networks converge correctly as reported in literatures: 23.7% top-1 error for ResNet50 after 90 epochs, perplexity 47.5

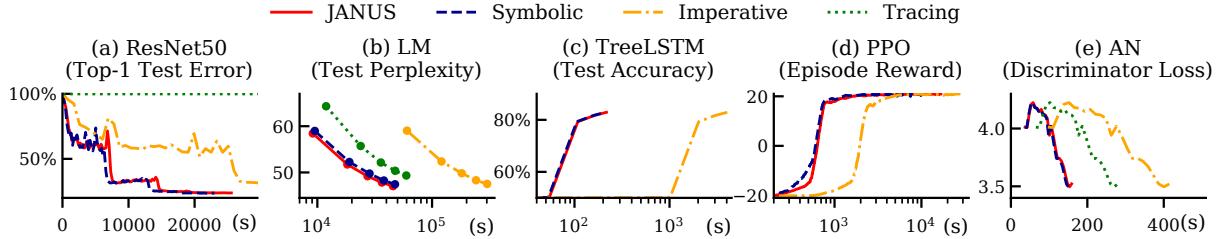


Figure 6: (a) The test error of ResNet50, (b) validation perplexity of LM, (c) test accuracy of TreeLSTM, (d) episode reward of PPO, and (e) discriminator loss of AN measured on JANUS, TensorFlow (Symbolic), TensorFlow Eager (Imperative), and TensorFlow defun (Tracing) according to the elapsed time in seconds. Each marker in (b) represents each training epoch, describing that per-epoch convergence is slower on TensorFlow defun compared to other frameworks.

for LM after 5 epochs, 82.0% binary accuracy for TreeLSTM after 4 epochs, 20.7 mean final score for PPO after 40M game frames, and 3.52 discriminator loss for AN after 30 epochs.⁸ Also, JANUS could make the model to converge up to 18.7 times faster than TensorFlow Eager, while executing the identical imperative program. The performance difference between JANUS and TensorFlow was within 4.0%.

On the other hand, trace-based TensorFlow defun failed to make the models to converge correctly. The ResNet50 model includes the conditional statement to distinguish the behavior of the batch-normalization [16] layer on model training and evaluation. If a user evaluates the initial accuracy before training the model by manipulating the model object attribute, TensorFlow defun converts the first execution trace into graph operations, which silently leads to an inaccurate result. Similarly, the LM model does not converge properly with TensorFlow defun, since it failed to capture state passing across sequences, due to its trace-based conversion mechanism. The TreeLSTM model could not be converted into the symbolic graph at all with TensorFlow defun, since it does not support recursive function call. We could not get the convergence metrics for PPO model with TensorFlow defun, as it does not support global state update statements. TensorFlow Eager converges slowly, since its training throughput is much lower than TensorFlow and JANUS. We next analyze the training throughput of the frameworks, excluding TensorFlow defun, which fails to make models converge correctly.

6.3 Training Throughput

6.3.1 Single-machine Throughput

Table 3 presents the training throughput of all models executed with JANUS, TensorFlow Eager, and TensorFlow on a single machine with a single GPU. As shown in the

⁸We measured the training loss with the official implementation in Tensorflow Eager [41].

Model	(A)	(B)	(C)	$\frac{(B)}{(A)}$	$\frac{(B)}{(C)} - 1$
	Imp.	JANUS	Sym.		
LeNet	7.94k	25.84k	26.82k	3.25x	-3.6%
ResNet50	188.46	200.37	207.39	1.06x	-3.4%
Inception-v3	108.36	119.32	124.33	1.10x	-4.0%
LSTM	2.75k	22.06k	22.58k	8.03x	-2.3%
LM	19.02k	40.18k	40.45k	2.11x	-0.7%
TreeRNN	20.76	988.72	928.66	47.6x	+6.5%
TreeLSTM	7.51	138.12	141.71	18.4x	-2.5%
A3C	220.66	1132.9	1178.6	5.13x	-3.9%
PPO	596.80	1301.0	1306.4	2.18x	-0.4%
AN	4.34k	11.33k	11.56k	2.61x	-2.1%
pix2pix	4.04	8.69	8.88	2.15x	-2.1%

Table 3: Training throughput of all models evaluated on a single machine with a single GPU in JANUS, TensorFlow (Sym.), and TensorFlow Eager (Imp.). The numbers represent processed images/s for CNN and GAN models, processed words/s for RNN models, processed sentences/s for TreeNN models, and processed frames/s for DRL models.

table, JANUS outperforms TensorFlow Eager (imperative execution) by up to 47.6 times, and shows throughput similar to TensorFlow (symbolic graph execution) by up to 4.0% performance degradation. JANUS even performs slightly better (+6.5%) for TreeRNN, since there is no need to pre-process the input sentences, which are the tree-structured Python objects.

JANUS achieves bigger performance gains on RNNs, TreeNNs, DRLs, and GANs than on CNNs, since those networks have many concurrently executable operations. In addition, the performance gain of JANUS on a single machine is larger on models with fine-grained graph operations such as LeNet, LSTM, TreeRNN, A3C, and AN, compared to the models with coarse-grained operations such as ResNet50, Inception-v3, LM, PPO, and pix2pix, since the gain from bypassing the Python interpreter and applying compiler optimizations is bigger when the computation time of each operation is short.

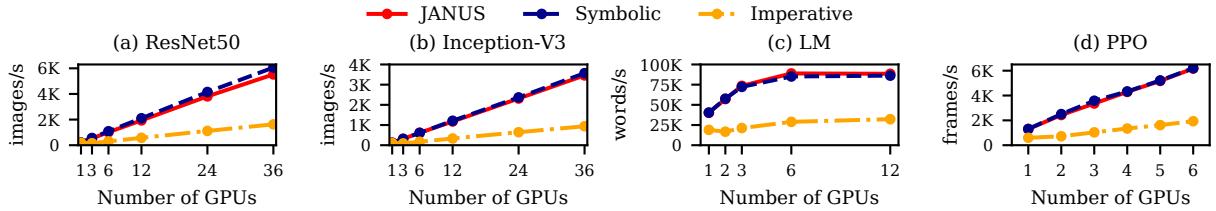


Figure 8: Training throughput for the ResNet50, Inception-v3, LM, and PPO models on JANUS, TensorFlow (Symbolic), TensorFlow Eager (Imperative), using varying numbers of GPUs.

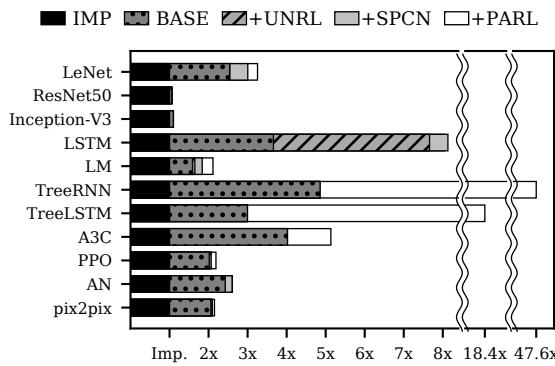


Figure 7: The contribution of optimizations to improve training throughput. Optimizations are cumulative. **+PARL** is the default configuration of JANUS.

For large CNN models such as ResNet50 and Inception-v3, optimized GPU kernel computation accounts for most of the computation time, which makes the performance difference among JANUS, TensorFlow, and TensorFlow Eager relatively small.

Optimization effect. Figure 7 analyzes the cause of the performance improvement of JANUS in detail. Converting the imperative program into the symbolic graph without any following optimizations (**BASE**) enabled up to 4.9x performance improvement compared to the imperative execution (**IMP**). It removes the Python interpreter and framework code overhead, which has the bigger effect when each graph operation is relatively smaller. Control flow unrolling (**+UNRL**) and type specialization (**+SPCN**) enable more aggressive compiler optimizations. On RNNs, **+UNRL** improved the performance of LSTM and LM by 2.09x and 1.04x, respectively. The control flow statements in CNNs, TreeRNNs and DRLs could not be unrolled due to their dynamicity. **+SPCN** enabled some compiler optimizations and improved the throughput up to 18.3% in small neural networks. Finally, executing multiple operations in parallel (**+PARL**) improved the throughput up to 9.81x. Especially higher gain could be achieved for TreeRNNs, since there exist many operations that could be executed in parallel in multiple

independent tree nodes.

We have also measured the effect of assumption validation, but the effect was negligible (in the error range), since the `AssertOps` can be executed with the main neural network in parallel.

6.3.2 Scalability

Figure 8 shows the scalability of ResNet50, Inception-v3, LM, and PPO models on JANUS, TensorFlow, and TensorFlow Eager on the cluster with 36 GPUs (12 GPUs for LM, 6 GPUs for PPO). We measured the scale factor, which is defined as *Multi-GPU Throughput / (Single-GPU Throughput × Number of GPUs)*. JANUS achieves similar scalability (scale factor 0.77, 0.81, 0.18 each) as TensorFlow (0.81, 0.80, 0.18 each), but TensorFlow Eager does not scale well (0.24, 0.24, 0.14 each), due its inability to overlap computation and communication.

The performance difference between JANUS and TensorFlow becomes smaller when the synthetic dataset is used, since the input processing of TensorFlow is highly optimized. The slight difference in the scalability of ResNet50 comes from the under-optimized input pipeline of TensorFlow Eager, which JANUS also uses. Optimizing the input processing pipeline for JANUS will further reduce the performance difference between JANUS and TensorFlow. We leave this optimization as future work.

7 Conclusion

In this paper, we introduced JANUS, a system that achieves the performance of symbolic DL frameworks while maintaining the programmability of imperative DL frameworks. To achieve the performance of symbolic DL frameworks, JANUS converts imperative DL programs into static dataflow graphs by assuming that DL programs inherently have the static nature. To preserve the dynamic semantics of Python, JANUS generates and executes the graph speculatively, verifying the correctness of such assumptions at runtime. Our experiments showed that JANUS can execute various deep neural networks efficiently while retaining programmability of imperative programming.

Acknowledgments

We thank our shepherd Srikanth Kandula and the anonymous reviewers for their insightful comments. This work was supported by Samsung Advanced Institute of Technology, the AWS Machine Learning Research Awards program, and Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

- [1] Martín Abadi et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.
- [2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. *Julia: A Fresh Approach to Numerical Computing*, 2017.
- [3] Greg Brockman et al. Openai gym. *CoRR*, abs/1606.01540, 2016.
- [4] Lars Buitinck et al. API design for machine learning software: experiences from the scikit-learn project. In *LML Workshop at ECML PKDD*, pages 108–122, 2013.
- [5] Ciprian Chelba. *1-billion-word-language-modeling-benchmark*. <https://github.com/ciprian-chelba/1-billion-word-language-modeling-benchmark>.
- [6] Tianqi Chen et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Workshop on Machine Learning Systems in NIPS*, 2015.
- [7] François Chollet et al. *Keras: The Python Deep Learning library*, 2015. <https://keras.io/>.
- [8] David E Culler. Dataflow architectures. *Annual review of computer science*, 1(1):225–253, 1986.
- [9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [10] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975.
- [11] Facebook. Caffe2, 2017. <https://caffe2.ai>.
- [12] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *SysML*, 2018.
- [13] Ian J. Goodfellow et al. Generative adversarial nets. In *NIPS*, 2014.
- [14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.
- [15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
- [16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [17] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.
- [18] P. Isola et al. Image-to-image translation with conditional adversarial networks. In *CVPR*, 2017.
- [19] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byeong-Gon Chun. Improving the expressiveness of deep learning frameworks with recursion. In *EuroSys*, 2018.
- [20] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. 2016.
- [21] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.
- [22] Yann LeCun and Corinna Cortes. *The MNIST Database of handwritten digits*, 2010. <http://yann.lecun.com/exdb/mnist/>.
- [23] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. In *ICLR*, 2017.
- [24] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yue-feng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. Device placement optimization with reinforcement learning. 2017.
- [25] Volodymyr Mnih et al. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.
- [26] MXNet. *Deep Learning Programming Style*, 2018. https://mxnet.incubator.apache.org/architecture/program_model.html.

- [27] MXNet Developers. *Gluon*, 2018. <http://gluon.mxnet.io/>.
- [28] Stefan C. Müller, Gustavo Alonso, and Adam Amara André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *OSDI*, 2014.
- [29] Graham Neubig et al. DyNet: The Dynamic Neural Network Toolkit, 2017. *arxiv preprint arXiv:1701.03980*.
- [30] RishiyurS Nikhil et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1990.
- [31] Adam Paszke et al. Automatic differentiation in pytorch. In *Autodiff Workshop in NIPS*, 2017.
- [32] Python Software Foundation. *Python programming language*. <https://www.python.org/>.
- [33] Olga Russakovsky et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.
- [34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [35] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.
- [36] Richard Socher et al. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.
- [37] Richard Socher et al. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.
- [38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.
- [39] Kai Sheng Tai et al. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, 2015.
- [40] TensorFlow. *Eager Execution*, 2018. https://www.tensorflow.org/programmers_guide/eager.
- [41] TensorFlow. Gan with tensorflow eager execution. 2018.
- [42] TensorFlow. *Swift for TensorFlow*, 2018. <https://github.com/tensorflow/swift>.
- [43] TensorFlow. *tf.contrib.eager.defun*, 2018. https://www.tensorflow.org/versions/master/api_docs/python/tf/contrib/eager/defun.
- [44] TensorFlow. *XLA Overview*, 2018. <https://www.tensorflow.org/performance/xla/>.
- [45] TensorFlow. *AutoGraph: Easy control flow for graphs*, 2019. <https://www.tensorflow.org/guide/autograph>.
- [46] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions, 2016. *arXiv preprint arXiv:1605.02688*.
- [47] Radim Tyleček and Radim Šára. Spatial pattern templates for recognition of objects with regular structure. In *GCPR*, 2013.
- [48] Stéfan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy Array: a Structure for Efficient Numerical Computation, 2011. *Computing in Science & Engineering*, 13, 2, 22-30.
- [49] Dong Yu et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014-112*, 2014.
- [50] Yuan Yu et al. Dynamic control flow in large-scale machine learning. In *EuroSys*, 2018.
- [51] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Appendix

A Python Syntax Coverage

Table 4 describes the entire set of opcode in the CPython [32] 3.5.2 interpreter, and maps them to the sections which describe the corresponding graph generation rules. Python programs whose opcodes are mapped to Section 4.3 can only be executed on the imperative executor, and the others can be executed on the graph executor. Python features that are not covered in previous sections are briefly discussed in the rest of this section.

Exceptions. A Python `raise` statement can be represented as an `AssertOp` in the dataflow graph. When the `AssertOp` for an exception aborts the graph execution,

Opcode	Num	Description	Section Ref.
POP_TOP, ROT_TWO, ROT_THREE, DUP_TOP, DUP_TOP_TWO, NOP, EXTENDED_ARG	7	stack manipulation	No conversion is necessary
LOAD_CONST	1	constant	Section 4.1
UNARY_INVERT, UNARY_NEGATIVE, UNARY_NOT, UNARY_POSITIVE, BINARY_ADD, BINARY_AND, BINARY_FLOOR_DIVIDE, BINARY_LSHIFT, BINARY_MATRIX_MULTIPLY, BINARY_MODULO, BINARY_MULTIPLY, BINARY_OR, BINARY_POWER, BINARY_RSHIFT, BINARY_SUBTRACT, BINARY_TRUE_DIVIDE, BINARY_XOR, INPLACE_ADD, INPLACE_AND, INPLACE_FLOOR_DIVIDE, INPLACE_LSHIFT, INPLACE_MATRIX_MULTIPLY, INPLACE_MODULO, INPLACE_MULTIPLY, INPLACE_OR, INPLACE_POWER, INPLACE_RSHIFT, INPLACE_SUBTRACT, INPLACE_TRUE_DIVIDE, INPLACE_XOR, COMPARE_OP	31	mathematical operators	Section 4.1
LOAD_FAST, STORE_FAST, DELETE_FAST, UNPACK_SEQUENCE, UNPACK_EX	5	local variables	Section 4.1
JUMP_ABSOLUTE, JUMP_FORWARD, JUMP_IF_FALSE_OR_POP, JUMP_IF_TRUE_OR_POP, POP_JUMP_IF_FALSE, POP_JUMP_IF_TRUE, POP_BLOCK, GET_ITER, FOR_ITER, BREAK_LOOP, CONTINUE_LOOP, SETUP_LOOP	12	dynamic control flow	Section 4.2.1
CALL_FUNCTION, CALL_FUNCTION_KW, CALL_FUNCTION_VAR, CALL_FUNCTION_VAR_KW, RETURN_VALUE, MAKE_FUNCTION	6	function call	Section 4.2.1, Section 4.3.1
LOAD_ATTR, STORE_ATTR, DELETE_ATTR	3	arbitrary object	Section 4.2.2, Section 4.2.3
BUILD_LIST, BUILD_LIST_UNPACK, LIST_APPEND, BUILD_MAP, BUILD_MAP_UNPACK, BUILD_MAP_UNPACK_WITH_CALL, MAP_ADD, BUILD_SET, BUILD_SET_UNPACK, SET_ADD, BUILD_SLICE, BUILD_TUPLE, BUILD_TUPLE_UNPACK, BINARY_SUBSCR, STORE_SUBSCR, DELETE_SUBSCR	16	list, set, map	Section 4.2.2, Section 4.2.3
LOAD_GLOBAL, LOAD_DEREF, LOAD_NAME, STORE_GLOBAL, STORE_DEREF, STORE_NAME, DELETE_GLOBAL, DELETE_DEREF, DELETE_NAME, LOAD_CLOSURE, MAKE_CLOSURE	11	non-local variables	Section 4.2.3
POP_EXCEPT, SETUP_EXCEPT, SETUP_FINALLY, RAISE_VARARGS, END_FINALLY	5	exception handling	Appendix A
SETUP_WITH, WITH_CLEANUP_FINISH, WITH_CLEANUP_START	3	with	Appendix A
YIELD_FROM, YIELD_VALUE, GET_YIELD_FROM_ITER	3	yield	Section 4.3.2
IMPORT_FROM, IMPORT_NAME, IMPORT_STAR	3	in-line import	Section 4.3.2
LOAD_BUILD_CLASS, LOAD_CLASSDeref	2	in-line class definition	Section 4.3.2
GET_AITER, GET_ANEXT, GET_AWAITABLE, BEFORE_ASYNC_WITH, SETUP_ASYNC_WITH	5	coroutine	Section 4.3.2
Total	113		

Table 4: The mapping of the full list of CPython opcode and the corresponding sections.

the fallback occurs, and the actual, Python-style exception can be safely raised on the imperative executor. Under the same principle, for `try-except-finally` statements, only the `try-finally` part is converted into the graph elements, and the `except` part is simply not converted, since the exception will never be caught by the symbolic graph. By avoiding exception handling inside the symbolic graph, we can protect users from having

to debug through symbolic graph execution traces, which are relatively more complicated than imperative execution traces.

Context manager. Since exception handling always occurs on the imperative executor as described in the previous paragraph, the `with` statement can be converted into the simple function calls to `__enter__` and `__exit__` of the corresponding context manager object.

BLAS-on-flash : An Efficient Alternative for Large Scale ML Training and Inference?

Suhas Jayaram Subramanya
Microsoft Research India
t-sujs@microsoft.com

Anil Kag
Microsoft Research India
t-anik@microsoft.com

Harsha Vardhan Simhadri
Microsoft Research India
harshashi@microsoft.com

Srajan Garg
IIT Bombay
srajan.garg@gmail.com

Venkatesh Balasubramanian
Microsoft Research India
t-venkb@microsoft.com

Abstract

Many large scale machine learning training and inference tasks are memory-bound rather than compute-bound. That is, on large data sets, the working set of these algorithms does not fit in memory for jobs that could run overnight on a few multi-core processors. This often forces an expensive redesign of the algorithm for distributed platforms such as parameter servers and Spark.

We propose an inexpensive and efficient alternative based on the observation that many ML tasks admit algorithms that can be programmed with linear algebra subroutines. A library that supports BLAS and sparse-BLAS interface on large SSD-resident matrices can enable multi-threaded code to scale to industrial scale datasets on a single workstation.

We demonstrate that not only can such a library provide near in-memory performance for BLAS, but can also be used to write implementations of complex algorithms such as eigensolvers that outperform in-memory (ARPACK) and distributed (Spark) counterparts.

Existing multi-threaded in-memory code can link to our library with minor changes and scale to hundreds of gigabytes of training or inference data at near in-memory processing speeds. We demonstrate this with two industrial scale use cases arising in ranking and relevance pipelines: training large scale topic models and inference for extreme multi-label learning.

This suggests that our approach could be an efficient alternative to expensive distributed *big-data* systems for scaling up structurally complex machine learning tasks.

1 Introduction

Data analysis pipelines in scientific computing as well as ranking and relevance often work on datasets that are hundreds of gigabytes to a few terabytes in size. Many algorithms in these pipelines, such as topic modeling [6], matrix factorizations [33], spectral clustering [32], ex-

treme multi-label learning [47], are memory limited as opposed to being limited by compute. That is, on large datasets, a training algorithm that requires a few hours of compute on a multi-core workstation would run out of DRAM for its working set.

This forces users to move the algorithm to distributed big-data platforms such as Apache Spark [63, 64] or systems based on Parameter Servers [18, 37, 60], which incurs three costs: (1) the cost of rewriting code in a distributed framework, (2) the cost of a cluster of nodes or non-availability in production environments, and (3) inefficiencies of the platform in using the hardware. Training on these platforms can require dozens of nodes for moderate speedups over single threaded code for non-trivial algorithms [22, 39]. This could be due to platform overheads as well as mismatch between the structure of the algorithm and the platform’s programming model [9, 17, 58], resulting in low processor utilization.

Several light-weight frameworks for single node workstations demonstrate that this inefficiency is unnecessary for many classes of algorithms that admit multi-threaded implementations that are orders of magnitude more efficient [16, 34, 52, 53]. It is also widely observed that many machine learning problems admit algorithms that are essentially compositions of linear algebra operations on sparse and dense matrices. High performance implementations of these algorithms typically invoke linear-algebra operations through standard APIs such as BLAS [10] and sparseBLAS [20]. High performance implementations for these standard APIs are provided by hardware vendors [26, 27, 43, 44].

Linear algebra kernels offer plenty of locality, so much so that the bandwidth required to run them on high-end multiprocessors can be provided by a non-volatile memory over PCIe or SATA bus [5, 13, 56]. Non-volatile memory is already widely deployed in cloud and developments in hardware and software eco-system position non-volatile memory as an inexpensive alternative to DRAM [4, 19, 49, 50]. Hardware technology and

interfaces for non-volatile memories have increasingly lower end-to-end latency (few μ s) [25] and higher bandwidth: from 8 GT/s in PCIe3.0 to 16GT/s in PCIe4.0 [45] and 32GT/s in PCIe5.0. Hardware manufactures are also packaging non-volatile memory with processing units, e.g. Radeon PRO SSG [2] to increase available memory.

These observations point to a cost-effective solution for scaling linear algebra based algorithms to large datasets in many scenarios – use inexpensive PCIe-connected SSDs to store large matrices corresponding to the data and the model, and exploit the locality of linear algebra to develop a library of routines that can operate on these matrices with a limited amount of DRAM. By conforming to the standard APIs, such a library could be a replacement for code that would have linked to BLAS libraries such as Intel MKL or OpenBLAS [59].

We present empirical evidence that this approach can be practical, easy, and fast, by developing a library which provides near in-memory speeds on NVM-resident data for subroutines on dense matrices and sparse matrices.

Performance of our *BLAS-on-flash* library is comparable to that of in-memory Intel MKL implementations for level-3 BLAS and sparseBLAS kernels such as `gemm` (dense-dense matrix multiplication) and `csrmm` (sparse-dense matrix multiplication) on multiprocessor machines with SSDs. The key to this performance is using the knowledge of data-access patterns arising in linear algebra kernels to effectively pipeline IO with computation. Using these kernels, we can implement algorithms such as k-means clustering that run at near in-memory speeds.

To illustrate that this approach is not limited to simple kernels, we consider one of the most structurally complex numerical algorithms – eigensolvers. Using the *BLAS-on-flash* library, we built a general purpose symmetric eigensolver, which is critical to dimensionality reduction (e.g. PCA) and spectral methods. Specifically, we adapted the restarted block Krylov-Schur [67] algorithm to compute thousands of eigenvectors on SSD-resident data faster than standard in-memory solvers based on the IRAM algorithm [54] (e.g., Spectra [48], ARPACK [35]). On large bag of words text datasets running into hundreds of gigabytes, our implementation running on one multi-core workstation with under 50GB DRAM outperforms Spark MLlib’s `computeSVD` [40] deployed on hundreds of executors, representing an order of magnitude efficiency gain in hardware utilization. Further, our solver can compute thousands of eigenvalues, while `computeSVD` is limited to 500 or fewer.

We present two use cases of the library for algorithms used in ranking and relevance pipelines that process hundreds of gigabytes of data: training topic models, and inference in Extreme Multi-Label learning.

Topic modeling [11] summarizes a corpus of documents, where each document is a collection of words

from a fixed vocabulary, as a set of *topics* that are probability distributions over the vocabulary. Although most large scale algorithms are based on approximating and scaling an intractable probabilistic model on parameter servers [14, 61, 62], recent research [6] has shown that linear algebra based approaches can be just as good qualitatively. We take a highly optimized version of the algorithm in [6] that already outperforms prior art on single node workstations, and link to the eigensolvers and clustering algorithms written using our framework. This allows the algorithm to train a 2000 topic model on a 60 billion token corpus (500GB on disk) in under 4 hours.

Extreme Multi-Label Learning (XML) is the problem of learning to automatically annotate a data point with the most relevant subset of labels from an extremely large label set (often many millions of labels). This is an important task with many applications in tagging, ranking, and recommendation [8]. Models in extreme multi-label learning tasks are often ensembles of deep trees with small classifier(s) at each node. e.g. PfasterXML [47], Parabel [46]. In production, models that exceed DRAM in size need to score (i.e. infer) several hundreds of millions sparse data points from a space with million+ dimensions every week on a platform that provides machines with moderate sized DRAM. As datasets grow in size, XML algorithms need to scale 10x along multiple axes: model size, number of points scored and dimensionality of the data.

In this work, we start with PfasterXML and Parabel models and a dataset that needed 440 and 900 compute hours respectively on a VM with large RAM. We optimized this code to reduce in-memory run time by a factor of six. When the optimized code is linked to our library, it runs at about 90% of in-memory speed with a much smaller memory footprint.

These results suggest that, for complex numerical algorithms, our approach is capable of running at near in-memory speeds on large datasets while providing significant benefits in hardware utilization as compared to general-purpose big-data systems. Further, we envision our library being useful in the following scenarios: (1) Environments without multi-node support for MPI, Spark etc., (2) Laptops and workstations or VMs in cloud with limited RAM but large non-volatile memories, (3) Batch mode periodic retraining and inference of large scale models in production data analysis pipelines, (4) Extending the capabilities of legacy single-node ML training code.

Roadmap. Sections 2, 3 and 4 provide an overview of the interface, design, and the architecture of the library. Section 5 presents an evaluation of the performance of our library and algorithms written using the library.

Source code for our library has been released at github.com/Microsoft/BLAS-on-flash.

2 BLAS-on-flash : Overview and Interface

The *BLAS-on-flash* library provides an easy way to write external memory parallel algorithms, especially numerical algorithms processing large matrices, that run at near in-memory speed on SSD-resident data. At its core, it pipelines calls to an existing math library (like Intel MKL or OpenBLAS) on in-memory data blocks. Coupled with prefetching and intelligent scheduling, *BLAS-on-flash* allows the programmer to define computation on inputs that are limited only by the size of storage.

Our library is intended for programmers who already write multi-threaded code in C++ using shared memory pointers. *BLAS-on-flash* provides a rich interface utilizing C++ templates and inheritance to allow easy integrations with existing code with minimal modifications.

Typically, programmers writing high-performance native code track data objects with *pointers* and manipulate these objects by passing their pointers to functions or linked libraries that perform operations such as matrix multiplication.

The *BLAS-on-flash* library provides a custom pointer type, `flash_ptr<T>`, to track large SSD-resident objects, and replaces the standard `T*` pointer type. A programmer can either invoke *BLAS-on-flash* library functions operating on `flash_ptr<T>` types or define new functions that operate on `flash_ptr<T>` types by specializing the `Task` class. The `Task` class allows a programmer to define inputs, outputs, and a compute function mapping inputs to outputs. A directed acyclic graph (DAG) of tasks defines a higher-level *kernel* (e.g. block matrix multiplication). In this section, we show how to use each of these functionalities.

2.1 The `flash_ptr<T>` type

The `flash_ptr<T>` is a replacement for standard `T*` pointers that allows programmers to handle large blocks of SSD-resident data. An object of type `flash_ptr<T>` can be created by one of two methods.

Allocation - Using an allocator provided by the library to allocate a large block on the disk. Akin to

```
int *mat=(int *)malloc(len);  
the library allows creation of a scratch space on SSD:  
flash_ptr<int> mat=flash_malloc<int>(len);
```

Mapping - Using a mapper provided by the library, one can create a `flash_ptr<T>` backed by an existing file. For example, `flash_ptr<float> mat_fptr = map_file<float>(matfile, READWRITE);` allows read/write access to the float matrix in `matfile`.

Using `flash_ptr<T>`, programmers can read and write to the backing file through our library calls. For example, one can write N elements to the file mapped to `mat_fptr` from an in-memory `mat_ptr` as follows:

```
flash::write_sync(mat_fptr, mat_ptr, N);
```

The `flash_ptr<T>` type supports pointer arithmetic and can be cast and used as a normal pointer through memory mapping for functionality not supported by the library (albeit with worse performance).

```
float* mmap_mat_ptr = mat_fptr.ptr;
```

2.2 Library Kernels

BLAS-on-flash kernels are functions that operate on `flash_ptr<T>` types, designed to be drop-in replacements for in-memory calls operating on `T*` types. Kernels we have implemented include:

- `gemm`: Takes two input matrices `A`, `B` of type `flash_ptr<float|double>` and outputs $C := \alpha \cdot \text{op}(A) * \text{op}(B) + \beta \cdot C$, where α and β are scalars, and $\text{op}(X)=X$ or X^T . The library allows striding and all layout choices a standard BLAS `gemm` call would offer.
- `csrmm` : Performs same computation as `gemm`, but on a sparse `A` in Compressed Sparse Row (CSR) format and allows for `op(·)` only on `B`. In addition to the version where all matrices are of type `flash_ptr<float>`, we also provide a variant where `B` and `C` are in memory pointers. The CSR format stores three arrays: the non-zeros values ordered first by row and then columns, the column index of each non-zero value, and the offsets into the two previous arrays where each row starts.
- `csrgemv` : Takes a sparse matrix `A` on disk and computes $c := \text{op}(A) * b$, where `b` and `c` are in-memory vectors and $\text{op}(X)=X$ or X^T .
- `srcsc` : Converts a sparse matrix in CSR form into its Compressed Sparse Column (CSC) form with both inputs and outputs as `flash_ptr<T>` types. This is equivalent to transposing the input matrix.

In addition to basic kernels, we also implemented some higher-level algorithms like:

- `kmeans` : Given seed centers and input data points, all as `flash_ptr<float>` types, the kernel runs a specified number of Lloyd's iterations and overwrites the seeds with final cluster centroids.
- `sort` : Parallel sample sort on a `flash_ptr<T>` array using a user-defined comparator.

Using *BLAS-on-flash* kernels, programmers can eliminate memory limitations of their in-memory variants. For example, using `csrmm` and `csrgemv`, one could implement an eigensolver for flash-resident matrices. In a later section, we describe complex algorithms using these and other custom kernels to process large amounts of flash-resident data.

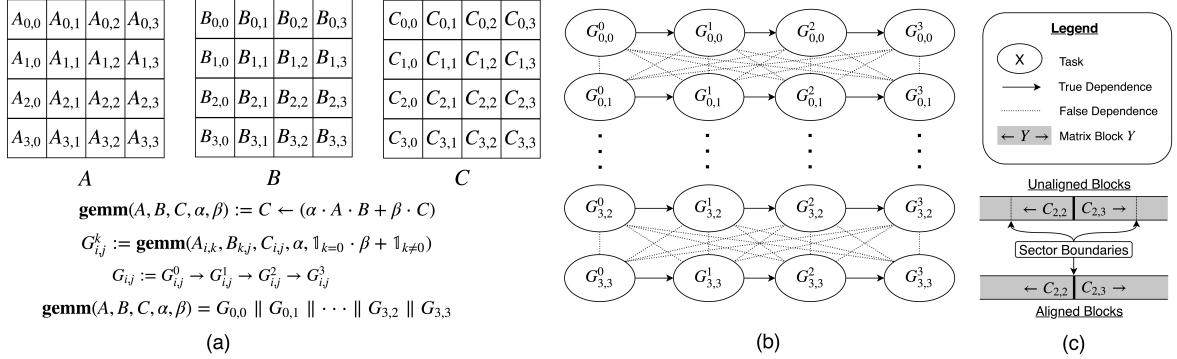


Figure 1: The `gemm` kernel, its DAG using the *Task* interface, and sector-sharing among adjacent output blocks in C.

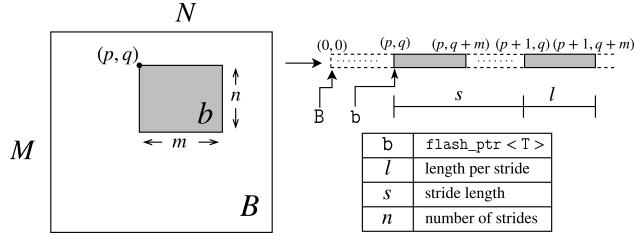


Figure 2: $\langle b, \{l, s, n\} \rangle$ is an access specifier for block b of a flash-resident matrix B stored in Row-Major layout.

2.3 Tasks and Computation Graphs

A *BLAS-on-flash* kernel operating on large inputs is composed of smaller units of computation called tasks. New tasks are defined using the `Task` interface of the library. The `Task` interface allows users to define in-memory computations on smaller portions of the input. It also provides a mechanism to compose a computation graph by allowing parent-child relationships between tasks to encode dependencies.

Task inputs and outputs are uniquely described using an *access specifier*: `<flash_ptr<T>, StrideInfo>`. Here, `flash_ptr<T>` points to the start of the data and `StrideInfo` describes an access pattern starting at `flash_ptr<T>`. An access pattern could be a:

- Strided access to retrieve a matrix block that touches a small *strip* – i.e. a subset – of each row/column of a dense matrix. This is specified using 3 parameters - number of strides, access length per stride (strip size) and the stride length before next access. For the matrix block b in Figure 2, these are n , l , and s respectively.
- Single contiguous access to a chunk of data, equivalent to a strided access with only one strip.

In addition to specifying the inputs and outputs, the user must implement the `execute` function that computes outputs using the inputs. The *BLAS-on-flash* runtime maps a `flash_ptr<T>` to an in-memory `T*` and makes this mapping available in `execute`. With inputs

and outputs available as `T*` types, the programmer must detail operations on inputs using only in-memory function calls to produce outputs.

Figure 1a illustrates a task $G_{i,j}^k$, its inputs ($A_{i,k}, B_{k,j}, C_{i,j}$) and the computation in its `execute` as a block-matrix multiplication on its inputs using an in-memory `gemm` call.

A user can create a new kernel by specifying a directed acyclic graph (DAG) with a task at each node and directed edges from parent tasks to their child tasks. Once a task's parents are specified, the user injects it through the *BLAS-on-flash* Scheduler interface. By allowing tasks to be injected into the scheduler at runtime, the user can specify data-dependent computation graphs required for certain algorithms like eigensolvers.

Figures 1a and 1b illustrate the `gemm` kernel and the DAG associated with its implementation using the Block Matrix Multiplication algorithm. For inputs A , B , and C , shown with 16 blocks for each matrix, an output block $C_{i,j}$ is given by $C_{i,j} := \beta \cdot C_{i,j} + \alpha \cdot \sum_{k=0}^{k=3} A_{i,k} \cdot B_{k,j}$. The inner summation is converted into an *accumulate* chain by using a task $G_{i,j}^k$ in Figure 1a, for each k . $G_{i,j}$ depicts the dependence between successive tasks in the accumulate chain using arrows from a parent task to its child task. Figure 1a illustrates the composition of the `gemm` kernel using accumulate chains and Figure 1b gives the complete DAG for A , B , and C as the inputs and C as the output. The parallel composition operator $X||Y$ allows both X and Y to execute in parallel while the serial composition operator $X \rightarrow Y$ allows Y to execute only after X .

The task injection and logic required for creating a DAG corresponding to a kernel are then packaged into a single function call. This method of packaging allows programmers to replace in-memory calls with *BLAS-on-flash* variants with minimal modifications to existing pipelines. We demonstrate this by replacing memory-intensive kernels in the ISLE topic modeling algorithm, with a *BLAS-on-flash* variant, one kernel at a time.

3 Library Design

BLAS-on-flash supports online scheduling of tasks from a user-defined dynamic graph using a limited DRAM budget with the aim of executing it at near in-memory performance. This requires addressing two resource management problems: (1) effective utilization of the limited DRAM budget by avoiding redundant copies of data shared between tasks, and (2) realizing effective pipelining of computation and IO by better utilization of the limited disk bandwidth offered by PCIe-based SSDs. The library addresses these problems by improving *buffer reuse*, and determining a task schedule likely to minimize disk reads and writes.

We use the `gemm` kernel operating on single precision floating point matrices as an example. The following calculation illustrates the gap between the running times of an in-memory and an SSD-based version on a machine, `test`, with 32 cores capable of 1TFLOPs, and an NVMe SSD with sustained read and write bandwidths of 3GB/s and 0.5GB/s, respectively. Assume that the input and output matrices are of size 32768×32768 each, blocked as in Figure 1. Assume that the matrix block size is 8192×8192 . Each task in the `gemm` kernel requires 1TFLOP of compute on 768MB of input to produce 256MB output. On the `test` system, each such task requires 0.75s of IO time for 1s of compute, when using all 32 threads for one task. Since every task has the same IO and compute requirements, a `gemm` kernel with 64 tasks would take 112s to execute out of memory without pipelining, instead of 64s if executed completely in memory. It is to be noted that, in reality, mixing reads and writes results in reduced read throughput [3]. We do not address this issue here. We instead focus on solving the two problems stated above within the constraints set by the hardware and OS. We are specifically interested in buffer management policies that optimize performance for DAGs arising from linear algebra kernels and algorithms involving matrix operations.

3.1 Buffer Reuse

A task scheduler executing the DAG in Figure 1b might execute tasks $G_{0,0}^1$ and $G_{1,0}^0$ concurrently. If the scheduler is naive, it might prefetch block $B_{0,0}$ twice, thus replicating it in memory. In addition to wasting limited DRAM, this would waste the limited disk bandwidth. Redundant reads can be eliminated, where possible, by enforcing uniqueness of data in memory. The *BLAS-on-flash* runtime ensures such uniqueness by using *reference counters* for in-memory buffers (described Section 4), allowing data reuse, where possible.

3.2 Prioritized Scheduling

Although Buffer Reuse reduces disk reads, the programmer still needs to carefully manage the order of task in-

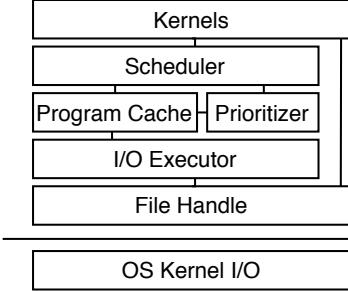


Figure 3: The *BLAS-on-flash* software stack.

jection to maximize data reuse between tasks active in memory. To avoid the programmer this burden, and allow the scheduler to take over this task, we propose a heuristic to select a task for prefetching, based on data currently buffered into memory and the IO requirements of the tasks in the ready list. Our heuristic selects the task that requires the minimum number of bytes to be prefetched given the current contents of the memory buffer. For kernels like `gemm` and `csrmm`, this heuristic minimizes the number of input matrix blocks read by scheduling tasks with high input and output locality. If all matrix blocks are uniform in size, this also reduces the number of write-back operations.

Suppose that, at some point, in an execution of the `gemm` DAG in Figure 1, $M = \{A_{0,0}, A_{1,0}, A_{1,1}, B_{0,0}, B_{1,0}, B_{1,1}, C_{0,0}, C_{1,0}, C_{1,1}\}$ is the set of blocks in memory, and the following tasks are executing concurrently.

$$\begin{aligned} G_{0,0}^1 &:= \text{gemm}(A_{0,1}, B_{1,0}, C_{0,0}, \alpha, 1) \\ G_{1,0}^0 &:= \text{gemm}(A_{1,0}, B_{0,0}, C_{1,0}, \alpha, \beta) \\ G_{1,1}^1 &:= \text{gemm}(A_{1,1}, B_{1,1}, C_{1,1}, \alpha, 1) \\ G_{1,0}^1 &:= \text{gemm}(A_{1,1}, B_{1,0}, C_{1,0}, \alpha, 1) \end{aligned}$$

If $G_{0,0}^1$, $G_{1,0}^0$, and $G_{1,1}^1$ are the latest 3 tasks to complete execution, $G_{1,0}^0$'s child task, $G_{1,0}^1$, is now ready for execution. By scheduling $G_{1,0}^1$ instead of the next-in-queue task, $G_{1,0}^1$ can immediately start execution without requiring any IO. Since outputs from the accumulate chains $G_{0,0}$, $G_{0,1}$, $G_{1,0}$, and $G_{1,1}$ exhibit high locality, our heuristic schedules tasks from such *nearby* accumulate chains to reduce disk operations.

4 Architecture

The *BLAS-on-flash* library implementation consists of the software stack in Figure 3. We describe the role of each of the 5 layers:

File Handle provides a read-write interface using access specifiers for all library calls. Implementations can be specialized for hardware interfaces (e.g. NVMe, SATA, or network) as required. We implement this interface for SSDs using the Linux kernel asynchronous

IO syscall interface – `io_submit` to submit IO jobs and `io_getevents` to reap job completions. Compared to user-space NVMe drivers like SPDK [24] and unvme [41], the `io_submit` syscall interface provides a simpler interface for sector-level unbuffered asynchronous IO with minimal performance penalties.

IO Executor maintains a thread-pool to service IO requests generated by *Program Cache*. To exploit parallelism and ensure correctness, *IO Executor* executes only *non-overlapping* requests in parallel. A pair of requests are *overlapping* if they modify at least one common sector on the disk. For example, consider Figure 1c. When the leading dimension of a matrix block is aligned to the device sector size, $C_{2,2}$ and $C_{2,3}$ can be operated on concurrently. Otherwise, writes to common sectors must be ordered to avoid data corruption. To detect overlaps between requests, each write request is advertised to other threads. A request is added to a thread-local backlog queue if it overlaps with an advertised request. Each thread in the thread-pool services its backlog queue with a higher priority in its next cycle.

Program Cache is the memory subsystem for *BLAS-on-flash*. It manages allocation, deallocation, prefetch, and eviction of in-memory buffers. *Program Cache* allows for buffer re-use by mapping *access specifiers* to reference-counted in-memory buffers. Each map entry is in one of four states - Active(A), Prefetch(P), Write-Back(W), or Zero-Reference(Z). An entry in state A indicates an active reference, i.e., at least one task has a reference to the buffer. An entry in P is a prefetch in progress, W is a write-back in progress, and Z is an entry with zero active references. Entries are one of 3 types - R-only, W-only and RW, corresponding to read-only, write-only and read-write entries. It uses this information to serve four types of requests.

- COMMIT - *commits* a task to memory by ensuring all inputs and outputs are mapped to in-memory buffers. If some inputs/outputs are not already mapped, it evicts some in-memory buffers to free up memory, allocates memory, and queues up prefetches to *IO Executor*. It also increases reference counts for mapped buffers. State is unchanged if the request fails because no entries were eligible for eviction.
- RELEASE - *returns* a task’s inputs and outputs; also decreases reference counts for returned buffers.
- UPDATE - Checks and updates status of pending IO operations.
- Batch HIT/MISS - Typical HIT/MISS queries on a cache to aid prioritization during scheduling.

Program Cache entries transition states according to Figure 4. R-Only, W-only and RW are transitions corresponding to read-only, write-only, and read-write entries, respectively. If a COMMIT request is successful

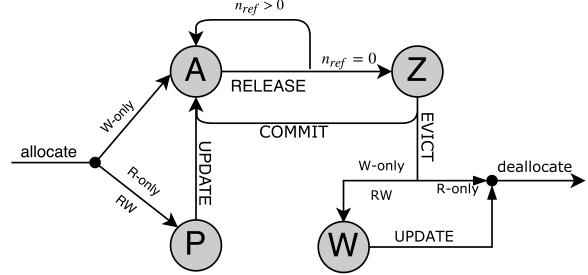


Figure 4: State transition diagram for *Program Cache* entries.

and a new entry is created, memory is allocated using `malloc`. If the entry requires data on disk to be read (R-only, RW), a prefetch is queued. Since entries in Z already contain prefetched data, COMMIT requests transition them directly to A , avoiding a redundant read. Entries enter state A with exactly one active reference. Additional COMMIT requests for entries in A only increase reference counts, and RELEASE requests decrease the same. Entries with zero active references in A transition to Z , making them available for eviction. Evicting a *dirty* entry (RW, W-only) queues a write-back and transitions the entry from Z to W . Entries in P transition into A , and those in W get de-allocated once their IO operations are complete

Prioritizer uses Batch HIT/MISS queries on *Program Cache* to rank the list of ready tasks in increasing order of their prefetch sizes given the current cache state.

Scheduler provides an interface to inject tasks at runtime. Once injected, tasks are executed using a 5-stage pipeline — Wait, Ready, Prefetch, Compute, and Complete. All tasks start out in **Wait** stage, and advance to **Ready** stage when all its parents have finished Compute stage. In each scheduling round, *Scheduler* tries a COMMIT request to *Program Cache* with the highest priority task obtained from *Prioritizer*. If successful, this task advances to **Prefetch** stage. When all its inputs and outputs are mapped to in-memory buffers, the task moves to **Compute** stage. Tasks in Compute stage are executed using a thread-pool maintained by *Scheduler*. Tasks finishing Compute stage are recorded as **Complete**, and *Scheduler* issues a RELEASE request to *Program Cache* with these completed tasks. Once all tasks in a kernel are complete, *Scheduler* allows the programmer to flush any outputs in *Program Cache* to persist results to disk.

5 Algorithms and Evaluation

We now discuss the implementation of the kernels provided by the library and complex algorithms built using these kernels, and compare the running times and memory requirements of in-memory and SSD-based versions. We implemented an eigensolver, an SVD-based

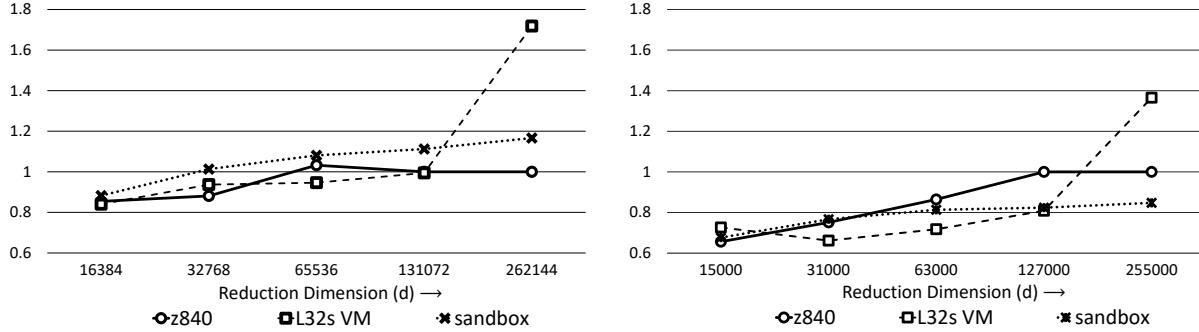


Figure 5: Ratio of in-memory MKL `gemm` to *BLAS-on-flash* `gemm` running times for 512-aligned (left) and unaligned (right) instances for various values of reduction dimension (d). The matrix dimensions are $2^{15} \times d \times 2^{15}$ and $31000 \times d \times 31000$ for the aligned and unaligned plots. *BLAS-on-flash* library has a 8GB Program Cache. `gemm` tasks in *BLAS-on-flash* library use 4 threads each. Program Cache budget determines the number of simultaneous tasks.

algorithm for topic modeling, and two inference algorithms for XML models. This choice of algorithms represents the state-of-the-art for a subset of non-deep learning problems used in ranking and relevance pipelines. Where available, we compare our implementations of these algorithms with prior implementations.

5.1 Experimental setup

The library allows the user to control the number of threads per task (T) and the maximum number of tasks that can be simultaneously executed (K). On a machine with N cores, one would typically choose $T \times K = N$. Within this constraint, the optimal values of T and K are determined by the compute-communication ratio of the task and the parallelism within the task. For the pipeline to execute K tasks in parallel in steady state, the *Scheduler* needs to hold $3K$ tasks in memory to account for K tasks each in Prefetch, Compute and Complete stages of the pipeline. Therefore, in the case of `gemm` and `csrmm` kernels, setting $T=1$ and $K=N$ increases pressure on disk and *Program Cache*. On the other hand, when $K=1$ with $T=N$, MKL does not realize T -fold parallelism with small block sizes. We find $T=4$, $K=N/4$ to be a good tradeoff, empirically.

Table 1 lists the configurations of machines used to evaluate our library. `sandbox` is a high-end bare-metal server with enterprise class Samsung PM1725a SSD capable of sustained read speeds of up to 4GB/s and write speeds of up to 1GB/s. `z840` is chosen to represent a typical bare-metal workstation machine configured with two Samsung 960EVO SSDs in RAID0 configuration, providing sustained read speed of about 3GB/s and write speed of about 2.2GB/s. `L32s VM` is a virtual machine on Azure configured for heavy IO with I/O throttled to a sustained 1.6GB/s or 160K IO ops/second. `M64-32ms VM` is a virtual machine on Azure with 1.7TB RAM that we'll use for running experiments with large mem-

Name	Processor	Cores	RAM	SSD
<code>sandbox</code>	Gold 6140	36	512GB	3.2TB
<code>z840</code>	E5-2620v4	16	32GB	2TB
<code>L32s VM</code>	E5-2698Bv3	32	256GB	6TB
<code>M64-32ms VM</code>	E7-8890v3	32	1.7TB	–
<code>DS14v2 VM</code>	E5-2673v3	16	112GB	–

Table 1: Intel Xeon-based machines used in experiments.

ory requirements. We use Intel MKL 2018 and Ubuntu 16.04LTS on all the machines listed above. Apache Spark instances run Apache Spark MLlib 2.1 on a cluster of Azure DS14v2 VM instances.

5.2 Matrix kernels

General Matrix Multiply (`gemm`) and Sparse (CSR) Matrix Multiply (`csrmm`) are perhaps the most used kernels in math libraries. Therefore, it is important to optimize their performance with careful selection of tiling patterns and prefetch and execution orders in order to minimize IO. For this, we build on well-established results on exploiting locality in matrix multiplications [5, 28, 30]. We also use the fact that BLAS and sparseBLAS computations can be tiled so that they write the output to disk just once [12, 13], thus saving on write bandwidth.

gemm. The block matrix multiplication algorithm in Figure 1 requires $O(n^3)$ floating point operations for $n \times n$ matrices. With block size b , it reads $O(n^3/b)$ bytes from disk and writes $O(n^2)$ bytes back. It is ideal for the library to increase the block size b as much as its in-memory buffer allows so as to decrease the amount of IO required. Figure 5 presents the ratio of running times of the in-memory MKL `gemm` call to that of our library for various *reduction dimension* sizes in two cases:

- **512-aligned.** A matrix is *512-aligned* if the size of its leading dimension is a multiple of 512. e.g., a 1000x1024 float matrix in row-major layout, that would require 4096 bytes for each row.

- **unaligned.** A matrix is *unaligned* if it is **not** 512-aligned, e.g., a 500×500 float matrix in row-major form, that would require 2000 bytes per row.

The distinction between 512-aligned and unaligned matrices is important as the two cases generate a different number of disk access when a block of the matrix is to be fetched or written to. Flushing an unaligned matrix block to disk requires two reads and one write per row – read the start and end sectors of each row in the block, and write-back the overwritten values. A 512-aligned block requires only one write per row.

We define the *reduction dimension* (RD) to be the dimension along which summation happens during matrix multiplication. Using notation from Figure 1, if A , B , and C are all stored in row-major form, the RD is the number of columns in A . Given a block size, increasing the RD increases the length of the accumulate chain, resulting in fewer disk writes per chain. Pipelining efficiency increases with longer accumulate chains, due to lower write-back operations per chain, as demonstrated by Figure 5. In fact, due to careful pipelining, our library outperforms in-memory MKL calls in many instances.

We also evaluated the performance of `gemm` when DRAM overflow is serviced by OS paging mechanisms. We timed a problem of dimension $49K \times 49K \times 49K$ (30GB size) on the `z840` machine with 32GB and 16GB of RAM. For runs with 16GB RAM, we pin a 128GB swap partition to the SSD. The OS-paged version with 16GB RAM ran $1.6\times$ slower than the in-memory version with 32GB RAM. On a larger problem size ($64K \times 64K \times 64K$, 48GB size) and 16GB RAM, OS paging results in a more substantial slowdown – about 13x slower than what in-memory version would have taken.

csrmm. The `csrmm` kernel performs $O(n^3s)$ floating point operations on $n \times n$ size input matrices with sparsity s , representing inputs of size $O(n^2(1+s))$ and output of n^2 size. For a matrix whose sparsity is uniform across rows and columns, with a block size of b , the compute to IO ratio is only $O(bs)$ as opposed to $O(b)$ for `gemm`. For sparse matrices such as those in Table 2 arising from text data (e.g. bag-of-words representation), sparsity can be as low as $s = 10^{-4}$. Therefore, although the execution of in-memory `csrmm` tasks is slower (sparse operations are $10 - 100\times$ slower than dense operations), the low locality (bs as opposed to b) makes it hard to always obtain near in-memory performance. Figure 6 demonstrates the effect of sparsity on `csrmm` by fixing the problem dimensions at $2^{20} \times 2^{17} \times 2^{12}$ and measuring the ratio of in-memory to *BLAS-on-flash* running times for $s \in \{10^{-4}, 10^{-3}, 10^{-2}\}$. It is evident that the efficiency of the `csrmm` kernel decreases with sparsity.

We also benchmark the `csrmm` call required to project the sparse bag-of-words datasets listed in Table 2 into

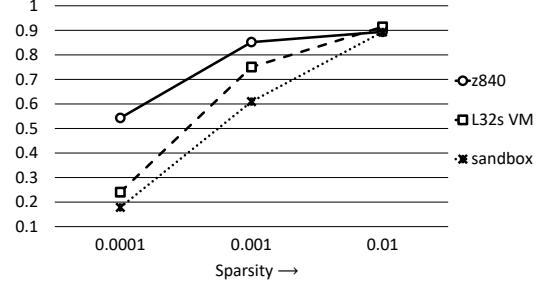


Figure 6: Ratio of in-memory MKL `csrmm` to *BLAS-on-flash* `csrmm` running times for $(2^{20} \times 2^{17} \times 2^{12})$ sized instances and various values of sparsity. *BLAS-on-flash* uses a 8GB Program Cache. Each `csrmm` task uses 4 threads and the number of simultaneous tasks is determined by number of cores in the system.

Dataset	#Cols	#Rows	NNZs	Tokens	Size
Small	8.15M	140K	428M	650M	10.3GB
Medium	22M	1.56M	6.3B	15.6B	151GB
Large	81.7M	2.27M	22.2B	65B	533GB

Table 2: Sparse matrices bag-of-words text data sets. Columns and rows of the matrix represent the documents and words in the vocabulary of a text corpora. The (i, j) -th entry of the matrix represents the number of times the j -th word in the vocabulary occurs in the i -th document.

a 1024-dimensional space (say, obtained from Principal Component Analysis). The dense input and output matrices are 512-aligned and in row-major format. A performance drop is expected in the unaligned case.

Table 3 compares the performance of the `csrmm` in *BLAS-on-flash* to the in-memory version provided by MKL on `z840`, `L32s VM` and `sandbox` machines. `z840` is too small to run the in-memory version for all three data sets because it has only 32GB RAM. Since projecting the Large dataset into 1024 dimensions requires 559GB of RAM, both `L32s VM` and `sandbox` are unable to do it in memory. As an approximation to the speed of an in-memory call on `L32s VM` we ran it on `M64-32ms VM` which has 1.7TB RAM.

Despite a sparsity of 2×10^{-4} , the `csrmm` in *BLAS-on-flash* is about 50% as fast as its in-memory counterpart on the Medium dataset (when the dense matrices are in row-major layout). We picked row-major order for dense matrices because our library was able to outperform MKL’s `csrmm` implementation for column-major order by a factor of $> 2 \times$ on Small and Medium datasets. We attribute this to poor multi-threading in MKL’s implementation.

5.3 Eigensolver

Eigen-decomposition is widely used in data analytics, e.g., dimensionality reduction. Given a symmetric matrix \mathbf{A} , a symmetric *eigensolver* attempts to find k

Dataset	z840 flash	L32s in-mem	VM flash	in-mem	flash
Small	34.7	8.2	24.5	6.9	35.2
Medium	135.75	58.5	101.3	49.5	98.0
Large	636.2	512.3*	390.6	—	354.9

Table 3: Running times in seconds for `csrmm` operations that project datasets in Table 2 into 1024-dimensions. *BLAS-on-flash* has a 16GB Program Cache. *This is run on M64-32ms VM as an approximation to L32s VM.

eigenvalue-eigenvector pairs $(\lambda_i, \mathbf{v}_i)$ such that

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \forall i$$

$$\mathbf{v}_i^T \mathbf{v}_j = 0, \|\mathbf{v}_i\|_2 = 1 \quad \forall i \neq j, |\lambda_1| \geq |\lambda_2| \dots \geq |\lambda_k|$$

Popular dimensionality reduction techniques like Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) use the symmetric eigenvalue decomposition (`syevd`) to compute the projection matrices required for dimensionality reduction. The SVD of a matrix \mathbf{M} can be formulated as a symmetric eigen-decomposition problem as follows:

$$\mathbf{M}\mathbf{u}_i = \sigma_i \mathbf{v}_i, \|\mathbf{v}_i\|_2 = \|\mathbf{u}_i\|_2 = 1 \forall i$$

$$\mathbf{u}_i^T \mathbf{u}_j = 0, \mathbf{v}_i^T \mathbf{v}_j = 0 \forall i \neq j, |\sigma_1| \geq |\sigma_2| \geq \dots \geq |\sigma_k|$$

$$\mathbf{M}\mathbf{M}^T \mathbf{u}_i = \sigma_i^2 \mathbf{u}_i, \mathbf{M}^T \mathbf{M} \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i$$

$$\text{svd}(\mathbf{M}) = \text{syevd}(\mathbf{M}\mathbf{M}^T) = \text{syevd}(\mathbf{M}^T \mathbf{M})$$

To showcase the versatility of our library, we implement a symmetric eigensolver and time it on large sparse matrices (in CSR format) obtained from text corpora in Table 2. Among the many flavors of eigensolvers, we picked the Krylov-subspace class of algorithms as they have been shown to be stable for a wide variety of matrices. These algorithms use iterated Sparse Matrix-Vector (`csrgemv`) products to converge on eigen-pairs.

Since `csrgemv` is bandwidth-bound, it is not suitable for an eigensolver operating on SSD-resident matrices. To overcome this limitation, we implement the Restarted Block Krylov-Schur (Block KS) algorithm [67]. The Block KS algorithm can potentially use fewer matrix accesses to achieve the same tolerance by using a `csrmm` kernel in place of `csrgemv`. Although the Block KS algorithm performs extra computation compared to its non-block variants, this extra work is highly parallel and the IO savings offset the extra compute.

Analysis of eigenvalues of our sparse matrices reveals a large gap between successive eigenvalues. Since time to convergence is inversely correlated with this gap, the Block KS algorithm converges quickly, to the desired tolerance, on our test datasets.

Evaluation. We benchmark both our in-memory and SSD-based single node implementations of the Block KS algorithm against single node and distributed implementations of the Implicitly Restarted Arnoldi Method

(IRAM) algorithm. The single node version is provided by *Spectra* [48], a C++ header-only implementation of *ARPACK* [35], while the distributed version (`computeSVD`) is provided by Apache Spark MLlib library v2.1. The Spark job was deployed on both a shared and a dedicated Hadoop cluster through YARN [55] to workers with 1 core and 8GB memory each and a driver node with 96GB memory. The shared cluster runs Xeon E5-2450L processors with 10Gb Ethernet, while the dedicated cluster uses DS14v2 VM nodes. Other distributed SVD solvers, such as those provided by *ScalAPACK* and Spark KeystoneML, do not adequately support sparse matrices, and are omitted from this comparison.

Table 4 compares the time taken to solve for the top singular values of sparse matrices in Table 2 to a tolerance of 10^{-4} (this is sufficient for the SVD-based topic modeling algorithm described in Section 5.4). It must be noted that `computeSVD` uses double precision floating point numbers while our algorithm uses single precision. We solve for 200 singular values on the large data set and 500 on the Medium data set because the Spark solver was unable to solve for more. Our implementation, on the other hand, easily scales to thousands of singular values on a single node.

The flash version of Block KS runs almost as fast as the in-memory version on datasets with sparsity up to 10^{-3} ; the gap widens as sparsity decreases below 10^{-4} . Further, both Block KS implementations outperform Spectra and Spark jobs in time to convergence. Spark does not see any benefit from adding more workers beyond a point; in fact it becomes slower. These results demonstrate that our flash-based eigensolver utilizes hardware order(s) of magnitudes more efficiently than distributed methods.

5.4 SVD-based Topic Modeling

Topic modeling involves the recovery of underlying *topics* from a text corpus where each document is represented by the frequency of words that occur in it. Mathematically, the problem posits the existence of a topic matrix M whose columns $M_{:,l}$ are probability distributions over the vocabulary of the corpus. The observed data is assumed to be generated by (1) picking a matrix W , whose columns sum to one and represent linear combinations of topic columns in M , (2) calculating $P = MW$, where the j -th column $P_{:,j}$ represents the probability of words in the document j , and (3) sampling the observed documents $A_{:,j}$ using a multinomial distribution based on the p.d.f. $P_{:,j}$. The computational problem is to recover the underlying topic matrix M , given the observations A .

ISLE, or Importance Sampling for Learning Edge topics, is a direct adaption of the TSVD algorithm [6] for recovering topic models [42]. Unlike the LDA class

Dataset (#eigenvalues)	Block Krylov-Schur				Spectra	computeSVD (shared)				computeSVD (dedicated)			
	L32s VM		sandbox			Number of Spark Executors				Number of Spark Executors			
	in-mem	flash	in-mem	flash		64	128	256	512	64	128	256	512
Medium(500)	76	182	63	95	934	320	275	365	450	460	225	228	226
Large (200)	154*	429	–	153	–	–	–	169	230	236	126	104	164

Table 4: Time, in minutes, to compute eigenvalues. For both Medium and Large datasets, Block KS is run with block=25. For Medium, nev=500 and ncv=2500 and for Large, nev=200 and ncv=1500. We run Block KS in-memory on M64–32ms VM as an approximation to L32s VM . Spark MLlib’s computeSVD was timed with 64, 128, 256, 512 workers with 8GB memory on both a shared and a dedicated cluster. The Large dataset needs at least 256 workers to run on the shared cluster. On stand alone cluster with 64 works, the Large dataset needed 10GB memory per worker.

of algorithms (based on MCMC techniques), ISLE uses linear-algebraic techniques to provably recover the underlying topic matrix under reasonable assumptions on the observed data. Empirically, it has been shown to yield qualitatively better topics on real world data. The open source implementation [42] is faster than other single node implementations of any topic modeling algorithms. It takes as input bag-of-words representation for documents in CSR or CSC format, and does the following steps: (1) threshold to *denoise* the data, (2) use SVD to compute a lower dimensional space to project the documents into, (3) cluster documents using k-means++ initialization and the k-means algorithm in the projected space, (4) use the resultant clusters to seed clustering in the original space using the k-means algorithm, and finally (5) construct the topic model. For large datasets, sampling techniques can be used to pick a subset of data for the expensive steps (2), (3), and (4). We adapt ISLE to use the *BLAS-on-flash* framework by leveraging our flash-based Block KS eigensolver and the clustering algorithms built using our framework.

Evaluation. Table 5 compares the running times of the in-memory version and a flash-based version using the *BLAS-on-flash* library. Using this redesigned pipeline, we were able to train a 5000-topic model with a DRAM requirement of 1.5TB on both L32s VM and sandbox machines with only 32GB allocated to *Program Cache*. We note that the number of tokens in this dataset (about 65 billion) is in the same ballpark as the number of tokens processed by LDA-based topic modeling algorithms in Parameter Server based systems that use multiple nodes [37].

On the Medium dataset, where it is possible to run an in-memory version, notice that the code linked to *BLAS-on-flash* achieved about 65 – 80% in-memory performance using less than 128GB RAM. On the Large dataset, the flash version run on *sandbox* is faster than the in-memory version on M64–32ms VM . We attribute this to newer hardware on *sandbox* , and near in-memory performance of eigensolver and kmeans kernels written with *BLAS-on-flash* .

Dataset (# Topics)	Sample Rate	sandbox		L32s VM	
		in-mem	flash	in-mem	flash
Small(1K)	1.0	15	27	18	37
Medium(1K)	0.1	46	66	63	72
Medium(2K)	0.1	119	144	158	212
Large(1K)	0.1	–	149	163*	172
Large(2K)	0.1	–	228	285*	279
Large(5K)	0.1	–	522	980*	664
Large(2K)	0.4	–	532	684*	869

Table 5: Running time of the ISLE algorithm in minutes.

*We use M64–32ms VM as an approximation to L32s VM for the Large dataset.

5.5 Extreme Multi-Label Learning

Extreme multi-label learning (XML) addresses the problem of automatically annotating a data point with the most relevant subset of labels from an extremely large label set. It has many applications in tagging, ranking and recommendation. Many popular XML algorithms use tree based methods due to their low training and prediction complexity. In this subsection, we present experiments with two such algorithms that use ensembles of trees: PfastreXML [29] and Parabel [46].

In a current deployment, both algorithms train an ensemble of trees (50 trees for PfastreXML, 3 for Parabel) using 40 million data points, each of which is a sparse vector in 4.5M dimensions. Once trained, each tree in the ensemble predicts label probabilities for 250M test data points. Both training and inference are difficult to scale – training requires weeks on a machine with few terabytes of RAM, and inference currently requires dozens of machines. As XML algorithms are applied to larger problems (e.g. web search), they need to scale to datasets with billions of points and hundreds of millions of labels, and train trees that are hundreds of gigabytes in size.

Because of the memory limitations of the platforms on which these algorithms are deployed, orchestrating data and models out of SSDs becomes critical. We demonstrate the capabilities of our library in such cases. We focus on inference since it is run more frequently than training. Similar techniques can be applied for training.

Algorithm 1 PfastreXML Inference

```
1: function CLASSIFY( $N, v$ )
2:   if  $N$  is leaf then
3:     return  $N.prob$ 
4:   else
5:     if  $\langle N.w, v \rangle + N.b > 0$  then
6:       return CLASSIFY( $N.right, v$ )
7:     else
8:       return CLASSIFY( $N.left, v$ )
```

Algorithm 2 Parabel Inference

```
1: function SCORE( $T, v, \alpha, k$ )
2:    $L \leftarrow [(T, 0.0)]$ 
3:   for each level in  $T$  from root to leaves do
4:      $L' \leftarrow []$ 
5:     for  $(N, s)$  in  $L$  do
6:        $s_l \leftarrow \langle N.w_l, v \rangle + N.b_l$ 
7:        $s_r \leftarrow \langle N.w_r, v \rangle + N.b_r$ 
8:        $s_l \leftarrow \alpha \cdot s - \max(0, 1 - s_l)^2$ 
9:        $s_r \leftarrow \alpha \cdot s - \max(0, 1 - s_r)^2$ 
10:      Append  $[(N.left, s_l), (N.right, s_r)]$  to  $L'$ 
11:    $L \leftarrow \text{top}_k(L', k)$ 
12:   return  $L$ 
```

PfastreXML: During training, trees are grown by recursively partitioning nodes starting at the root until each tree is *fully grown*. A node N is split by learning a hyperplane $N.w$ and bias $N.b$ to partition training points between its left and right children, $N.left$ and $N.right$. Node partitioning terminates when a node contains fewer points than a threshold. Leaf nodes contain a probability distribution over the label set ($N.prob$). During inference, a tree with root R assigns the probability vector over labels for a point v dictated by $\text{CLASSIFY}(R, v)$.

Parabel: During training, a tree T is grown by recursively partitioning its nodes to distribute the labels. Labels assigned to a node N are partitioned in equal numbers to its two children, $N.left$ and $N.right$. A node N containing fewer labels than a threshold is split into multiple leaf nodes with one label per leaf node. Each tree node N contains two probabilistic linear classifiers, with weights and biases $(N.w_l, N.b_l)$ and $(N.w_r, N.b_r)$, that decide whether the data point has relevant labels in its left and right subtrees. These classifiers are trained to maximize the a-posteriori probability distribution over the training data. The Parabel inference algorithm is described in Algorithm 2. α is a discount factor and k is the beam width for beam search on tree T . $\text{top}_k(L, k)$ returns the top k entries in list L , ordered by their scores in descending order. Given a point, v , and the root node, R , likely labels and their associated scores for v are contained in the return value of $\text{SCORE}(R, v, \alpha, k)$.

The inference code downloaded for both algorithms from the XML repository [8] is single-threaded and

takes about 440 hours and 900 hours for PfastreXML and Parabel inference, respectively, on Azure D14v2 nodes with 112GB RAM and 16 cores. The orchestration required to complete the inference in under two days is complex and increases the likelihood of failures.

PfastreXML inference involves a depth-first traversal of a non-balanced binary tree while Parabel inference requires breadth-first beam search on a balanced binary tree. In both cases, we noticed that the baseline code was inefficient and modified the code to take a batch of test data points (about 2-4 million per batch) and perform a level-by-level, or breadth-first, traversal of the tree. With this transformation, the new inference code was about 6× faster on nodes with a large amount of RAM. We think this is close to the limit of how fast this inference can run with DDR3 memory.

We use the *BLAS-on-flash* library to orchestrate the level-by-level traversal of each tree for a batch of points. For both algorithms, we construct one task for each (level, batch) pair. For PfastreXML inference, the DAG is data-dependent, while for Parabel, it is dependent only on the tree height. Since inference is data parallel, *BLAS-on-flash* can run tasks corresponding to multiple batches concurrently. It also orders the prefetches of tree levels and data to maximize re-use.

Evaluation. We compare the in-memory and *BLAS-on-flash* variants of the inference code on models in two regimes – Medium and Large. The Medium-sized models consist of 20GB trees containing about 25 million nodes each, while the Large Parabel model consists of 122GB trees. The Medium-sized models fit in the memory of the largest machines used in the inference platform, while the Large-sized model does not fit in the memory of any machine in the platform. We use a total of 50 trees for PfastreXML and 3 for Parabel inference. Our test data consists of 250 million points, each a sparse vector in 4.3M dimensions and taking up 500GB of storage when stored in a compressed sparse format.

We benchmark both inference algorithms on `z840`, `L32s VM`, and `sandbox` and use 2^{21} points/batch for `z840` and 2^{22} points/batch for `L32s VM` and `sandbox`. The size of *Program Cache* for *BLAS-on-flash* is set at 20GB for `z840` and 40GB for `L32s VM` and `sandbox`. We use 32 compute threads on `z840` and `L32s VM` and 64 threads on `sandbox`.

Table 6 presents the running times and memory requirements of our code on the Medium and Large-sized models. Inference code written with *BLAS-on-flash* runs at over 90% of in-memory speed using only a third of the required memory. The memory requirement can be further reduced by decreasing the test batch size or by splitting each (level, batch) task into multiple tasks in an accumulate chain. This reduction in working set, with practically no impact on performance, critically enables

	PfastreXML (50 trees)		Parabel (3 trees)		
	in-mem	flash	in-mem	flash	
sandbox	45 (155)	51.0 (42)	27.3 (125)	25.3 (47.6)	
L32s VM	69.2 (149)	67.0 (42)	44.3 (123)	45.8 (48)	
z840	–	118 (26.2)	–	71.5 (30.5)	

	Time (hours)		RAM (GB)		
	in-mem	flash	in-mem	flash	
sandbox	51.7	57.0	241.3	80.1	
L32s VM	108.4	118.2	235.5	80.9	

Table 6: Running time in hours and peak DRAM usage in GB (inside parenthesis) for XML inference on 250×10^6 data points using an ensemble of medium-sized trees (left) and large Parabel trees (right). We used 64 threads on sandbox and 32 threads on L32s VM and z840. Inference with large Parabel tree uses 70GB Program Cache.

us to execute inference on larger models, that can provide greater accuracy for ranking and relevance tasks.

6 Conclusion

We have demonstrated that (a) dense and sparse linear algebra kernels can be designed to run at near in-memory speeds on large SSD-resident datasets, (b) memory-intensive algorithms built using the library can match in-memory implementations, and (c) for complex numerical algorithms like eigensolvers, careful co-design of algorithm and software stack can offer large gains in hardware utilization and keep the costs of data analytics pipelines low.

Our results suggest that operating on data stored in fast non-volatile memory on a single node could provide an efficient alternative to distributed big-data systems for training and inference of industrial scale machine learning models for algorithms with large memory requirements. We do not make such claims about computationally intensive workloads such as training CNNs using GPUs. Further, our library provides a higher value proposition for the large quantity of NVM storage already deployed as storage in data centers. Our library can also be adapted to support GPU and other PCIe storage devices like Optane with minor changes.

7 Other Related Work

Recent work [7, 12, 13] has studied parallel and sequential external memory algorithms in the setting where writes to non-volatile memories are much more expensive than reads. They conclude that for kernels like sorting and FFTs, decreasing writes to non-volatile external memory is possible at the price of more reads. However, this is not the case in the case of linear algebra. Simple reordering of the matrix tiles on which the in-memory computation is performed can achieve asymptotic reduction in the amount of writes for `gemm` and `csrmm` calls without increase in reads. We use this observation extensively in our work.

FlashEigen [65] implements the Block KS eigensolver for large-scale graph analysis using a custom filesystem on an array of SSDs. While FlashEigen supports only a limited set of matrix operations, our library allows execution of user-defined computation graphs on

user-defined data structures. Our library uses separate IO threads to effectively pipeline IO with computation resulting in a narrow-gap with in-memory performance, while FlashEigen worker threads perform IO and then computation on matrix blocks assigned to them.

Partitioned Global Address Space systems such as FaRM [19] and UPC [15, 21, 31, 66] that present an unified view of the entire memory available in a distributed system present an alternative for programs considered here to scale to larger data and model sizes. However, the network bandwidth available presents a barrier to the scalability of sparse kernels just as in the case of Spark. Further, with careful co-design, we feel that a large range of workloads (of up to a few terabytes in size) can be processed on a single node without the cost overhead of a cluster of RDMA-enabled nodes. Scaling our library to such systems remains future work.

The problem of smart buffer-cache management for SSDs and other non-volatile memories has been studied in the database community. For example, Ma et al. [38] evaluate design choices such as paging policies that arise when one tries to extend in-memory database to hard-drives, SSDs, 3D XPoint, etc. LeanStore [36] proposes a new storage management system to extend in-memory databases to SSDs with little overhead. In contrast, our library relies on a task scheduler designed to better utilize the buffer-cache for access patterns that typically arise in linear algebra.

While our system uses existing processing and memory hardware, new hardware and accelerators that move computation to the memory have been proposed. For example, [1] proposes how expensive access patterns such as shuffle, transpose, pack/unpack might be performed in accelerator co-located with DRAM, and analyzes potential energy gains for math kernels from such accelerators. Further, systems that proposes moving entire workloads to memory systems have been proposed [23, 51, 57].

8 Acknowledgments

The authors would like to thank Anirudh Badam, Ravi Kannan, Muthian Sivathanu, and Manik Varma for their useful comments and advice.

References

- [1] AKIN, B., FRANCHETTI, F., AND HOE, J. C. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ISCA '15, ACM, pp. 131–143.
- [2] AMD. Radeon™ Pro SSG. <https://pro.radeon.com/en/product/pro-series/radeon-pro-ssg/>, 2018.
- [3] ANANDTECH. Mixed Random Read/Write Performance - Samsung 960 EVO (1TB) Review. <https://www.anandtech.com/show/10833/the-samsung-960-evo-1tb-review/8>, 2016.
- [4] ARULRAJ, J., AND PAVLO, A. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, ACM, pp. 1753–1758.
- [5] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Minimizing Communication in Linear Algebra. *SIAM Journal on Matrix Analysis and Applications* 32, 3 (2011), 866–901.
- [6] BANSAL, T., BHATTACHARYYA, C., AND KANNAN, R. A provable SVD-based algorithm for learning topics in dominant admixture corpus. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, pp. 1997–2005.
- [7] BEN-DAVID, N., BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., GU, Y., MCGUFFEY, C., AND SHUN, J. Parallel Algorithms for Asymmetric Read-Write Costs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (2016), SPAA '16, ACM, pp. 145–156.
- [8] BHATIA, K., DAHIYA, K., JAIN, H., PRABHU, Y., AND VARMA, M. The extreme classification repository: Multi-label datasets and code. <http://manikvarma.org/downloads/XC/XMLRepository.html>.
- [9] BILENKO, M., FINLEY, T., KATZENBERGER, S., KOCHMAN, S., MAHAJAN, D., NARAYANAMURTHY, S., WANG, J., WANG, S., AND WEIMER, M. Salmon: Towards Production-Grade, Platform-Independent Distributed ML. In *The ML Systems Workshop at ICML* (2016).
- [10] BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETIET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151.
- [11] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (Mar. 2003), 993–1022.
- [12] BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., GU, Y., AND SHUN, J. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (2015), SPAA '15, ACM, pp. 1–12.
- [13] CARSON, E., DEMMEL, J., GRIGORI, L., KNIGHT, N., KOANANTAKOOL, P., SCHWARTZ, O., AND SIMHADRI, H. V. Write-Avoiding Algorithms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2016), pp. 648–658.
- [14] CHEN, J., LI, K., ZHU, J., AND CHEN, W. WarpLDA: a Cache Efficient O(1) Algorithm for Latent Dirichlet Allocation. *Proc. VLDB Endow.* 9, 10 (June 2016), 744–755.
- [15] CHEN, W.-Y., BONACHEA, D., DUELL, J., HUSBANDS, P., IANCU, C., AND YELICK, K. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing* (2003), ICS '03, ACM, pp. 63–73.
- [16] DHULIPALA, L., BLELLOCH, G., AND SHUN, J. Julianne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (2017), SPAA '17, ACM, pp. 293–304.
- [17] DINH, D., SIMHADRI, H. V., AND TANG, Y. Extending the Nested Parallel Model to the Nested Dataflow Model with Provably Efficient Schedulers. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (2016), SPAA '16, ACM, pp. 49–60.
- [18] DMTK. Multiverso: Parameter Server for Distributed Machine Learning. <https://github.com/Microsoft/Multiverso>, 2015.
- [19] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)* (April 2014).
- [20] DUFF, I. S., HEROUX, M. A., AND POZO, R. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Trans. Math. Softw.* 28, 2 (June 2002), 239–267.
- [21] EL-GHAZAWI, T., AND SMITH, L. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), SC '06, ACM.
- [22] GITTEENS, A., DEVARAKONDA, A., RACAH, E., RINGENBURG, M., GERHARDT, L., KOTTALAM, J., LIU, J., MASCHHOFF, K., CANON, S., CHHUGANI, J., SHARMA, P., YANG, J., DEMMEL, J., HARRELL, J., KRISHNAMURTHY, V., MAHONEY, M. W., AND PRABHAT. Matrix Factorizations at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies. *ArXiv e-prints* (July 2016).
- [23] GUO, Q., GUO, X., BAI, Y., AND İPEK, E. A Resistive TCAM Accelerator for Data-Intensive Computing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44, ACM, pp. 339–350.
- [24] INTEL. Storage Performance Development Kit (SPDK), 2016.
- [25] INTEL®. Optane™ memory. <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html>, 2017.
- [26] INTEL®. Math Kernel Library. <https://software.intel.com/en-us/mkl>, 2018.
- [27] INTEL®. Math Kernel Library Sparse BLAS level 2 and 3 routines. <https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-level-2-and-level-3-routines>, 2018.
- [28] IRONY, D., TOLEDO, S., AND TISKIN, A. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.* 64, 9 (Sept. 2004), 1017–1026.

- [29] JAIN, H., PRABHU, Y., AND VARMA, M. Extreme Multi-label Loss Functions for Recommendation, Tagging, Ranking and Other Missing Label Applications. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (August 2016).
- [30] JIA-WEI, H., AND KUNG, H. T. I/O Complexity: The Red-Blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (1981), STOC '81, ACM, pp. 326–333.
- [31] KAMIL, A., ZHENG, Y., AND YELICK, K. A local-view array library for partitioned global address space C++ programs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (2014), ARRAY'14, ACM, pp. 26:26–26:31.
- [32] KANNAN, R., VEMPALA, S., AND VETTA, A. On Clusterings: Good, Bad and Spectral. *J. ACM* 51, 3 (May 2004), 497–515.
- [33] KUMAR, A., SINDHWANI, V., AND KAMBADUR, P. Fast Conical Hull Algorithms for Near-separable Non-negative Matrix Factorization. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (2013), ICML'13, JMLR.org, pp. I-231–I-239.
- [34] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.
- [35] LEHOUcq, R., MASCHHOFF, K., SORENSEN, D., AND YANG, C. ARPACK Software. <http://www.caam.rice.edu/software/ARPACK/>, 2009.
- [36] LEIS, V., HAUBENSCHILD, M., KEMPER, A., AND NEUMANN, T. LeanStore: In-Memory Data Management Beyond Main Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering* (2018).
- [37] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 583–598.
- [38] MA, L., ARULRAJ, J., ZHAO, S., PAVLO, A., DULLOR, S. R., GIARDINO, M. J., PARKHURST, J., GARDNER, J. L., DOSHI, K., AND ZDONIK, S. Larger-than-Memory Data Management on Modern Storage Hardware for In-Memory OLTP Database Systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (2016), DaMoN '16, ACM, pp. 9:1–9:7.
- [39] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! But at what COST? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.
- [40] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., XIN, D., XIN, R., FRANKLIN, M. J., ZADEH, R., ZAHARIA, M., AND TALWALKAR, A. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241.
- [41] MICRONSSD. UNVMe - A User Space NVMe Driver, 2016. <https://github.com/MicronSSD/unvme>.
- [42] MICROSOFT. ISLE: Importance sampling-based algorithms for large scale topic modeling. <https://github.com/Microsoft/ISLE>, 2018.
- [43] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (Mar. 2008), 40–53.
- [44] NVIDIA. cuSPARSE library. <http://docs.nvidia.com/cuda/cusparse/index.html>, 2017.
- [45] PCI-SIG. PCI Express Base Specification Revision 4.0, Version 1.0. <https://members.pcisig.com/wg/PCI-SIG/document/10912?downloadRevision=active>, October 2017.
- [46] PRABHU, Y., KAG, A., HARSOLA, S., AGRAWAL, R., AND VARMA, M. Parabel: Partitioned Label Trees for Extreme Classification with Application to Dynamic Search Advertising. In *Proceedings of the International World Wide Web Conference* (April 2018).
- [47] PRABHU, Y., AND VARMA, M. FastXML: A Fast, Accurate and Stable Tree-classifier for eXtreme Multi-label Learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014), KDD '14, ACM, pp. 263–272.
- [48] QIU, Y. Spectra - Sparse Eigenvalue Computation Toolkit as a Redesigned ARPACK, 2015. <https://spectralib.org>.
- [49] ROCKLIN, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015), pp. 130–136.
- [50] SCALEMP™. vSMP Foundation Flash Expansion. <http://www.scalemp.com/products/flx/>, 2018.
- [51] SHAFIEE, A., NAG, A., MURALIMOHAR, N., BALASUBRAMONIAN, R., STRACHAN, J. P., HU, M., WILLIAMS, R. S., AND SRIKUMAR, V. ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 14–26.
- [52] SHUN, J., AND BLELLOCH, G. E. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), PPoPP '13, ACM, pp. 135–146.
- [53] SHUN, J., ROOSTA-KHORASANI, F., FOUNTOULAKIS, K., AND MAHONEY, M. W. Parallel Local Graph Clustering. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1041–1052.
- [54] SORENSEN, D. C. Implicit Application of Polynomial Filters in a k-Step Arnoldi Method. *SIAM Journal on Matrix Analysis and Applications* 13, 1 (1992), 357–385.
- [55] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.
- [56] VITTER, J. S. External Memory Algorithms and Data Structures: Dealing with MASSIVE Data. *ACM Comput. Surv.* 33, 2 (June 2001), 209–271.

- [57] WANG, K., ANGSTADT, K., BO, C., BRUNELLE, N., SADREDINI, E., TRACY, II, T., WADDEN, J., STAN, M., AND SKADRON, K. An Overview of Micron’s Automata Processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (2016), CODES ’16, ACM, pp. 14:1–14:3.
- [58] WEIMER, M., CHEN, Y., CHUN, B.-G., CONDIE, T., CURINO, C., DOUGLAS, C., LEE, Y., MAJESTRO, T., MALKHI, D., MATUSEVYCH, S., ET AL. REEF: Retainable Evaluator Execution Framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1343–1355.
- [59] XIANYI, Z. OpenBLAS. <http://www.openblas.net/>, 2017.
- [60] XING, E. P., HO, Q., DAI, W., KIM, J.-K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015), KDD ’15, ACM, pp. 1335–1344.
- [61] YUAN, J., GAO, F., HO, Q., DAI, W., WEI, J., ZHENG, X., XING, E. P., LIU, T.-Y., AND MA, W.-Y. LightLDA: Big Topic Models on Modest Computer Clusters. In *Proceedings of the 24th International Conference on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2015), WWW ’15, International World Wide Web Conferences Steering Committee, pp. 1351–1361.
- [62] YUT, L., ZHANG, C., SHAO, Y., AND CUI, B. LDA*: A robust and large-scale topic modeling system. *Proc. VLDB Endow. 10*, 11 (Aug. 2017), 1406–1417.
- [63] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 15–28.
- [64] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.
- [65] ZHENG, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. An SSD-based eigensolver for spectral analysis on billion-node graphs. *arXiv preprint arXiv:1602.01421* (2016).
- [66] ZHENG, Y., KAMIL, A., DRISCOLL, M. B., SHAN, H., AND YELICK, K. UPC++: A PGAS extension for C++. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS ’14, IEEE Computer Society, pp. 1105–1114.
- [67] ZHOU, Y., AND SAAD, Y. Block Krylov–Schur method for large symmetric eigenvalue problems. *Numerical Algorithms 47*, 4 (Apr 2008), 341–359.

Tiresias: A GPU Cluster Manager for Distributed Deep Learning

Juncheng Gu¹, Mosharaf Chowdhury¹, Kang G. Shin¹, Yibo Zhu^{2,3}
Myeongjae Jeon^{2,4}, Junjie Qian², Hongqiang Liu⁵, Chuanxiong Guo³

¹University of Michigan, ²Microsoft, ³Bytedance, ⁴UNIST, ⁵Alibaba

Abstract

Deep learning (DL) training jobs bring some unique challenges to existing cluster managers, such as unpredictable training times, an all-or-nothing execution model, and inflexibility in GPU sharing. Our analysis of a large GPU cluster in production shows that existing big data schedulers cause long queueing delays and low overall performance.

We present Tiresias, a GPU cluster manager tailored for distributed DL training jobs, which efficiently schedules and places DL jobs to reduce their job completion times (JCTs). Given that a DL job’s execution time is often unpredictable, we propose two scheduling algorithms – *Discretized Two-Dimensional Gittins index* relies on partial information and *Discretized Two-Dimensional LAS* is information-agnostic – that aim to minimize the average JCT. Additionally, we describe when the consolidated placement constraint can be relaxed, and present a placement algorithm to leverage these observations without any user input. Experiments on the Michigan ConFlux cluster with 60 P100 GPUs and large-scale trace-driven simulations show that Tiresias improves the average JCT by up to $5.5\times$ over an Apache YARN-based resource manager used in production. More importantly, Tiresias’s performance is comparable to that of solutions assuming perfect knowledge.

1 Introduction

Deep learning (DL) is gaining rapid popularity in various domains, such as computer vision, speech recognition, etc. DL training is typically compute-intensive and requires powerful and expensive GPUs. To deal with ever-growing training datasets, it is common to perform distributed DL (DDL) training to leverage multiple GPUs in parallel. Many platform providers have built GPU clusters to be shared among many users to satisfy the rising number of DDL jobs [1, 3, 4, 9]. Indeed, our analysis of Microsoft traces shows a $10.5\times$ year-by-year increase in the number of DL jobs since 2016. Efficient job scheduling and smart GPU allocation (i.e., job placement) are the keys to minimizing the cluster-wide average JCT and maximizing resource (GPU) utilization.

Due to the unique constraints of DDL training, we observe two primary limitations in current cluster manager designs.

1. Naïve scheduling due to unpredictable training time. Although shortest-job-first (SJF) and shortest-remaining-time-first (SRTF) algorithms are known to minimize the average JCT [23, 24], they require a job’s (remaining) execution time, which is often unknown for DL training jobs. Optimus [34] can predict a DL training job’s remaining execution

time by relying on its repetitive execution pattern and assuming that its loss curve will converge. However, such proposals make over-simplified assumptions about jobs having smooth loss curves and running to completion; neither is always true in production systems (§2.2).

Because of this, state-of-the-art resource managers in production are rather naïve. For example, the internal solution of Microsoft is extended from Apache YARN’s Capacity Scheduler that was originally built for big data jobs. It only performs basic orchestration, i.e., non-preemptive scheduling of jobs as they arrive. Consequently, users often experience long queuing delays when the cluster is over-subscribed – up to several hours even for small jobs (Appendix A).

2. Over-aggressive consolidation during placement. Existing cluster managers also attempt to consolidate a DDL job onto the minimum number of servers that have enough GPUs. For example, a job with 16 GPUs requires at least four servers in a 4-GPUs-per-server cluster, and the job may be blocked if it cannot find four completely free servers. The underlying assumption is that the network should be avoided as much as possible because it can become a bottleneck and waste GPU cycles [31]. However, we find that this assumption is only partially valid.

In this paper, we propose Tiresias, a shared GPU cluster manager that aims to address the aforementioned challenges regarding DDL job scheduling and placement (§3). To ensure that Tiresias is practical and readily deployable, we rely on the analysis of production job traces, detailed measurements of training various DL models, and two simple yet effective ideas. In addition, we intentionally keep Tiresias transparent to users, i.e., all existing jobs can run without any additional user-specified configurations.

Our first idea is a new scheduling framework (2DAS) that aims to minimize the JCT when a DL job’s execution time is unpredictable. We propose two scheduling algorithms under this framework: *Discretized 2D-LAS* and *Discretized 2D-Gittins index*. The Gittins index policy [8, 21] is known to be the optimal in the single-server scenario in minimizing the average JCT when JCT distributions are known. Similarly, the classic LAS (Least-Attained Service) algorithm [33] has been widely applied in many information-agnostic scenarios, such as network scheduling in datacenters [13, 17]. Both assign each job a priority – the former uses the Gittins index while the latter directly applies the service that job has received so far – that changes over time, and jobs are scheduled in order of their current priorities.

Adapting these approaches to the DDL scheduling problem faces two challenges. First, one must consider both the spatial (how many GPUs) and temporal (for how long) dimensions of a job when calculating its priorities. We show that simply considering one is not enough. Specifically, a job’s total attained service in our algorithms jointly considers both its spatial and temporal dimensions.

More importantly, because relative priorities continuously change as some jobs receive service, jobs are continuously preempted. Although this may be tolerable in networking scenarios where starting and stopping a flow is simpler, preempting a DDL job from its GPUs can be expensive because data and model must be copied back and forth between the main memory and GPU memory. To avoid aggressive job preemptions, we apply *priority discretization* atop the two classic algorithms – a job’s priority changes after fixed intervals.

Overall, when the cluster manager has the distribution of previous job execution times that may still be valid in the near future, our scheduling framework chooses the *Discretized 2D-Gittins index*. If no prior knowledge is available, *Discretized 2D-LAS* will be applied.

Our second idea is to use model structure to loosen the consolidated placement constraint whenever possible. We observe that only certain types of DL models are sensitive to whether they are consolidated or not, and their sensitivity is due to skew in tensor size distributions in their models. We use this insight to separate jobs into two categories: jobs that are sensitive to consolidation (high skew) and the rest. We implement an RDMA network profiling library in Tiresias that can determine the model structure of DDL jobs through network-level activities. By leveraging the profiling library and the iterative nature of DDL training, Tiresias can transparently and intelligently place jobs. Tiresias first runs the job in a trial environment for a few iterations, and then determines the best placement strategy according to the criteria summarized from previous measurements.

We have implemented Tiresias¹ and evaluated using unmodified TensorFlow DDL jobs on a 15-server GPU cluster (each server with four P100 GPUs with NVlink) using traces derived from a Microsoft production cluster. We further evaluate Tiresias using large-scale trace-driven simulations. Our results show that Tiresias improves the average JCT by up to $5.5 \times$ w.r.t. current production solutions and $2 \times$ w.r.t. Gandiva [41], a state-of-the-art DDL cluster scheduler. Moreover, it performs comparably to solutions using perfect knowledge of all job characteristics.

In summary, we make the following contributions:

- Tiresias is the first information-agnostic resource manager for GPU clusters. Also, it is the first that applies two-dimensional extension and priority discretization into DDL job scheduling. It can efficiently schedule and place unmodified DDL jobs without any additional information

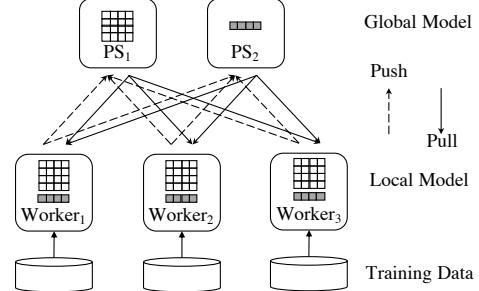


Figure 1: Data parallelism & parameter server architecture. This DDL job has two parameter servers (PS) and three workers.

from the users. When available, Tiresias can leverage partial knowledge about jobs as well.

- Tiresias leverages a simple, externally-observable, model-specific criteria to determine when to relax worker GPU collocation constraints.
- Our design is practical and readily deployable, with significant performance improvements.

2 Background and Motivation

2.1 Distributed Deep Learning (DDL)

As DL models become more sophisticated and are trained on larger datasets, distributed training is becoming more prevalent (see Appendix A). Here, we focus on *data parallelism*, which is the most common option for DDL training in popular DDL frameworks.² As Figure 1 shows, each worker occupies a GPU and works on its local copy of the DL model. The training dataset is divided into equal-sized parts to feed the workers. All jobs are trained in the *synchronous* mode which has been observed often to achieve faster convergence than asynchronous distributed training over GPUs [19].

Periodic iterations. DL training works in an iterative fashion. In each *iteration*, workers first perform *forward-backward* computation with one chunk of its training data (*minibatch*). Workers then aggregate local results to update the DL model with each other, which is referred to as *model aggregation*. Since the computation load and the communication volume are exactly the same across iterations, the iteration time of a DDL job is highly predictable.

Parameter server architecture. The parameter server (PS) architecture [30] (Figure 1) is the most popular method for model aggregation. The parameter server hosts the master copy of the DL model. It is in charge of updating the model using the local results from all workers. The workers pull back the updated model from the parameter server at the beginning of each iteration. There can be multiple parameter servers in a single DDL job.

Trial-and-error exploration. Training a DL model is not an one-time effort and often works in a *trial-and-error*

¹<https://github.com/SymbioticLab/Tiresias>

²There are other parallelization architectures, like model parallelism, which is job-specific and much less popular.

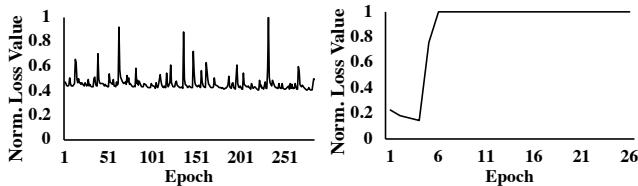


Figure 2: The training loss of two production jobs from Microsoft.

manner. DL model exposes many *hyperparameters* that express the high-level properties of the model. To get a high-quality model, the combinations of hyperparameters need to be explored in a very large search space; this is known as *hyperparameter-tuning* [41, 44]. Users can use AutoML [2] to perform this exploration efficiently and automatically by using some searching tools [14]. In AutoML, many DL jobs with different hyperparameter configurations are generated to train the same job. Most of those jobs will be killed because of random errors, or low quality of improvement. With the feedbacks from early trials, AutoML can search new configurations and spawn new jobs. Only a very small portion of those jobs with good qualities can run to completion.

2.2 Challenges

We highlight three primary challenges faced by DDL cluster managers in production. These challenges originate from the nature of DDL training and are not specific to the Microsoft cluster. See Appendix A for more details about the Microsoft cluster and its workload.

Unpredictable job duration. Current solutions that predict DL job training times [34] all assume DL jobs to (1) have smooth loss curves and (2) reach their training targets and complete. However, for many poor models during a *trial-and-error* exploration, their loss curves are not as smooth as the curves of the best model ultimately picked at the end of exploration. We show two representative examples from Microsoft in Figure 2. The spikes in the first example and the non-decreasing curve in the second example make it challenging to predict when the target will be hit. In addition to these proprietary models, popular public models sometimes also show non-smooth curves [25]. Additionally, the termination conditions of DL jobs are non-deterministic. In AutoML, most of the trials are killed because of quality issues which are determined by the searching mechanism. Usually, users also specify a maximum epoch number to train for cases when the job cannot achieve the training target. Therefore, a practical resource manager design should not rely on the accuracy/loss curve for predicting eventual job completion time.

Over-aggressive job consolidation. Trying to minimize network communication during model aggregation is a common optimization in distributed training because the network can be a performance bottleneck and waste GPU cycles [31]. Hence, many existing GPU cluster managers blindly follow a consolidation constraint when placing DDL jobs – specif-

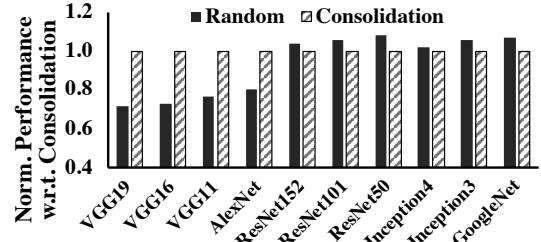


Figure 3: 4 concurrent 8-worker jobs with different placement schemes. The performance values are normalized by the value of the consolidation scheme. We use the median value from 10 (20) runs for consolidation (random) scheme.

ically, they assign all components (parameter servers and workers) of the job to the same or the minimum number of servers. A DDL job will often wait when it cannot be consolidated, even if there are enough spare resources elsewhere in the cluster. Although this constraint was originally set for good performance, it often leads to longer queuing delays and resource under-utilization in practice.

To understand the importance of this constraint, we run four concurrent 8-GPU jobs using different placement (random and always-consolidate) strategies on eight 4-GPU servers. Similar to [45], each job uses eight parameter servers – the same as the number of workers. Figure 3 shows that the locality of workers mainly impacts the VGG family and AlexNet. Nevertheless, neither the cluster operator nor the users can tell which category a job belongs to.

Time overhead of preemption. The current production cluster does not preempt jobs because of large time overhead. To show this, we manually test pausing and resuming a DDL job on our local testbed. Upon pausing, the chief worker checkpoints the most recent model on a shared storage. The checkpointed model file will be loaded by all workers when the job is resumed. Figures 4 and 5 show the detailed numbers. Whenever Tiresias preempts a job, we must take this overhead into account.

2.3 Potential for Benefits

We can achieve large gains by mitigating two common myths.

Myth I: jobs cannot be scheduled well without exact job duration. Despite the fact that DDL job durations are often unpredictable, their overall distribution can be learned from history logs. The Gittins index policy [21], which is widely used for solving the classic multi-armed bandit problem [21], can decrease the average JCT as long as the job duration distribution is given. Even without that information, the LAS algorithm can efficiently schedule jobs based on their attained service.

Myth II: DDL jobs should always be consolidated. While it is true that consolidated placement of a job may minimize its communication time, we find that some DDL jobs are insensitive to placement. We identify that the core factor is the model structure (§3.3).

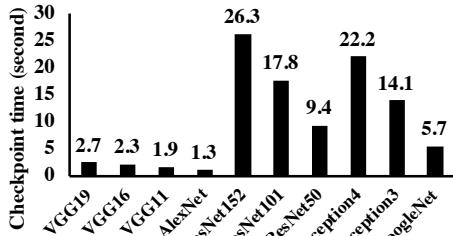


Figure 4: Time overhead of pausing a DDL job in Tensorflow. Only the chief worker checkpoints the most updated model.

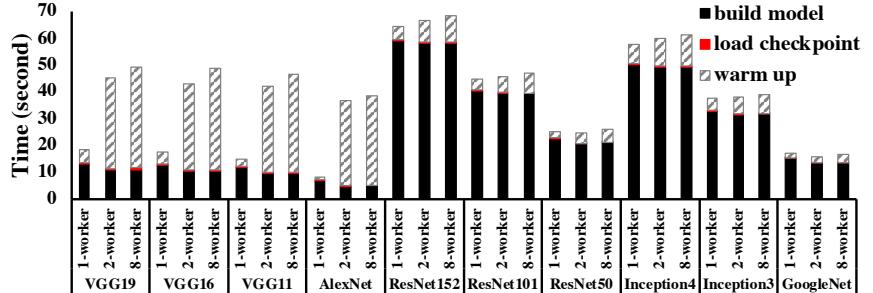


Figure 5: Time overhead of resuming a DDL job in Tensorflow. Each model is tested with different number of workers.

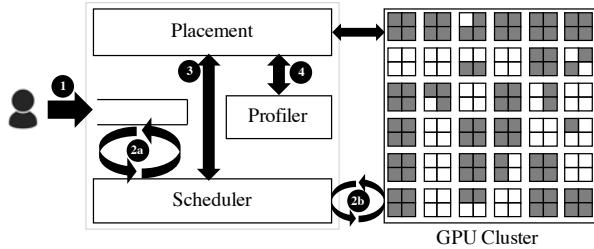


Figure 6: Tiresias components and their interactions. Job lifecycle under Tiresias is described in Section 3.1. In this figure, each machine has four GPUs; shaded ones represent GPUs in use.

In the rest of this paper, we demonstrate that Tiresias – using smarter job placement and scheduling strategies – can improve the average total job completion time by more than $5\times$ when running the same set of jobs.

3 Tiresias Design

This section describes Tiresias’s architecture, followed by descriptions of its two key components – scheduler and placement manager – and the profiler that learns the job characteristics during runtime.

3.1 Overall Architecture

Tiresias is a bespoke resource manager for GPU clusters, where the primary workload is DL training. It deals with both allocating GPUs to individual jobs (i.e., job placement) and scheduling multiple jobs over time. So, it has two primary objectives: one user-centric and the other operator-centric.

1. *Minimizing the average JCT*: Jobs should complete as fast as possible regardless of their requirements.
2. *High GPU utilization*: All the GPUs in the cluster should be utilized as much as possible.

Tiresias has an additional goal to balance between operator- and user-centric objectives.

3. *Starvation freedom*: Jobs should not starve for arbitrarily long periods.

Constraints and assumptions: Tiresias must achieve the aforementioned objectives under realistic assumptions highlighted in prior sections:

1. *Online job arrival*: Jobs are submitted by users (trial-and-error exploration mechanisms such as AutoML) in an online fashion. The resource requirements of a job J (i.e., the number of parameter servers PS_J and workers W_J) are given but unknown prior to its arrival. Model and data partitions are determined by the DL framework and/or the user [9, 16, 42]. Tiresias only deals with resource allocation and scheduling.

2. *Unknown job durations*: Because of non-smooth loss curves and non-deterministic termination in practice, a DL job’s duration cannot be predicted. However, the overall distribution of job duration may sometimes be available via history logs.

3. *Unknown job-specific characteristics*: A user does not know and cannot control how the underlying DL framework(s) will assign tensors to parameter servers and the extent of the corresponding skew.

4. *All-or-nothing resource allocation*: Unlike traditional big data jobs where tasks can be scheduled over time [11], DL training jobs require all parameter servers and workers to be simultaneously active; i.e., all required resources must be allocated together.

Job lifecycle: Tiresias is designed to optimize the aforementioned objectives without making any assumptions about a job’s resource requirements, duration, or its internal characteristics under a specific DL framework.

Figure 6 presents Tiresias’s architecture along with the sequence of actions that take place during a job’s lifecycle. As soon as a job is submitted, its GPU requirements become known, and it is appended to a WAITQUEUE (1). The scheduler (§3.2) periodically schedules jobs from the WAITQUEUE and preempts running jobs from the cluster to the WAITQUEUE (2a and 2b) on events such as job arrival, job completion, and changes in resource availability. When starting a job for the first time or resuming a previously preempted job, the scheduler relies on the placement module (§3.3) to allocate its GPUs (3). If a job is starting for the first time, the placement module first profiles it – the profiler identifies job-specific characteristics such as skew in tensor distribution – to determine whether to consolidate the job or not (4).

3.2 Scheduling

The core of Tiresias lies in its scheduling algorithm that must (1) *minimize the average JCT* and (2) *increase cluster utilization* while (3) *avoiding starvation*.

We observe that preemptive scheduling is necessary to satisfy these objectives. One must employ preemption to avoid head-of-line (HOL) blocking of smaller/shorter jobs by the larger/longer ones – HOL blocking is a known problem of FIFO scheduling currently used in production [41]. Examples of preemptive scheduling algorithms include time-sharing,³ SJF, and SRTF. For example, DL jobs in Gandiva [41] are scheduled by time-sharing. However, time-sharing based algorithms are designed for isolation via fair sharing, not minimizing the average JCT. SJF and SRTF are also inapplicable because of an even bigger uncertainty: it is difficult, if not impossible, to predict how long a DL training job will run. At the same time, size-based heuristics (i.e., how many GPUs a job needs) are not sufficient either, because they ignore job durations.

3.2.1 Why Two-Dimensional Scheduling?

By reviewing the time- or sized-based heuristics, we believe that considering only one aspect (spatial or temporal) is not enough when scheduling DDL jobs on a cluster with limited GPU resources. In an SRTF scheduler, large jobs with short remaining time can occupy many GPUs, causing non-negligible queuing delays for many small but newly submitted jobs. If the scheduler is smallest-first (w.r.t. the number of GPUs), then large jobs may be blocked by a stream of small jobs even if they are close to completion.

To quantify the approaches, we ran trace-driven simulations on three different schedulers using the Microsoft production trace: (1) smallest-first (SF); (2) SRTF; and (3) shortest-remaining-service-first (SRSF). Of them, the first two are single-dimensional schedulers; the last one considers both spatial and temporal aspects. The remaining service in SRSF is the multiplication of a job’s remaining time and the number of GPUs. For this simulation, we assume that job durations are given when needed.

Table 1 shows that SRSF outperforms the rest in minimizing the average JCT. SRSF has a much smaller tail JCT than the single-dimensional counterparts as well. Altogether, we move forward in building a DDL scheduler that considers both spatial and temporal aspects of resource usage.

Note that, among the three, SF is not a time-based algorithm; hence, it does not actively attempt to minimize the average JCT. As for the rest, SRTF is not always worse than SRSF, either. For example, large-and-short jobs that have many GPUs but short service time can mislead the SRSF scheduler and block many smaller jobs. However, in DL training, multiple GPUs are typically allocated to the jobs that have well-tuned hyperparameters and run to completion.

Table 1: Normalized performance of single-dimensional schedulers w.r.t. SRSF.

	Avg. JCT	Med. JCT	95th JCT
Smallest-First (SF)	1.52	1.20	3.45
SRTF	1.03	1.01	1.55

Therefore, the fraction of large-and-short jobs is often small in practice.

3.2.2 Two-Dimensional Attained Service-Based Scheduler (2DAS)

We address the aforementioned challenges with the 2DAS scheduler, which schedules DL jobs without relying on their exact durations while taking their GPU requirements into consideration. 2DAS generalizes the classic least-attained service (LAS) scheduling discipline [33] as well as the Gittins index policy [21] to DL job scheduling by considering both the *spatial* and *temporal* aspects of such jobs as well as their all-or-nothing characteristic. At a high-level, 2DAS assigns each job a priority based on its attained service. The attained service of a job is calculated based on the number of GPUs it uses (W_J) and the amount of time it has been running so far (t_J). The former becomes known upon the job arrival, while the latter continuously increases.

The priority function in 2DAS can be changed based on different prior knowledge. When *no job duration information* is provided, the priority function applies the LAS algorithm where a job’s priority is inverse to its attained service. If the cluster operator provides *the distribution of job duration* from previous experience, then a job’s priority equals its Gittins index value (Pseudocode 1). In the Gittins index-based algorithm, the ratio (Line 11) is between (1) the probability that the job will complete within the service quantum of Δ (i.e., the possibility of reward when adding up to Δ overhead on all subsequent jobs) and (2) the expected service that Job J will require for completion.

Both LAS and Gittins index take job’s attained service as their inputs. LAS prefers jobs that received less service. All jobs start with the highest priority, and their priorities decrease as they receive more service. The Gittins index value of job represents how *likely* the job that has received some amount of service can complete within the next service quantum. Higher Gittins index value means higher priority.

Example: Let us consider an example that illustrates both the algorithms and compares them against SRSF that has complete information (Figure 7). Three DL jobs arrive at a two-GPU machine at the same time. The resource requirement of each job is represented using (number of GPUs, duration) pairs. Only SRSF has prior knowledge of job duration. 2D-Gittins index knows the distribution, while 2D-LAS has no related information. The average JCTs in this example are 9.3, 10 and 11.7 units of time for SRSF, 2D-Gittins index, and 2D-LAS, respectively. In general, algorithms with more information perform better in minimizing the average JCT.

³ Also known as processor-sharing.

Pseudocode 1 Priority function in 2DAS

```

1: procedure PRIORITY(Job J, Distribution  $\mathbb{D}$ )
2: if  $\mathbb{D}$  is  $\emptyset$  then ▷ w/o distribution, apply LAS
3:    $R_J = -W_J \times t_J$ 
4: else ▷ w/ distribution, apply Gittins index
5:    $R_J = \text{Gittins\_Index}(J, \mathbb{D})$ 
6: return  $R_J$ 
7: end procedure
8:
9: procedure GITTINS_INDEX(Job J, Distribution  $\mathbb{D}$ )
10:   $a_J = W_J \times t_J$ 
11:   $G_J = \sup_{\Delta > 0} \frac{\mathbf{P}(S-a_J \leq \Delta | S > a_J)}{\mathbf{E}[\min\{S-a_J, \Delta\} | S > a_J]}$ 
12:  ▷  $\mathbf{P}$  is the probability and  $\mathbf{E}$  is the mean, both of which are calculated from  $\mathbb{D}$ .  $\Delta$  is the service quantum.
13:  return  $G_J$ 
14: end procedure

```

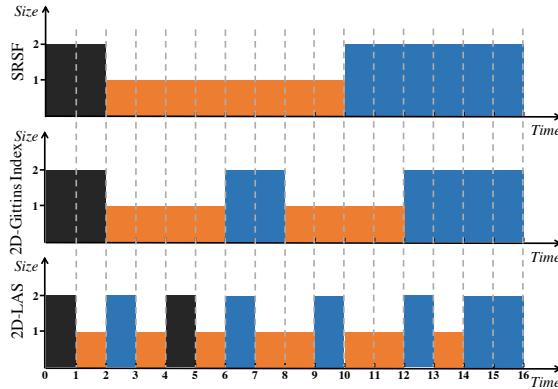


Figure 7: Time sequence of three jobs with three different two-dimensional scheduling algorithms. Job 1 (black) is (2, 2), job 2 (orange) is (1, 8), and job 3 (blue) is (2, 6). The first value in each tuple is the number of GPUs while the second is duration. The scheduling interval is one unit of time. The 2D-Gittins index values for this example are shown in Appendix C. Job index is used to break ties.

3.2.3 Priority Discretization

As observed in prior work [17], using continuous priorities can lead to a sequence of preemptions and subsequent resumptions for all jobs. Unlike preempting a network flow or a CPU process, preempting and resuming a DL job on GPU(s) can be time-consuming and expensive (§2.2). The excessive cost can make 2DAS infeasible. Furthermore, continuous preemption degenerates 2DAS to fair sharing by time-division multiplexing, which increases the average JCT.

We address these challenges by adopting the *priority discretization* framework based on the classic Multi-Level Feedback Queue (MLFQ) algorithm [12, 17, 18].

Discretized 2DAS: Instead of using a continuous priority spectrum, we maintain K logical queues (Q_1, Q_2, \dots, Q_K), with queue priorities decreasing from Q_1 to Q_K (Figure 8). The i -th queue contains jobs of attained service ($W_J t_J$) values within $[Q_i^{\text{lo}}, Q_i^{\text{hi}}]$. Note that $Q_1^{\text{lo}} = 0$, $Q_K^{\text{hi}} = \infty$, and $Q_{i+1}^{\text{lo}} = Q_i^{\text{hi}}$.

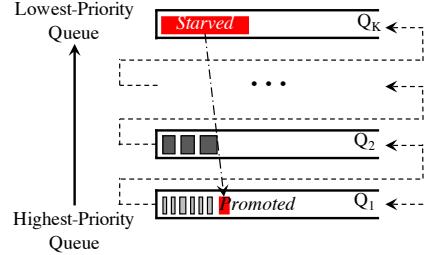


Figure 8: Discretized 2DAS with K queues. Starving jobs are periodically promoted to the highest priority queue.

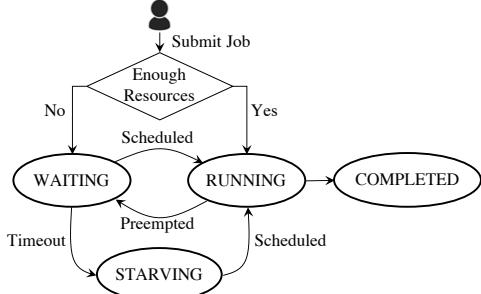


Figure 9: State transition diagram of a job in Tiresias.

Actions taken during four lifecycle events determine a job’s priority (Figure 9).

- **Arrival:** If there are available resources, a new job enters the highest priority queue Q_1 when it starts.
- **Activity:** A job is demoted to Q_{i+1} from Q_i , when its $(W_J t_J)$ value crosses queue threshold Q_i^{hi} .
- **Starvation:** A job’s priority is reset if it had been preempted for too long.
- **Completion:** A job is removed from its current queue upon completion.

The overall structure ensures that jobs with similar ($W_J t_J$) values are kept in the same queue. Jobs with highly different ($W_J t_J$) values are kept in different priority levels.

When LAS is used, jobs in the same queue are scheduled in a FIFO order of their start time (i.e., when they were first scheduled) without any risk of HOL blocking. Because of the all-or-nothing nature of DDL jobs, high-priority jobs without enough GPUs must be skipped over to increase utilization; as such, FIFO ordering on submission time instead of start time can lead to unnecessary preemptions.

The service quantum Δ in Gittins index is also discretized. For jobs in Q_i , Δ_i equals Q_i^{hi} which is the upper limit of Q_i . When a job consumes all its service quantum, it will be demoted to the lower priority queue. For Gittins index, jobs in the same queue are scheduled according to their Gittins index values. In the last queue, Q_K , Δ_K is set to ∞ . In this extreme case, Gittins index performs similar to that of LAS, and jobs in the last queue are scheduled in the FIFO order.

Determining K and queue thresholds: While the discretization framework gives us the flexibility to pick K and

corresponding thresholds, optimally picking them is an open problem [13, 17]. Instead of frequently solving an integer linear programming (ILP) [13] formulation or using a heavy-weight deep learning mechanism [15], we leverage the classic foreground-background queueing idea [33], which has been shown to perform well for heavy-tailed distributions. There are only two queues ($K = 2$) and only one threshold. Our sensitivity analysis shows that using $K = 2$ performs close to that of larger K values, ignoring preemption overheads. In practice, $K = 2$ limits the number of times a job can be preempted, which reduces job completion time.

Avoiding starvation: Using Discretized 2DAS, some jobs can starve if a continuous stream of small-and-short jobs keep arriving. This is because jobs in the same queue may be skipped over due to the lack of free GPUs. Similarly, jobs in lower priority queues may not receive sufficient GPUs either.

To avoid starvation, we promote a job to the highest-priority Q_1 if it has been WAITING for longer than a threshold: STARVELIMIT (Line 6).

This poses a tradeoff: while promotion can mitigate starvation, promoting too often can nullify the benefits of discretization altogether. To this end, we provide a single knob (PROMOTEKNOB) for the cluster operator to promote a job if its WAITING time so far (δ_J) is PROMOTEKNOB times larger than its execution time so far (t_J); i.e.,

$$\delta_J \geq \text{PROMOTEKNOB} * t_J$$

Setting PROMOTEKNOB = ∞ disables promotion and focuses on minimizing the average JCT. As PROMOTEKNOB becomes smaller, 2DAS becomes more fair, sacrificing the average JCT for tail JCT.

Note that both t_J and δ_J are reset to zero to ensure that a promoted job is not demoted right away.

3.3 Placement

Given a job J that needs PS_J parameter servers and W_J workers, if there are enough resources in the cluster, Tiresias must determine how to allocate them. More specifically, it must determine whether to consolidate the job’s GPUs in as few machines as possible or to distribute them. The former is currently enforced in Microsoft production clusters; as a result, a job may be placed in the WAITQUEUE even if there are GPUs available across the cluster.

Taking this viewpoint to its logical extreme, we created an ILP formulation to optimally allocate resources in the cluster to minimize and balance network transfers among machines (see Appendix D). The high-level takeaways from such a solution are as follows. First and foremost, it is extremely slow to solve the ILP for a large-scale cluster with many DDL jobs. Second, from small-scale experiments, we found that explicitly minimizing and balancing the load of the network does not necessarily improve DL training performance.

Pseudocode 2 2DAS Scheduler

```

1: procedure 2D-LAS(Jobs  $\mathbb{J}$ , Queues  $Q_1 \dots Q_K$ , Distribution
       $\mathbb{D}$ ) ▷ §3.2
2:     $\mathbb{P} = \{\}$  ▷ Tracks jobs to preempt
3:    for all Job  $J \in \mathbb{J}$  do
4:      if  $J$  is RUNNING then
5:         $r_J = \text{PRIORITY}(J, \mathbb{D})$  ▷ calculate job’s priority
6:      if  $J$  is WAITING longer than STARVELIMIT then
7:        Reset  $t_J$ 
8:      Enqueue  $J$  to  $Q_1$  ▷ Promote if  $J$  is STARVING
9:    while Cluster has available GPUs do
10:     for all  $i \in [1, K]$  do ▷ Prioritize across queues
11:       if  $\mathbb{D}$  is not  $\emptyset$  and  $i \in [1, K - 1]$  then
12:         Sort_Gittins_Index( $Q_i$ ) ▷ Sort jobs in  $Q_i$ 
13:       for all Job  $J \in Q_i$  do ▷ From the first  $J$  in  $Q_i$  to the end
14:         if Available GPUs  $\geq W_J$  then ▷  $J$  can run
15:           Mark  $W_J$  GPUs as unavailable
16:         else ▷  $J$  cannot run
17:            $\mathbb{P} = \mathbb{P} \cup J$ 
18:         Preempt  $J$  if it is already RUNNING
19:       for all Job  $J \in \mathbb{J}$  and  $J \notin \mathbb{P}$  do
20:         if  $J$  is not already RUNNING then
21:           if  $J$  was not profiled before then
22:             Profile  $J$  ▷ §3.3.1
23:           Store  $J$ ’s start time ▷ Used for FIFO in Discretized 2D-LAS
24:           Assign GPUs by comparing  $S_J$  to PACKLIMIT ▷ §3.3
25:    end procedure

```

How important is consolidation? Given the infeasibility of an ILP-based formulation, we focused on developing a faster solution by asking a simple question: *which jobs benefit from consolidation?*

We found that the skew of the model structure (S_J) can be a good predictor. The DL models whose performance are sensitive to consolidated placement (Figure 3) have huge tensor(s); their largest tensor size dominates the whole model (Table 6). This is because messages sizes in model aggregation are closely related to the structure of the model. For example, a model in TensorFlow consists of many tensors. Each tensor is wrapped as a single communication message.⁴ Therefore, the message size distribution in DDL depends on the tensor size distribution of the model. The tensor sizes are often unevenly distributed; sometimes there is a huge tensor which holds most of the parameters in those models. Hence, aggregating larger tensors suffers from network contention more severely, while transmissions of smaller tensors tend to interleave better with each other.

Leveraging this insight, we design Tiresias profiler that finds out the skew level of each model, which is then used by the Tiresias placement algorithm.

3.3.1 Profiler

For a given job J , Tiresias’s profiler identifies the amount of skew in tensor distributions across parameter servers (S_J)

⁴Other frameworks may split each tensor into multiple messages, but still, these messages are sent out in clear batches for each tensor.

Table 2: Comparison of DL cluster managers.

	YARN-CS	Gandiva [41]	Optimus [34]	Tiresias(Gittins index)	Tiresias(LAS)
Prior Knowledge	None	None	JCT prediction	JCT distribution	None
Scheduling Algorithm	FIFO	Time-sharing	Remaining-time-driven	Gittins index	LAS
Scheduling Input	Arrival time	N/A	Remaining time	Attained service	Attained service
Schedule Dimensions	Temporal	None	Temporal	Spatial & temporal	Spatial & temporal
Job Priority	Continuous	Continuous	Continuous	Discretized queues	Discretized queues
Job Preemption	N/A	Context switch	Model checkpoint	Model checkpoint	Model checkpoint
Minimizing Average JCT	No	No	Yes	Yes	Yes
Starvation Avoidance	N/A	N/A	Dynamic resource	Promote to Q_1	Promote to Q_1
Job Placement	Consolidation	Trial-and-error	Capacity-based	Profile-based	Profile-based

without user input and in a framework-agnostic manner. The skew is a function of the tensor size distribution of the DL job and tensor-to-parameter server mapping of the DL framework (e.g., TensorFlow assigns tensors in a round-robin fashion). Instead of forcing users to design DL models with equal-sized tensors or making assumptions about the tensor assignment algorithm of a given DL framework, we aim to automatically identify the skew via profiling.

Because each parameter server periodically sends out its portion of the updated model to each worker (§2.1), observing these network communications can inform us of the skew. Given that most production DL jobs use RDMA (e.g., InfiniBand in Microsoft) for parameter server-worker communication, and to the best of our knowledge, there exists no RDMA-level traffic monitoring tool, we have built one for Tiresias.

Tiresias’s profiler intercepts communication APIs – including the low-level networking APIs like RDMA ibverbs – in each machine to collect process-level communication traces. Whether a DDL job uses RDMA directly or through GPUDirect, Tiresias can capture detailed meta-data (e.g., message sizes) about all RDMA communications.

During the profiling run of a job, Tiresias aggregates information across all relevant machines to determine S_J for job J . Because each iteration is exactly the same from a communication perspective, we do not have to profile for too many iterations. This predictability also enables us to identify a job’s iteration boundaries, model size, and skew characteristics. Tiresias’s placement algorithm uses this information to determine whether the GPU allocation of a job should be consolidated or not.

3.3.2 The Placement Algorithm

Tiresias’s placement algorithm compares S_J with a threshold (PACKLIMIT); if S_J is larger than PACKLIMIT, Tiresias attempts to consolidate the job in as few machines as possible. As explained above, a job with a large skew performs worse due to a skewed communication pattern if it is not consolidated. For the rest, Tiresias allocates GPUs in machines to decrease fragmentation. Albeit simple, this algorithm is very effective in practice (§5). It performs even better than the pre-

vious ILP-based design because the ILP cannot capture the different effects of consolidation on different models.

Determining PACKLIMIT: We rely on job history to periodically update PACKLIMIT. Currently, we use a simple linear classifier to periodically determine the PACKLIMIT value using a job’s placement and corresponding performance as features. More sophisticated mechanism to dynamically determine PACKLIMIT can be an interesting future work.

3.4 Summary

Compared to Apache YARN’s Capacity Scheduler (YARN-CS) and Gandiva, Tiresias aims to minimize the average JCT. Unlike Optimus, Tiresias can efficiently schedule jobs without or with partial prior knowledge (Table 2). Additionally, Tiresias can smartly place DDL jobs based on the model structure automatically captured by the Tiresias profiler.

4 Implementation

We have implemented Tiresias as a centralized resource manager. The Discretized 2DAS scheduler, the placement algorithm, and the profiler are integrated into the central master, and they work together to appropriately schedule and place DDL jobs. Similar to using current DDL clusters, users submit their DDL jobs with the resource requirements, primarily the number of parameter servers (PS_J) and the number of GPUs/workers (W_J). The resource manager then handles everything, from resource allocation when a job starts to resource reclamation when it completes.

As mentioned earlier, Tiresias makes job placement decisions based on profiling via a network monitoring library. This library is present in every server of the cluster and communicates with the central profiler so that Tiresias can determine the skew of each new DDL job.

Central master: In addition to starting new jobs and completing existing ones, a major function of the master is to preempt running jobs when their (GPU) resources are assigned to other jobs by the scheduler. Because of the iterative nature of DL jobs, we do not need to save all the data in GPU and main memory for job preemption. Currently, we use the checkpoint function provided by almost every DL framework and just

Table 3: DL jobs are put into bins by their number of GPUs (**S**mall and **L**arge) and their training time (**S**hort and **L**ong)

Bin	1 (S\$)	2 (SL)	3 (LS)	4 (LL)
% of Jobs	63.5%	12.5%	16.5%	7.5%

save the most updated model for the preempted job. When a preemption is triggered, the job is first paused; then its chief worker checkpoints its model to a cluster-wide shared file system. When a paused job is resumed again by the scheduler, its most recent checkpoint will be loaded before it is restarted. The central master also determines a job’s placement using the placement algorithm and the profiler.

Distributed RDMA monitoring: Because RDMA is widely used in GPU clusters for DDL jobs, we implement the profiler as a loadable library that intercepts RDMA ibverbs APIs. Therefore, it can record all the RDMA activities on each server, such as building connections, sending and receiving data. The RDMA-level information of all relevant workers and parameter servers are then aggregated at the central profiler. Based on the aggregated information (e.g., message size and the total amount of traffic), Tiresias can resolve the detailed model information of a given DDL job, including its skew. Though implemented for RDMA networks, the profiler can easily be extended to support TCP/IP networks by intercepting socket APIs.

5 Evaluation

We have deployed Tiresias on a 60-GPU cluster and evaluated it using experiments and large-scale simulations using production traces from Microsoft. The highlights are:

- In testbed experiments, Tiresias improves the average JCT by up to $5.5\times$ and the makespan by $1.21\times$ compared to YARN-CS. It also performs comparably to SRTF, which uses complete prior information (§5.2). Tiresias’s benefits are due to job placement benefits for skewed DDL jobs and reduction in queueing delays during scheduling.
- Tiresias’s benefits hold for large-scale simulation of the production trace from Microsoft (§5.3).
- Tiresias is robust to various configuration parameters and workload variations (§5.4).

In this section, Tiresias-G (Tiresias-L) represents Tiresias using the *Discretized 2D-Gittins index* (*Discretized 2D-LAS*).

5.1 Experimental Setup

Testbed. Our testbed consists of 15 4-GPU PowerNV 8335-GTB machines from IBM in the Michigan ConFlux cluster. Each machine has 4 NVIDIA Tesla P100 GPUs with 16 GB GPU memory, two 10-core (8 threads per core) POWER8 CPUs, 256 GB DDR4 memory, and a 100 Gbps EDR Mellanox InfiniBand adapter. There is also a high-performance cluster file system, GPFS [35], shared among those machines. In Tiresias, the checkpoint files used in job

preemptions are written to and read from GPFS. The read and write throughput of GPFS from each machine is 1.2 GB/s.

Simulator. We developed a discrete-time simulator to evaluate Tiresias at large scale using a real job trace from Microsoft. It simulates all job events in Tiresias, including job arrival, completion, demotion, promotion, and preemption. However, it cannot determine job training time with the dynamic cluster environment; instead, it uses actual job completion times.

Workload. Given the scale of our GPU cluster, we generate our experimental workload of 480 DL/DDL jobs by scaling down the original job trace. Job requirements (number of GPUs, and training time) in our workload follow the distributions of the real trace. Half of these jobs are single-GPU DL jobs; the rest are DDL ones (40 2-GPU jobs, 80 4-GPU jobs, 90 8-GPU jobs, 25 16-GPU jobs, and 5 32-GPU jobs). The number of parameter servers in each DDL job is the same as its GPU number. Each model in Table 6 has 48 jobs. Each job has a fixed number of iterations to run. The training time of jobs varies from 2 mins to 2 hours. Jobs arrive following a Poisson process with an average inter-arrival time of 30 seconds. We run the jobs in synchronous data parallelism mode using TensorFlow 1.3.1 with RDMA extension and using model files from the TensorFlow benchmark [5].

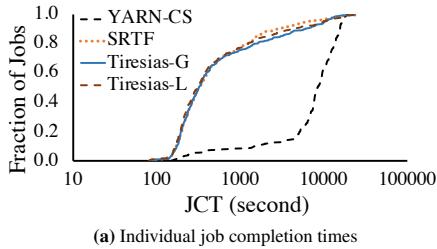
Job Bins. We category our jobs based on both their spatial (number of GPUs) and temporal (job training time) characteristics (Table 3). For the original trace, we consider a job to be *small* if it does not need more than 8 GPUs (Microsoft uses 8-GPU machines) and *short* if its training time is less than 4 hours. After scaling down, we consider a job to be small if it needs at most 4 GPUs (we are using 4-GPU machines) and short if it trains for less than 800 seconds.

Baselines. We compare Tiresias to an Apache YARNs capacity scheduler (YARN-CS) used in Microsoft (Appendix A). For comparison, we also implement an *SRTF* scheduler that has complete information, i.e., the eventual training time that in practice, cannot be obtained before running the job. Note that job durations are unknown to both Tiresias and YARN-CS. SRTF uses Tiresias’s placement mechanism. We also comapre Tiresias with the time-sharing scheduler in Gandyva [41] in the large-scale simulation.

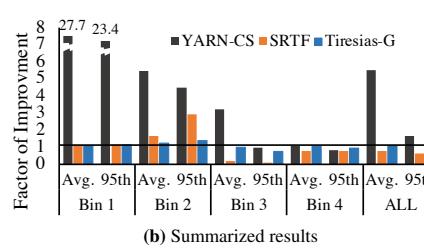
Metric. Our key metric is the improvement in the average JCT (i.e., time from submission to completion):

$$\text{Factor of Improvement} = \frac{\text{Duration of an Approach}}{\text{Duration of Tiresias-L}}$$

To clearly present the performance of Tiresias, unless otherwise specified, the results of all schedulers (including Tiresias-G) are normalized by that of Tiresias-L. Factor of improvement (FOI) greater than 1 means Tiresias-L is performing better, and vice versa.



(a) Individual job completion times



(b) Summarized results

Figure 10: Improvements in the average JCT using Tiresias w.r.t. YARN-CS and SRTF.

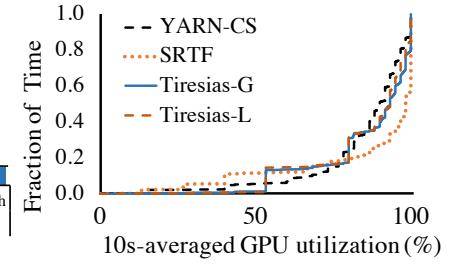


Figure 11: Cluster-wide GPU utilization.

5.2 Tiresias in Testbed Experiments

In testbed experiments, we compare the performance of YARN-CS, SRTF, Tiresias-G and Tiresias-L. For Tiresias, there are two priority queues with a threshold of 3200 GPU seconds. The PROMOTEKNOB for avoiding starvation is disabled in Testbed experiments.

5.2.1 JCT Improvements

Tiresias-L achieves $5.5\times$ improvement in terms of the average JCT w.r.t. to YARN-CS (Figure 10). If we look at the median JCT, then Tiresias-L is $27\times$ better than YARN-CS. Tiresias-G has almost the same performance as Tiresias-L ($1.06\times$ in average, $1.05\times$ in median). Its negligible performance loss is due to more job preemptions (§5.2.4). Half of all jobs avoid severe queueing delays using Tiresias. Moreover, Tiresias is not far from SRTF either.

The key idea of Tiresias's scheduler is avoiding queueing delays to small or short jobs, thus saving them from large or long jobs. When using Tiresias-L (Tiresias-G), the average JCT of jobs in Bin1 (\$\\$S\$) is just 300 (330) seconds, which is $27.6\times$ ($25.2\times$) better than that using YARN-CS. On the other hand, jobs in Bin4 (LL) have almost the same average JCT in both Tiresias and YARN-CS.

5.2.2 Cluster-Wide GPU Utilization

Figure 11 shows the averaged GPU utilizations of our cluster over time. While there are some small variations, overall utilizations across solutions look similar. However, Tiresias reduces the makespan compared to YARN-CS. The makespan of Tiresias-L (27400 seconds) was $1.21\times$ smaller than that of YARN-CS (33270 seconds), and it was similar to Tiresias-G (27510 seconds) and SRTF (28070 seconds).

5.2.3 Sources of Improvements

Smaller queueing delays. Tiresias's scheduler can reduce the average queueing delay of all jobs (Table 4), especially for small and short jobs. The average queueing delay is reduced from over 8000 seconds to around 1000 seconds when comparing YARN-CS and Tiresias. More importantly, half of the jobs are just delayed for less than or equal to 13 (39) seconds in Tiresias-L (Tiresias-G), which is negligible compared to the median delay in YARN-CS. Note that while Tiresias's average queueing delay is higher than SRTF, smaller jobs actually experience similar or shorter delays.

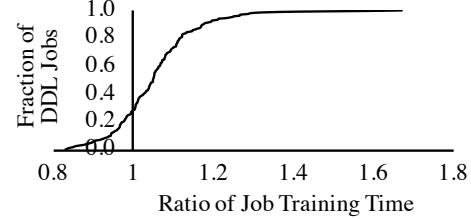


Figure 12: Performance improvement from job placement in Tiresias-L. We pick all the DDL jobs and compare their training times when Tiresias-L is running with and without placement.

Table 4: Queueing delays for DDL jobs for different solutions.

	Average	Median	95th
YARN-CS	8146s	7464s	15327s
SRTF	593s	32s	3133s
Tiresias-G	1005s	39s	7933s
Tiresias-L	963s	13s	7755s

Faster training. Albeit smaller, another source of performance improvement is Tiresias's job placement algorithm. To illustrate this, we rerun the experiment using Tiresias-L but without its profiler; i.e., jobs are randomly placed on the cluster. We compare the training time of DDL jobs in Tiresias-L without job profiling versus in the original Tiresias-L. We use the ratio of training time in random placement to Tiresias-L as the factor of improvement. In Figure 12, large ratio means random placement slows down the training, and vice versa; single-GPU jobs are excluded from the figure. Tiresias-L achieves up to $1.67\times$ improvement w.r.t. random placement, because it can identify sensitive jobs and place them on minimal number of machines for better performance. Fewer than 30% of DDL jobs experience limited performance loss.

Because of the highly-skewed job distribution and the variety of model types, the major improvement comes from the job scheduling by avoiding HOL blocking of small/short jobs by the large/long ones.

5.2.4 Overheads

Because Tiresias uses preemption in its scheduling algorithm, its major overhead comes from preempting DDL jobs. The Discretized 2DAS scheduler in Tiresias provides *discretized* priority levels to jobs. Hence, two cases trigger job preemptions in Tiresias: job arrivals/promotions and demotions that change the priority queue of a job. In our experiments,

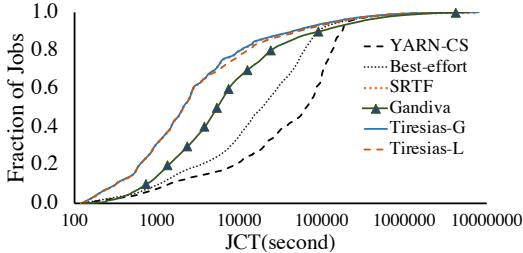


Figure 13: JCT distributions using different solutions in the trace-driven simulation. The x-axis is in logarithmic scale.

Table 5: Improvements in JCT using Tiresias in simulation. Numbers are normalized by that of Tiresias-L.

	Average	Median	95th
YARN-CS	2.41×	30.85×	1.25×
Best-effort	1.50×	9.03×	1.08×
SRTF	1.00×	1.00×	0.84×
Gandiva	2.00×	2.59×	2.08×
Tiresias-G	0.97×	1.00×	0.85×

Tiresias-L spent 13724 seconds performing 221 preemptions; Tiresias-G triggered 297 preemptions with 17425 seconds overhead in total. There are more preemptions in Tiresias-G because jobs in the same queue are sorted based on their Gittins index value at every event. In Tiresias-L, jobs will not be re-sorted because of FIFO ordering.

In contrast, job priorities in SRTF are *continuous*. Whenever short jobs come in, jobs with longer remaining time (lower priorities) may be preempted due to lack of resources. Overall, SRTF spent 18057 seconds for 316 preemptions.

Note that the exact overhead of each preemption depends on the specific job and cluster conditions.

5.3 Tiresias in Trace-Driven Simulations

Here we evaluate Tiresias’s performance on the Microsoft job trace. We compare it against YARN-CS, SRTF, and Best-effort, where Best-effort is defined as YARN-CS but without HOL blocking – i.e., it allows small jobs to jump in front of large jobs that do not have enough available GPUs.

5.3.1 Simulator Fidelity

We replayed the workload used in our testbed experiments in the simulator to verify the fidelity of our simulator. We found the simulation results to be similar to that of our testbed results – $5.11 \times (1.50 \times)$ average (95th percentile) improvement w.r.t. YARN-CS, $0.74 \times (0.55 \times)$ w.r.t. SRTF, and $1.01 \times (1.13 \times)$ w.r.t. Tiresias-G. Because the simulator cannot capture overheads of preemption, the impact of placement, or cluster dynamics, the results are slightly different.

5.3.2 JCT Improvements

We then simulated the job trace from Microsoft to identify large-scale benefits of Tiresias. Tiresias-L improves the average JCT by $2.4 \times$, $1.5 \times$, and $2 \times$ over YARN-CS, Best-effort, and Gandiva, respectively (Table 5). In addition, Tiresias-

L reduces the median JCT by $30.8 \times (9 \times)$ w.r.t. YARN-CS (Best-effort). This means half of the Microsoft jobs would experience significantly shorter queueing delays using Tiresias. Compared to Tiresias-L, Tiresias-G has almost the same (median JCT) or slightly better (average and 95th percentile JCT) performance. More importantly, Tiresias performs similar to SRTF that uses complete knowledge.

5.4 Sensitivity Analysis

Here we explore Tiresias’s sensitivity to K (number of priority queues), queue thresholds (server quantum Δ), and PROMOTEKNOB. By applying the Discretized 2D-LAS algorithm, Tiresias relies on K and corresponding thresholds to differentiate between jobs. In this section, we use $(K, \text{threshold1}, \text{threshold2}, \dots)$ to represent different settings in Tiresias. For example, $(2, 1\text{h})$ means Tiresias has 2 priority queues and the threshold between them is 1 hour GPU time.

5.4.1 Impact of Queue Thresholds

We use Tiresias with $K=2$ and increase the threshold between the two priority queues (Figure 14a and 15a). We observe that $(2, 0.5\text{h})$ is slightly worse than others who have larger thresholds in terms of the average JCT in Tiresias-L. When the threshold is larger than or equal to 1 hour, Tiresias-L’s performance almost does not change. For Tiresias-G, different Δ values have almost the same performance. These are because 1h GPU time can cover more than 60% of all the jobs.

5.4.2 Impact of K (number of priority queues)

Next, we examine Tiresias’s sensitivity to K . We evaluate Tiresias with K set to 2, 3 and 4, and pick the best thresholds in each of them. The number of priority queues does not significantly affect Tiresias (Figure 14b and 15b). The 3- and 4-queue Tiresias only improves the average JCT by 1% in comparison to the 2-queue Tiresias-L.

5.4.3 Impact of PROMOTEKNOB

This simulation is based on $(2, 1\text{h})$. We pick the initial PROMOTEKNOB as 1, and increase it by the power of 2. When PROMOTEKNOB is infinite, Tiresias does not promote. Smaller PROMOTEKNOB means more frequent promotions of long-delayed jobs back to the highest priority queue. For Tiresias-G, the maximal JCT is cut down by PROMOTEKNOB (Figure 15c). However this trace is not sensitive to the different value of PROMOTEKNOB. In Figure 14c, the 95th JCT minutely changes when we use smaller PROMOTEKNOB in Tiresias-L – the key reason PROMOTEKNOB has little impact for this trace is due to its heavy-tailed nature [33].

6 Discussion and Future Work

Formal analysis. Although Discretized 2DAS has advantages in minimizing the average JCT of DL jobs, formal analyses are still needed to precisely present its applicable boundaries (in terms of cluster resources and DL jobs’ requirements). This will simplify Tiresias configuration in practice.

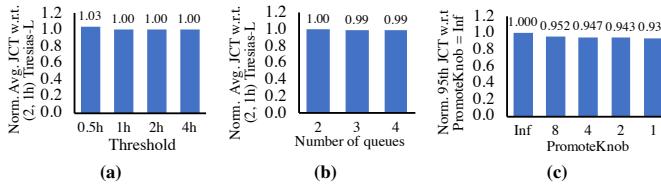


Figure 14: Sensitivity analysis of Tiresias-L. The queue settings in (b) are (2, 1h), (3, 1h, 2h), and (4, 1h, 2h, 4h) for each bar.

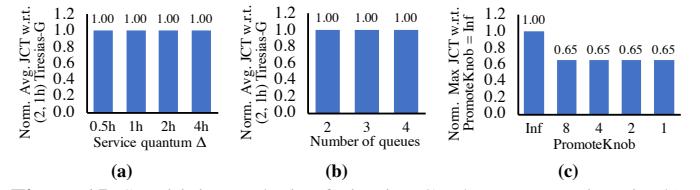


Figure 15: Sensitivity analysis of Tiresias-G. The queue settings in (b) are (2, 1h), (3, 1h, 2h), and (4, 1h, 2h, 4h) for each bar.

Lightweight preemption. Existing preemption primitives for DL jobs are time-consuming. To reduce the number of preemptions, Tiresias adopts priority discretization using MLFQ (§3.2.3). A better way of preempting DL jobs has been proposed in Gandiva [41]. However, that approach requires DL framework modifications. At the same time, its overhead is still non-negligible. With lightweight preemption mechanisms, many classic and efficient algorithms in network flow and CPU scheduling can be applied for DDL scheduling.

Fine-grained job placement. Tiresias’s profile-based placement scheme coarsely tries to avoid network transfers when necessary. However, there can be interferences within the server (e.g., on the PCIe bus) when too many workers and parameter servers are collocated. Further investigations on how placement can affect job performance are required. To this end, possible approaches include topology-aware schemes [10] and fine-grained placement of computational graphs in DL jobs [32].

7 Related Work

Cluster Managers and Schedulers. There are numerous existing resource managers and schedulers for CPU-based clusters for heterogenous workloads [20, 22–24, 26, 29, 38–40, 46] or for traditional machine learning jobs [27, 37, 44]. As explained in Section 1, these frameworks are not designed to handle the unique characteristics of DDL jobs – e.g., all-or-nothing task scheduling, and unpredictable job duration and resource requirements – running on GPU clusters.

Resource Management in DDL Clusters. Optimus [34] is an online resource scheduler for DDL jobs on GPU clusters. It builds resource-performance model on the fly and dynamically adjusts resource allocation and job placement for minimizing the JCT. It is complementary to Tiresias in terms of job placement, because the latter focuses on the efficiency of the initial job placement based on job characteristics, while the former performs online adjustment according to a job’s realtime status. However, Optimus assumes that the remaining time of a DL job is predictable, which is not always true in practice (§2.2). Tiresias can schedule jobs without any or with partial prior knowledge, and it does not rely on such assumptions. Gandiva [41] is a resource manager for GPU clusters that gets rid of the HOL blocking via GPU time sharing. However, the time-slicing scheduling approach in Gandiva brings limited improvement in terms of the average JCT.

Resource Management with Partial or No Information. To the best of our knowledge, Tiresias is the first cluster scheduler for DDL training jobs that minimizes the average JCT with partial or no information. While similar ideas exist in networking [13, 17] and CPU scheduling [12, 18], GPU clusters and DDL jobs provide unique challenges with high preemption overheads and all-or-nothing scheduling. There exist all-or-nothing gang schedulers for CPU, but they are not information-agnostic. While fair schedulers do not require prior knowledge [6, 7, 43], they cannot minimize the average JCT. Similar to the Gittins index policy, shortest-expected-remaining-processing-time (SERPT) [36] just needs partial knowledge of job durations. However, the Gittins index policy is proven to be better because it prioritizes a larger number of potentially shorter jobs [36].

8 Conclusion

Tiresias is a GPU cluster resource manager that minimizes distributed deep learning (DDL) jobs’ completion times with partial or no a priori knowledge. It does not rely on any intermediate DL algorithm states (e.g., training loss values) or framework specifics (e.g., tensors-to-parameter server mapping). The key idea in Tiresias is the 2DAS scheduling framework that has two scheduling algorithms (*Discretized 2DLAS* and *Discretized 2D-Gittins index*). They can respectively minimize the average JCT with no and partial prior knowledge. Additionally, Tiresias’s profile-based job placement scheme can maintain the resource (GPU) utilization of cluster without hurting job performance. Compared to a production solution (Apache YARN’s Capacity Scheduler) and a state-of-the-art DDL cluster scheduler (Gandiva), Tiresias shows significant improvements in the average JCT.

Acknowledgments

Special thanks go to the ConFlux team from the University of Michigan, especially Karthik Duraisamy and Todd Raeker, for reserving enough GPU servers to make Tiresias experiments possible. We would also like to thank the anonymous NSDI reviewers, our shepherd, KyoungSoo Park, and SymbioticLab members, especially Yiwen Zhang, for their constructive comments and feedback that helped improve the paper. This work was supported in part by NSF grants CCF-1629397, CNS-1563095, and CNS-1617773. Computing resources were provided by the NSF via OAC-1531752.

References

- [1] Amazon EC2 Elastic GPUs. <https://aws.amazon.com/ec2/elastic-gpus/>.
- [2] AutoML. <http://www.ml4aad.org/automl/>.
- [3] GPU-Accelerated Microsoft Azure. <https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/>.
- [4] GPU on Google Cloud. <https://cloud.google.com/gpu/>.
- [5] TensorFlow Benchmark Code. <https://github.com/tensorflow/benchmarks>.
- [6] YARN Capacity Scheduler. <http://goo.gl/cqwcp5>.
- [7] YARN Fair Scheduler. <http://goo.gl/w5edEQ>.
- [8] S. Aalto, U. Ayesta, and R. Righter. On the gittins index in the m/g/1 queue. *Queueing Systems*, 63(1-4):437, 2009.
- [9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.
- [10] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinader. Topology-aware gpu scheduling for learning workloads in cloud environments. In *SC*, 2017.
- [11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.
- [12] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Scheduling: The multi-level feedback queue. In *Operating Systems: Three Easy Pieces*. 2014.
- [13] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI*, 2015.
- [14] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.
- [15] L. Chen, J. Lingys, K. Chen, and F. Liu. Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*, 2018.
- [16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.
- [17] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.
- [18] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Spring Joint Computer Conference*, pages 335–344, 1962.
- [19] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *EuroSys*, 2016.
- [20] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.
- [21] J. Gittins, K. Glazebrook, and R. Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.
- [22] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *OSDI*, 2016.
- [23] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.
- [24] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.
- [25] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE CVPR*, 2016.
- [26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
- [27] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.
- [28] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. *arXiv preprint arXiv:1901.05758*, 2019.
- [29] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayananmurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

- [30] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.
- [31] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. *arXiv preprint arXiv:1805.07891*, 2018.
- [32] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.
- [33] M. Nuyens and A. Wierman. The Foreground-Background queue: A survey. *Performance Evaluation*, 65(3):286–307, 2008.
- [34] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.
- [35] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.
- [36] Z. Scully, M. Harchol-Balter, and A. Scheller-Wolf. Soap: One clean analysis of all age-based scheduling policies. In *SIGMETRICS*, 2014.
- [37] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *IEEE Smart Computing (SMARTCOMP)*, 2017.
- [38] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.
- [39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.
- [40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.
- [41] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.
- [42] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Research, October 2014.
- [43] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.
- [44] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *ACM SoCC*, 2017.
- [45] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *USENIX ATC*, 2017.
- [46] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. In *VLDB*, 2014.

A Characteristics of Production Cluster

We describe Project Philly [28], the cluster manager in one of Microsoft internal production clusters, referred as P . P is shared by several production teams that work on projects related to a search engine. It is managed by an Apache YARN-like resource manager, which places and schedules DDL jobs submitted by users via a website/REST API front end. P supports various framework jobs, including TensorFlow, Caffe and CNTK. In 2016, P consisted of around 100 4-GPU servers. In 2017, due to the surging demand of running DDL, P is expanded by more than 250 8-GPU servers. So, the total number of GPUs has grown by $5\times$. Servers in P are interconnected using a 100-Gbps RDMA (InfiniBand) network.

We collect traces from P over a 10-week period from Oct. 2017 to Dec. 2017. This cluster runs *Ganglia* monitoring system, which collects per-minute statistics of hardware usage on every server. Since some jobs are quickly terminated because of bugs in user’s job configuration, we only show the data of jobs that run for at least one minute. Also, we collect the per-job logs output by the DL framework which include the time for each iteration and the model accuracy along the running time. The network-level activities are monitored by Tiresias profiler which is explained in Section 3.3.1, that logs every RDMA network operation, e.g., the send and receive of every message, and their timestamps. In addition, we add hooks that intercept the important function calls in a DDL framework, e.g., the start of an iteration or aggregation, and log their timestamps.

Although we cannot disclose the details of proprietary DL models in P , we present the results of several public and popular models, some of which are also run in P .

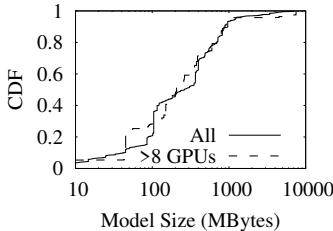


Figure 16: The CDF of DL model sizes being trained.

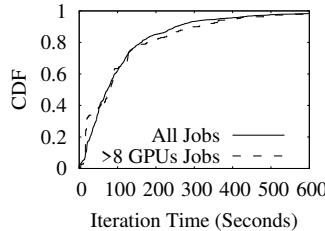


Figure 17: The CDF of average iteration time per job.

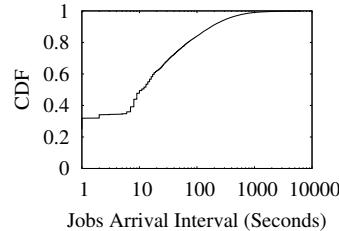


Figure 18: The CDF of DDL job arrival intervals.

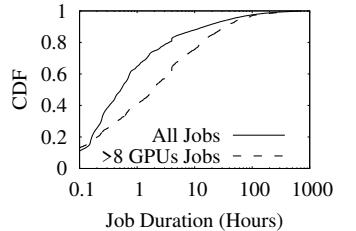


Figure 19: The CDF of DDL job duration.

Large and different DL model sizes. As shown in Figure 16, production DL models range from a few hundreds of megabytes to a few gigabytes. The model size distribution is rather independent from the number of GPUs used. According to the cluster users, the number of GPUs used often depends more on the training data volume and the urgency of jobs, and less on model sizes. Larger model sizes mean heavier communication overhead per iteration in distributed training. The largest one is 7.5GB. It may cause network congestion even with 100 Gbps network and greatly hurt the job-level performance.⁵ To minimize this overhead, existing job placement strategy intuitively consolidates DDL jobs as much as possible.

A staggering increase in the number DDL jobs. We compare the number of DDL jobs (with at least two GPUs) during ten weeks from Oct. 2017 to Dec. 2017, and the number of DDL jobs during the same ten weeks in 2016. The total number of DDL jobs has grown by $10.5 \times$ year over year. We refer to jobs using more than 8 GPUs as “large jobs,” since such jobs have to run on multiple servers (8 GPUs per server in P). Large jobs have grown by $9.4 \times$. The largest job run on 128 GPUs in 2017, while the number was 32 GPUs in 2016. We expect this trend to continue as DL jobs are trained on ever larger data sets.

Long job queuing time in production clusters. We also observe that the number of DDL jobs is increasing faster than the speed of cluster expansion. As a result, some jobs have to wait in a queue when the cluster is overloaded. From the trace, we see the average queuing delay of all jobs is 4102 seconds! A brute-force solution is to add GPUs as fast as the demand. However, this poses significant monetary costs – each 8-GPU server in P costs around 100K US Dollars based on public available GPU price. Thus, the DDL cluster service providers are seeking ways to improve job completion time (including the queuing time) given limited GPU resources.

Unpredictable job arrivals. Since the cluster is shared by multiple teams and jobs are submitted on demand, the job arrival intervals are naturally unpredictable. Figure 18 shows that the job arrival interval is mostly less than one hour. Many

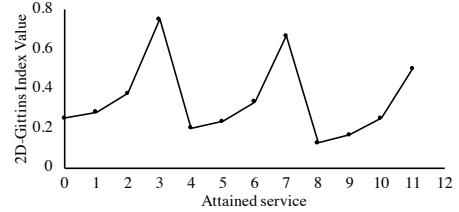


Figure 20: 2D-Gittins index value in §3.2.2. Jobs have required service 4, 8, and 12, each with probability 1/3.

arrival intervals are less than one second, suggesting that they are generated by AutoML to sweeping hyperparameters.

Various aggregation frequency depending on algorithm demands. The communication overhead also depends on how frequently aggregations are performed, which depends on the minibatch sizes. The size of minibatches is determined by the model developers – the larger the minibatches, the larger the learning step, which may help the learning process avoid local optimas but risk final convergence due to too-large steps. Thus, it is usually chosen by the users based on the requirements of specific models.

Figure 17 shows that the per iteration time varies significantly across jobs. However, the distribution of large jobs is very close to all jobs. This means that users probably do not choose minibatch sizes based on how many GPUs are used in each job.

B Characteristics of Popular DNN models

In Table 6, we pick 10 popular DNN models and present the details of their model structures for their TensorFlow implementations [5]. For the VGG family and AlexNet, the size of each model is dominated by its largest tensor. For the rest, their tensor size distributions are less skewed.

C 2D-Gittins Index Value in Section 3.2.2

When using 2D-Gittins index scheduling algorithm, the priorities of the jobs is determined by their corresponding 2D-Gittins index value mapped to their attained service. The three jobs in Figure 7 follow the same 2D-Gittins index in Figure 20.

⁵Section 3.3 shows that in fact it mostly depends on the model structure.

Table 6: Characteristics of 10 popular DNN models in TensorFlow

Model	Model size (MB)	#Tensors	#Large tensors ($\geq 1\text{MB}$)	Largest tensor size (MB)	Largest tensor ratio
VGG19	548.1	39	15	392.0	71.5%
VGG16	527.8	33	12	392.0	74.3%
VGG11	506.8	23	9	392.0	77.3%
AlexNet	235.9	17	7	144.0	61.0%
ResNet152	230.2	778	48	9.0	3.9%
ResNet101	170.4	523	35	9.0	5.3%
ResNet50	97.7	268	18	9.0	9.2%
Inception4	162.9	599	81	5.9	3.6%
Inception3	91.0	397	21	7.8	8.6%
GoogleNet	26.7	117	7	3.9	14.6%

D ILP Formula for DDL Placement

When placing a DDL job on to a shared GPU cluster, the network traffic generated by that job affects not only itself, but also all the jobs that share the same machines or network links. The existing network status can affect the newly-placed DDL job as well. Therefore, the objective of placing a DDL job is to maximize the overall performance of the entire cluster. To achieve this, we have to minimize the total network traffic and also balance the network load on individual machines in the cluster. In our ILP formulation, the objective function is to minimize the maximal network load of machines when placing a new DDL job onto the cluster.

By default, we assume all DDL jobs have the same number of parameter servers (PS) and GPU worker, which is a common practice [45]. Actually, changing number of parameter servers does not affect the total amount of data in aggregation in the parameter server architecture. There are N GPU nodes in the cluster. N_i is the i -th node whose network traffic from existing DDL jobs is t_i . And N_i has g_i free GPUs before placing any new jobs. We assume a new DDL job J with model size M is going to be placed. There are W GPU workers and K parameter servers in it. The total size of tensors hosted by the j -th parameter server is s_j . For J , w_i is the number of GPU workers placed on N_i . p_{ji} is a binary variable. It will be 1 if the j -th parameter server is placed on N_i , and vice versa.

The total network traffic of N_i comes from three parts: (1) existing traffic, (2) traffic from the workers of J on it, and (3) traffic from the parameter servers of J on it. For collocated parameter servers and workers, the traffic between them has to be deducted. Therefore, the total network traffic T_i is:

$$T_i = t_i + w_i \cdot (M - \sum_{j \in K} p_{ji} \cdot s_j) + \sum_{j \in K} p_{ji} \cdot s_j \cdot (W - w_i)$$

The overall objective can then be expressed as:

$$\text{minimize } \max_{i \in N} \{T_i\}$$

The corresponding constraints are the following:

$$\forall_{i \in N} w_i \leq g_i \quad (1)$$

$$\sum_{i \in N} w_i = W \quad (2)$$

$$\forall_{j \in K} \sum_{i \in N} p_{ji} = 1 \quad (3)$$

The first one is GPU resource constraints on all nodes. The second one requires the consistency of total number of GPU workers in J . The last one means every parameter server must have exactly one host machine. Of course, more constraints, such as CPU and host memory limitations, can be added into this ILP formulation.

Correctness and Performance for Stateful Chained Network Functions

Junaid Khalid Aditya Akella
University of Wisconsin - Madison

Abstract: Network functions virtualization (NFV) allows operators to employ NF chains to realize custom policies, and dynamically add instances to meet demand or for failover. NFs maintain detailed per- and cross-flow state which needs careful management, especially during dynamic actions. Crucially, state management must: (1) ensure NF *chain-wide* correctness and (2) have good performance. To this end, we built CHC, an NFV framework that leverages an external state store coupled with state management algorithms and metadata maintenance for correct operation even under a range of failures. Our evaluation shows that CHC can support $\sim 10\text{Gbps}$ per-NF throughput and $< 0.6\mu\text{s}$ increase in median per-NF packet processing latency, and chain-wide correctness at little additional cost.

1 Introduction

NFV vastly improves network management. It allows operators to implement rich security and access control policies using NF chains [5, 14, 10, 6, 1]. Operators can overcome NF failure and performance issues by spinning up additional instances, and dynamically redistributing traffic [15, 29].

To be applicable to enforcing policies correctly, NFV must provide *chain output equivalence* (COE): given an input packet stream, at any point in time, the collective action taken by all NF instances in an NFV chain (Figure 1a) must match that taken by an hypothetical equivalent chain with infinite capacity always available single NFs (Figure 1b). COE must hold under dynamics: under NF instance failures/slow-downs, traffic reallocation for load balancing/elastic scaling, etc. Given that NFV is targeted for cloud and ISP deployments, COE should not come at the cost of *performance*.

These goals are made challenging by NFs' *statefulness*. Most NFs maintain detailed internal state that could be updated as often as per packet. Some of the state may be shared across instances. For example, the IDS instances in Figure 1a may share *cross-flow state*, e.g., per port counters. They may also maintain *per-flow state*, e.g., bytes per flow, which is confined to within an instance.

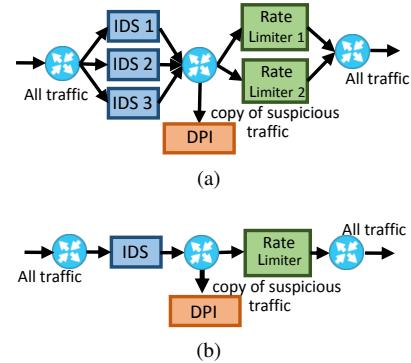


Figure 1: (a) Example NFV chain with many instances per NF (b) logical view with infinite capacity NFs/links for COE.

Ensuring COE under statefulness requires that, as traffic is being processed by many instances, or being reassigned across instances, updates to state at various NFs must happen in a “correct” fashion. For example, shared state updates due to packets arriving at IDS1 must be reflected at IDS2; likewise, when reallocating a flow, say f_1 , from IDS1 to 2, f_1 's state should be updated due to in-flight f_1 packets arriving at both IDSes 1 and 2. Finally, *how* the state is updated can determine an NF's action. For example, the off-path Trojan detector [12] in Figure 2 relies on knowing the exact order in which connection attempts were made. When there is a discrepancy in the order observed w.r.t. the true order – e.g., due to intervening NFs running slow or failing – the Trojan detector can arrive at incorrect decisions, violating COE.

Many NFV frameworks exist today [29, 25, 26, 20, 11, 17, 32]. Several of them focus on managing NF state migration or updates upon traffic reallocation during scaling or failover [29, 25, 26, 16, 32]. However, they either violate COE, or suffer from poor performance (or both).

First, most systems **ignore shared state** [29, 25, 26, 20]. They assume that NFs do not use cross-flow state, or that traffic can be split across NF instances such that sharing is completely avoided. Unfortunately, neither assumption is valid; many NFs [21, 30, 8, 22] have cross-flow state, and the need for fine-grained traffic partitioning for load balanc-

ing can easily force cross-flow state sharing across instances. Because shared state is critical to NF processing, ignoring how it is updated can lead to inconsistent NF actions under dynamics, violating COE (§2.2).

Second, existing approaches **cannot support chain-level consistency**. They cannot ensure that the order of updates made to an NF’s state (e.g., at the Trojan detector [12] in Figure 2) are consistent with the input packet stream. This inability can lead to NFs arriving at incorrect decisions, e.g., missing out on detecting attacks (as is the case in Figure 2), violating COE. Similar issues arise in the inability to correctly suppress spurious duplicate updates observed at an NF due to recovery actions at upstream NFs (§2.1).

Finally, existing frameworks impose **high overhead** on state maintenance, e.g., 100s of milliseconds to move per-flow state across instances when traffic is reallocated (§2.2).

We present a new NFV framework, CHC (“correct, high-performance chains”), which overcomes these drawbacks. For COE, CHC uses three building blocks. CHC stores NF state in an in-memory **external state store**. This ensures that state continues to be available after NF instances’ recover from failure, which is necessary for COE. Second, it maintains simple **metadata**. It adds a “root” at the entry of a chain that: (1) applies a unique logical clock to every packet, and (2) logs packets whose processing is still ongoing in the chain. At the store and NFs, CHC tracks packet clocks along with update operations each NF issues. Clocks help NFs to reason about relative packet ordering irrespective of intervening NFs’ actions, and, together with datastore logs, help suppress duplicates. We develop failure recovery protocols which leverage clocks and logs to ensure correct recovery from the failure. In the extended version of our paper [18], we prove their correctness by showing that the recovered state is same as if no failure has occurred, thereby ensuring COE.

State externalization can potentially slow down performance of state reads/writes. Thus, for performance, CHC introduces **NF-aware algorithms** for shared state management. It uses scope-awareness of state objects to partition traffic so as to minimize cross-instance shared state coordination. It leverages awareness of the state access patterns of NFs to implement strategies for shared state caching. Because most NFs today perform a simple set of state update operations, CHC offloads operations to the state store, which commits them in the background. This speeds up shared state updates – all coordination is handled by the store which serializes the operations issued by multiple NF instances.

We built a multi-threaded C++ prototype of CHC along with four NFs. We evaluate this prototype using two campus-to-EC2 packet traces. We find that CHC’s state management optimizations reduce latency overhead to **0.02 μ s - 0.54 μ s** per packet compared to traditional NFs (no state externalization). CHC failover offers **6X** better 75%-ile per packet latency than [29]. CHC is 99% faster in updating strongly

consistent shared state, compared to [16]. CHC obtains per-instance throughput of 9.42Gbps – **same as maximum achievable** with standalone NFs. CHC’s support for chain-wide guarantees adds little overhead, but **eliminates false positives/negatives** seen when using certain security NFs in existing NFV frameworks. Thus, CHC is the only framework to support COE, and it does so at state-of-the-art performance.

2 Motivation

NFV allows operators to connect NFs together in chains, where each type of NF can use multiple instances to process input traffic demand. Use of software NFs and SDN [24] means that when incoming traffic load spikes, or processing is unbalanced across instances, operators can scale up by adding NF instances and/or reallocate flow processing across instances. Furthermore, hot-standby NFs can be used to continue packet processing when an instance crashes. Due to these benefits, cloud providers and ISPs are increasingly considering deploying NFV in their networks [4].

2.1 Key Requirements for COE

NFV chains are central to security and compliance policies, they must always operate correctly, i.e., ensure COE (§1). Ensuring COE is challenging: (1) NFs are stateful; they maintain state objects for individual and group of flows. These state objects may be updated on every packet and the value of these state objects may be used to determine the action on the packet. This requires support for fine grained NF state management. (2) In addition to this, COE also require that the per-NF and chain-wide state updates are consistent with the input packet stream. (3) Since chaining may create a dependency between the action taken in upstream instances and its downstream instances, it is important that the impact of a local action taken for failure recovery should be isolated from the rest of the chain. These challenges naturally map to three classes of requirements for supporting COE:

State Access: The processing of each packet requires access to up-to-date state; thus, the following requirement are necessary to ensure COE under dynamics:

- (R1) *State availability*: When an NF instance fails, all state it has built up internally disappears. For a failover instance to take over packet processing it needs access to the state that the failed instance maintained just prior to crashing.

- (R2) *Safe cross-instance state transfers*: When traffic is reallocated across NF instances to rebalance load, the state corresponding to the reallocated traffic (which exists at the old instance where traffic was being processed) must be made available at the reallocated traffic’s new location.

Consistency: Action taken by a given NF instance may depend on shared-state updates made by other instances of the

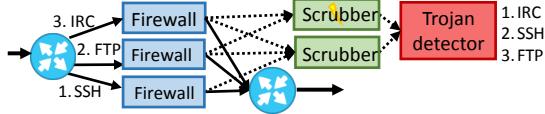


Figure 2: Illustrating violation of chain-wide ordering.

same NF, or state actions at upstream NFs in the chain. Ensuring that said NF instances’ actions adhere to COE boils down to following requirements:

- (R3) *Consistent shared state*: Depending on the nature of an NF’s state, it may not be possible to completely avoid sharing a subset of it across instances, no matter how traffic is partitioned (e.g., port counts at the IDSEs in Figure 1a). Such state needs to be kept consistent across the instances that are sharing; that is, writes/updates made locally to shared state by different instances should be executed at all other instances sharing the state in the same global order. Otherwise, instances may end up with different views of shared state leading to inconsistent and hence incorrect actions.

- (R4) *Chain-wide ordering*: Some NFs rely on knowing the order in which traffic entered the network. Consider Figure 2. The off-path Trojan detector [12] works on a copy of traffic and identifies a Trojan by looking for this sequence of steps: (1) open an SSH connection; (2) download HTML, ZIP, and EXE files over an FTP connection; (3) generate IRC activity. When a Trojan is detected, the network blocks the relevant external host. A different order does not necessarily indicate a Trojan. It is crucial that the Trojan detector be able to reason about the true arrival order as seen at traffic input.

In Figure 2, either due to one of the scrubbers being slowed down due to resource contention or recovering from failure [29], the order of connections seen at the Trojan detector may differ from that in the traffic arriving at the input switch. Thus, the Trojan detector can either incorrectly mark Trojan traffic as benign, or vice versa. When multiple instances of the Trojan detector are used, the problem is compounded because it might not be possible to partition traffic such that all three flows are processed at one instance.

- (R5) *Duplicate suppression*: In order to manage straggler NFs, NFV frameworks can adopt the following approach: (a) deploy clones initialized with the state of a slow NF instance; (b) use packet replay to bring the clone up to speed with the straggler’s state since state initialization; and (c) replicate packets to the straggler and clone (§5.3). Depending on when the clone’s state was initialized, replay can lead to duplicate state updates at the straggler. Also, the original and clone instances will then both generate duplicate output traffic. Unless such duplicate updates and traffic are suppressed, the actions of the straggler and of downstream NFs can be impacted (spurious duplicates may trigger an anomaly). The need for duplicate suppression also arises during fault recovery (§5.4).

Isolation: NFs in a chain should not be impacted by failure recovery of other NFs. Specifically:

- (R6) *Safe chain-wide recovery*: When NF failures occur and recovery takes place, it is important that the state at each NF in the chain subsequent to recovery have the same value as in the no-failure case. In other words, actions taken during recovery should not impact the processing, state, or decisions of NFs upstream or downstream from the recovering NF — we will exemplify this shortly when we describe failings of existing systems in meeting this requirement.

The network today already reorders or drops packets. Our goal is to ensure that NF replication, chaining, and traffic re-allocation together do not induce artificial ordering or loss on top of network-induced issues. This is particularly crucial for many important off-path NFs (e.g., DPI engines and exfiltration checkers) which can be thwarted by artificially induced reordering or loss.

2.2 Related work, and Our Contributions

A variety of NFV frameworks exist today [29, 16, 17, 25, 26, 20, 23, 11, 9, 28, 14, 32]. We review their drawbacks below.

Incomplete support for correctness requirements: Most existing frameworks focus on handling requirements R1 and/or R2. Split/Merge [26], OpenNF [16] and S6 [32] support cross-instance state transfers (R2). FTMB [29] and Pico Replication [25] focus on state availability (R1).

More fundamentally, Split/Merge, Pico Replication and FTMB focus on availability of the state contained entirely within an NF instance. They either ignore state shared across instances, or focus on the small class of NFs where such state is not used. Thus, these frameworks cannot handle R3.

Among existing frameworks, only OpenNF and S6 can support consistency for shared state (R3), but this comes at high performance cost. For example, OpenNF imposes a $166\mu s$ per packet overhead to ensure strong consistency! (§7). Similarly, S6 cannot support frequent updates to strongly consistent shared state.

Equally crucially, all of the above frameworks focus on a single NF; they cannot handle chains. Thus, none of them support chain-wide ordering (R4).

Support for R5 is also missing. StatelessNF [17] and S6 [32] update shared state in an external store or remote NF, respectively, but they do not support atomic updates to all state objects an instance can access. Thus, when a clone is created to mitigate a straggler off-path NF (as outlined above), the straggler may have updated other state objects that are not reflected in the clone’s initialized state. Upon replay, the straggler can make duplicate state updates (likewise, duplicate packets can also arise). For the same reason, R6 is also violated: when an NF fails over, replaying packets to bring the recovery NF up to speed can result in duplicate processing in downstream NFs.

State management performance is poor: FTMB’s periodic checkpointing significantly inflates NF packet processing latency (§7). As mentioned above, OpenNF imposes performance overhead for shared state. The overhead is high

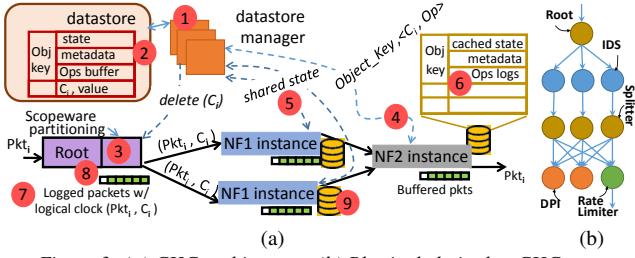


Figure 3: (a) CHC architecture; (b) Physical chain that CHC runs.

even for cross-instance transfers of per-flow state: this is because such transfers require extracting state from an instance and installing it in another while ensuring that incoming packets are directed to the state’s new location.

Our contributions: How do we support requirements R1-R6 while ensuring good state management performance? Some NFs or operating scenarios may just need a subset of R1-R6. However, we seek a *single framework* that meets all requirements/scenarios because, with NFV becoming mainstream, we believe we can no longer trade-off general correctness requirements for performance or functionality (specific NFs). Thus, we identify *basic building blocks* and study how to synthesize them into one framework. We have set ourselves the ambitious goal of designing a single generic NFV framework to support all of these requirements, though some NFs may only need support for a subset of these requirements. Building such a framework is especially challenging because we must carefully deal with shared state and NF chaining.

Our system, CHC, has three building blocks (Figure 3a): We maintain NF state in an in-memory **state store external to NFs** (1; §4). NFs access the store to read/write relevant state objects. This ensures state availability (R1). The store’s state object metadata simplifies reasoning about state ownership and concurrency control across instances (2; §4.3). This makes state transfer safety (R2) and shared state consistency (R3) simple and efficient (§5.1).

We propose **NF state-aware algorithms** for good state read/write performance which is a key concern with state externalization. These include (§4.3): automatic state scope-aware traffic partitioning to minimize shared-state coordination (3); asynchronous state updates for state that is updated often but read infrequently; this allows packet processing to progress unimpeded (4); NFs sending update operations, as opposed to updated state, to the store, which simplifies synchronization and serialization of shared-state updates (5); scope- and access pattern-aware state caching strategies, which balances caching benefits against making cache updates immediately visible to other instances (6).

Finally, we maintain a small amount of **metadata – clocks and logs**. We insert per packet logical clocks (7; §5) which directly supports cross-instance ordering (R4). We couple clocks with logs to support duplicate suppression (R5; §5.3) and COE under failover of NFs and framework components

(R6; §5.4). We log every packet that is currently being processed at some NF in the chain (8). Logged packets are replayed across the entire chain during failover. At the state store, we store logical clocks of packets along with the state updates they resulted in, which aids duplicate suppression. At each NF, we store packet clocks along with the update operations issued and the most recently read state value (9). Together with state store snapshots, these NF-side logs support COE under datastore recovery.

Though StatelessNF [17] first advocated for externalizing state, but it has serious issues. Aside from a lack of support for R4–R6, it lacks atomic state updates: when a single NF fails after updating some but not all state objects, a failover NF can boot up with incorrect state! It requires locks for shared state updates, which degrades performance. Also, it assumes Infiniband networks for performance.

3 Framework: Operator View

In CHC, operators define “logical” NF chains (such as Figure 1b) using a DAG API. We elide low level details of the API, such as how policies are specified, and focus on aspects related to correctness and performance. Each “vertex” of the DAG is an NF and consists of operator supplied NF code, input/output, configuration, and state objects. Edges represent the flow of data (packets and/or contextual output).

The CHC framework compiles the logical DAG into a physical DAG with logical vertex mapped to one or more *instances* (Figure 3b). For example, the IDS in Figure 1b is mapped to three instances in Figure 3b. The operator can provide default parallelism per vertex, or this can be determined at run time using operator-supplied logic (see below). CHC deploys the instances across a cluster. Each instance processes a partition of the traffic input to the logical vertex; CHC automatically determines the traffic split to ensure even load distribution (§4).

The CHC framework supports chain elastic scaling and straggler mitigation. Note that the logic, e.g., when to scale is not our focus; we are interested in high performance state management and COE during such actions. Nevertheless, we outline the operator-side view for completeness: operators must supply relevant logic for each vertex (i.e., scaling¹; identifying stragglers²). CHC executes the logic with input from a “vertex manager”, a logical entity is responsible for collecting statistics from each vertex’s instances, aggregating them, and providing them periodically to the logic.

Based on user-supplied logic, CHC redirects traffic to (from) scaled up (down) NF instances or clones of straggler NFs. CHC manages state under such dynamic actions to ensure COE. CHC also ensures system-wide fault tolerance. It automatically recovers from failures of NFs or of

¹e.g., “when input traffic volume increased by a certain θ ”

²when an instance processing $\theta\%$ slower than other instances

CHC framework components while always preserving COE.

4 Traffic and State Management

We discuss how CHC processes traffic and manages state. The framework automatically partitions traffic among NF instances (§4.1) and manages delivery of packets to downstream NFs (§4.2). As packets flow, different NFs process them and update state in an external store; CHC leverages several algorithms for fast state I/O; the main challenge here is dealing with shared state (§4.3).

4.1 Traffic partitioning

CHC performs *scope-aware partitioning*: traffic from an upstream instance is partitioned across downstream instances such that: (1) each flow is processed at a single instance, (2) groups of flows are allocated to instances such that most state an instance updates for the allocated flows is not updated by other instances, and (3) load is balanced. #1 and #2 reduce the need for cross-instance coordination for shared state.

In CHC, state scope is a *first-class entity*. A function `.scope()` associated with a vertex program returns a list of scopes i.e., the set of packet header fields which are used to key into the objects that store the states for an NF; i.e., these are the different granularities at which states can be queried/updated. CHC orders the list from the most to least fine grained scope. Suppose the DPI vertex in Figure 1b has two state objects: one corresponding to records of whether a connection is successful or not; and another corresponding to the number of connections per host. The scope for the former is the 5-tuple (*src IP, dst IP, src port, dst port, protocol*); the scope for the latter is *src IP*.

CHC first attempts to partition traffic at instances immediately upstream (which, for the DPI in Figure 1b would be the IDSes) based on the most coarse-grained state scope (for the DPI this is *src IP*); such splitting results in no state sharing at the downstream (DPI) instances. However, being coarse grained, it may result in uneven load across instances. The framework gathers this information via the (DPI) vertex manager. It then considers progressively finer grained scopes and repeats the above process until load is even.

The final scope to partition on is provided in common to the *splitters* upstream. The framework inserts a splitter after every NF instance (Figure 3b). The splitter partitions the output traffic of the NF instance to instances downstream.

The *root* of a physical DAG is a special splitter that receives and splits input traffic. Roots can use multiple instances to handle traffic; in CHC, we fix root parallelism to some constant R . Network operators are required to statically partition traffic among the R roots such that the traffic processed by a root instance has no overlap in any of the 5-tuple dimensions with that processed by another instance.

Scope	Any	Per-flow	Cross-flow	Cross-flow
Access pattern	Write mostly, read rarely	Any	Write rarely (read heavy)	Write/read often
	Non-blocking ops. No caching	Caching \w periodic non-blocking flush	Caching \w callbacks	Depends upon traffic split. Cache, if split allows; flush periodically

Table 1: Strategies for state management performance

4.2 Communication

Inter-NF communication is asynchronous and non-blocking. Each NF’s outputs are received by the CHC framework which is responsible for routing the output to downstream instances via the splitter. The framework stores all the outputs received from upstream instances in a queue per downstream instance; downstream instances poll the queue for input. This approach offers three benefits: (a) upstream instances can produce output independent of the consumption rate of downstream instances, (b) the framework can operate on queue contents (e.g., delete messages before they are processed downstream), which is useful for certain correctness properties, e.g., duplicate suppression (§5), (c) user logic can use persistent queues to identify stragglers/uneven load.

4.3 State Maintenance

CHC *externalizes* NF state and stores it in an external distributed key-value datastore. Thus, state survives NF crashes, improving availability and satisfying requirement R1 (§2). All state operations are managed by the datastore (Figure 3a). As described below, CHC incorporates novel algorithms and metadata to improve performance (Table 1).

State metadata: The datastore’s client-side library appends metadata to the key of the state that an NF instance stores. This contains `vertex_ID` and `instance_ID`, which are immutable and are assigned by the framework. In CHC, the key for a per-flow (5 tuple) state object is: `vertex_ID + instance_ID + obj_key`, where `obj_key` is a unique ID for the state object. The `instance_ID` ensures that only the instance to which the flow is assigned can update the corresponding state object. Thus, this metadata simplifies reasoning about ownership and concurrency control. Likewise, the key for shared objects, e.g., `pkt_count`, is: `vertex_ID + obj_key`. All the instances of a logical vertex can update such objects. When two logical vertices use the same key to store their state, `vertex_ID` prevents any conflicts.

Offloading operations: Most NFs today perform simple operations on state. Table 2 shows common examples. In CHC, an instance can *offload operations* and instruct the datastore to perform them on state on its behalf (developed contemporarily with [32]). Developers can also load custom operations. The benefit of this approach is that NF instances do not have to contend for shared state. The datastore serializes operations issued by different instances for the same shared state object and applies them in the background (In

Operation	Description
Increment/ decrement a value	Increment or decrement the value stored at key by the given value.
Push/pop a value to/from list	Push or pop the value in/from the list stored at the given key.
Compare and update	Update the value, if the condition is true.

Table 2: Basic operations offloaded to datastore manager

[18], we prove that updates will always result in consistent state.). This offers vastly better performance than the natural approach of acquiring a lock on state, reading it, updating, writing it back, and releasing the lock (§7).

Non-blocking updates: In many cases, upon receiving a packet, an NF updates state, but does not use (read) the updated value; e.g., typical packet counters (e.g., [21, 22, 30]) are updated every input packet, but the updated value is only read infrequently. For such state that is written mostly and read rarely, we offer *non-blocking updates* (Table 1): the datastore immediately sends the requesting instance an ACK for the operation, and applies the update in the background. As a further optimization, NFs do not even wait for the ACK of a non-blocking operation; the framework handles operation retransmission if an ACK is not received before a timeout. If an instance wishes to read a value, the datastore applies all previous outstanding updates to the value, in the order NFs issued them, before serving the read.

Caching: For all the objects which are not amenable to non-blocking updates, we improve state access performance using novel caching strategies that leverage state objects’ scope and access patterns (ready-heavy vs. not).

Per-flow state: CHC’s scope-aware partitioning ensures that flows that update per-flow state objects are processed by a single instance; thus, these objects do not have cross-instance consistency requirements. The datastore’s client-side library caches them at the relevant instance, which improves state update latency and throughput. However, for fault tolerance, we require local updates made to cached objects to be flushed to the store; to improve performance, these flush operations have non-blocking semantics (Table 1).

Cross-flow state: Cross-flow state objects can be updated by multiple instances simultaneously. Unlike prior works that largely ignore such state, CHC supports high performance shared state management. Some shared objects are rarely updated; developers can identify such objects as read-heavy. CHC (1) caches such an object at the instances needing them; and (2) the client-side library at each of these instances registers a *callback* with the store, which is invoked whenever the store updates the object on behalf of another instance. The NF developer does not need to provide callbacks to update state; they are handled by the client-side library.

The cached objects only serve *read requests*. Whenever an (rare) update is issued by an instance - operation is immediately sent to the store, The store applies the operation and sends back the updated object to the update initiator. At the same time, the client-side library of other instances re-

ceives callback from the store and updates the locally cached value (Table 1). We prove this approach results in consistent updates to shared state in [18]

For other cross-flow objects (not rarely-updated), the datastore allows them to be cached at an instance only as long as no other instance is accessing them (Table 1); otherwise, the objects are flushed. CHC notifies the client-side library when to cache or flush the state based on (changes to) the traffic partitioning at the immediate upstream splitter.

For **scale and fault tolerance** we use multiple datastore instances, each handling state for a subset of NF instances. Each datastore instance is multi-threaded. A thread can handle multiple state objects; however, each state object is only handled by a single thread to avoid locking overhead.

5 Correctness

So far, we focused on state management and its performance. We also showed how CHC supports requirement R1 (state availability) by design. We now show how it supports the requirements R2–R6. This is made challenging both by shared state and by chaining. To support R2–R6, CHC maintains/adds metadata at the datastore, NFs and to packets. We first describe how the most basic of the metadata – logical packet clocks and packet logs – are maintained. We describe other metadata along with the requirements they most pertain to.

Logical clocks, logging: The root (§4.1) attaches with every input packet a unique *logical clock* that is incremented per packet. The root also *logs* in the datastore each packet, the packet clock, and to which immediate downstream instance the packet was forwarded. When the last NF in a chain is done processing a packet, updating state and generating relevant output, it informs the CHC framework. CHC sends a “delete” request with the packet’s clock to the root which then removes the packet from the log. Thus, at any time, the root logs all packets that are being processed by one or more chain instances. When any NF in the chain cannot handle the traffic rate, the root log builds in size; CHC drops packets at the root when this size crosses a threshold to avoid buffer bloat. When multiple root instances are in use (§4.1), we encode the identifier of the root instance into the higher order bits of the logical clock inserted by it to help the framework deliver “delete” requests to the appropriate root instance.

5.1 R2, R3: Elastic scaling

In some situations, we may need to reallocate ongoing processing of traffic across instances. This arises, e.g., in elastic scaling, where a flow may be processed at an “old” instance and reallocated to a “new” scaled up instance. We must ensure here that the old and new instances operate on the correct values of per- and cross-flow state even as traffic is reassigned (requirements R2 and R3).

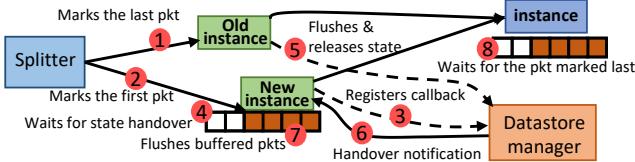


Figure 4: State handover.

Specifically, for cross-flow shared state, we require that: *updates made to the shared state by every incoming packet are reflected in a globally consistent order irrespective of which NF instance processed the corresponding packet.*

Existing systems achieve this at high overhead: OpenNF [16] copies shared internal state from/to the instances sharing it, each time it is updated by an incoming packet! In contrast, ensuring this property in CHC is straightforward due to externalization and operation offloading (§4.3): when multiple instances issue update operations for shared state, the datastore serializes the operations and applies in the background. All subsequent accesses to the shared state then read a consistent state value.

Per-flow state’s handling must be correctly reallocated across instances, too (R2). One approach is to disassociate the old instance from the state object (by having the instance remove its instance_ID from the object’s metadata) and associate the new instance (by adding its instance_ID). But, this does not ensure correct handover when there are in-transit packets that update the state: even if the upstream splitter immediately updates the partitioning rules and the traffic starts reaching the new instance, there might be packets in-transit to, or buffered within, the old instance. If the new instance starts processing incoming packets right away then state updates due to in-flight/buffered packets may be disallowed by the datastore (as a new instance is now associated with the state object) and hence the updates will be lost.

Thus, to satisfy R2, we require: *Loss-freeness, i.e., the state update due to every incoming packet must be reflected in the state object.* Furthermore, some NFs may also need *order-preservation: updates must happen in the order of packet arrivals into the network.*

These properties are crucial for off-path NFs, e.g., IDS. Such NFs cannot rely on end-to-end retransmissions to recover from lost updates induced by traffic reallocation [16]. Similarly, they may have to process packets in the order in which they are exchanged across two directions of a flow, and may be thwarted by a reordering induced by reallocation (resulting in false positives/negatives).

Figure 4 shows the sequence of steps CHC takes for R2: ① The splitter marks the “last” packet sent to the old instance to inform the old instance that the flow has been moved. This mark indicates to the old instance that it should flush any cached state associated with the particular flow(s) to the datastore and disassociate its ID from the per flow state, once it has processed the “last” packet. ② The splitter

also marks the “first” packet from the traffic being moved to the new instance. ③ When the new instance receives the “first” packet, it tries to access the per flow state from the datastore. If the state is still associated with the old instance_ID, it registers a callback with the datastore to be notified of metadata updates. ④ The new instance starts buffering all the packets associated with the flow which is being moved. ⑤ After processing the packet marked as “last”, the old instance flushes the cached state and updates the metadata to disassociate itself from the state. ⑥ The datastore notifies the new instance about the state handover. ⑦ The new instance associates its ID with the state, and flushes its buffered packets.

The above ensure that updates are not lost *and* that they happen in the order in which packets arrived at the upstream splitter. In contrast, OpenNF provides *separate algorithms* for loss-freeness and order-preservation; an NF author has the arduous task of choosing from them!

Note also that packets may arrive out of order at a *downstream* instance, causing it to make out-of-order state updates. To prevent this: ⑧ The framework ensures that packets of the moved flow emitted by the new instance are not enqueued at the downstream instance, but instead are buffered internally within the framework until the packet marked as “last” from the old instance is enqueued at the new instance.

5.2 R4: Chain-wide ordering

To support R4, we require that: *Any NF in a chain should be able to process packets, potentially spread across flows, in the order in which they entered the NF chain.* CHC’s logical clocks naturally allow NFs to reason about cross-flow chain-wide ordering and satisfy R4. E.g., the Trojan detector from §2.1 can use packets’ logical clocks to determine the arrival order of SSH, FTP and IRC connections.

5.3 R5: Straggler mitigation

R5 calls for the following: *All duplicate outputs, duplicate state updates, and duplicate processing are suppressed.*

A key scenario in which duplicate suppression is needed is straggler mitigation. A straggler is a slow NF that causes the entire NF chain’s performance to suffer. We first describe CHC’s mechanism for straggler mitigation (which kicks in once user-provided logic identifies stragglers; §3), followed by duplicate suppression.

Clone and replay: To mitigate stragglers CHC deploys *clones*. A clone instance processes the same input as the original in parallel. CHC retains the faster instance, killing the other. CHC initializes the clone with the straggler’s latest state from the datastore. It then replicates incoming traffic from the upstream splitter to the straggler and the clone.

This in itself is not enough, because we need to satisfy R2, i.e., ensure that the state updates due to packets that were

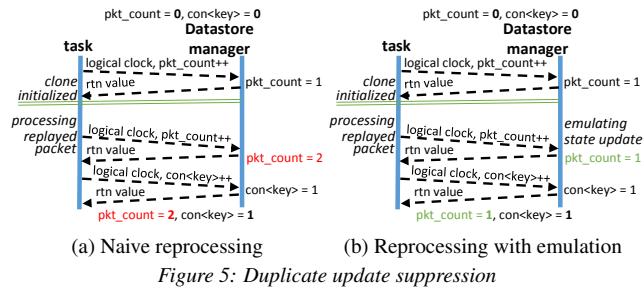


Figure 5: Duplicate update suppression

in-transit to the straggler at the time the clone’s state was initialized are reflected in the state that the clone accesses. To address this, we *replay* all logged packets from the root. The root continues to forward new incoming packets alongside replayed ones. The clone processes replayed traffic first, and the framework buffers replicated traffic. To indicate end of replay traffic, the root marks the “last” replayed packet (this is the most recent logged packet at the time the root started replaying). When replay ends (i.e., the packet marked “last” was processed by the clone), the framework hands buffered packets to the clone for processing.

Given the above approach for straggler mitigation, there are three forms of duplicates that can arise. CHC suppresses them by maintaining suitable metadata.

1. Duplicate outputs: Replicating input to the clone results in duplicate outputs. Here, the framework suppresses duplicate outputs associated with the same logical clock at message queue(s) of immediate downstream instance(s).

2. Duplicate state updates: Some of the replayed packets may have already updated some of the stragglers’ state objects. For example, an IDS updates both the total packet count and the number of active connections per host. A clone IDS may have been initialized after the straggler updated the former but not the latter. In such cases, processing a replayed packet can *incorrectly* update the same state (total packet count) multiple times at the straggler (Figure 5a). To address this, the datastore logs the state value corresponding to each state update request issued by any instance, as well as the logical clock of the corresponding packet. This is only done for packets that are currently being processed by some NF in the chain. During replay, when the straggler or clone sends an update for a state object, the datastore checks if an update corresponding to the logical clock of the replayed packet has already been applied; if so, the datastore *emulates* the execution of the update by returning the value corresponding to the update (Figure 5b). In [18], we describe how CHC handles non-deterministic state update operations.

3. Duplicate upstream processing: NFs upstream from the clone/straggler would have already processed some of the in-transit packets. In such cases, reprocessing replayed packets leads to incorrect actions at upstream NFs (e.g., an IDS may raise false alarms). To address this, each replayed packet is marked and it carries the ID of the clone where it will be processed. Such packets need special handling: the

intervening instances recognize that they are not suspicious duplicates; if necessary, the instances read the store for state corresponding to the replayed packet, make any needed modifications to the packet’s headers, and produce relevant output; the instances can issue updates to state, too, but in such cases the datastore *emulates* updates as before. The clone’s ID is cleared once it processed the packet.

5.4 R6: Safe Fault Recovery

Our description of R6 in §2 focused on NF failures; however, since CHC introduces framework components, we generalize R6 to cover other failures as well. Specifically, we require the following general guarantee:

Safe recovery Guarantee: *When an NF instance or a framework component fails and a recovery occurs, we must ensure that the state at each NF in the chain has the same value as under no failure.*

We assume the standard fail-stop model, that a machine/node can crash at any time and that the other machines/nodes in the system can immediately detect the failure.

First, we show how CHC leverages metadata to handle the failure of individual components. Then, we discuss scenarios involving simultaneous failure of multiple components.

NF Failover: When an NF fails, a failover instance takes over the failed instance’s processing. The datastore manager associates the failover instance’s ID with relevant state. Packet replay brings state up-to-speed (from updates due to in-transit packets). Similar to cloning (§5.3), we suppress duplicate state updates and upstream processing.

Since “delete” requests are generated after the last NF is done processing a packet, failure of such an NF needs special handling: consider such an instance T failing after generating an output packet for some input packet P, but before the framework sends a “delete” request for P. When P is replayed, T’s failover instance produces output again, resulting in duplicate packets at the receiving end host. To prevent this, for the last NF in the chain, our framework sends the “delete” request for P *before* the NF generates the output packet. If the NF fails before the “delete” request, then P will be replayed, but this does not result in duplicate downstream processing since the NF did not generate output. If the NF fails after the “delete” request but before generating output, then P is not replayed, and hence the end host will not receive any output packet corresponding to P. To the host, this will appear as a packet being dropped by the network, causing P to be retransmitted from the source and resulting in correct overall behavior. In [18], we show that using this protocol an NF instance recovers with state similar to that under no failure.

Non-blocking operations: Non-blocking updates, where NF instances don’t wait for ACKs, instead relying on the framework to handle reliable delivery, can introduce the following failure mode: a instance may fail after issuing state

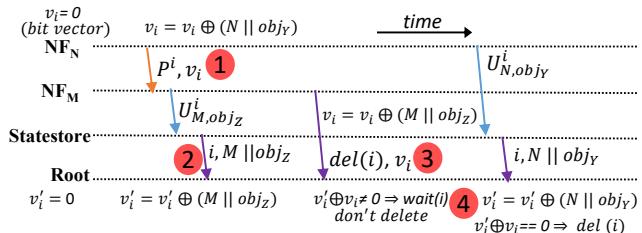


Figure 6: Recovery under non-blocking operations. Consider a packet P^i which is processed by NF_N , followed by NF_M , the last NF in the chain. NF_N and NF_M update objects obj_Y and obj_Z , respectively.

update but before the update is committed and an ACK was received. In such cases, to ensure R6, we need that *the framework must re-execute the incomplete update operation*.

Suppose an instance N fails after processing packet P_i (i is the logical clock) but before the corresponding state update operation $U_{N,obj}^i$ (obj is the state object ID) completes. P_i may have induced such operations at a subset of NF instances $\{N\}$ along the chain. A natural idea to ensure the above property is to replay packets from the root to reproduce $U_{N,obj}^i$ at various N 's. For this, however, P_i must be logged and should not have been deleted. If P_i is deleted it can't be replayed.

We need to ensure P_i continues to be logged as long as there is some N for which $U_{N,obj}^i$ is not committed. Our approach for this is shown in Figure 6: ① Each packet carries a 32-bit vector v_i (object ID and instance ID; 16b each) that is initialized to zero. Each NF instance where processing the packet resulted in a state update XORs the concatenation of its ID and the corresponding state objects' IDs into the bit vector. ② When committing a given NF's state update, the state store signals to the root the clock value of the packet that induced the update as well as the concatenated IDs. ③ The last instance sends the final vector along with its “delete” request to the root. ④ When a delete request and the final vector are received, the root XORs the concatenated IDs with the concatenated IDs reported by each signal from the state store in step 2. If the result is zero, this implies that updates induced by the packet at all NF instances $\{N\}$ were committed to the store; the root then proceeds to delete the packet from the log. Otherwise, the packet updated state at some NF, but the NF has not yet reported that the state was committed; here, the root does not delete the packet.

Root: To ensure R6 under root failover, we need that a new root must start with the logical clock value and current flow allocation at the time of root failure. This is so that the new root processes subsequent packets correctly. To ensure this, the failover root reads the last updated value of the logical clock from the datastore, and retrieves how to partition traffic by querying downstream instances' flow allocation. The framework buffers incoming packets during root recovery. We prove this approach ensures recovery with a state similar to that under no failure in [18].

Datastore instance: Recall that different NFs can store their states in different storage instances (§4.3). This ensures

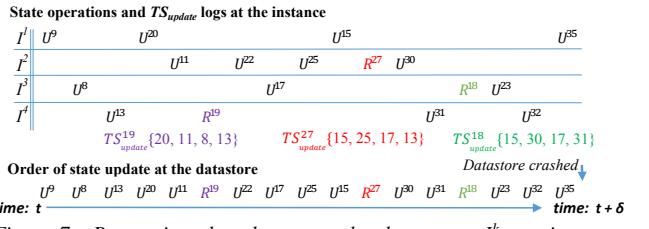


Figure 7: Recovering shared state at the datastore. I^k are instances. $U^{logical_clock}$ and $R^{logical_clock}$ represent “update” and “read”.
time: t time: $t + \delta$

that store failures impact availability of only a portion of the overall state being tracked. Now, to ensure R6 under the failure of a datastore instance, we need that the recovered state in the new store instance must represent the value which would have resulted if there was no failure. The recovered state must also be consistent with the NF instances' view of packet processing thus far (i.e., until failure).

To support this property we distinguish between per-flow and shared state. For the former, we leverage the insight that all the NFs already maintain an updated cached copy of per-flow state. If a datastore instance fails, we can simply query the last updated value of the cached per-flow state from all NF instances that were using the store.

Recovering shared state is nuanced. For this, we use checkpointing with write-ahead logging [19]. The datastore periodically checkpoints shared state along with the metadata, “TS”, which is the set of logical clocks of the packets corresponding to the last state operation executed by the store on behalf of each NF instance. Each instance locally writes shared-state update operations in a write-ahead log. Say the latest checkpoint was at time t and failure happens at $t + \delta$. A failover datastore instance boots with state from the checkpoint at t . This state now needs to be “rolled forward” to $t + \delta$ and made consistent with the NF instances' view of packet processing at $t + \delta$. Two cases arise:

(Case 1) If NF instances that were using the store instance don't read shared state in the δ time interval, then to recover shared state, the framework re-executes state update operations from the local write-ahead log on behalf of each NF, starting from the logical clocks included in the metadata TS in the checkpoint. Recall that in our design the store applies updates in the background, and this update order is unknown to NF instances. Thus, our approach ensures that the state updates upon re-execution match that produced by a *plausible* sequence of updates that the store may have enforced prior to failure. This consistency property suffices because, in Case 1, NFs are not reading shared state in the δ interval.

(Case 2) Say an NF instance issues a read between t and $t + \delta$; e.g., I^3 in Figure 7 issues R^{18} . Following the above approach may lead to an order of re-execution such that the actual state I^3 read in R^{18} is different from the state in the store after recovery. To ensure that the store's state is consistent with all I^k 's current view, the framework must re-execute operations in such an order that the datastore would have pro-

	NF instance	Root
Store instance	✓*	✓*
NF instance	✓	✓

Table 3: Handling of correlated failures (* Cannot recover if component and the store instance storing its state fail together).

duced the same value for each read in $[t, t + \delta]$.

To ensure this, on every read operation, the datastore returns TS along with the latest value of the shared state (e.g., TS^{19} is returned with I^4 ’s R^{19}). The instance then logs the value of the shared state along with the corresponding TS . Re-execution upon failure then needs to select, among all TS ’s at different instances, the one corresponding to the most recent read from the store prior to the crash (i.e., TS^{18} , since R^{18} is the most recent read; most recent clock does not correspond to most recent read). How selection is done is explained shortly; but note that when the framework re-executes updates starting from the clock values indicated by this selected TS that would bring the store in sync with all NFs. In our example, TS^{18} is the selected TS ; we initialize the store state with the value in the corresponding read (R^{18}). From the write-ahead log of each NF, the framework re-executes update operations that come after their corresponding logical clocks in TS^{18} . At instance I^1 , this is the update after U^{15} , i.e., U^{35} . At I^3 and I^4 these are U^{23} and U^{32} , respectively. Shared state is now in sync with all NFs.

TS selection works as follows: first we form a set of all the TS ’s at each instance, i.e., $Set = \{TS^{18}, TS^{19}, TS^{27}\}$. Since the log of operations at an instance follows a strict clock order we traverse it in the reverse order to find the latest update operation whose corresponding logical clock value is in Set . For example, if we traverse the log of I^1 , we find that the logical clock of U^{15} exists in Set . After identifying such a logical clock value, we remove all the entries from Set which do not contain the particular logical clock value (such TS s cannot have the most recent read); e.g., we remove TS^{19} as it does not contain logical clock 15. Similarly, we remove TS^{27} , after traversing I^2 ’s log. Upon doing this for all instances we end up selecting TS^{18} for recovery. In [18], we prove that using this protocol the store recovers with state similar to that under no failure.

Correlated failures: Using the above approaches, CHC can also handle correlated failures (Table 3) of multiple NF instances, root, and storage instances. However, CHC cannot withstand correlated failure of a store instance with any other component that has stored its state in that particular instance. Replication of store instances can help recover from such correlated failures, but that comes at the cost of increasing the per packet processing latency.

6 Implementation

Our prototype consists of an execution framework and a datastore, implemented in C++. NFs runs in LXC containers [3] as multithreaded processes. NFs are implemented

NF	Description of state object	Scope; access pattern
NAT	Available ports	Cross-flow; write/read often
	Total TCP packets	Cross-flow; write mostly, read rarely
	Total packets	Cross-flow; write mostly, read rarely
	Per conn. port mapping	Per-flow; write rarely, read mostly
Trojan detector	Arrival time of IRC, FTP and SSH flows for each host	Cross-flow; write/read often
Portscan detector	Likelihood of being malicious (per host)	Cross-flow; write/read often
	Pending conn. initiation req. along with its timestamp	Per-flow; write/read often
Load balancer	Per server active # of conn.	Cross-flow; write/read often
	Per server byte counter	Cross-flow; write mostly, read rarely
	Conn. to server mapping	Per-flow; write rarely, read mostly

Table 4: NFs and description of their state objects

using our CHC library that provides support for input message queues, client side datastore handling, retransmissions of un-ACK’d state updates (§4.3), statistics monitoring and state handling. Packet reception, transmission, processing and datastore connection are handled by different threads.

For low latency, we leverage Mellanox messaging accelerator (VMA) [31] which enables user-space networking with kernel bypass similar to DPDK [13]. In addition to this, VMA also supports TCP/UDP/IP networking protocols and does not require any application modification. Even though we use VMA, we expect similar performance with other standard kernel bypass techniques. Protobuf-c [7] is used to encode and decode messages between a NF instance and the datastore. Each NF instance is configured to connect to a “framework manager” to receive information about its downstream instances (to which it connects via tunnels), datastore instances and other control information.

The framework manager can dynamically change the NF chain by instantiating new types of NFs or NF instances and updating partitioning information in upstream splitters³. Our datastore implements an in-memory key-value store and supports the operations in Table 2. We reimplemented four NFs atop CHC. Table 4 shows their state objects, along with the state’s scope and access patterns.

NAT: maintains the dynamic list of available ports in the datastore. When a new connection arrives, it obtains an available port from the datastore (The datastore pops an entry from the list of available ports on behalf of the NF). It then updates: 1) per-connection port mapping (only once) and, 2) (every packet) L3/L4 packet counters.

Portscan detector [27]: detects infected port scanner hosts. It tracks new connection initiation for each host and whether it was successful or not. On each connection attempt, it updates the likelihood of a host being malicious, and blocks a host when the likelihood crosses a threshold.

Trojan detector: implementation here follows [12].

Load balancer: maintains the load on each backend server. Upon a new connection, it obtains the IP of the least loaded server from the datastore and increments load. It then updates: 1) connection-to-server mapping 2) per server #connections and, 3) (every packet) per server byte counter .

³based on statistics from vertex managers

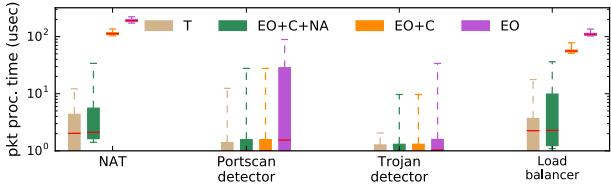


Figure 8: 5%ile, 25%ile, median, 75%ile and 95%ile pkt processing times. (T = Traditional NF, EO = Externalized state operations, C = with caching, NA = without waiting for the ACK)

7 Evaluation

We use two packet traces (Trace{1,2}) collected on the link between our institution and AWS EC2 for trace-driven evaluation of our prototype. Trace1 has 3.8M packets with 1.7K connections and Trace2 has 6.4M packets with 199K connections. The median packet sizes are 368B and 1434B. We conducted all experiments with both traces and found the results to be similar; for brevity, we only show results from Trace2. We use six CloudLab [2] servers each with 8-core Intel Xeon-D1548 CPUs and a dual-port 10G NIC. One port is used to forward traffic, and the other for datastore communication and control messages. To process at 10Gbps, each NF instance runs multiple processing threads. CHC performs scope-aware partitioning of input traffic between these threads. Our datastore runs on a dedicated server.

7.1 State Management Performance

Externalization: We study three models which reflect the state access optimizations discussed in (§4.3): #1) All state is externalized and non-blocking operations are used. #2) Further, NFs cache relevant state objects. #3) Further, NFs do not wait for ACKs of non-blocking operations to state objects; the framework is responsible for retransmission (§4.3). The state objects per NF that benefit from #2 and #3 can be inferred from Table 1 and Table 4; e.g., for NAT, per-connection port mapping is cached in #2, and the two packet counters benefit from non-blocking updates in #3. We compare these models with a “traditional” NF where all state is NF-local. We study each NF type in isolation first.

Figure 8 shows the per packet processing times. The median times for traditional NAT and load balancer are $2.07\mu s$ and $2.25\mu s$, respectively. In model #1, this increases by $190.67\mu s$ and $109.87\mu s$, respectively, with network RTT contributing to most of this (e.g., NAT needs three RTTs on average per packet: one for reading the port mapping and other two for updating the two counters). We don’t see a noticeable impact for scan and Trojan detectors (they don’t update state on every packet).

Relative to #1, caching (#2) further lowers median processing times by $111.98\mu s$ and $55.94\mu s$ for NAT⁴ and load balancer. For portscan and Trojan detector, reduces it by $0.54\mu s$ and $0.1\mu s$ (overhead becomes $+0.1\mu s$ as compared to

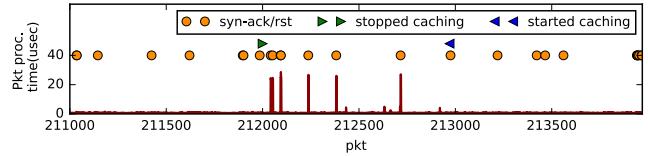


Figure 9: Per packet processing latency with cross-flow state caching

traditional NFs) as CHC caches the cross-flow state. Later, we evaluate the benefits of cross-flow caching in detail. Finally, #3 results in median packet processing times of $2.61\mu s$ for NAT (which now needs 0 RTTs on average) and $2.27\mu s$ for load balancer. These represent small overheads compared to traditional NFs: $+0.54\mu s$ for NAT, and $+0.02\mu s$ for the load balancer (at the median). Note that for portscan and Trojan detector the performance of #3 is comparable to #2 as they don’t have any blocking operations.

We constructed a simple chain consisting of one instance each of NAT, portscan detector and load balancer in sequence, and the Trojan detector operating off-path attached to the NAT. With model #3, the median end-to-end overhead was $11.3\mu s$ compared to using traditional NFs.

Operation offloading: We compare CHC’s operation of floating against a naive approach where an NF first reads state from the datastore, updates it, and then writes it back. We turn off caching optimizations. We now use two NAT instances updating shared state (available ports and counters). We find that the median packet processing latency of the naive approach is $2.17\times$ worse ($64.6\mu s$ vs $29.7\mu s$), because it not only requires 2 RTTs to update state (one for reading the state and the other for writing it back), but it may also have NFs wait to acquire locks. CHC’s aggregate throughput across the two instances is $>2\times$ better.

Cross-flow state caching: To show the performance of our cross-flow state caching schemes (Table 1; Col 5), we run the following experiment: we start with a single portscan detector. After it has processed around 212K packets, we add a second instance and split traffic such that for particular hosts, captured by the set \mathcal{H} , processing happens at both instances. At around 213K packets, we revert to using a single instance for all processing. Figure 9 shows the benefits of caching the shared state. At 212K packets, when the second instance is added, the upstream splitter signals the original instance to flush shared state corresponding to hosts $\in \mathcal{H}$ (Table 4). From this point on, both instances make blocking state update operations to update the likelihood of hosts $\in \mathcal{H}$ being malicious on every successful/unsuccessful connection initiation. Thus, we see an increase in per packet processing latency for every SYN-ACK/RST packet. At packet number 213K, all processing for \mathcal{H} happens at a single instance which can start caching state again. Thus, the processing latency for SYN-ACK/RST packets drops again, because now state update operations are applied locally and updates are flushed in a non-blocking fashion to the store.

Throughput: We measure degradation in per NF throughput for models #1 and #3 above compared to traditional NFs.

⁴NAT needs 2 RTTs to update counters as port mapping is cached.

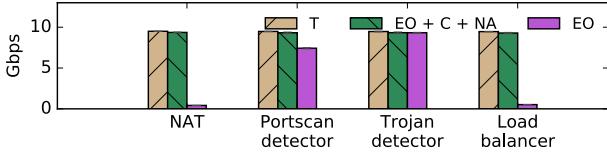


Figure 10: Per instance throughput. (T = Traditional NF, EO = Externalized state operations, C = with caching, NA = without waiting for the ACK)

Figure 10 shows that the max. per NF throughput for traditional NFs is around 9.5Gbps. Under model #1, load balancer and NAT throughput drops to 0.5Gbps. The former needs to update a byte counter (which takes 1 RTT) on every packet; likewise, the NAT needs three RTTs per packet. The port scan and Trojan detectors do not experience throughput degradation because they don't update state on every packet. Model #3 increases throughput to 9.43Gbps, matching traditional load balancer and NAT. We repeated our experiment with the aforementioned single-instance NF chain and observed similar maximal performance (9.25Gbps with both CHC and traditional NFs) in Model #3.

Datastore performance We benchmarked the datastore using the workload imposed by our state operations. We used 128bits key and 64bits value to benchmark the datastore. The datastore was running four threads. Each thread handled 100k unique entries. As discussed in §4.3, state is not shared between these threads. We found that a single instance of our datastore supports $\sim 5.1\text{M}$ ops/s (increment at 5.1M ops/s, get at 5.2M ops/s, set at 5.1M ops/s; Table 2). The datastore can be easily scaled to support a greater rate of operations by simply adding multiple instances; each state object is stored at exactly one store node and hence no cross-store node coordination is needed.

7.2 Metadata Overhead

Clocks: The root writes packet clocks to the datastore for fault tolerance. This adds a $29\mu\text{s}$ latency per packet (dominated by RTT). We optimize further by writing the clock to the store after every n^{th} packet.⁵ The average overhead per packet reduces to $3.5\mu\text{s}$ and $0.4\mu\text{s}$ for $n = 10, 100$.

Packet logging: We evaluated two models of logging: 1) locally at the root, 2) in the datastore. The former offers better performance, adding $1\mu\text{s}$ latency per packet, whereas the latter adds $34.2\mu\text{s}$ but is more fault tolerant (for simultaneous root and NF failures). We also studied the overhead imposed by the framework logging clocks and operations at NFs, the datastore logging clocks and state, and the XOR-ing of identifiers (§5.4); the performance impact for our chain (latency and throughput overhead) was negligible (< 1%).

XOR check and delete request: (§5.4) XOR checks of bit vectors are performed asynchronously in the background and do not introduce any latency overhead. However, ensur-

⁵After a crash, this may lead the root to assign to a packet an already assigned clock value. To overcome this issue, the root starts with $n + \text{last update}$ so that clock values assigned to packets represent their arrival order.

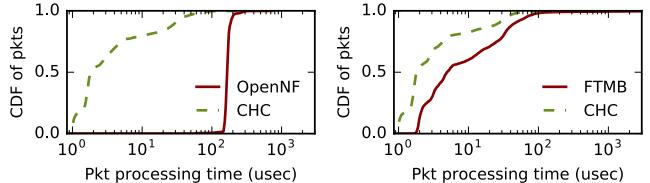


Figure 11: State sharing.

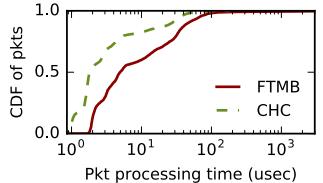


Figure 12: Fault recovery.

ing the successful delivery of “delete” request to root before forwarding the packet introduces a median latency overhead of $7.9\mu\text{s}$. Asynchronous “delete” request operation eliminates this overhead but failure of the last NF in a chain may result in duplicate packets at the receiver end host.

7.3 Correctness Requirements: R1–R6

R1: State availability: Using our NAT, we compare FTMB’s [29] checkpointing approach with CHC writing all state to a store. We could not obtain access to FTMB’s code; thus, we emulate its checkpointing overhead using a queuing delay of $5000\mu\text{s}$ after every 200ms (from Figure 6 in [29]). Figure 12 (with 50% load level) shows that checkpointing in FTMB has a significant impact: the 75th%-ile latency is $25.5\mu\text{s}$ – which is **6X** worse than that under CHC (median is **2.7X** worse). FTMB’s checkpointing causes incoming packets to be buffered. Because of externalization in CHC, there is no need for such checkpointing. Also, FTMB does not support recovery of the packet logger [29]. CHC intrinsically supports this (§5.4), and we evaluate it in §7.3.

R2: Cross-instance state transfers: We elastically scale up NAT as follows: we replay our trace for 30s through a single instance; midway through replay, we reallocate 4000 flows to a new instance, forcing a move of the state corresponding to these flows. We compare CHC with OpenNF’s loss-free move; recall that CHC provides both loss-freeness and order preservation. CHC’s move operation takes **97% or 35X** less time (0.071ms vs 2.5ms), because, unlike OpenNF, CHC does not need to transfer state. It notifies the datastore manager to update the relevant instance_IDs. However, when instances are caching state, they are required to flush cached state operations before updating instance_IDs. Even then, CHC is 89% better because it *flushes only operations*.

R3: Cross-instance state sharing: We compare CHC against OpenNF w.r.t. the performance of strongly consistent shared state updates across NAT instances, i.e., updates are serialized according to some global order. Figure 11 (with 50% load level) shows that CHC’s median per-packet latency is 99% lower than OpenNF’s (1.8 μs vs 0.166ms). The OpenNF controller receives all packets from NFs; each is forwarded to every instance; the next packet is released only after all instances ACK. CHC’s store simply serializes all instances’ offloaded operations.

R4: Chain-wide ordering: We revisit the chain in Figure 2. Each scrubber instance processes either FTP, SSH, or

	30%load	50%load
Duplicate packets	13768	34351
Duplicate state updates	233	545

Table 5: Duplicate packet and state update at the downstream portscan detector without duplicate suppression.

IRC flows. To measure the accuracy of the Trojan detector, we added the signature of a Trojan at 11 different points in our trace. We use three different workloads with varying upstream NF processing speed: W1) One of the upstream NFs adds a random delay between 50-100 μ s to each packet. W2) Two of the upstreams add the random delay. W3) All three add random delays. We observed that CHC’s use of chain-wide logical clocks helps the Trojan detector identify **all 11 signatures**. We compare against OpenNF which does not offer any chain-wide guarantees; we find that **OpenNF misses 7, 10, and 11 signatures** across W1-W3.

R5: Duplicate suppression: Here, we emulated a straggler NAT by adding a random per packet delay between between 3-10 μ s. A portscan detector is immediately downstream from the NAT. CHC launches a clone NAT instance according to §5.3. We vary the input traffic load. Table 5 shows the number of duplicate packets generated by the NAT instances under different loads, as well as the number of duplicate state updates at the portscan detector – which happen whenever a duplicate packet triggers the scan detector to *spuriously* log a connection setup/teardown attempt. Duplicate updates create both false positives/negatives and their incidence worsens with load. No existing framework can detect such duplicate updates; CHC simply suppresses them.

R6: Fault Tolerance: We study CHC failure recovery.

NF Failure: We fail a single NAT instance and measure the recovery and per packet processing times. Our NAT performs non-blocking updates without waiting for the framework ACK; here, we use the 32bit vector (§5.4) to enable recovery of packets whose non-blocked operations are not yet committed to the store. To focus on CHC’s state recovery, we assume the failover container is launched immediately. Figure 13 shows the average processing time of packets that arrive at the new instance at two different loads. The average is calculated over 500 μ s windows. Latency during recovery spikes to over 4ms, but it only takes **4.5ms** and **5.6ms** at 30% and 50% loads, respectively, for it to return to normal.

Root failure: Recovering a root requires just reading the last updated logical clock from the datastore and flow mapping from downstream NFs. This takes < 41.2 μ s.

Datastore instance failure: Recovering a datastore instance failure requires reading per-flow state from NFs using it, and replaying update operations to rebuild shared state. Reading the latest values of per-flow state is fast. Recovering shared state however is more time-consuming. Figure 14 shows the time to rebuild shared state with 5 and 10 NAT instances updating the same state objects at a single store instance. We replayed the state update operation logs generated by these instances. The instances were processing

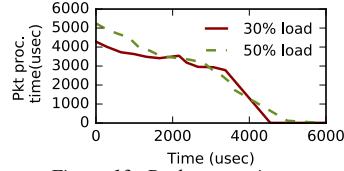


Figure 13: Packet proc. time.

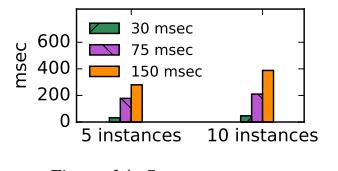


Figure 14: Store recovery.

9.4Gbps of traffic; periodic checkpoints occurred at intervals of 30ms, 75ms, and 150ms. The recovery time is $\leq 388.2\text{ms}$ for 10 NATs with checkpoints at 150ms intervals. In other words, a storage instance can be quickly recovered.

8 Conclusion

We presented a ground-up NFV framework called CHC to support COE and high performance for NFV chains. CHC relies on managing state external to NFs, but couples that with several caching and state update algorithms to ensure low latency and high throughput. In addition, it leverages simple metadata to ensure various correctness properties are maintained even under traffic reallocation, NF failures, as well as failures of key CHC framework components.

Acknowledgements We thank the reviewers and our shepherd Katerina Argyraki. This research is supported by NSF grants CNS-1302041 and CNS-1717039. Aditya is supported by a H. I. Romnes Faculty Fellowship, and gifts from Google and Huawei.

References

- [1] Cisco Network Service Header: draft-quinn-sfc-nsh-03.txt. <https://tools.ietf.org/html/draft-quinn-sfc-nsh-03>.
- [2] Cloud lab. <http://cloudlab.us/>.
- [3] LXC - Linux containers . <https://linuxcontainers.org/lxc/introduction/>.
- [4] Network functions virtualisation – update white paper. https://portal.etsi.org/nfv/nfv_white_paper2.pdf.
- [5] Network functions virtualisation: Introductory white paper. http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf.
- [6] NFV Management and Orchestration: An Overview. <https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf>.
- [7] Protobuf-c. <https://github.com/protobuf-c/protobuf-c>.

- [8] A. Anand, V. Sekar, and A. Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.
- [9] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS ’12*, pages 49–60, 2012.
- [10] M. Boucadair, C. Jacquet, R. Parker, D. Lopez, P. Yegani, J. Guichard, and P. Quinn. Differentiated Network-Located Function Chaining Framework. Internet-Draft [draft-boucadair-network-function-chaining-02](#), IETF Secretariat, July 2013.
- [11] A. Bremer-Barr, Y. Harchol, and D. Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 511–524. ACM, 2016.
- [12] L. De Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.
- [13] Intel. Data Plane Development Kit. <http://dpdk.org/>.
- [14] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 543–546, 2014.
- [15] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical report, Technical Report, 2013.
- [16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM ’14*, pages 163–174. ACM, 2014.
- [17] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, 2017.
- [18] J. Khalid and A. Akella. Correctness and performance for stateful chained network functions. <https://arxiv.org/abs/1612.01497>, 2018.
- [19] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.
- [20] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.
- [21] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7, SSYM’98*, pages 3–3, 1998.
- [22] PRADS. <https://gamelinux.github.io/prads/>.
- [23] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM ’15*, pages 29–42. ACM, 2015.
- [24] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM ’13*, pages 27–38. ACM, 2013.
- [25] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 1. ACM, 2013.
- [26] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.
- [27] S. Schechter, J. Jung, and A. Berger. Fast detection of scanning worm infections. In *Recent Advances in Intrusion Detection*, pages 59–81. Springer, 2004.
- [28] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI’12*, pages 24–24, 2012.

- [29] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, pages 227–240. ACM, 2015.
- [30] Squid. <http://www.squid-cache.org/>.
- [31] Mellanox. Messaging Accelerator (VMA). http://www.mellanox.com/page/software_vma.
- [32] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI’18*, 2018.

Performance Contracts for Software Network Functions

Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli,
Katerina Argyraki, and George Canea

EPFL, Switzerland

Abstract

Software network functions (NFs), or middleboxes, promise flexibility and easy deployment of network services, but face the serious challenge of unexpected performance behaviour. We propose the notion of a *performance contract*, a construct formulated in terms of *performance critical variables*, that provides a precise description of NF performance. Performance contracts enable fine-grained prediction and scrutiny of NF performance for arbitrary workloads, without having to run the NF itself.

We describe BOLT, a technique and tool for computing such performance contracts for the entire software stack of NFs written in C, including the core NF logic, DPDK packet processing framework, and NIC driver. BOLT takes as input the NF implementation code and outputs the corresponding contract. Under the covers, it combines pre-analysis of a library of stateful NF data structures with automated symbolic execution of the NF’s code. We evaluate BOLT on four NFs—a Maglev-like load balancer, a NAT, an LPM router, and a MAC bridge—and show that its performance contracts predict the dynamic instruction count and memory access count with a maximum gap of 7% between the real execution and the conservatively predicted upper bound. With further engineering, this gap can be reduced.

1 Introduction

The goal of our work is to enable network operators and developers to predict and scrutinise the performance of software network functions without having to run them. A network function (NF) performs packet processing inside the network, such as packet forwarding, load balancing, or network address translation (NAT). NF development has been moving away from custom hardware toward software running on commodity hardware. This change increases flexibility and reduces development costs and time-to-market [22, 37, 38], but arguably makes it harder to predict the NF’s performance. Unexpected NF performance behaviour makes it

harder for network operators to provision their networks and exposes a new attack surface for adversaries seeking to degrade network performance.

We propose the construct of a *performance contract* for NFs. A contract $C_N^U(i)$ answers the question of what the performance of the NF N is like when processing packets from an arbitrary input packet class i , with performance measured in units of U . To illustrate, N could be a particular implementation of a router, U — the number of x86 instructions it executes per packet, and i_1 (respectively i_2) — the class of valid (respectively invalid) packets arriving at the router. The contract predicts performance in terms of human readable expressions. These expressions are functions of what we call *performance critical variables* (PCVs), which summarise the impact of input history and configuration on the given NF’s state and execution. In our example, the contract could return functions $p_1(l)$ (respectively $p_2(l)$) for valid (respectively invalid) packets, where l is the length of the IP prefix that matches the input packet’s destination IP address. l is a PCV. For a NAT, a PCV could be the occupancy rate of the NAT’s flow table. In this paper, we consider three performance metrics: number of executed instructions, number of memory accesses, and number of execution cycles. In general, we consider an NF implementation to be the software stack plus the hardware architecture it runs on.

Our work draws upon ideas from earlier work on analysing/predicting performance and worst-case execution time (WCET), either of software in general [26, 24, 45] or of NFs in particular [15, 32]. The way performance contracts differ from classic performance prediction and WCET analysis is that, rather than producing a performance number, they express performance as a function of critical parameters—the PCVs. This enables contracts to expose the entire range of values of the NF’s performance, not just a worst-case bound, as well as explain how these values relate to different workloads. Performance contracts also strike a favorable balance between accuracy, utility, and human legibility.

We present BOLT, a technique and tool that analyses NF code, without actually running it, to generate NF perfor-

mance contracts. We draw inspiration from Vigor [47], a technique for verifying that NFs written in C satisfy semantic properties and are memory-safe. Vigor assumes a clear separation of NF code into (a) a library of commonly used NF data structures, which is written and formally verified by experts, and (b) stateless NF logic that uses the library, is written by NF developers, and is verified automatically by Vigor using symbolic execution. In the same spirit, BOLT starts from manually pre-computed performance contracts for basic common data structures as a base case, and then automatically generates performance contracts for the NF code that uses these data structures. Contracts can be computed recursively for chains of NFs as well.

To help operators and developers handle the PCVs in contracts, BOLT comes with a tool we call the Distiller, which takes as input a packet trace and computes the PCV values that result from that trace being processed by the target NF.

We evaluate the accuracy and utility of BOLT-generated performance contracts for four NFs written in C using DPDK: a NAT, a Maglev[17]-like load balancer, an LPM router, and a MAC bridge. The contracts for these NFs predict the dynamic instruction count and memory access count with a maximum gap of 7% between real executions and the conservatively predicted upper bound. We explain the origin of this gap and argue that it can be reduced to 0 with further engineering. BOLT also generates contracts for the number of execution cycles, in which case it relies on a hardware model; the result is as accurate as the model allows. We close our evaluation with example use cases where BOLT uncovers performance issues and helps understand how to fix them.

In summary, we make two contributions in this paper:

- We propose the concept of a performance contract for NFs, which expresses NF performance as a function of performance critical variables.
- We demonstrate, using our BOLT prototype, that it is possible to compute performance contracts that are accurate, useful, and human-legible.

The BOLT source code is available as open source [1].

In the rest of the paper we define the performance contract construct (§2), then describe how BOLT generates performance contracts (§3) and how the BOLT Distiller helps obtain a concrete performance number from a performance contract given a particular packet trace (§4). Finally, we present BOLT’s evaluation (§5), discuss its limitations (§6), present related work (§7), and conclude (§8).

2 Performance Contracts

In this section, we define the performance contract construct, and we use a running example to illustrate this definition.

2.1 Running Example: LPM Router

Algorithm 1 shows pseudocode for a simplified longest prefix match (LPM) IPv4 router that stores the forwarding table in a Patricia trie. The router first classifies packets based on whether they are IPv4 or not (line 2). Invalid packets are immediately dropped (line 6), thus incurring a constant performance cost. Valid packets lead to a lookup in the LPM data structure (line 3), which has a more complex performance profile (lines 10–17), with the number of loop iterations being data-dependent (see lines 12 and 15).

Algorithm 1: Simple LPM Router

```

1 function processPacket (packet pkt)
2   if pkt.etherType == IPv4 then
3     | dst_port = lpmGet (pkt.ipv4.dst_addr)
4     | FORWARD (pkt, dst_port)
5   else
6     | DROP (pkt)
7   end
8 function lpmGet (bit ip[32])
9   node = lpmRoot
10  for i in 0..31 do
11    | b = ip[i]
12    | if exists node.children[b] then
13      |   | node = node.children[b]
14    | else
15    |   | break
16    | end
17  end
18  return node.port

```

For clarity of exposition, the running example assumes that the packet processing framework and every layer below has zero impact on performance.

2.2 Definition

A performance contract describes the performance of NF software running on a particular hardware configuration.

Contract $C_N^U : I \rightarrow F$ is a map from input classes to functions, i.e., C_N^U maps input class $i \in I$ to a function $p_i(v_1, v_2, \dots) \in F$. Input class i is a specification that describes which inputs (e.g., packets) belong to that class, such as a symbolic expression for “all valid IPv4 packets without IP options.” The contract’s domain I spans the entire input space of the program N . Function p_i expresses the performance exhibited by N when processing an arbitrary input that belongs to class i . p_i is a function of performance-critical variables v_1, v_2, \dots , and its value is measured in units of U . p_i can be as simple as a constant function.

In general, a performance contract can be formulated for any program/procedure P , not just an NF. The performance

of P is determined by its input and its state at the time of processing the input. P 's state, in turn, is determined by the inputs it has processed in the past, its configuration parameters, and its environment.

Performance-critical variables (PCVs) capture the influence on performance of anything other than P 's input.

Performance is expressed through metrics, such as number of memory accesses or number of execution cycles. Performance contracts are metric-specific. An essential property of a performance contract is that, for any real execution that satisfies the contract's assumptions (configuration, input history, etc.), the measured performance is guaranteed to be no more than the metric value predicted by the contract.

A performance contract for P is a recursive composition of the performance contracts for its constitutive parts. P as used above could be a chain of NFs, one individual NF, a part of an NF, or—in the base case—a data structure method whose contract has been derived manually by an expert.

Table 1 shows two performance contracts for our example LPM router, corresponding to two performance metrics: instruction count and memory accesses. There are two input classes that emerge naturally from this NF's code structure: invalid and valid packets. To process any packet in the first class, the NF executes 2 instructions and 1 memory access. For the second class, it executes $4 \cdot l + 5$ instructions and $l + 3$ memory accesses, where l is the matched prefix length. This example ignores all layers below the NF code, so the matched prefix length l fully captures how anything other than the input packet (in particular, the configuration of the LPM table) influences performance. Hence it is the sole PCV used by both contracts.

Input Class	Instructions	Memory Accesses
Invalid packets	2	1
Valid packets	$4 \cdot l + 5$	$l + 3$

Table 1: Two stylised performance contracts for an example LPM router. PCV l is the matched prefix length.

2.3 Rationale Behind PCVs in Contracts

Designing a performance contract involves a trade-off between the level of detail that the contract exposes about the code and the contract's legibility (i.e., how easy it is for a human to parse and draw useful conclusions from).

Different users may need different balances between detail and legibility. For instance, an NF developer who uses performance contracts to debug and optimise the performance of their own code will likely want more detail than a network operator who has no access to NF implementations and uses performance contracts solely to provision their network.

We chose to express performance as a function of PCVs, because this enables one to navigate fluidly this trade-

off, i.e., to generate contracts that achieve different detail/legibility balance. For example, consider a hypothetical NF whose only operation is to update, for every observed packet, per-flow state stored in a hash map. The performance of this NF is determined, in a straightforward manner, by the collision rate of the hash map, which is itself determined, in a complicated manner, by the workload and hash map configuration. So, one can express the performance of this NF as a complicated function of workload and configuration, or as a simple function of the hash map's collision rate. The former arguably provides all the detail that any user of a performance contract would ever care for, but may be too much for a human to digest and draw conclusions from; the latter hides some of this detail but still provides insight into what determines NF performance and how, just at a different level. So, it is possible to generate performance contracts that favor detail or legibility by choosing the proper level of PCVs.

We are concerned about exposing, in a performance contract, implementation-specific notions like collision rates or matched prefix lengths. This may be fine for the developer who chose or implemented the NF's data structures, but awkward for an operator who knows nothing about the NF's implementation. Still, we do not think that it is possible to design meaningful performance contracts that do not leak non-trivial information about implementation. In the end, when a network operator is debugging an unexpectedly slow network device, they do end up digging into the device's implementation and trying to understand how that interacts with the given workload. A performance contract that distills how implementation affects performance into a simple expression would arguably be welcome in such cases.

If desired, operators and developers can bind the PCVs in the performance expressions to values chosen by themselves or by the BOLT Distiller. The latter, given a packet trace, computes the concrete values of the PCVs at the end of the NF's processing of that trace.

3 Generating Performance Contracts

In this section, we describe BOLT, a technique and tool that generates performance contracts. Our current prototype works with three performance metrics: number of executed instructions, memory accesses, and execution cycles.

We first provide background on the techniques BOLT employs (§3.1), then describe how to obtain contracts for data structures (§3.2), entire NFs (§3.3), and chains of NFs (§3.4). We close with a few implementation details (§3.5).

3.1 Background

The conceptually simplest way to explore all possible behaviours of a program is to execute it with every possible input. As this does not typically scale to real-world programs, a more efficient approach is to group inputs in non-

overlapping classes, such that all inputs in the same class follow the same execution path through the program (hence induce the same behaviour); then we can explore induced behaviour once per input class. For example, a program that takes as input a 64-bit number and takes one of two possible actions depending on whether the number is positive or negative has 2^{64} possible inputs but only 2 input classes that induce different behaviours.

Symbolic execution (SE) [8, 21] is a commonly used technique for exploring feasible execution paths of a program and identifying the input class that triggers each one. SE relies on a special program interpreter called a symbolic execution engine (SEE), which uses *symbols* to represent inputs and propagates these symbols through the program. For instance, if a program takes as input an integer x , the engine associates with x a symbol α ; if the program assigns to a variable y the value $x + 1$, the engine associates with y the expression $\alpha + 1$. If the program branches on a symbolic value, the engine explores both paths, and keeps track of the *constraints* that led down each path, such as $\alpha < 0$. The engine uses a constraint solver [13, 20] to ensure that it explores only feasible paths and to identify the input class that triggers each one. However, it can typically not identify *all* the feasible paths of a program due to path explosion [6]: the number of paths is generally exponential in the number of branches in the code, and symbolic pointers make things worse, as the engine sometimes needs to concretise them, i.e., fork a new path for each possible address that a symbolic pointer may reference.

Vigor [47] leverages SE to verify semantic properties and memory safety of a stateful NF. It assumes a clear separation of NF code into: a library of common NF data structures that is written and verified by experts; and stateless NF code that uses the library and is written by NF developers. The experts that verify the library produce a semantic contract for each library method, which specifies pre- and post-conditions for the method; this is a tedious process that requires time and expertise, but it needs to be done only once per library method, hence its cost is amortised when multiple NFs use the library.

Vigor uses SE to automatically explore all the feasible paths through the stateless NF code. The Vigor toolchain includes an SEE tailored to the domain of NFs so that SE of the stateless NF code does not suffer from path explosion. Vigor automatically combines this analysis with the semantic contracts for the library methods used by the NF, and generates a proof that the NF as a whole satisfies the target semantic properties and is memory-safe. Note that Vigor’s semantic contracts are unrelated to our performance contracts and do not provide any performance-related information.

BOLT reuses Vigor’s toolchain and adopts a similar NF development process: A team of experts writes the library of common NF data structures and their performance contracts and symbolic models. NF developers write stateless NF code

Input Class	Instructions	Memory Accesses
Unconstrained	$4 \cdot l + 2$	$l + 1$

Table 2: Performance contract for `lpmGet`.
PCV l is the matched prefix length.

that uses this library, and use BOLT to generate performance contracts for the NF.

3.2 Base Case: Contracts for Data Structures

The first step is to manually generate performance contracts for the parts of the code that BOLT does not analyse automatically; for our current prototype, these are all the methods for accessing data structures that keep NF state. In the same spirit as Vigor, we rely on a library of common data structures that are analysed once and then reused across multiple NFs. Table 2 shows manually generated performance contracts for the `lpmGet` method used by our LPM router. Like the performance contracts for the entire router, the ones for its data structure express performance as a function of the matched prefix l , which is the only PCV.

Part of this process—perhaps the hardest one—is picking a set of PCVs so as to achieve a target balance between precision and legibility. For example, the `lpmGet` method uses pointer arithmetic (line 12), which the compiler unfolds into a series of conditional jumps; as a result, the performance of the method varies slightly, depending on whether each bit in the matched IP prefix is 0 or 1. One option is to expose each bit in the matched IP prefix as a PCV, in which case the contract precisely predicts the performance of any real executions. Another option is to assume that each bit has the value that results in the worst-case performance (essentially coalesce multiple execution paths into the one among them with the worst performance) and expose only the length of the matched prefix as a PCV; in this case, the contract predicts performance conservatively, i.e., overestimates the number of execution cycles and memory accesses. This is an example of how a higher-level PCV sacrifices a small amount of precision for a more concise, hence legible contract.

3.3 Contracts for NFs

BOLT (Algorithm 2) takes as input the NF code (line 1) and generates a special build where all calls to stateful methods are replaced at link time with calls to corresponding symbolic models (line 2). For example, in our LPM router, the call to `lpmGet` is replaced with a call to the symbolic model shown as Algorithm 3. Next, BOLT symbolically executes this special build exhaustively and obtains all feasible execution paths through the stateless NF code (line 3). For our LPM router, this results in 2 paths, one for valid IPv4 packets and one for invalid packets. For each execution path,

Algorithm 2: GetPerformanceContract

```
1 function GetPerformanceContract
  Input : function Fn,
    <opt> map <function, model> Models
    <opt> map <function, perfContract> Contracts
  Output: perfContract Perf_Contract
2 stubbFn := SubstituteModels(Fn, Models)
3 paths := GetAllPaths(stubbFn)
4 Perf_Contract :=  $\emptyset$ 
5 foreach path in paths do
6   inputs := GetInputsForPath(path)
7   traceInstr := GetInstrTrace(stubbFn, inputs)
8   perf :=  $\emptyset$ 
9   foreach instr in traceInstr do
10     if instr is a call to a stateful function fn then
11       perf+= Perf(Contracts[fn],
12                     path.constraints)
13     else
14       perf+= Perf(instr)
15     end
16   end
17   Perf_Contract.append(path, perf)
18 end
19 return Perf_Contract
```

Algorithm 3: lpmGet function model.

```
1 function lpmGet ( bit ip[32] );
2 return <new symbol>
```

BOLT also obtains *symbolic path constraints*, which consist of two categories of constraints: (1) constraints on NF inputs that cause it to go down the particular execution path and (2) constraints on the abstract state of each data structure, before and after each call to a stateful method. The second category of constraints tells BOLT how stateless and stateful code interact along the execution path.

Once it has obtained all feasible execution paths and their path constraints, BOLT analyses each path: First, it passes the path's constraints to a solver to obtain concrete inputs that exercise the path (line 6); these inputs include a packet, as well as values for any symbols generated by the symbolic models of the stateful methods. For our LPM router, one path will yield a concrete invalid IPv4 packet, while the other will yield a concrete valid IPv4 packet and a concrete port that would result from the LPM lookup (e.g., port 0). Next, for each of these concrete inputs, BOLT replays the NF execution and obtains a unique trace of machine instructions (line 7).

Finally, BOLT characterises the performance of each feasible execution path by stepping through the corresponding instruction trace: it traverses the trace, adding up the cost of

each instruction (line 13), until it hits a call to a modelled method; when this occurs, it picks the right branch of the method's performance contract based on the constraints on the abstract state of the data structure (line 11). In the case of *lpmGet*, the performance contract has no branches. This will typically not be the case for more complex data structures and methods, e.g., the performance contract of a flow table *get* method will have different formulae depending on whether the flow is present or absent in the flow table. In such a scenario, BOLT uses the path constraints to pick the right formula.

3.4 Contracts for NF chains

We summarise how BOLT can be extended to generate contracts for chains of NFs. This can be useful in scenarios where one NF's worst-case performance is masked by another NF on the same chain. For example, consider the scenario where, in front of our LPM router, an operator deploys a firewall that drops all packets matching prefixes that exceed a given length. In this scenario, we will get a more accurate performance prediction by using a performance contract generated for the NF chain as a whole, than by using two separate contracts generated for each NF and adding their predictions.

We can extend BOLT for joint analysis of multiple chained NFs as follows: First, generate a performance contract for each individual NF as before. Next, pair together execution paths from two connected NFs; for each such path pair, AND together their respective path constraints and add equality constraints connecting the symbolic expression for the packet sent by the first NF to the symbol representing the packet received by the second NF. Next, use a solver to check if the paths are compatible. Finally, generate a global performance contract for the NF pair that sums up the performance for each compatible path pair, while ignoring incompatible ones.

For longer chains, rather than fully enumerating the entire combinatorial explosion of all path tuples, BOLT could piece together compatible paths across the chained NFs one at a time in sequence, following a procedure similar to joint symbolic execution [31]. This process further generalises to more complex networks, so long as the topology forms a directed acyclic graph (DAG).

3.5 Implementation Details

Instruction Replay. While replaying each execution path, we use an instruction tracer based on Intel's pin dynamic binary instrumentation tool [25] to log the x86 instructions along with memory locations touched along that path. During replay, we ensure that despite the difference between the analysed code (linked against models) and production code (linked against actual data structures), BOLT remains

conservative. This is done by compiling the stateless NF code separately from the models and disabling any link-time-optimisations when linking them together. While this leads to slight over-estimations, it ensures that BOLT never under-approximates performance.

Hardware model employed. For metrics that rely on the underlying hardware (e.g., cycles), the BOLT prototype employs a simple, conservative hardware model that does not model CPU components that are either too complex or constitute trade secrets. For compute instructions, BOLT conservatively assumes the worst case performance cost of each instruction as reported in the Intel manual [2] due to the proprietary nature of out-of-order (OoO) instruction scheduling within the processor. For memory instructions, we only model the private L1 Data Caches. We do not model proprietary features such as the slice selection algorithm in the L3 cache, memory-level parallelism (MLP), or prefetching. Consequently, BOLT conservatively assumes that every memory access is serviced from main memory unless it can definitively prove otherwise (by tracking spatial and temporal locality of memory accesses in the L1D cache).

These proprietary features significantly improve NF performance, given that an NF repeatedly performs the same tasks (e.g., flow expiration, hash-ring traversal) in a tight loop. Consequently, our hardware model causes BOLT to over-estimate performance and in our experimental evaluation §5, we find that BOLT comes within $4\times$ for typical workloads and $9\times$ for pathological workloads. However, we show that it should be possible to improve the accuracy of BOLT’s contracts by plugging in a better hardware model.

Including DPDK and NIC driver code. BOLT allows analysis of the NF code at two different levels of abstraction: 1) Only the NF code sitting atop DPDK or 2) the entire software stack including the NF logic, DPDK and the NIC driver.

Doing this at the level of just the NF is relatively simple. We stub out the DPDK send and receive calls and replace them with models that inject the packet symbols. We then filter the stateless instruction traces to include only the instructions between these two calls.

Building on recent work [34] that applied Vigor to the entire NF software stack, BOLT can also include DPDK and the NIC drivers in its performance predictions. The insight behind this work is that while such frameworks as a whole may be complex, simple NFs require only a small subset that primarily reads and writes to device registers. This subset has simple control flow and so can be symbexed along with the stateless NF code. In this case, we include in the trace instructions from the beginning of the driver receive function until the end of the corresponding send/drop function.

4 The BOLT Distiller

Once BOLT has built a performance contract, users can predict the performance of the NF under varying assumptions. However, performance contracts can have several hundred to a few thousand execution paths each, with their own unique assumptions. Often, it is not obvious which assumptions are reasonable or typical in the real world. To reason about this, BOLT provides an additional tool called the *Distiller*.

The Distiller takes as input the NF code and a sample of real-world traffic (as PCAP files). It feeds the traffic through the NF code and logs the values that are induced in each model parameter. For our running example, this would mean linking the stateless code with a slightly modified version of the data structure that traces the number of loop iterations that occur, logging the matched prefix length. With these traces, the Distiller computes a detailed breakdown of which assumptions hold for each packet, and how that relates to predicted performance. Note, the distiller does not affect the generated performance contract in any way; it merely tells the user which assumptions held for each packet in the given trace allowing the user to then extrapolate and identify execution paths of interest in the performance contract.

The distiller also enables users to perform a sensitivity analysis. For our example LPM, the user could, for instance, see that most packets match prefixes that are 16 to 24 bits long. Longer prefixes lead to 32% worse performance (133 vs 101 instructions) but may (hypothetically) account for only 1% of traffic.

An operator can leverage the Distiller to balance risk with resource utilisation to decide how to provision the network. A developer can understand how any assumptions that they have made regarding which scenarios are more common may be wrong, guiding further optimisation efforts. We illustrate further, the utility of the distiller in §5.2 and §5.3.

5 Evaluation

We now examine whether BOLT works, i.e., whether it produces correct performance contracts for software NFs (§5.1), and illustrate, through example use cases, how BOLT can help network operators (§5.2) and NF developers (§5.3).

5.1 Does BOLT work?

We experiment with a MAC bridge (Br), an LPM router implemented with DPDK’s LPM data structure [3] (LPM), the NAT from [47] (NAT), and a Maglev-like [17] Load Balancer (LB).

Testbed. We use two directly connected servers: a device under test (DUT) and a traffic generator and sink (TG). Both servers have Intel Xeon E5-2667v2 3.3GHz CPUs with 32GB of RAM; they are connected over Intel 82599ES 10Gb

NICs. The DUT runs one of the NFs, while the TG uses MoonGen [18] to replay a PCAP file, one packet at a time, to avoid any queuing or pipelining effects.

Methodology. First, Bolt generates performance contracts for each of the 4 NFs for three different metrics — number of executed instructions, number of memory accesses and number of execution cycles. Each such contract subsumes from several hundred to a few thousand unique execution paths.

We compare the performance predicted by each contract for various input packet classes to actual measurements. We use broad input packet classes (e.g., “unconstrained traffic” or “broadcast traffic”), each of them covering thousands to millions of possible packets and exercising hundreds to thousands of execution paths in the NFs. Given these broad input classes, BOLT being conservative reports the predicted performance value of the execution path with the worst predicted performance. Said differently, BOLT predicts the worst performance that an input packet from this class could encounter.

For most packet classes, we programmatically produced a PCAP file that samples the packet class, i.e., contains a large number of packets from that packet class, and obtained measurements from our testbed. For one specific packet class (discussed below), we could not produce a representative PCAP file; in this case, to obtain ground truth, we modified the NF to synthesise the expected state (so that it did not need to be built from actual packet history). For each NF and input packet class, we measure the performance metrics predicted by BOLT: instruction count (IC), number of memory accesses (MA), and latency (cycles); IC and MA are measured using the binary instrumentation described in §3.5, while latency is measured using high precision CPU clocks (TSC) during separate non-instrumented runs.

Input packet classes. For each NF, we first consider unconstrained traffic (scenarios Br1, LPM1, NAT1, and LB1). Given this input packet class, the performance contract generated by BOLT predicts the absolute worst-case performance of the NF. For Br, NAT, and LB, which maintain and expire per-flow state, BOLT determines that the worst-case performance happens when the NF’s MAC/flow table is full, all the entries have collided with each other, and all the entries are sufficiently aged so as to induce a mass expiry event that completely clears the table when the current input packet arrives. We were unable to produce a PCAP file that led to this pathological scenario, but still wanted to verify that, if this scenario did occur, the NF’s performance would indeed be the one predicted by BOLT. This is why we modified the NF to synthesise the necessary state (as stated above). To generate unconstrained traffic for the LPM, we used the CASTAN framework [32] which specialises in generating adversarial workloads for NFs.

For the NFs that maintain per-flow state, we also consider

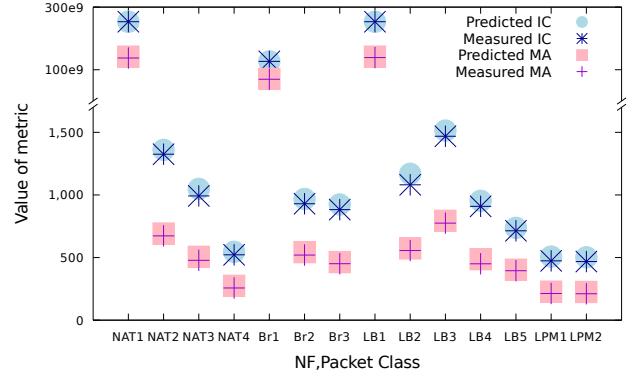


Figure 1: Accuracy of Performance contracts for multiple NFs and packet classes in terms of “Instruction Count”(IC) and “Number of Memory Accesses”(MA).

a few representative classes of input packets that do not encounter hash collisions or entry expirations. For the Bridge: broadcast (Br2) and unicast (Br3) packets. For the NAT: packets arriving from the internal network that belong to new (NAT2) and established (NAT3) connections, and packets arriving from the external network that do not belong to an established connection and are dropped (NAT4). For the Load Balancer: packets arriving from the external network that belong to new flows (LB2), existing flows with unresponsive (LB3) and live (LB4) backend servers and heartbeat packets from backend servers (LB5).

The LPM uses DPDK’s two-tiered lookup table, which is structured such that any packet with a matched prefix of ≤ 24 bits incurs exactly one lookup and all other packets incur exactly two lookups. Hence, any input packet class where the packets are constrained to matched prefixes of > 24 bits can incur the same performance as unconstrained traffic. In addition, we consider input packets that are constrained to matched prefixes of ≤ 24 bits (LPM2).

Results for hardware-independent metrics. Figure 1 shows the results for the metrics IC and MA, and we see that BOLT predicts them accurately, with a maximum over-estimation of 7.5% and 7.6%, respectively. It is possible that our generated test traffic may not have incurred the actual worst case performance, hence the above numbers represent an upper bound on Bolt’s over-estimation. In the pathological scenarios that correspond to unconstrained traffic (Br1, NAT1, LB1) for NFs that maintain and expire per-flow state, both the predicted and the actual performance is 8 orders of magnitude worse than in the other scenarios (in these extreme scenarios, a packet could take over a minute to be processed). Even so, BOLT’s IC and MA predictions are accurate (and conservative) with maximum over-estimation 2.36% and 3.03%, respectively. The over-estimation in IC and MA predictions comes from two sources: (1) imprecision introduced when we coalesce execution paths within the

NF+Class	Predicted Bound	Measured Cycles	Ratio
NAT1	591,948,908,371	65,217,699,390	9.08
NAT2	7,401	2,376	3.11
NAT3	5,142	1,789	2.87
NAT4	2,956	884	3.34
Br1	295,984,939,878	32,383,472,634	9.14
Br2	7,329	2,013	3.64
Br3	7,383	1,808	4.08
LB1	591,969,879,756	66,062,284,173	8.96
LB2	5,299	2,386	2.22
LB3	8,108	2,541	3.19
LB4	4,300	2,310	1.86
LB5	4,837	2,079	2.33
LPM1	1,419	967	1.46
LPM2	1,015	545	1.86

Table 3: Accuracy of execution cycle performance contracts for multiple NFs and packet classes.

stateful performance contract, and (2) small differences between the analysed code (linked against models) and the production build (linked against real data structure implementations).

Results for our hardware-dependent metric. Table 3 shows the results for the number of execution cycles; BOLT predicts them conservatively, within a factor of $4.08\times$ for typical workloads and $9.26\times$ for the pathological (unconstrained) workloads. This is not surprising, given our simplistic, conservative hardware model.

If we had a more accurate hardware model, we would expect BOLT’s results to be more accurate too. To validate this hypothesis without investing significant time in a sophisticated model, we performed a simple experiment.

We used three simple programs that traverse a non-contiguously allocated linked list (P_1), a linked list allocated in a contiguous chunk of memory (P_2), and an array (P_3), respectively, and had BOLT compute their performance contracts. P_1 lacks opportunities for MLP or prefetching, and BOLT’s latency prediction was within 5% of the measured value. P_2 benefits from prefetching but not MLP, and BOLT’s prediction was $6\times$ higher than measured. P_3 has ample opportunity for both prefetching and MLP, and BOLT’s predicted latency was $9\times$ greater than measured latency. These measurements indicate that the more the hardware behaves like the model, the more accurate BOLT becomes. In future work, we plan to bring the model closer to real hardware.

5.2 NF operator use-cases

In this subsection, we illustrate the utility of performance contracts for NF operators and answer the following questions: Can performance contracts enable NF operators to 1) Understand the NF’s performance for a variety of workloads, in particular, when the NF is under attack? 2) Reason about

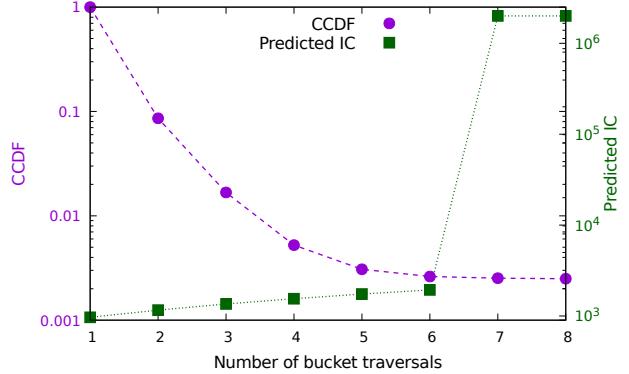


Figure 2: Predicted latency as a function of hashring bucket traversals, alongside the CCDF of traversals for a uniform random workload. The Distiller allows the operator to make an informed choice regarding where to position the threshold.

the performance of a sequence of NFs, in order to design and configure NF chains that meet their performance targets?

Understanding the performance of the NF under attack. Performance contracts that detail the performance of every feasible execution path through the NF code can be of particular utility to network operators for reasoning about the performance of their NF, when under attack. We use the MAC bridge as a motivating example to illustrate this use-case.

The bridge uses a MAC learning hash-table that defends itself from collision attacks by incorporating a random key in the hash algorithm. If the number of buckets traversed in the hash-table during a put operation exceeds a certain threshold, the key is renewed and the table re-hashed accordingly. This rehashing is designed to be a defence from attackers that know the hashing algorithm, but not the random key. However, this rehashing is particularly expensive and results in a performance cliff (Table 4).

Given its performance cost, the rehashing mechanism should be used only when a deliberate attack is suspected. The threshold that triggers the re-hashing should be carefully picked to avoid it occurring under normal circumstances. In such a scenario, the contracts and the Distiller enable the operator to easily understand the risks and trade-offs involved.

Figure 2 shows the analysis generated by the Distiller. The CCDF shows that less than 0.2% packets incur more than 6 traversals under a uniform random test workload. Setting the threshold to 6 results in the performance prediction shown in the overlaying line. The instruction count is predicted to always be less than $1939 = (144 \times 5 + 50 \times 6 + 918)$ for typical traffic.

Ability to reason about the performance of a network. Typically, operators deploy NFs in chains with packets being processed by each NF in a sequence. In these scenarios, the worst-case for one NF can often be masked by another,

Traffic Type	Instructions
Known Source MAC	$245 \cdot e + 144 \cdot c + 36 \cdot t + 82 \cdot e \cdot c + 19 \cdot e \cdot t + 882$
Unknown Source MAC; No Rehashing	$245 \cdot e + 144 \cdot c + 50 \cdot t + 82 \cdot e \cdot c + 19 \cdot e \cdot t + 918$
Unknown Source MAC; Rehashing	$245 \cdot e + 144 \cdot c + 50 \cdot t + 124 \cdot o + 82 \cdot e \cdot c + 19 \cdot e \cdot t + 14 \cdot t \cdot o + 984069$

Table 4: Bridge performance contract. Instructions are described as a function of the number of expired MAC entries (e), the number of hash collisions (c) and bucket traversals (t) incurred in the hash table, and its occupancy (o).

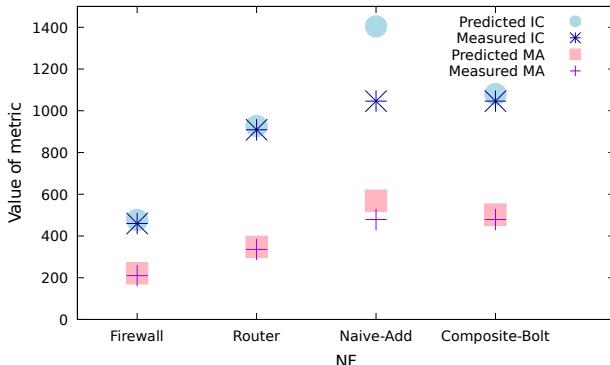


Figure 3: Composite NF of firewall + IP router. Naive-Add represents the predictions obtained from trivial addition of the individual contracts, while Composite-Bolt represents the contract for the chain produced by Bolt. Bolt’s predictions are more accurate, since it correctly takes into account the inter-NF dependencies.

resulting in tighter bounds for the aggregate than the sum of the parts (§3.4).

To evaluate BOLT’s ability to reason about such chains, we sequence together a firewall and a static IP router. Unlike our example LPM, this router can process IP options (particularly the timestamp option [36]), but doing so can be expensive, as can be seen in the contract in Table 5b. We set up the firewall (contract shown in Table 5a) to drop any packets with any IP options included.

The combined contract for the sequence, shown in Table 5c, accurately reflects the best of both NFs. Packets with IP options are quickly dropped, incurring no further cost, and the remaining packets go through the fast path on the router.

The composition gap is further illustrated in Figure 3, the worst-case performance prediction for the composite NF is more accurate than that which would be obtained by naïvely adding their individual performance contracts.

5.3 NF developer use-cases

We now illustrate the utility of performance contracts for NF developers and answer the following questions: Can performance contracts enable NF developers to 1) Identify design flaws that lead to the NF performing poorly for certain workloads/packet traces? and 2) Pick appropriate data structure

implementations when multiple implementations exist?

Debugging configuration bottlenecks. The performance contracts generated by BOLT provide the developer insight into possible performance bottlenecks. We illustrate this utility with a performance bug we found in VigNAT [4].

When dealing with traffic with high churn, VigNAT consistently incurred significant latency ($> 3\mu s$) but only for around 1.5% of packets, as can be seen in the latency CCDF in Figure 4. Building the performance contract for VigNAT (Table 6) allowed us to realise that such long tails were likely an artifact of a large number of flows expiring at once since the PCV “ e ” is dominant (by an order of magnitude) in the performance contract. This is further corroborated by the Distiller (Table 7), as the number of expired flows follows a similar distribution and coincides with the worse performance predictions.

As it turns out, VigNAT was inadvertently batching flow expiry. This was due to VigNAT time-stamping flows only at the granularity of a second. As a result, any packet arriving at the change of the second on the clock would induce the expiry of all of the flows that were supposed to expire during the entire previous second.

After we increased the granularity of the timestamp, ensuring a more uniform expiry of flows, VigNAT no longer exhibited such a long tail. The resulting change can be seen both in the latency CCDF (Figure 4) and Table 8 . The median per-packet latency rises since more packets are affected by flow expiry; however, the long tail has been eliminated.

Picking the appropriate data structure implementation. Often, developers need to make a choice between multiple implementations of a data structure that can deliver varying performance depending on the characteristics of the incoming traffic. In such scenarios, the predictive power of BOLT greatly simplifies this decision and lessens the need for more elaborate A/B testing.

We illustrate this utility using two implementations of the port allocator for a NAT (Allocator A & Allocator B) which differ in subtle ways. Note, that the difference in performance cannot always be captured in the big-O performance specification: both allocators are $O(1)$ in the common case but have different constant factors in different scenarios. Allocator A, implemented as a doubly-linked list has similar constants for allocating & de-allocating a new port, regardless of churn or flow-table occupancy. Allocator B, implemented using an array and a singly linked list, has a similar

Traffic Type	Instructions
No IP options	477
IP Options	298

(a) Firewall.

Traffic Type	Instructions
No IP options	603
IP Options	$79 \cdot n + 646$

(b) Static Router.

Traffic Type	Instructions
No IP options	1053
IP Options	298

(c) Firewall+Router chain.

Table 5: NF performance contracts for the Firewall, the Static Router, and a combination of the two in a chain. Instructions are described as a function of the number of IP options in the packet (n).

Traffic Type	Instructions
Invalid packets (dropped)	$359 \cdot e + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 425$
Known flows (forwarded)	$359 \cdot e + 30 \cdot c + 18 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 1030$
New external flows (dropped)	$359 \cdot e + 30 \cdot c + 18 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 528$
New internal flows; table full (dropped)	$359 \cdot e + 30 \cdot c + 18 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 639$
New internal flows; table not full (forwarded)	$359 \cdot e + 30 \cdot c + 44 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 1316$

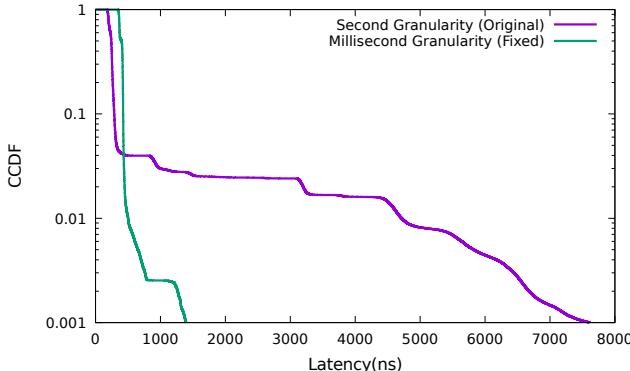
Table 6: VigNAT performance contract. Instructions are described as a function of the number of expired flows (e) and the number of hash collisions (c) and bucket traversals (t) incurred in the hash table.

Figure 4: CCDF of packet latencies. Second granularity refers to the NF from [47]. Millisecond Granularity is the NF after the timestamp granularity was increased.

constant to allocator A for de-allocation but trades off a faster allocation at low flow-table occupancies for a much slower allocation at high flow-table occupancies.

The performance contracts capture this trade-off precisely. By specifying the characteristics of the expected traffic, the developer gains complete insight into the expected performance of the NF for each data structure under consideration. Figure 5 shows how the performance contracts precisely encapsulate the performance difference between the two implementations. Figures 6 & 7, show that the performance predicted by the contract is reflected in the performance of the NAT, for each scenario. In scenarios with a large number of long-lived flows (low churn), Allocator A outperforms Allocator B by approximately 33%, while in high churn scenarios with few, short-lived flows, Allocator B outperforms Allocator A by approximately 10%. BOLT predicts a performance difference of 30% and 8% in the two scenarios, respectively.

Number of Expired Flows	Probability Density(%)
0	98.459
1 – 63	0.0066
64	0.93
65	0.6
66+	0.0044

Table 7: Distiller report for expired flows for VigNAT for uniform random traffic, clearly indicating batching.

Number of Expired Flows	Probability Density(%)
0	16.1
1	83.6
2	0.28
3+	0.02

Table 8: Distiller report for expired flows for VigNAT for uniform random traffic after the timestamp granularity was increased. Clearly flows are expired more uniformly

6 Limitations

In this section we describe limitations of our current BOLT prototype.

Since BOLT builds upon Vigor [47], it requires NFs to be written with a clean stateless/stateful separation and to use a library of pre-analysed data structures. An NF that does not follow this design cannot be analysed accurately by BOLT.

The current BOLT prototype does not extend to multi-threaded NFs with shared state. We expect the biggest challenge here to be the generation of performance summaries for concurrent data structures that properly account for the effects of cross-core interference.

The current BOLT prototype works for NFs that are im-

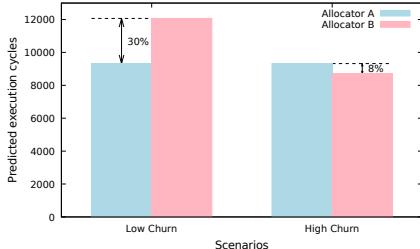


Figure 5: NAT latency predictions with both allocators in different scenarios.

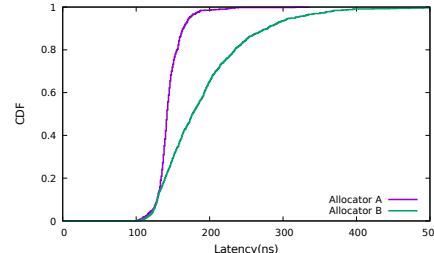


Figure 6: Allocator A outperforms Allocator B in scenarios with low churn.

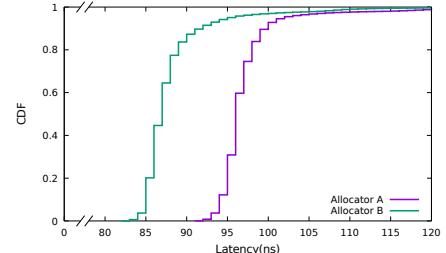


Figure 7: Allocator B outperforms Allocator A in scenarios with high churn.

plemented using the DPDK [16] framework, using a single processor core. Since DPDK is a kernel bypass framework, BOLT does not concern itself with the Linux kernel. With DPDK, best practices dictate pinning the NF to a core guaranteeing exclusive access to the L1 cache. An NF that performs Linux system calls during packet processing or that does not have exclusive non-preemptible use of a CPU core cannot be analysed accurately by BOLT.

As mentioned previously, BOLT is accurate for metrics that are independent of the underlying hardware. The slight over-estimation (7% error) arises from two sources: 1) coalescing two or more execution paths into the most expensive of them all in stateful performance summaries and 2) disabling link-time optimisations to ensure no under-approximation of performance due to differences between the analysed and production code. From our experience, the first factor is dominant and can be reduced to 0 if we choose to expose additional PCVs that are less intuitive. For hardware dependent metrics, BOLT is only as accurate as the hardware model it employs. The simple, conservative hardware model described in §3.5 leads to a 4× over-estimation for typical workloads and 9× over-estimation for pathological workloads. Additionally, it does not account for scenarios in which multiple co-located NFs contend for resources such as memory controllers [42] since it assumes a constant performance cost for memory accesses. We plan to improve the hardware model in future work.

The performance summaries for the stateful data structures were generated manually, by studying the assembly code. While the effort can be amortised across all NFs that use the data structure, the approach is laborious and potentially error-prone. In future work, we plan to automate this.

BOLT currently requires access to the NF source code, though we argue that this is not fundamental and merely an artifact of our current research prototype. Having access to the source significantly simplifies the manual analysis of the stateful code and allows us to attach semantic meaning to PCVs. That said, much of the needed information could also potentially be gleaned from debug information that compilers optionally include in binaries or otherwise deduced via reverse engineering. Likewise, access to the source facilitates attaching semantic meaning to different traffic classes in the stateless code and also gives us more

freedom in preparing analysis builds for the SEE. While our current SEE analyses LLVM bit-code, which requires a special build process, other engines [11] can directly analyse X86 binaries and our stateful/stateless separation could be rendered as separate object files. We leave redesigning our system around such considerations to future work.

BOLT currently quantifies performance in terms of three metrics (IC, MA, cycles) that provide a concrete first step into understanding NF performance. Nevertheless, we plan to extend BOLT to reason about more commonly used metrics such as throughput and end-to-end latency. We expect the major challenge to arise from modelling of the PCIe bus, the NIC and queueing delays (for latency) and modelling instruction and memory level parallelism (for throughput).

7 Related Work

Performance evaluation and diagnosis. There exists a large body of work that focuses on generating and analysing adversarial workloads that attack software performance. [12, 5, 39] describe manually generated, adversarial attacks on specific data structures and network applications (e.g., IDS). Others generate these workloads automatically: [41, 33, 35] use fuzzing to find bottlenecks in individual methods and data structures, [29] automatically detects certain complexity attacks in web services and [32] automatically generates adversarial inputs for NFs. All of these systems, focus only on adversarial workloads, while performance contracts characterise performance in the face of any arbitrary workload, whether typical, ideal or adversarial.

Others focus on deriving formal upper bounds on performance: Worst-Case Execution Time (WCET) Analysis [45]. Again, as with adversarial workloads, this only looks at one aspect of the performance profile: the absolute worst-case. These techniques are popular in the real-time systems domain where performance guarantees are a part of functional correctness. However, because of this requirement, real-time systems tend to avoid dynamic data structures and input-dependent memory accesses, aspects that are commonplace in NFs. Though not primarily designed as a WCET analysis tool, BOLT can also be used to deduce worst-case bounds (when generating contracts without any assumptions).

Performance side-channel attacks have also been analysed using static analysis techniques. [7] uses symbolic analysis to identify cost differentials between execution paths, rather than to predict absolute performance.

Another large body of work predicts performance for large-scale data analytics applications [43, 30]. This work solves a different problem from Bolt and operates on a different scale with different challenges and effects to account for. Rather than predicting performance for varying workloads on a given hardware platform, this work focuses on predicting performance for a given workload but on varying hardware configurations.

Other instances of previous work address the same problem as Bolt, but operate under different assumptions and thus tackle different challenges. Like Bolt, [28] makes parametric performance predictions, but for probabilistic programs that implement randomised algorithms where the challenge lies in the probabilistic reasoning, rather than cache-effects and other low-level details. TiML [44] also similarly predicts performance but requires applications to be implemented in the TiML functional language which uses type annotations to enable more rigorous reasoning about performance bounds. While Bolt assumes that NFs use a pre-analysed data structure library, it permits developers to continue to use low-level languages like C, as is the norm for NFs. Bolt also reasons about low-level hardware behaviour, such as cache effects.

Runtime performance analysis monitors systems during execution to identify performance issues; we focus here on work related to NFs. NFVPerf [27] passively monitors traffic to find hardware and software bottlenecks in software NFs. PerfSight [46] extracts low-level performance information from software data planes and allows operators to find which network functions are responsible for performance issues. FlowTags [19] modifies middleboxes to tag sent packets and use tags from received packets to enforce network-wide policies, including performance.

Program Analysis Applied to NFs. Several instances of prior work have proposed using static analysis to help understand, debug, and verify software NFs. StateAlyzr [23] statically analyses NFs to identify their per-flow and global state to enable efficient state migration and redistribution. Other approaches, use symbolic execution to find bugs or formally verify correctness. [48, 9, 10] leverage this technique to automate bug finding and test-case generation. [40] symbolically executes NF models to reason about network properties like reachability and loops. [47, 14] use exhaustive symbolic execution to formally verify functional correctness.

8 Conclusion

In this work, we propose the notion of performance contracts for NFs. Performance contracts precisely characterise NF performance for any arbitrary incoming workload, whether

typical, ideal or adversarial. We express performance contracts in terms of Performance Critical Variables (PCVs) which succinctly characterise how NF state and configuration parameters affect the performance of the incoming packet. Additionally, we present and evaluate BOLT, a tool that automatically derives performance contracts with accurate performance predictions from the NF code. Finally, we walk through a series of use-case scenarios that illustrate how network operators and NF developers can use BOLT to understand and mitigate NF performance unpredictability.

References

- [1] Bolt Project Repository. <https://github.com/bolt-perf-contracts/bolt>.
- [2] Intel®64 and IA-32 Architectures Optimization Reference Manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- [3] LPM library - data plane development kit 18.08.0 documentation. https://doc.dpdk.org/guides/prog_guide/lpm_lib.html.
- [4] VigNAT source code repository. <https://github.com/vignat/vignat>.
- [5] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Making DPI engines resilient to algorithmic complexity attacks. *IEEE/ACM Trans. on Networking*, 2016.
- [6] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [7] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Intl. Symp. on Software Testing and Analysis*, 2018.
- [8] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.
- [9] M. Canini, D. Kostic, J. Rexford, and D. Venzano. Automating the testing of OpenFlow applications. *Intl. Workshop on Rigorous Protocol Engineering*, 2011.
- [10] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test openflow applications. In *Symp. on Networked Systems Design and Implem.*, 2012.

- [11] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.
- [12] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symp.*, 2003.
- [13] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [14] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Symp. on Networked Systems Design and Implem.*, 2014.
- [15] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Symp. on Networked Systems Design and Implem.*, 2012.
- [16] DPDK: Data plane development kit. <https://dpdk.org>.
- [17] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Huelscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Symp. on Networked Systems Design and Implem.*, 2016.
- [18] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A scriptable high-speed packet generator. In *Internet Measurement Conf.*, 2015.
- [19] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Symp. on Networked Systems Design and Implem.*, 2014.
- [20] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification*, 2007.
- [21] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.
- [22] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.
- [23] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr. In *Symp. on Networked Systems Design and Implem.*, 2016.
- [24] M. Kuhnemann, T. Rauber, and G. Runger. A source code analyzer for performance prediction. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004.
- [25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.
- [26] K. Meng and B. Norris. Mira: A framework for static performance analysis. In *2017 IEEE International Conference on Cluster Computing*, 2017.
- [27] P. Naik, D. K. Shaw, and M. Vutukuru. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *IEEE Conf. on Network Function Virtualization and Software Defined Networks*, 2016.
- [28] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Intl. Conf. on Programming Language Design and Implem.*, 2018.
- [29] O. Olivo, I. Dillig, and C. Lin. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Conf. on Computer and Communication Security*, 2015.
- [30] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Symp. on Operating Systems Principles*, 2017.
- [31] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *Symp. on Networked Systems Design and Implem.*, 2015.
- [32] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *ACM SIGCOMM Conf.*, 2018.
- [33] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conf. on Computer and Communication Security*, 2017.
- [34] S. Pirelli, A. Zaostrovnykh, and G. Candea. A formally verified NAT stack. In *ACM SIGCOMM Workshop on Kernel-Bypass Networks*, 2018.
- [35] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symp.*, 1998.

[36] RFC 781. <https://tools.ietf.org/html/rfc781>, 1981.

[37] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Symp. on Networked Systems Design and Implem.*, 2012.

[38] V. Sekar and P. Maniatis. Verifiable resource accounting for cloud computing services. In *Cloud Computing Security Workshop*, 2011.

[39] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a NIDS. In *Annual Computer Security Applications Conf.*, 2006.

[40] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM Conf.*, 2016.

[41] L. D. Toffola, M. Pradel, and T. R. Gross. Synthesizing programs that expose performance bottlenecks. In *Intl. Symp. on Code Generation and Optimization*, 2018.

[42] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling sls in network function virtualization. In *Symp. on Networked Systems Design and Implem.*, 2018.

[43] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Symp. on Networked Systems Design and Implem.*, 2016.

[44] P. Wang, D. Wang, and A. Chlipala. Timl: A functional language for practical complexity analysis with invariants. 2017.

[45] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

[46] W. Wu, K. He, and A. Akella. PerfSight: Performance diagnosis for software dataplanes. In *Internet Measurement Conf.*, 2015.

[47] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Canea. A formally verified NAT. In *ACM SIGCOMM Conf.*, 2017.

[48] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Intl. Conf. on Emerging Networking Experiments and Technologies*, 2012.

FlowBlaze: Stateful Packet Processing in Hardware

Salvatore Pontarelli^{1,2}, Roberto Bifulco³, Marco Bonola^{1,2}, Carmelo Cascone⁴,
Marco Spaziani^{2,5}, Valerio Bruschi^{2,5}, Davide Sanvito⁶, Giuseppe Siracusano³,
Antonio Capone⁶, Michio Honda³, Felipe Huici³ and Giuseppe Bianchi^{2,5}

¹Axbryd, ²CNIT, ³NEC Laboratories Europe, ⁴Open Networking Foundation,
⁵University of Rome Tor Vergata, ⁶Politecnico di Milano

Abstract

While programmable NICs allow for better scalability to handle growing network workloads, providing an expressive yet simple abstraction to program stateful network functions in hardware remains a research challenge. We address the problem with FlowBlaze, an open abstraction for building stateful packet processing functions in hardware. The abstraction is based on Extended Finite State Machines and introduces the explicit definition of flow state, allowing FlowBlaze to leverage flow-level parallelism. FlowBlaze is expressive, supporting a wide range of complex network functions, and easy to use, hiding low-level hardware implementation issues from the programmer. Our implementation of FlowBlaze on a NetFPGA SmartNIC achieves very low latency (in the order of a few microseconds), consumes relatively little power, can hold per-flow state for hundreds of thousands of flows, and yields speeds of 40 Gb/s, allowing for even higher speeds on newer FPGA models. Both hardware and software implementations of FlowBlaze are publicly available.

1 Introduction

Network infrastructures need a continuously evolving set of network functions to be operated reliably [45, 35]; NAT, path and server load balancing, traffic shaping, security functions such as access control, and DoS protection are just a few examples. Given the flexibility of modern networks and the need to continuously support new applications, operators have turned to pure software implementations for such functions, which have a number of benefits, including programmability and ease of management [8].

However, while essential to network operations, network functions introduce overheads. Most notably, they increase network packets’ end-to-end delay, since they are on the path of network flows, and increase the overall cost of running the infrastructure, needing additional computation resources, i.e., CPU cores. These overheads become particularly critical in low latency fabrics such as those of modern datacen-

ters [50]. For instance, server-to-server communication delays are expected to be in the order of a few tens of μ s in a cloud datacenter [29]. For comparison, it could take tens of μ s for packets to go from a NIC, over the PCIe bus, to a CPU that executes a network function and back to the NIC.

To address this issue, past years have seen the introduction of efficient programmable network devices that can offload network packets processing from the CPU. For example, Microsoft deployed FPGA-based SmartNICs in their datacenters [24]. These devices save CPU usage and reduce the amount of traffic on a server’s PCIe bus, improving a network function’s packet processing latency by more than an order of magnitude. As a downside, programming a SmartNIC to support a new network function requires hardware design expertise. While a tech giant can build and assign a dedicated team to the task [24], this is usually not the case for a large majority of companies, e.g., smaller cloud or network operators. As a result, recent network programming abstractions, such as P4 [13] have the explicity goal of simplifying the programming of FPGA-based network devices [60, 2]. However, they have limitations in describing network functions that need to keep per-flow state [24, 58], a common requirement in the world of network functions.

In this paper we address these shortcomings by introducing FlowBlaze, an abstraction that extends match-action languages such as P4 or Microsoft’s GFT [24] to simplify the description of a large set of L2-L4 stateful functions, while making them amenable to (line-rate) implementations on FPGA-based SmartNICs. To benefit the community at large, we build FlowBlaze on open SmartNIC technology (NetFPGA [65]), and provide our hardware and software implementations as open source. Our contributions are:

- The FlowBlaze abstraction. FlowBlaze adapts match-action tables to describe the evolution of a network flow’s state using Extended Finite State Machines (EFSM).
- A hardware implementation of FlowBlaze on the NetFPGA SUME card that can forward packets at 40Gb/s line rate while keeping state for hundreds of thousands of flows, consuming relatively little power, and providing

- packet processing latency of less than 1 μ s.
- A comprehensive analysis of FlowBlaze’ expressiveness through the implementation of over 10 different use cases including stateful firewalls, load balancers, NATs, traffic polices and SYN flood detection and mitigation.
 - Two different high performance software implementations (for mSwitch [31] and the Linux kernel) of FlowBlaze. Such implementations can be used to handle VM-to-VM communications; to help implement fail-over in software scenarios, e.g., during hardware maintenance; to support end systems that do not have a SmartNIC available; or to implement network functions that are deployed in less performance-demanding scenarios.

FlowBlaze implementations, documentation and additional results are available at [25].

2 Requirements and state-of-the-art

Our goal is to provide a system that allows a programmer with little hardware design expertise to quickly implement, or update, stateless and stateful packet processing functions at high speed, on FPGA-based SmartNICs. It should be noted that our focus is on functions that operate on packet headers, generally at L2-L4 of the network stack, such as firewalls, L4 load balancers and NATs. These are common building blocks for many other functions, and can often be completely executed within a NIC. Functions that operate on packet payloads such as DPIs and L7 firewalls are out of the scope of this work. More formally, we target the following requirements:

- **R1: High Performance:** support of network functions that achieve throughput in the range of 40-100Gb/s while providing per-packet processing latency of at most a few μ s.
- **R2: State Scalability:** support for functions that operate on fine-grained per-flow state and the ability to store per-flow state for large numbers of network flows (e.g., up to several 100Ks). The number of flows should not affect the processing latency.
- **R3: Easy to Use:** allow a programmer to focus solely on implementing the functionality needed and not get bogged down in tricky, time-consuming hardware performance optimizations. Further, hardware constraints should have reasonably little impact on application design, and the programmer should need little to no hardware design expertise to implement a function.
- **R4: Expressiveness:** the system’s programming abstraction should allow users to describe a large range of potentially *stateful* functions, including complex ones (e.g., anomaly detection, connection tracking, etc.).

2.1 Existing systems

Taking these requirements into account, we review the state of the art and find that existing systems only meet these re-

	High Perf	State Scal	Ease	Expressiv
General programming frameworks				
HDL	✓	✓	✗	✓
HLS [52]	✗	✓	✓	✓
ClickNP [42]	-	✓	-	✓
Match-action abstractions				
P4 [13]	-	-	✓	-
Domino [58]	✓	✓	-	-
OpenState [11]	✓	✓	✓	✗
FAST [51]	✗	✓	✓	✗

Table 1: Qualitative comparison of stateful abstractions. A dash means a requirement is only partly met.

quirements partially, for an FPGA target (See Table 1). In effect, we can split previous approaches in two categories:

General programming frameworks are solutions that rely entirely on FPGA re-programmability features to implement new functions and therefore focus on languages and frameworks to simplify FPGA programming. Here we include Hardware Description Languages (HDLs) and High Level Synthesis (HLS) systems.

HDLs such as Verilog are a low-level programming method for FPGAs, requiring extensive hardware design expertise. HLS systems like those based on OpenCL, can improve ease of use (R3) by adopting high-level languages. However, hardware expertise is still needed since the code has to be properly designed and annotated to ensure the synthesis process succeeds in providing a high-performance implementation [52]. For example, ClickNP [42] may require the programmer of a functional element to apply specific hardware-related optimizations, when the compiler fails to apply its automated optimization [42]. Further, updating a function requires a new synthesis and flashing of the FPGA design, a process that takes hours.

Match-action abstractions are based on the match-action tables (MATs) model. MATs are an effective and widely applied tool to describe network functions as a pipeline composed of a parser and a variable number of match and action blocks. The parser and the actions logic are generally rather stable and are therefore programmed at configuration time, while the match table’s entries are inserted at runtime and can be used to change the implementation of functions on-the-fly.

MATs are a good tool to describe L2-L4 network functions [47], but currently available MAT abstractions only support *stateless* functions so that a programmer cannot specify functions where the processing of a packet should depend on a previously received packet. Older versions of the P4 language, which targets a MAT model, left the implementation of state-related constructs such as registers, to platform-specific features, which reduces portability and complicates the work of a programmer. As a matter of fact, solutions that use P4 to implement FPGA-based network functions require the programmer to provide stateful func-

tions using HDLs [2, 60].

A step forward in this direction was first provided by Domino [58], which extends the match-action model to include, in the action blocks, small and fast registers that can keep state. These registers can be accessed during the processing of any packet handled by the action block, thus providing a global state access model. A similar solution is also supported by the @atomic construct in the latest P4 language version, i.e., P4-16 [20] (the previous version was P4-14 [19]). Unfortunately, the global state model provides little information for hardware implementations to optimize state operations. Thus, Domino-like solutions are designed to address a worst-case access pattern, which leads to very constrained state update functionality (R4).

Do we meet all four requirements at once? The widespread application of MATs suggests that they could be a good starting point for providing an effective abstraction to describe network functions, and works like Domino already point towards a direction that extends MATs to support stateful functions. Thus, our problem boils down to supporting per-flow states in a match-action abstraction.

Here, we begin by making the observation that many network functions already partition their state on a per-flow basis [54]. This means that programmers are familiar with the concept of flow state, and that this inherent, per-flow parallelism can be leveraged to optimize state operations and memory accesses in a hardware implementation.

Can flow state become a first class citizen? Admittedly, FAST [51] and OpenState [11] follow this direction to provide stateful packet processing abstractions, explicitly defining flow state. In both cases, packets are grouped in programmer-defined network flows (like in MATs), and flows are associated with a Finite State Machine (FSM). Programmers then define the FSMs' transition tables which are used to update the corresponding FSMs' states as packets are processed. Following this line of reasoning, it would seem that FSMs would be a good choice to transform a stateless MAT into a stateful element. Related work has shown FSMs to be programmer-friendly, allowing for the definition of a host of packet forwarding behavior [40, 62] (R3, R4). Further, a FSM can be efficiently represented by its transitions table, which can be implemented in a straightforward way in hardware since it is functionally equivalent to a MAT (R1).

Unfortunately, FSMs are not scalable (R2). Briefly, an FSM is described by the set of states S , inputs I , outputs O and by the transition relation $T : S \times I \rightarrow S \times O$. Since FSMs need to explicitly define all the possible states $s_i \in S$ the system can be in, this may lead to a phenomenon known as *state explosion* [34]; the next section explains how FlowBlaze solves this issue.

3 FlowBlaze Design

To keep most of the good properties of FSMs while providing a scalable abstraction, we resort to Extended Finite State

Machines (EFSMs) [4]. An EFSM extends the FSM model by introducing: (i) variables D to describe the state; (ii) enabling functions F on such variables to trigger transitions ($f_i : D \leftarrow \{0, 1\}, f_i \in F$); and (iii) update functions for the variable values ($u_i : D \leftarrow D, u_i \in U$). The transition relation of an EFSM is expressed as $T : S \times F \times I \rightarrow S \times U \times O$.

Example. Figure 1 shows the EFSM representation of an application that identifies whether a single flow f_1 (e.g., identified by IP destination) is large by marking all packets after the 100th one. Using a conventional graph representation, the nodes of Figure 1 are states, while edges represent transitions. Each node is named using the corresponding state label. Transitions are marked with the quadruple $\{enabling\ functions, event, update\ functions, output\}$. Enabling and update functions operate on the variable $D(f_1)$, which is the variable of D we selected to store a flow's number of packets. The event $pkt(f_1)$ represents the reception of any packet belonging to a flow f_1 . Finally, the outputs *mark* and *fwd* are high-level descriptions of a packet header rewriting action and a generic forwarding action. The dashed line shows a transition triggered by a timeout event (e.g., an idle timeout), which brings the EFSM back to its starting state.

3.1 The FlowBlaze Abstraction

While adopting EFSMs partly helps in dealing with state explosion, we still need to adapt them to ensure an efficient hardware implementation. We need to address two issues:

- **State scalability:** standard EFSMs would require a separate transition table (i.e., the EFSM's description), for *each* flow in the system.
- **Flow parallelism:** an EFSM state definition does not include the concept of flow state, which we need in order to leverage flow-level parallelism.

To address the above issues FlowBlaze uses EFSM *definitions* to describe the behavior of a group of flows. Here, the observation is that often many flows share the same forwarding behavior [39], even if each flow, at any given point in time, may be in a different state of such EFSM. For example, in Figure 1 we specify an EFSM for a single flow, but we usually want to apply the same EFSM to all the other flows seen by our function, so to mark *any* flow with more than 100 packets. In other words, the forwarding logic is the same for all flows, but they are actually associated with different EFSM *instances*.

State types The introduction of such instances has an immediate side-effect: there is no way for two different instances to read or write each other's states. For example, we would be unable to extend the EFSM of Fig. 1 to also count the total number of flows that sent more than 100 packets.

We address the issue by introducing the notion of *global state*, which can be read and modified by all the EFSM instances generated from the same EFSM definition. Formally, the FlowBlaze abstraction divides the EFSM variable space

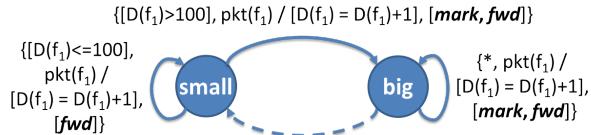


Figure 1: EFSM description of an application that identifies flows generating more than 100 packets

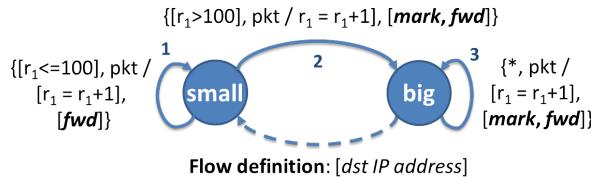


Figure 2: Description of the application of Fig. 1 using a generic flow definition.

D in two parts: the global registers $g_i \in G$ that are part of the global state; and the flow registers $r_i \in R$, which, together with the state label, constitute the flow state. The global state exists for the entire lifetime of the system and accesses to the global state are serialized, similar to [58]. Flow state, on the other hand, only exists while its corresponding flow does, and accesses to it are serialized for packets *within* a flow, i.e., when processing packets belonging to different flows it is possible modify their respective flow states in parallel.

To illustrate, Figure 2 extends the example of Figure 1 to describe the same application but this time using an EFSM to identify all large flows, with a flow defined as a set of packets with the same destination IP address. In this case, each flow is associated with an EFSM instance that has its own current state ("small" or "big") and a variable r_1 . Notice that in Figure 2 variables are no longer accessed using a flow id (in Fig. 1 we used $D(f_1)$), since each packet is now associated with its corresponding flow's EFSM instance. Similarly, the packet reception event does not specify the flow the packet belongs to anymore, since that information is captured already by the generic flow definition associated with the EFSM. Table 2 shows the transition table needed to implement this function; we will give a full explanation of it and of how to program FlowBlaze later in this section.

Composition In a match-action abstraction there is usually a pipeline of MATs. Similarly, FlowBlaze allows for the pipelining of EFSMs, which is equivalent to their sequential composition [40]. That is, in the most general case a network function is described by an ordered list of EFSM definitions, where the output of the EFSM i can be used as input for EFSM $i + 1$. Each EFSM definition has its own scope for the flow and global states: an EFSM definition's global state is only accessible by that EFSM's instances.

To move information between two sequential EFSMs, FlowBlaze associates *packet state*, i.e., metadata, with each packet. Such state is created when a packet is received, and deleted once its processing is completed. For example, after

#	Cond.	Event	S	Nxt S	Update	Pkt act.
1	$r_1 \leq 100$	pkt	sml	sml	$r_1 = r_1 + 1$	fwd
2	$r_1 > 100$	pkt	sml	big	$r_1 = r_1 + 1$	mrk, fwd
3	*	pkt	big	big	$r_1 = r_1 + 1$	mrk, fwd

Table 2: Transition table for the example of Fig. 2.

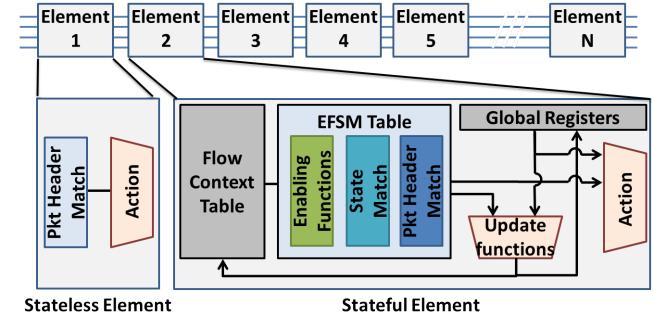


Figure 3: FlowBlaze machine model

the EFSM of Figure 2 one could add another EFSM which uses metadata tagged by the first EFSM to classify packets differently, based on whether they belong to a large or a small flow.

3.2 Machine Model

Having described the FlowBlaze abstraction, we now proceed to show how this abstraction is formalized to a machine model that can be implemented in hardware.

More specifically, FlowBlaze's machine model (see Figure 3) extends the MATs pipeline model described by RMT [14]. Like in RMT, FlowBlaze packet headers (including packet metadata) are processed through the pipeline's elements to define the forwarding actions. Each element can be either stateless or stateful. A stateless element is a MAT, similar to the ones employed in RMT¹. Unlike RMT, a stateful element implements an EFSM definition. As a result, a pipeline can combine both stateless and stateful elements.

The architecture of a stateful element has two notable differences from a MAT: (1) such an element has a *Flow Context Table* before the usual match part, and (2) the element splits the actions into (state) update functions and packet actions. In greater detail, as shown in Figure 3 (the box labeled "Stateful Element"), packet processing involves the following sequential steps:

1. Flow Context Table. When a packet header enters the element, it is associated with a corresponding flow context. The context is extracted from the Flow Context Table using, as search key, a list of header fields (e.g., the TCP/UDP 4-tuple, optionally in conjunction with the packet's metadata). The search key is specified at configuration time and corresponds to FlowBlaze's EFSM flow definition. The context, i.e., a table's entry, includes a state label s and an array of

¹FlowBlaze assumes packets headers are already parsed when passed to the pipeline, taking advantage of RMT-like programmable packet parsing [28] and reconfigurable match tables [14].

registers $\vec{R} = \{r_0, r_1, \dots, r_{(k-1)}\}$. Flow contexts are also associated with hard and idle timeouts. If no context is found for a given key, a default context is used (i.e., with all values set to 0). A single flow context identifies an EFSM instance.

2. EFSM Table. The packet’s header and metadata, plus the extracted flow context, are passed to the *EFSM table*. Such table is an extension of a traditional MAT, which in addition to supporting matching on the packet header fields, can also match on the state label s and evaluate enabling functions. An enabling function can be specified as a logical AND of up to m arithmetic comparisons ($\vec{C} = \{c_0, c_1, \dots, c_{(m-1)}\}$). The comparisons’ operands can be selected among any combination of flow registers and packet header fields. For each entry in the table, a programmer can specify (i) packet modification and forwarding operations, (ii) the next state label s , and (iii) a list of instructions to update the flow context registers \vec{R} and the global registers $\vec{G} = \{G_0, G_1, \dots, G_{(h-1)}\}$. In short, the EFSM table acts as the state machine’s transition table.

3. Update Functions. The header and metadata, the update instructions, and the new value of the state label are passed to the *update functions block*. The block performs the required update instructions to update the values stored in both the flow context registers \vec{R} and global registers \vec{G} . Such instructions can range from simple integer sums, for instance to update the value of a register representing a packet or byte counter, to more complex ones, e.g., multiplications, depending on the specific implementation and target performance.

4. Action. Like in a MAT, this block applies actions on the packet header. In contrast to a MAT, the values of the flow context registers as well as those of the global registers can also be used (e.g., to rewrite some packet header fields).

3.3 FlowBlaze Programming

FlowBlaze is deployed bump-in-the-wire, in the NIC, and its programming is similar to programming a P4 device. At configuration time, the programmer has to define the parser, the match fields for MATs and EFSM transition tables, and the actions, which now include also the state update functions. Like in [2], changing these components requires a new synthesis of the FPGA design. Luckily, these are the parts of a function that change less frequently [23].

At runtime, the programmer defines the logic of her network functions by configuring the flow definition for the stateful elements, selecting a subset of the parsed header fields, and writing the required entries in the MATs and EFSM transition tables. This is analogous to the runtime programming of P4 and OpenFlow devices. In fact, we extend the OpenFlow protocol to write such entries from a python-based RYU OpenFlow controller [25].

One thing worth highlighting is that the programmer can quickly experiment and update her functions logic on-the-fly, since programming the network functions logic is as quick as writing entries to tables, unlike other solutions that need a new synthesis and flashing of the FPGA design [42].

Use case	Entries	Reg.
Server Load Balancer	2,2	0+1, 1
UDP Stateful Firewall	5	0
Port Knocking Firewall	6	0
Flowlet load balancer	2, 4, 9	1, 0+2, 0+4
Traffic Policer	4	2+2
Big Flow Detector	3	1
SYN flood Detection and Mitigation	4	1
TCP optimistic ACK detection	8	3
TCP super spreader detection	8	1
Dynamic NAT	3, 4	1+1, 2
vEPC subscriber’s quota verification	9	1
Switch Paxos Coordinator	1	0+1
Switch Paxos Acceptor	3	3+1
In-network KVS cache	6	2

Table 3: Implemented use cases. The entries column reports the number of EFSM table entries needed by a stateful element, with each comma-separated number representing a different element. The registers column lists the number of flow+global registers for each element. More in the Appendix.

3.4 Expressiveness: A Case Study

To demonstrate FlowBlaze’s expressiveness we have implemented a large set of network functions ranging from NATs to load balancers and anomaly detection, to name a few (full listings in Table 3). For each we state the number of stateful elements, and the respective number of entries and registers required by FlowBlaze to support them. The different use cases can be combined to provide more complex functions.

Beyond these applications, and to provide a better understanding of the level of complexity that our system can support, we now show an example of a FlowBlaze configuration that implements a TCP connection tracking function (Figure 4). Connection tracking is required by network operators to protect their networks from a number of attacks [33], but its implementation is fairly complex and challenging (e.g., it requires per-packet flow state updates to check TCP sequence numbers, and we cannot know, at least not right away, whether a packet we see is actually delivered or lost).

Assuming a parser can extract the relevant TCP header fields [28], FlowBlaze allows us to implement the function using two stateful elements. The first element has 7 transitions and tracks the TCP connection establishment, i.e., the three-way handshake, and computes receive window (RCW) boundaries. A connection is identified with a bi-directional 4-tuple network flow. In particular, the flow key specification is defined as *biflow*, meaning that both directions of the flow will be associated with the same flow context. The flow context contains the IP address of the initiator of the connection (r_0), the last acknowledged sequence numbers (SEQs) (r_1, r_2), the last seen SEQs (r_3, r_4), and the RCW for each direction (r_5, r_6). A packet is sent to the second element when a connection is in the *estbl* state, in which case the left and right boundaries of the RCW are computed and copied

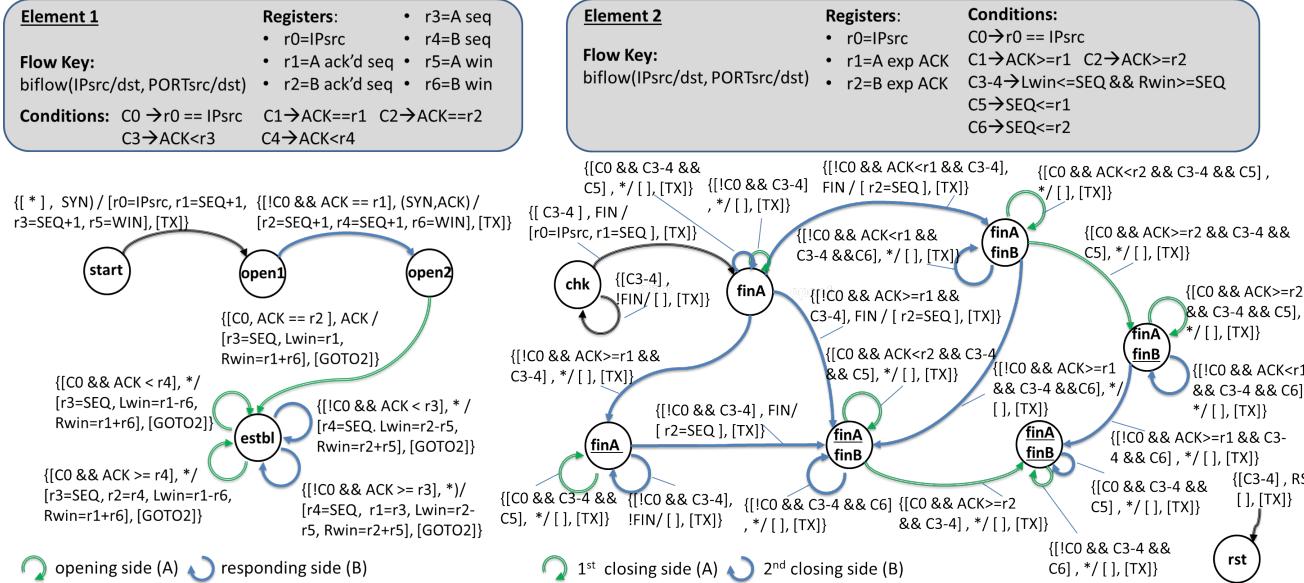


Figure 4: A complex, TCP connection tracking use case to show FlowBlaze’s expressiveness. All of element 2’s states can transition to the `rst` state. Transitions triggered by timeouts are omitted for clarity.

into the packet’s metadata. The second element has 29 transitions and checks if the packet’s SEQ is within the RCW boundaries and handles connection termination. The flow context contains the IP source of the connection termination initiator (r_0) and the last expected ACK number for each direction (r_1, r_2). Since connection terminations do not happen at packet processing speed, and since it is safe to keep state for terminated connections for a short time, we leave it to software (e.g., running on a CPU), to clean up the state of terminated connections: the software reads the terminated connections from element 2 and clears the state for those connections from element 1.

It is worth pointing out that making a few assumptions about the implemented network functions can seriously reduce the required FlowBlaze resources. For instance, we can reduce the number of registers from 7 to 4 in element 1 by assuming that one side of the connection is trusted [30], e.g., because we control the TCP stacks of those machines. Likewise, we could simplify the implementation assuming a fixed size for the receive window, as done in some hardware firewall implementations [55]. Finally, we could easily change the implementation behavior for SEQ checking from *window shifting* to *window advancing* by simply introducing an additional condition (i.e., $newSEQ > storedSEQ$) to the transitions that update the stored SEQ in element 1.

4 Hardware Design and Implementation

In this section we present FlowBlaze’s hardware design, mapping the abstraction and machine model described in the previous section to an actual hardware implementation. Note that we focus mainly on the architecture of a stateful element, since that is where most of the research contribution

in this section lies (we leave out details about packet header parsing). We begin by giving an overview of the stateful element’s architecture, and follow that by a description of the issues we encountered and how we addressed them.

4.1 Stateful Element Architecture

FlowBlaze’s hardware design extends a state-of-the-art packet processing pipeline by introducing stateful elements. As mentioned in Section 2, one of the advantages of selecting an EFSM-based abstraction is that the EFSM’s transition table can be directly mapped to a MAT, which constitutes the starting point for our stateful element architecture (see Fig. 5): the TCAM, instruction RAM and packet header action blocks are components of a regular MAT implementation; the remaining blocks are required to implement the FlowBlaze abstraction.

In greater detail, the stateful element works as a pipeline. A packet header is received and hashed by the key extractor, which is configured to generate a flow key according to the provided EFSM’s flow definition, and first handled by the scheduler block. This block uses the hash to place the header in one of its queues, guaranteeing no per-flow re-ordering, and then serves headers from the queues with a scheduling policy that provides per-flow state access consistency.

A scheduled header is then used to look up the corresponding flow state in the Flow Context Table. The state includes the state label s and the registers \vec{r} , values which are then fed as input to the conditions evaluation block. This block selects the operands for the conditions from the registers, the header fields and/or a constant value using a crossbar, and outputs the result to a vector \vec{c} , which, together with the state label s and the header fields, is used to perform a look-up in the EFSM table. The result of the look-up is the instruc-

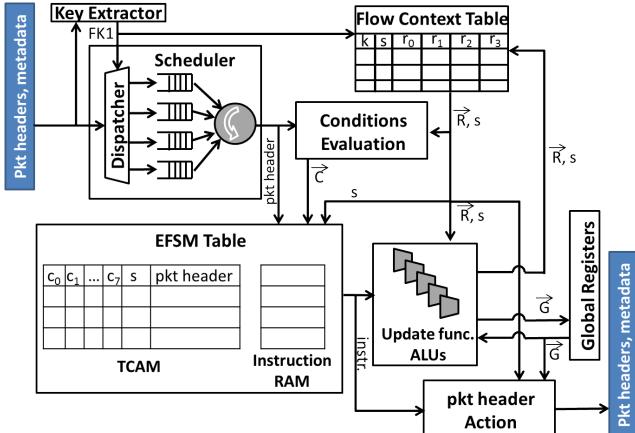


Figure 5: FlowBlaze’s stateful element architecture.

tions that should be executed by the update function’s ALUs and the packet header action block. These two blocks perform their operations in parallel: the former updates the flow state in the Flow Context Table and the value of the global registers, while the latter modifies the header fields.

Having given an overview of the architecture, we dedicate the rest of the section to describing how we solve issues to do with the scalability of the Flow Context Table and with guaranteeing consistency.

4.2 Scalability of Flow Context Tables

In order to implement the flow-oriented addressing required by the forwarding tables and by FlowBlaze’s Flow Context Table we need to rely on optimized hash tables. A typical solution in this space is to use cuckoo hash tables [63] which support higher load factors, thereby improving SRAM usage efficiency. However, an entry insertion in a cuckoo hash table may actually need multiple operations when there is a hash collision; this makes insertion times variable and potentially long for a loaded table, severely impacting performance. While current designs usually perform entry insertion and collision handling in the device’s control plane [24, 49], in FlowBlaze we need to handle entry insertions in the data plane while guaranteeing quick and constant insertion times². FlowBlaze solves the issue by implementing the cuckoo insertion logic completely in hardware and extending the hash table with a small stash memory to hold entries waiting for insertion. The stash allows FlowBlaze to hide the variable insertion time of a cuckoo hash. We implement a four-choices cuckoo hash table that offers a 97% load factor [22, 26], using a dual port (read-after-write) SRAM to support two accesses in the same clock cycle for concurrent read and write operations.

The key extractor of Figure 5 generates the four hash keys

²For reference, the device’s control plane can insert/modify entries at a speed that is usually 1000-10000x slower than the packet processing rate. Even in cases where such a slow update rate is tolerable, consistency problems for the state and scalability concerns for the control plane arise [49].

required to address the table, and the table itself is coupled with a stash memory that can host 8 entries [41]. In contrast to a typical cuckoo with stash, a new entry is always inserted first in the stash, which guarantees constant insertion time (1 cycle). In parallel, the insertion logic moves entries from the stash to the hash table and operates as a regular cuckoo+stash implementation. The insertion logic can execute an entry insertion or a movement (needed to resolve collisions) per cycle, in about 6.4ns in our implementation. It is worth noting that while a very loaded table can cause the number of movements to grow by as much as 100x [26], in section 5 we empirically show that our 8-entry stash memory is enough to avoid packet losses for the rather large network traces we test against. The observations here is that unlike state write operations which happen for each packet arrival, an insertion happens only when a new network flow starts, a smaller and thus manageable rate.

Handling Corner Cases Despite these mechanisms, a Flow Context Table (and its stash) may exceptionally become full. At this stage, the right strategy is dependent on the network function being run (e.g., for a stateful firewall the right approach might be to reject any new connections).

FlowBlaze provides a programmer with the ability to implement the logic to handle such cases. When a packet belonging to a new flow is received, FlowBlaze checks, in addition to the look-up in the Flow Context Table, the table’s occupancy level. If the table is full, a flag is set in the packet metadata, essentially indicating that there is no more space to save flow state for this flow. This flag can then be matched in the EFSM table, allowing the programmer to program the state machine to handle the case, e.g., by dropping the packet (and the flow) or by sending the packet out for further processing, e.g., in software. Further, when a table is full, FlowBlaze provides the option to install a new flow state entry by replacing an existing one. This is handled with a configurable eviction policy: the insertion logic can be configured to read the entries’ flow register R_0 and to select for eviction the entry with the highest (or lowest) value. That is, a network function’s FSM can use R_0 to enforce a custom eviction logic. For example, R_0 can be used to store the timestamp of the last seen packet in order to implement a logic that evicts the least active flows; alternatively, R_0 could store a packet counter to evict the smallest (or biggest) flow.

4.3 Guaranteeing Flow State Consistency

Flow-oriented memory addressing implies that a memory location is accessed only when a given flow’s packet is processed. As a result, a given memory location is accessed at a rate that may be just a fraction of the overall packet processing rate (i.e., one access per cycle), and enables read/modify/write operations that span multiple clock cycles. This feature, however, introduces a state consistency problem, since a memory location’s access times may vary depending on the traffic pattern, potentially leading to a concurrent read/write

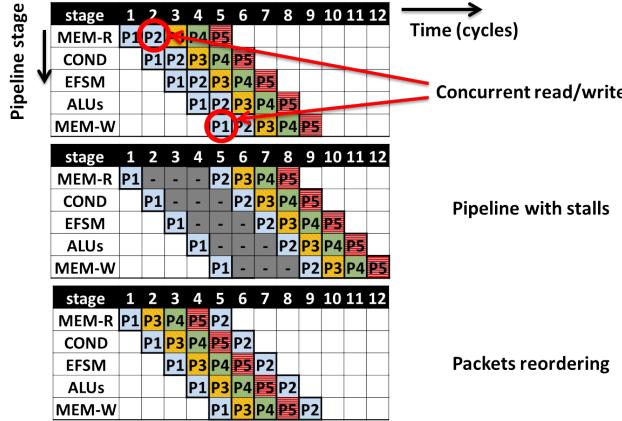


Figure 6: Stateful element scheduling scenarios. P1 (Packet 1) and P2 belong to the same flow and use the same flow context. P3, P4 and P5 belong to different flows and can concurrently access memory.

of the same location. For example, when two packet headers that access the same flow context entry are processed in short sequence, the second packet’s flow state read may happen before the first packet’s flow state update has been written back to memory (see top table in Figure 6). Stalling the pipeline while waiting for the state to be updated would guarantee consistency at the cost of performance (middle table).

To resolve the issue, [58] uses read/modify/write state operations that are performed in a single clock cycle, but at the cost of very constrained update operations. FlowBlaze, on the other hand, leverages the inherent parallelism given by the presence of different flows that access different flow context entries, hence, memory areas. In particular, the scheduler (recall Figure 5) guarantees flow context consistency by conservatively locking the pipeline only when two packets need to access the same flow context. To achieve this, the scheduler recognizes the flow a packet belongs to by using one of the hash keys (*FK1*) generated by the key extractor. When a new packet arrives, the scheduler feeds it to the pipeline if no other packets with the same hash key are being processed. Otherwise, the scheduler stalls the pipeline.

To mitigate any potential head-of-line blocking issues, in our design we arrange the state update ALUs in a parallel fashion. This effectively reduces the number of stalled cycles at the cost of constraining the complexity of state updates. That is, we limit state updates to one single operation per operand, which is anyway sufficient to implement the use cases described in Sec. 3.4. By doing so, as we will see in Sec. 5, a single queue in the scheduling block is enough to achieve the target performance.

Here, notice that this design decision may be changed with relatively minor modifications, e.g., arranging ALUs in a serial fashion, thereby providing richer state update semantics. However, in such a case, the number of stalled cycles would increase, and so would the risk of having head-of-line block-

Param.	Value	Descr.
k	4x32b	Flow context’s registers
m	8	Maximum number of conditions.
z	64b	Size of metadata moved between elements
h	8x32b	Global registers
ALUs	5	The maximum number of ALUs dictates the maximum number of update functions that can be performed for a given transition.
mqS	20	max queue size, in number of packets
nq	1	number of queues

Table 4: Parameters of a FlowBlaze stateful element in our hardware implementation.

Resource type	Reference switch	FlowBlaze
# Slice LUTs	49436 (11%)	71712 (16%)
# Block RAMs	194 (13%)	393 (26%)

Table 5: NetFPGA’s resource requirements for FlowBlaze with a single stateful element compared to those of a reference, single-stage Ethernet switch.

ing issues. Thus, our general architecture includes the option of using multiple waiting queues for packets belonging to different flows.

The scheduling block uses *FK1* to assign packets to Q different queues, guaranteeing that packets belonging to the same flow are always enqueued in the same queue, thus keeping the original ordering for packets belonging to the same flow (cf. Figure 6). The scheduler serves the queues in a round-robin fashion. When a queue is selected, it verifies if a packet with the same hash *FK1* is already in the pipeline. If that is the case, the scheduler examines the next queue, until it finds a queue whose first packet has a different hash. If no other queues are available, the pipeline is stalled. Otherwise, the scheduler extracts the current queue’s first packet and feeds it to the pipeline.

4.4 Hardware Implementation

Our implementation is based on the NetFPGA SUME [65] SmartNIC, an x8 Gen3 PCIe adapter card containing a Xilinx Virtex-7 690T FPGA [3] and four SFP+ transceivers providing four 10G Ethernet links. The system is clocked at 156.25MHz and designed to forward 64B minimum-size packets at line rate. We synthesized FlowBlaze using the standard Xilinx design flow.

Our prototype fixes the machine model’s parameters as in Table 4 and uses a non-programmable packet parser, a configurable size flow context table, and a fixed-size EFSM table. The Flow Context Table is implemented with BRAM blocks. Each entry has 128b for the flow key and 146b for the value (16b of state label plus 4 registers of 32b and 2b acting as internal flags). The EFSM table is implemented by a small TCAM of 32 entries of 160 bits. The limited size is due to the technical challenges of implementing a TCAM on FPGAs, which is still an open research issue [36, 59, 37]. Nonetheless, Table 3 shows that such number of entries is al-

ready sufficient for a large range of use cases. As described earlier, the scheduler block has a single queue, for up to 20 packets, which is enough to provide the required throughput and forwarding latency for the tested workloads. We studied the implications of different queue sizes and scheduler configurations in [17].

Table 5 lists the logic and memory resources (in terms of absolute numbers and fraction of available FPGA resources) used by a FlowBlaze implementation with a single stateful element and a Flow Context Table with 16k entries. For reference, we also list those required for the NetFPGA SUME single-stage reference switch, i.e., a simple Ethernet switch. The reported resources include the overhead of several blocks, such as the microcontroller for the FlowBlaze configuration, the input/output FIFO for the 10G interfaces, etc., which are required to operate the FPGA and do not need to be replicated for each element. We successfully implemented on the NetFPGA SUME up to 6 stateful elements for a total of about 200k flow context entries, using around 57% of LUTs and 85% of BRAM blocks.

4.5 Software Implementation

We implemented FlowBlaze’s pipeline design as a software data plane to enable any network functions written using the FlowBlaze abstraction to run in software. Briefly, we provide two implementations: a FlowBlaze module on the mSwitch [31] platform in native C (open source [25]); and for the Linux kernel network stack using eBPF/XDP [1]. We evaluate these implementations in the next section.

5 Evaluation

Methodology We experimentally measure FlowBlaze’s performance with end-to-end tests and microbenchmarks, resorting to simulations to test corner cases scenarios or to unveil details that would not be visible with black-box testing. To test the workload-dependent behavior of FlowBlaze we used a number of traffic traces collected at various operational networks. Here we report the results for publicly available traces (see Table 6) selected from carrier networks (CHI15 [16], MW15 [46]) and from university datacenters (UNI1, UNI2) [10, 9]. For the hardware implementation tests, the achieved performance is independent of the particular tested application and only influenced by the number of pipeline elements, thus we report tests only for a UDP Stateful Firewall application (NetFPGA-FW-FB). For the software implementations we show the performance of the UDP Stateful Firewall (XDP-FW-FB) and Big Flow Detector applications (mSw.-FB), to compare them with custom implementations of the same functions and evaluate the overhead of the FlowBlaze abstraction. Here, we use as comparison plain mSwitch [31], Linux’s XDP [1] and VPP [27] on DPDK. The VPP functions are from the official project repository (VPP-FW), while XDP-FW is a functionally equivalent implementation of the UDP Stateful Firewall

Trace	Max # active flows			Max # new flows/ms			Mean Pkt Size (B)
	IP s	IP s,d	5 tpl	IP s	IP s,d	5 tpl	
UNI1	575	997	4k	13	19	39	697
UNI2	948	3k	7k	20	42	42	751
MW15	12k	130k	152k	38	112	114	540
CHI15	92k	147k	178k	135	144	144	778

Table 6: Max number of active flows for 10s time windows and max number of new flows/ms in the examined traces.

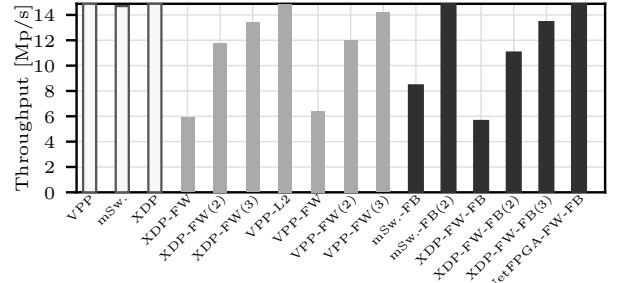


Figure 7: Packet forwarding rates of bare packet I/O engines (white), stateful packet processing w/o FlowBlaze (light gray) and that with FlowBlaze (dark gray): Numbers in () indicate the number of CPU cores if not 1. FlowBlaze does not add overhead (XDP-FW vs XDP-FW-FB) and it scales well (XDP-FW-FB vs XDP-FW-FB(2) and mSw.-FB vs mSw.-FB(2)). The NetFPGA implementation can always forward at line rate while saving up to 3 CPU cores.

function implemented by FlowBlaze.

Testbed For NetFPGA experiments, we use a single machine equipped with Xeon X3470 CPU clocked at 2.93 Ghz, the quad-port NetFPGA board (cf. Sec. 4) and a single dual-port Intel 82599 10 GbE NIC. Each 10 GbE port is connected to each of two active NetFPGA ports. For experimenting with software implementations, we use two servers connected back-to-back with Ethernet cables: each has an Intel Xeon E3-1231 v3 CPU (3.4GHz) and a single dual-port Intel 82599 10 GbE NIC. One server is used to generate and terminate test traffic; the other is used to forward packets.

5.1 Throughput

End-to-end tests We measure the end-to-end FlowBlaze throughput when running different applications using both our hardware and software implementations. The mSwitch and XDP implementations are configured with the Big Flow Detector and UDP Stateful Firewall, respectively, both identifying flows by the 5-tuple. The same application is configured also on the NetFPGA implementation.

Figure 7 summarizes the measured packet forwarding rates of minimum-sized (64B) packets with various systems. All the *bare* packet I/O frameworks (white bars) achieve line rate (14.88 Mp/s) as they do not touch packet headers. When implementing packet processing logic on top of them, rates decrease (gray bars) and we need more CPU cores to reach line rate. Implementing network functions with FlowBlaze

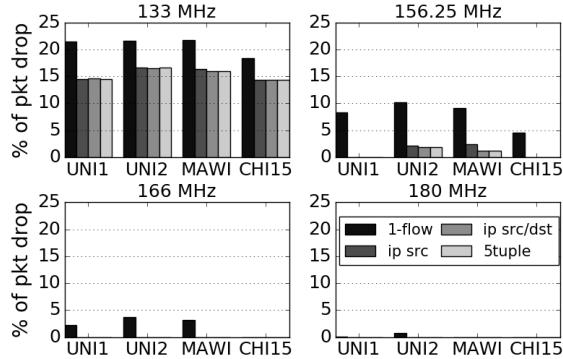


Figure 8: Drop rate of FlowBlaze when clocked at different frequencies and forwarding the traffic traces of Table 6, at 40Gb/s line rate. The 1-flow label shows the case of a pipeline without the scheduler block. With more granular flow definitions, e.g., 5-tuple, there is a higher degree of flow-level parallelism, which reduces the need to stall the pipeline. This can be seen looking at the change of drop rates for different flow definitions in the 156.25MHz case.

(black bars) does not add much overhead, as we see in the comparison between XDP-FW and XDP-FW-FB. Further, FlowBlaze scales to multiple CPU cores well (see XDP-FW-FB and mSw.-FB). Notice that the software implementations are forwarding only 4 flows, and thus we are showing a best case scenario for the achieved forwarding rate. The NetFPGA implementation can forward packets at 10Gbps line rate, and the performance is independent of the number of flows being forwarded. These results show that, even for a relatively simple use case, FlowBlaze can free several CPU’s cores from network processing tasks.

Hardware A clock-cycle detailed simulation of the hardware design shows that our prototype could in principle forward 40 Gb/s for all packet sizes when clocked at 156.25MHz. However, the introduction of pipeline stalls to guarantee flow state consistency may actually lower the achieved throughput for some traffic mixes. Thus, in order to have a better understanding of FlowBlaze performance, we use the traces described earlier to test it with different workloads. The traces show a largely bi-modal distribution, with at least 30% (or more) of the packets being minimum size. Furthermore, we try different versions of the design, clocked at 133, 156.25, 166 and 180MHz. This helps to highlight the effect of the packet scheduler, with respect to the option of running the system with a higher frequency.

We use the `moongen` packet generator on a dedicated server to replay the traffic traces, connecting the server’s 4 10GbE ports back-to-back with the NetFPGA. The traces are replayed while removing any inter-packet gap, i.e., we generate traffic at 40Gb/s line rate. We consider 4 different flow definitions: 1-flow, i.e., no distinction of network flows and thus no scheduler block, IP src, IP src/dst and 5-tuple. Fig. 8 reports the results.

For a frequency of 133MHz the FlowBlaze pipeline is un-

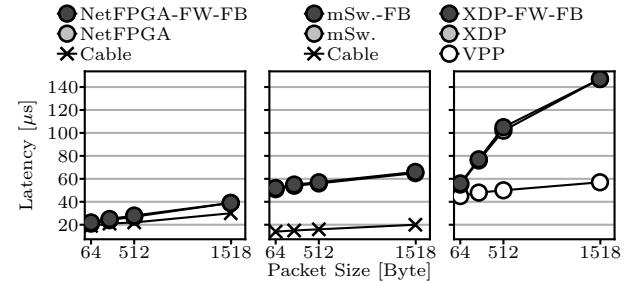


Figure 9: End-to-end RTT: offloading NFs to hardware reduces latency by avoiding PCIe and CPU overheads (dark gray plots). FlowBlaze adds almost no latency to the hardware and software baselines (comparison between light and dark gray plots within each graph).

able to sustain line rate: the roughly 15% packet drop is independent of the stalling and of flow definitions. Our prototype’s selected clock frequency, i.e., 156.25MHz, is the minimum one that sustains 40Gb/s with our design, but stalling reduces the actual throughput: this is visible in the 1-flow case, which results in a 4.6-10.20% packet drop rate, depending on the trace. The introduction of the scheduler block allows FlowBlaze to reduce (or completely remove) packet drops to 0-1.8% when using a 5-tuple flow definition. Slightly rising the frequency to 166MHz allows FlowBlaze to completely remove packet drops for all the traces and all flow definitions. In contrast, in the 1-flow case, even further increasing the frequency to 180MHz does not achieve that, with 0.7% of drops with the UNI2 trace.

5.2 Latency

In terms of latency, offloading stateful processing to an I/O peripheral can significantly reduce end-to-end latency by avoiding transfers over PCIe. Recall that end-to-end RTTs between a pair of client and server machines over Ethernet, TCP, a socket API and using a simple HTTP parser are a few tens of μ s [32]; RTTs over datacenter fabrics, thanks to sophisticated congestion control algorithms [5, 6, 61] and RDMA deployment [64, 50], can be lower than 100 μ s [29].

We connect two 10G NIC regular ports to two NetFPGA ports to measure the end-to-end latency; we then run a netmap `pkt-gen` program on one regular port to send and receive packets while instrumenting another `pkt-gen` on the other regular port to echo back received packets. The measured latency includes two passes through the NetFPGA and the latency of the `pkt-gen` (including overheads of moving packets through the PCIe bus 4 times).

Figure 9 plots the RTT measured at the `pkt-gen` generator. In the left graph the NetFPGA implementation adds only 2-9 μ s to cable, and up to 1 μ s to plain FPGA forwarding (no FlowBlaze). This is because packets are avoiding PCIe bus transfers and FlowBlaze is optimized to process a packet in just 8 clock cycles. Here, notice that multiple pipelined elements may slightly increase the processing latency of Flow-

Blaze, with each element adding about 50ns. Also, a full queue in the scheduler block may add up to 384ns of waiting time. In any case, even with these additional delays, FlowBlaze packet processing latency is well below one μ s.

The middle and right graphs show the RTTs when using mSwitch and XDP implementations. Since software packet processing requires moving packets over the PCIe bus (4 times for round trip), in this case the RTT increases by 38–46 μ s with mSwitch (mSw.-FB over Cable) and by 41–127 μ s with XDP (XDP-FW-FB over Cable). However, this additional latency does not come from the FlowBlaze abstraction but from packet I/O frameworks; we can see that by comparing them against mSw and XDP, respectively. While not visible in the graphs, we note that FlowBlaze adds up to 1 μ s in both the mSwitch and XDP implementations.

In summary, offloading stateful functions to SmartNICs is important for low latency end-to-end services. Furthermore, since FlowBlaze does not add latency to base packet I/O frameworks, operators can start deploying it in software, allowing for incremental deployment of FlowBlaze-enabled SmartNICs that implement performance critical network functions.

5.3 Power Consumption

The NetFPGA consumes 16W when idle and configured with a no-op bitstream. When the FlowBlaze bitstream is loaded, the consumption grows to 22W and is independent of the packet rate and of the network function programmed on FlowBlaze. This consumption has to be considered in addition to the overall system’s power consumption, which is 85W when the CPU is idle (For a total of about 107W). In contrast, the software implementations mSwitch and FB-mSwitch consume 124W and 123W of power during operation, respectively, and 119W when not forwarding packets; FB-eBPF consumes 123W in operation and 118W otherwise. In all, FlowBlaze provides significant power savings over software-based implementations, while supporting much higher packet forwarding rates.

5.4 Flow Scalability

The maximum number of entries a FlowBlaze’s stateful element can host depends on the amount of state required by the application (e.g., number of registers); our NetFPGA implementation can host about 200k entries, which are enough for all the traces listed in Table 6. It should be noted that using a simpler but less efficient hash scheme for the FlowBlaze design, such as a 4-left hash, would have made the system unable to deal with CHI15. In fact, a 4-left hash table, with 65–70% maximum load factor could only host about 140k entries in the NetFPGA’s memory. Further, it is worth noting that the NetFPGA SUME uses a fairly old FPGA generation, with less than 10MB of SRAM blocks. Modern FPGAs could host more than 5x times such number of entries. In any case, related work such as [24] shows that these numbers are in line with current datacenter requirements.

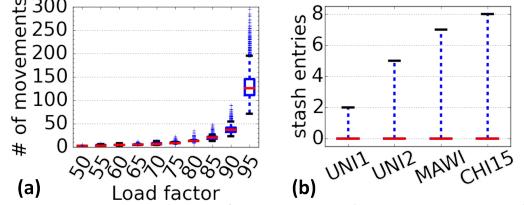


Figure 10: (a) Number of moves to insert an entry in the Flow Context Table. (b) Number of entries in the stash.

5.5 Flow Insertion Performance

Recall that FlowBlaze extends Cuckoo hashing for constant time insertion in hardware. Since slow insertion could lead to dropping flows, we are interested in whether FlowBlaze can handle high flow arrival rates.

To analyze its behaviour, we implement the FlowBlaze insertion algorithm in software, and measure the number of entry movements required for a new entry insertion while increasing occupancy of the Flow Context Table. We run two tests. First, for each traffic trace, we first fill the table to the required occupancy level, e.g., 50%, then we try to insert the next trace’s flows and measure the required number of movements for each such insertion. In a second test, we fill the table with randomly generated keys, and then try to insert new keys that are also randomly generated, performing 10k independent measurements. In all cases, the hash table is provided with memory to host all the entries when loaded at 95%. The results are similar for the two tests; for the sake of brevity, Fig. 10a shows them only for the second test.

For a table 95% full, the median number of movements is 125 per insertion, with the highest outlier requiring about 300 movements. Recall that a movement takes 6.4ns, thus 300 movements equates to about 1.9us and so FlowBlaze is able to scale to insertion rates in the order of millions of entries per second. However, recall that FlowBlaze uses a stash that can host at most 8 entries waiting for insertion. If the flow arrival rate is faster than the insertion rate in the hash table, the stash could become full and flows would be dropped. We measured the stash occupancy when using the traffic traces described earlier, simulating a challenging scenario in which each entry insertion in the table takes 450 movements, i.e., 1.5 \times the worst outlier of Figure 10a. The results in Figure 10b show that in all the cases, with the most fine-grained 5-tuple flow definition, the stash has at most 8 entries (meaning no flow is dropped) while being empty for most of the time.

6 Discussion

Ease of use While we did not run large scale surveys, we can report our experience in using FlowBlaze to implement network functions. In line with the findings of [40], we found the EFSM abstraction requires the programmer to adapt to a model that is different from regular procedural programming, but relatively simple to adopt. Once adopted, the EFSM model helps in focusing on the required function’s

state and on how it evolves. We found this particularly helpful in designing more complex functions, such as the one of Fig. 4. Here, we would first focus on the different states the function could be in, and then describe the inputs that would make a function evolve from one state to the next, leading to a very linear design and implementation process.

Security Even if a smart state encoding, such as the one adopted in [49], could enable handling state for millions of flows, FlowBlaze will always have some hard memory limit which could be potentially exploited, for example by DoS attacks. To deal with this, FlowBlaze provides primitives that allow a programmer to explicitly handle cases where the Flow Context Table is full. Still, it is up to the programmer to define functions that are robust to DoS attacks. For example, the function in Figure 4 could be preceded by an element that implements a SYN flooding detection function so that traffic from a host that performs a SYN attack could be dropped, avoiding the creation of a large number of entries in the Flow Context Tables for the elements that follow.

7 Related Work

The MAT abstraction is the starting point for FlowBlaze and was formalized by OpenFlow [48]. RMT [14] extends that model with programmable packet parsers, re-configurable forwarding tables to match on different headers and programmable actions. dRMT [18] is an alternative implementation of the RMT instruction set for a high-performance switching ASIC. Languages such as P4 [13] and PX [15] are used to describe such configurable MATs. Our work extends RMT to perform stateful packet processing.

SNAP [7] introduces a stateful packet processing abstraction for the control plane. The network is programmed as if it was a single big stateful switch, then, a compiler distributes state variables to the network devices. State is represented by an array of values that can be indexed by e.g., packet header fields' values. SNAP does not focus on the implementation of stateful operations in the data plane, which is instead the focus of our work. In fact, FlowBlaze can be used as an implementation target for SNAP. However, FlowBlaze provides a predefined per-flow state access model, which may not suite all the SNAP's programs. That is, SNAP's abstraction is less constrained, therefore there may be programs that cannot be mapped to a FlowBlaze target.

Perhaps the closest works to FlowBlaze are FAST [51] and OpenState [11]. Both works define an FSM abstraction, but (i) do not define a state access model that allows for both per-flow and global consistency, and (ii) do not deal with issues related to the integration of FSMs in an RMT machine model. FlowBlaze fills these gaps and provides a hardware implementation that addresses problems that have to do with quick per-flow state insertion and state update consistency. In contrast, FAST provides only a software implementation, while OpenState can only support much simpler Mealy Ma-

chines.

VFP [23] presents a MAT-like abstraction, GFT, to offload some network functions to a SmartNIC, and defines concepts similar to the *biflow* we use in FlowBlaze. AccelNet[24] implements GFT in a FPGA-based SmartNIC. Their design differs from FlowBlaze in several ways. First, AccelNet requires the first packet of a flow to be handled in software. For short flows this introduces additional delays that may be critical for real-time applications, thus FlowBlaze allows for functions to be entirely executed in the FPGA instead. Second, AccelNet uses a small (2k flows) cache in SRAM backed by a larger DRAM to host flow entries. This is also related to Marple, which extends data plane state memory using off-chip DRAM [53] to support network monitoring functions in high-performance switching ASICs. In contrast, FlowBlaze places all the flow entries in a highly optimized hash-table in SRAM, which guarantees constant delays for all flows' state reads and writes. Unlike FlowBlaze, AccelNet does not provide an abstraction to program FPGA functions, though it does implement a microcode programmable action block which allows for reconfiguration of the actions without requiring a change in the FPGA design; this is complementary to our work.

Examples of functions offloaded to programmable hardware are presented in [49, 57, 43, 21, 44, 56, 38]. FlowBlaze provides an abstraction to implement such use cases, and addresses issues in those implementations to do with control plane scalability and limited state memory. We described the implementation of a few datacenter use cases with an earlier software version of FlowBlaze in [12].

8 Conclusion

We presented FlowBlaze, an EFSM-based abstraction able to describe network functions targeted at high-performance data plane implementations. FlowBlaze is flexible and can implement complex network functions while being compatible with the wide-spread MATs pipeline abstraction. Leveraging the flow state concept, we provided an efficient hardware implementation that can run, at line-rate, stateful network functions that keep large, per-flow state. FlowBlaze is built on top of the NetFPGA open platform and both hardware and software sources are publicly available [25].

Acknowledgements

We would like to thank the anonymous NSDI reviewers and our shepherd Anirudh Sivaraman for their valuable feedback. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 761493 ("5GTANGO") and No. 762057 ("5G-PICTURE"). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

- [1] Linux socket filtering aka berkeley packet filter (bpf). <https://www.kernel.org/doc/Documentation/networking/filter.txt>.
- [2] P4-NetFPGA. <https://github.com/NetFPGA/P4-NetFPGA-public/wiki>.
- [3] Virtex-7 Family Overview. <http://www.xilinx.com>.
- [4] V. S. Alagar and K. Periyasamy. *Extended Finite State Machine*, pages 105–128. Springer London, London, 2011.
- [5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM ’10, pages 63–74, New York, NY, USA, 2010. ACM.
- [6] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, 2012.
- [7] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM ’16, pages 29–43, New York, NY, USA, 2016. ACM.
- [8] AT&T, BT, CenturyLink, China Mobile, Colt, Deutsche Telekom, KDDI, NTT, Orange, Telefom Italia, Telefonica, Telstra, and Verizon. Network function virtualization - white paper. http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf.
- [9] T. Benson. Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
- [10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM IMC*, ACM SIGCOMM IMC ’10, 2010.
- [11] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM CCR*, 44(2):44–51, 4 2014.
- [12] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, and G. Bianchi. Implementing advanced network functions with stateful programmable data planes. In *Local and Metropolitan Area Networks (LANMAN)*, *2017 IEEE International Symposium on*, pages 1–2. IEEE, 2017.
- [13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.
- [14] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM ’13*, ACM SIGCOMM ’13, pages 99–110. ACM, 2013.
- [15] G. Brebner and W. Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, Jan 2014.
- [16] CAIDA. The CAIDA UCSD anonymized internet traces - chicago 2015-02-19. http://www.caida.org/data/passive/passive_2015_dataset.xml.
- [17] C. Cascone, R. Bifulco, S. Pontarelli, and A. Capone. Relaxing state-access constraints in stateful programmable data planes. *ACM SIGCOMM Computer Communication Review*, 48(1):3–9, 2018.
- [18] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Varagaitik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’17, pages 1–14, New York, NY, USA, 2017. ACM.
- [19] T. P. L. Consortium. The p4 language specification - version 1.0.5, 5 2018.
- [20] T. P. L. Consortium. P416 language specification, 5 2018.
- [21] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, May 2016.
- [22] U. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *7th Workshop on Distributed Data and Structures (WDAS’06)*, Santa Clara, CA, January 2006.
- [23] D. Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, 2017. USENIX Association.

- [24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.
- [25] FlowBlaze. Repository with FlowBlaze source code and additional material. <http://axbryd.com/FlowBlaze.html>.
- [26] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, Feb 2005.
- [27] L. Foundation. FD.io. <https://fd.io/>.
- [28] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *ACM/IEEE ANCS '13*.
- [29] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.
- [30] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, volume 2001, 2001.
- [31] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mswitch: A highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ACM SOSR '15, pages 1:1–1:13. ACM, 2015.
- [32] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *Proc. USENIX NSDI*, 2018.
- [33] H. Hong, H. Choi, D. Kim, H. Kim, B. Hong, J. Noh, and Y. Kim. When cellular networks met ipv6: Security problems of middleboxes in ipv6 cellular networks. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 595–609, April 2017.
- [34] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [35] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, 2017. USENIX Association.
- [36] B. Jean-Louis. Using block RAM for high performance read/write TCAMs, 2012. Xilinx XAPP204.
- [37] W. Jiang. Scalable ternary content addressable memory implementation using fpgas. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pages 71–82, Oct 2013.
- [38] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 121–136, New York, NY, USA, 2017. ACM.
- [39] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.
- [40] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, 2015. USENIX Association.
- [41] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, Dec. 2009.
- [42] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *ACM SIGCOMM '16*.
- [43] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, 2016. USENIX Association.
- [44] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGOPS Oper. Syst. Rev.*, 51(2):795–809, Apr. 2017.
- [45] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems*

- Design and Implementation*, USENIX NSDI'14, pages 459–473. USENIX Association, 2014.
- [46] MAWI. MAWILab traffic trace - 2015-07-20. <http://www.fukuda-lab.org/mawilab/v1.1/2015/07/20/20150720.html>.
- [47] N. McKeown. Programmable forwarding planes are here to stay. In *ACM SIGCOMM 2017 The Third Workshop on Networking and Programming Languages (NetPL)*, 2017.
- [48] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 3 2008.
- [49] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.
- [50] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015.
- [51] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ACM HotSDN '14, pages 61–66. ACM, 2014.
- [52] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017.
- [53] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM.
- [54] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, 2017. USENIX Association.
- [55] Z. Qian and Z. M. Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 347–361, Washington, DC, USA, 2012. IEEE Computer Society.
- [56] A. Sapiro, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.
- [57] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, 2017. USENIX Association.
- [58] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.
- [59] Z. Ullah, M. Jaiswal, Y. Chan, and R. Cheung. FPGA Implementation of SRAM-based Ternary Content Addressable Memory. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE IPDPSW 2012, 2012.
- [60] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.
- [61] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.
- [62] Y. Yuan, D. Lin, R. Alur, and B. T. Loo. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 34:1–34:13, New York, NY, USA, 2015. ACM.
- [63] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ACM CoNEXT '13, pages 97–108. ACM, 2013.
- [64] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and

- M. Zhang. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 523–536. ACM, 2015.
- [65] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro '14*, 34(5):32–41, 2014.

A: ALU Instructions

Type	Instructions	Description
Logic	NOP	do nothing
	NOT	$OUT \leftarrow NOT(IN1)$
	XOR, AND, OR	$OUT \leftarrow IN1 op IN2$
Arithmetic	ADD, SUB	$OUT \leftarrow IN1 op IN2$
	ADDI, SUBI	$OUT \leftarrow IN1 op IMM$
	LSL (Logical shift left) LSR (Logical shift right) ROR (Rotate right)	$OUT \leftarrow IN1 << IMM$ $OUT \leftarrow IN1 >> IMM$ $OUT \leftarrow IN1 ror IMM$

Table 7: FlowBlaze ALU-supported instructions.

B: Pipeline Simulations

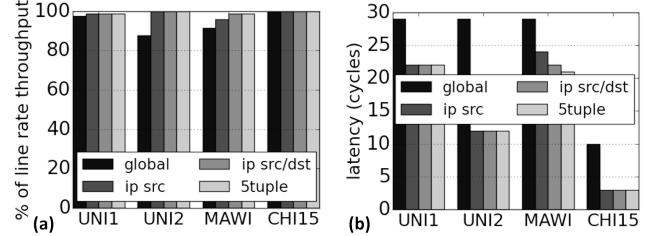


Figure 11: sim1 (a) throughput and (b) latency.

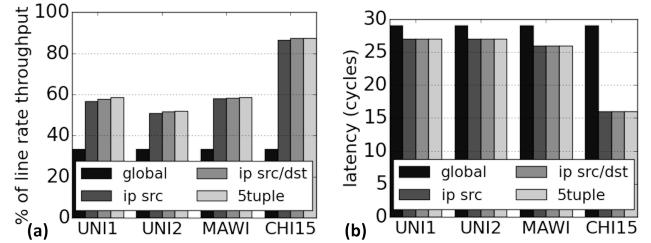


Figure 12: sim2 (a) throughput and (b) latency.

In order to understand FlowBlaze’s design performance (when clocked at 156.25MHz) with respect to a worst case scenario, we implemented a custom FlowBlaze simulator and used modified versions of the traces described in Sec. 5. The traces show a largely bi-modal distribution, with at least 30% (or more) of the packets being minimum size. It should be noted that we used this custom pipeline simulator since a clock-cycle level hardware simulator would be too slow to perform trace-based simulations at this scale.

We perform two simulations: *sim1* simulates line rate by removing any time gap between packets; this results are used to validate the simulator and are comparable to our experimental evaluation results (cf. Fig. 8); *sim2* modifies all packets to be minimum size and so represents a worst-case workload. Figures 11 and 12 plot throughput relative to FlowBlaze’s line rate, with the scheduler block (cf. Section 4.3) and without it (global label), as well as the 99th percentile forwarding latency which is increased by pipeline stalls or queuing. The former case is tested against three flow definitions: IPsrc, IPsrc-IPdst, and 5-tuple. In the global label case, we force the pipeline to stall for 3 cycles for every minimum size packet in order to guarantee state consistency (recall the middle table in Figure 6).

The results show the effects of having the scheduler block (cf. traces details in Table 6): depending on flow definitions and traces, it improves the throughput by 50–160% in the worst case (*sim2*), and up to 12% in *sim1*. In this last case, the scheduler reduces the per-packet latency by 30–70% .

SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks

Yilong Geng¹, Shiyu Liu¹, Zi Yin¹, Ashish Naik²,
Balaji Prabhakar¹, Mendel Rosenblum¹, and Amin Vahdat²

¹Stanford University

²Google Inc.

Abstract

It is important to perform measurement and monitoring in order to understand network performance and debug problems encountered by distributed applications. Despite many products and much research on these topics, in the context of data centers, performing accurate measurement at scale in near real-time has remained elusive. There are two main approaches to network telemetry—switch-based and end-host-based—each with its own advantages and drawbacks.

In this paper, we attempt to push the boundary of edge-based measurement by scalably and accurately *reconstructing* the full queueing dynamics in the network with data gathered entirely at the transmit and receive network interface cards (NICs). We begin with a Signal Processing framework for quantifying a key trade-off: reconstruction accuracy versus the amount of data gathered. Based on this, we propose SIMON, an accurate and scalable measurement system for data centers that reconstructs key network state variables like packet queuing times at switches, link utilizations, and queue and link compositions at the flow-level. We use two ideas to speed up SIMON: (i) the hierarchical nature of data center topologies, and (ii) the function approximation capability of multi-layered neural networks. The former gives a speedup of 1,000x while the latter implemented on GPUs gives a speedup of 5,000x to 10,000x, enabling SIMON to run in real-time. We deployed SIMON in three testbeds with different link speeds, layers of switching and number of servers. Evaluations with NetFPGs and a cross-validation technique show that SIMON reconstructs queue-lengths to within 3-5 KBs and link utilizations to less than 1% of actual. The accuracy and speed of SIMON enables sensitive A/B tests, which greatly aids the real-time development of algorithms, protocols, network software and applications.

1 Introduction

Background and motivation. Measurement and telemetry are long-standing important problems in Networking; there’s

a lot of research on these topics and there are several products providing these functionalities (e.g., [42, 21, 19, 56, 55, 57, 27, 13, 20, 50, 44, 40, 49, 7, 1, 2, 3, 4]). The primary use cases are monitoring the health of networks, measuring their performance, billing, traffic engineering, capacity planning, troubleshooting in the case of breakdowns or failures, and for detecting anomalies and security threats. The key challenges are: (i) accuracy: how to accurately observe and measure events or phenomena of interest; (ii) scalability: how to scale the measurement method to large networks, involving hundreds or thousands of nodes and high line rates, hence a very large “event frequency”; and (iii) speed: how to perform accurate and scalable measurement in near real-time as opposed to offline. Since these are conflicting requirements, most solutions seek to make effective trade-offs.

Measurement methods can be classified as “switch-based” or “edge-based”. Switch-based methods can be approximate or exact. We survey the literature on this topic in Section 7. For now, it suffices to say that most early work (and products; e.g., NetFlow [50] and sFlow [44]) consider approximate measurement since accurate measurement was deemed prohibitively expensive. These methods only give approximate counts of packets/bytes passing through a single switch, requiring a lot of extra processing to stitch together network-wide, flow-level views. Further, they also require extra bandwidth to move the measurement data to the network’s edge for processing. Recent developments in programmable switches and in-band network telemetry [58, 30, 32, 28] enable accurate, per-packet measurement. However, they generate a lot of data (per-packet, per-switch), whereas we shall see that network phenomena of interest can be captured with a lot smaller data. The effectiveness of INT also relies on all nodes being able to perform it. Finally, because switches are not adjacent to the end-hosts (in the way that NICs are), they cannot easily relate network bottlenecks to application performance.

Edge-based methods record “events” at end-hosts with little or no help from the network. The events are used to infer some network state that is of interest to an application or to the operator. Since storage is distributed and resources

in the end-hosts are abundant, these methods are inherently scalable. The question is how much network state can be inferred and how accurately? Existing work, surveyed in Section 7, only obtains a partial or approximate view of the network state from edge observations, such as end-to-end delay distributions, which link dropped a packet, detecting traffic spikes, silent packet drops, load imbalance, or routing loops.

By contrast, our work, which is also edge-based, obtains a near exact reconstruction of network state variables.¹ That is, we obtain key variables like queuing delays, link utilizations and queue/link compositions over small time intervals and on a per-packet or per-flow basis. Our approach is based on *network tomography*.

Tomography. The goal of network tomography is to use the “individual records”² of unicast or multicast probes and packets collected at the *edge* of the network and determine *internal* network state quantities such as delays and backlogs at individual links. The general philosophy of network tomography is this: While each record conveys a limited amount of information about network conditions, it may be possible to combine the records of all the probes or packets to get a detailed picture of internal network state and conditions.

Despite much research (surveyed in Section 7), network tomography hasn’t proved successful in wide area networks. As [25] notes, one major impediment is ignorance of the underlying network topology. This leads tomography algorithms to make unrealistically simple assumptions, which, in turn, lead to inaccurate inferences. Even if the exact topology were known, the end-to-end traversal times in the wide area setting are at least a few milliseconds and typically a few tens of milliseconds, much longer than the queuing times at routers. So two probes whose network dwell times overlap might encounter quite different queuing times at a common router buffer on their path. Since the accurate determination of queuing times is infeasible in the wide area setting, [25] advocates determining their *distributions* instead.

Reconstructing data center state variables. We revisit network tomography in the data center context. By restricting ourselves to data centers, we sidestep the problems plaguing tomography in wide area networks and obtain the following advantages. (a) A data center is operated by a single administrative entity, hence, the network topology is easy to know. The path followed by a probe or packet is also knowable (e.g., using traceroute or because the hash functions which assign packets to paths are known). (b) A modular network topology of the Clos type provides multiple paths between any pair of nodes, making congestion and bottlenecks sparse. (c) As a consequence of (a) and (b), the network traversal time of a probe or packet is dominated by queuing times at

one or two queues and *the wire times are negligible*.³

It is important to note that we do not reconstruct the *instantaneous values* of network state variables, rather we reconstruct a *I-average* of these quantities, where *I* is a short interval (e.g., 0.25 msec–1 msec in 10–40 Gbps networks). In Section 2.1, we demonstrate that packet queuing times and backlog processes viewed at the granularity of packet enqueueing and dequeuing times are very noisy. By analyzing the queuing process in the frequency domain (specifically, by looking at its *power spectral density*), we propose to reconstruct the *I*-averaged queuing times and link utilizations and show that these quantities retain 97.5% of the power of the corresponding instantaneous quantities, and are practically the same in value except for the noise. A major benefit is that the *I*-averaged network state quantities are obtained with much less data and processing effort! For example, in a 10 Gbps, 256-server network with 3 tiers of switching operating at 40% load, going from per-packet to per-millisecond data capture at the edge reduces total storage by 60x and speeds up computation by 40x with negligible reduction in accuracy (see Table 1).

If it works in the data center setting, the advantages of a tomography-based measurement system are several: (i) it doesn’t require any modification to the existing switching infrastructure since it only needs data to be gathered at the edge, and most current-generation NICs are able to timestamp packets at wirespeed, (ii) by injecting extra probes at roughly 0.3% of the link bandwidth to obtain network state information, its bandwidth overhead is negligible when compared with switch-centric approaches which need to send data collected at the switches to the network’s edge, (iii) being edge-based, it is readily able to relate application-level performance to network bottlenecks, and (iv) most importantly, it has the potential to be accurate, scalable and near real-time.

Our contributions.

We propose SIMON, a sensing, inference and measurement system *for data centers* that reconstructs key network state variables such as queuing times, link utilizations and queue and link compositions (i.e., breaking down the packets in a queue according to flow ids). SIMON uses a mesh of probes to cover all the linearly independent paths in the network, and the delays of the probes in a reconstruction interval are processed by the LASSO inference algorithm [53] to obtain the queue size and other related variables. We present:

(1) A signal processing framework for analyzing the basic elements of tomography-based measurement methods (Section 2.1). The main finding is that queue sizes and wait times fluctuate noisily when viewed at packet enqueueing and dequeuing times, but a low-pass filtered version of these processes is both easier to reconstruct and carries more than

¹“near exact reconstruction” is defined in Section 2.1.

²An individual record, described formally later, consists of the standard 5-tuple, transmit and receive timestamps, and the byte-count.

³For example, the propagation time is 5ns for 1 meter or 0.5 microsecs for 100 meters which is comparable to the raw switching (zero queuing) time at a node.

97.5% of the signal power.

(2) SIMON, a probe-based sensing, inference and measurement system for accurately determining network state variables (Section 3). In an ns-3 simulation of a 10 Gbps, 256 server DCN, SIMON achieves an RMSE of 4.14KB in reconstructing queue compositions, and a 1% error in reconstructing link utilization for each traffic class (see Figure 11). In a real-world 1 Gbps, 128 server testbed, SIMON achieves an RMSE of 5.1KB (just over 3 1500B packets) with respect to ground truth.

(3) Exploiting the hierarchical (Clos or fat-tree type) structure of modern data center topologies to devise a modular, fast version of SIMON (Section 4.1). The resulting speedup is 1,000x.

(4) Using the function approximation capability of multi-layered neural networks (NNs) [29] to hardware-accelerate SIMON, more specifically, to hardware-accelerate the LASSO inference algorithm used by SIMON. We find that even unoptimized NNs running on standard GPUs, give up to 5,000–10,000x acceleration (Section 4), enabling SIMON to run in near real-time.

(5) A verification of the accuracy and scalability of SIMON on a 128 server, 1 Gbps data center and a 240 server, 40 Gbps testbed. We comment on the deployment experience from these testbeds as well as a mixed 10G-40G, 288 server network (Section 6).

As a consequence of the speed and accuracy of SIMON, it can support sensitive A/B tests in near real-time, hence enabling the rapid development of algorithms, protocols, data center software and high-level applications. It can also be coupled with edge-based control in “Smart NICs” which are a recent and major focus of the industry [37, 10, 14].

2 Network reconstruction using tomography

In this section we describe how to use data sensed from the edge of the network (servers) to reconstruct key performance measures in a data center network (DCN). Figure 1 shows a 3-stage Clos (or fat-tree) DCN, with the path taken by a probe or data packet shown in red. We introduce the following concepts and terminology.

Probe. A probe is a 64 byte UDP or TCP packet sent from one server to another in the DCN. It travels on the same priorities and through the same queues as regular data packets. Its purpose is to incur the same queuing times as the payload in its priority and, hence, provide data for reconstruction.

Probe mesh. This is a graph connecting the N servers in a DCN to one another along whose edges probes are exchanged. Each server picks K other servers randomly and uniformly (without replacement) from the set of all servers. Suppose server i picked servers j_1, \dots, j_K . Then i sends probes to each j_l ($1 \leq l \leq K$) at a frequency F Hz. Each j_l sends probes back to i at the same frequency. The path in the DCN followed by probes between any pair of servers is cho-

sen uniformly at random from the set of all available paths between them, and independently of the choice of paths for other pairs.⁴ The paths are chosen once and held fixed. Note that each server sends a total of $2KF$ probes per second on average.

Remark. We shall later demonstrate (Section 3) that with $10 \leq K \leq 20$ we can sample all the links (hence queues) in any *Clos (fat-tree) DCN*; that is, a constant value of K suffices. We shall also comment on F .

Handling Ethernet priorities. There are 8 priorities in Ethernet, the dominant L2 technology in DCNs. For convenience, in this paper we assume that the DCN uses only one priority, although the verification in the 40 Gbps testbed was conducted in a multi-priority setting. In order to handle multiple priorities, we simply launch a probe mesh in each priority and perform reconstruction per priority. The probes will use the same transport protocol (TCP or UDP) as the traffic in the given priority and encounter the same queuing delays as the payload. Indeed, the 40 Gbps testbed also employs priority flow control (IEEE 802.1Qbb), where packets in a priority may be paused. Reconstruction works in this case as well.

Individual record. The individual record of a probe or a data packet is captured at the transmitter network interface card (NIC) and the receiver NIC. It consists of the 5-tuple header information (source and destination addresses, source and destination port numbers, protocol port number), the transmit and receive timestamps at the corresponding NICs, and the length of the packets.

Remark. We assume that the clocks of all the NICs are accurately synchronized using techniques in [24], [31] or [33]. Note that, even though these techniques can synchronize clocks up to a 10s of nanoseconds, it suffices for our purposes that the clocks be synchronized to about 1 microsecond.

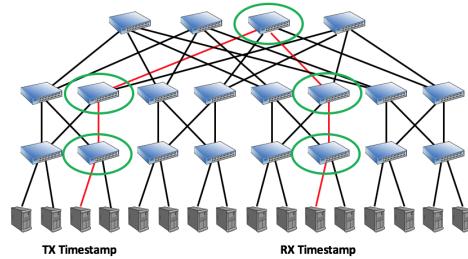


Figure 1: Reconstruction from Edge Timestamps

Output-queue assumption. Switches in DCNs can have queues on both the ingress and egress line cards and packets can queue at both places. However, switch implementations

⁴The paths of the probe mesh do not have to be chosen at random; indeed, it may sometimes be desirable to choose the path deliberately so as to cover the DCN’s links more evenly or unevenly, depending on some objective. Further, probes from i to j can follow a different path in the DCN than probes from j to i . Whatever the choice, the paths of the probe mesh must cover all the links in the network uniformly, and cover all the “linearly independent paths” with an adequate number of probes passing through each link per unit time so as to enable a good reconstruction of that link. Random path selection is adequate for these purposes.



Figure 2: The network in ns-3 simulation setup (NS)

employ a fabric speedup [16], ensuring that most queuing takes place at the egress line cards. Throughout this paper we assume that queuing takes place in the output queues of switches, and measurements done in a real testbed in Section 6.2 validate this assumption. Hence, each directed link in the DCN has a queue associated with it; namely, the output queue that drains into it.⁵

2.1 What to reconstruct

Suppose we have a DCN with a total of N^q queues. Under some traffic load, let $Q_i(t)$ be the queue length at time t and $W_i(t)$ be the waiting (or queuing) time of a packet arriving at time t to the i^{th} queue. With a single priority $W_i(t) = Q_i(t)/L$, where L is the line rate in Bytes/sec, so the waiting time and queue size are related by a constant.⁶ Consider a preliminary statement of the reconstruction problem:

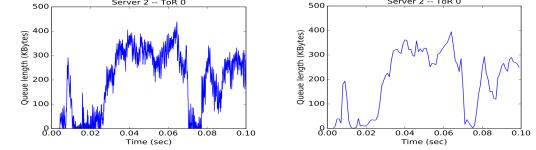
Given the individual records of all the probe packets during some time $[0, T]$, obtain a reconstruction $\hat{Q}_i(t)$ of $Q_i(t)$ so as to minimize $\mathbb{E} [Q_i(t) - \hat{Q}_i(t)]^2$ for $t \in [0, T]$.

ns-3 simulation (NS) setup. To gain an understanding of what is involved in solving this problem, let us consider an ns-3 simulation of a 10 Gbps, 3-layer, 256-server network. We shall reuse this simulation set up in the next section as well; hence, we shall refer to it as the NS setup. This network has 32 ToR switches (8 servers per rack), 32 spine layer 1 switches and 8 spine layer 2 switches, connected as shown in Figure 2. All switches are output-queued and have 1MB of buffering per queue. The DCN operates under an incast traffic⁷ load of 40%. We have also used a “long flow” workload at 40% load, where one file is sent from a server to another randomly chosen one, and the file sizes are uniformly in [10MB, 1GB] with an average of 505MB.

⁵Note that the only directed links with which no queues need to be associated are those connecting a NIC to a top-of-the-rack (ToR) switch. These links don't get oversubscribed.

⁶Note that the inference equations will be in terms of $W_i(t)$, as in equation (3). The queue wait times in each priority come from the probes in that priority. We're looking at the queue lengths here rather than queue wait times because the lengths vary directly with packet arrivals and departures.

⁷The incast traffic pattern is taken from DCTCP [6]. Each server maintains a certain level of load by requesting data simultaneously from a random number (30% 1, 50% 2 and 20% 4) of other servers, referred as the “fanout” of the request. The gap between adjacent requests are independent exponentials with load-dependent rate. The file sizes are distributed according to a heavy-tailed distribution in the range [10KB, 30MB] with an average file size of 2.4MB. We have used incast workload in this paper because it captures some generic scenarios (long file transfers and short RPC-type traffic) and causes congestion. In deployments we've used map-reduce-type batch workloads and RDMA-type traffic.



(a) Queue sampled every $1\mu s$ (b) The 1ms average
Figure 3: Queue length in a 10Gbps network

$Q_i(t)$, measured every microsecond, is plotted in Figure 3a. (Note that packet enqueueing/dequeueing times on a 10G link are close to 1 microsecond.) The 1 ms-averaged version of $Q_i(t)$, $\bar{Q}_i(t)$, in Figure 3b) is obtained by averaging the 1000 consecutive values of $Q_i(t)$ in each millisecond. The two graphs are essentially identical; the $Q_i(t)$ is a noisy version of $\bar{Q}_i(t)$. We shall next show that there is a straightforward relationship between $Q_i(t)$ and $\bar{Q}_i(t)$; indeed, $\bar{Q}_i(t)$ has almost all of the *signal power* in $Q_i(t)$. In Section 3 we present a method for reconstructing $\bar{Q}_i(t)$ using the LASSO algorithm.

Averaging: empirical evidence. Define the autocorrelation function of $Q_i(t)$ as follows:

$$R_Q(\tau) = \mathbb{E}[Q_i(t + \tau)Q_i(t)], \text{ for } \tau \geq 0. \quad (1)$$

The autocorrelation function captures the rate of decay of correlations in $Q_i(t)$. The power spectral density (PSD) of $Q_i(t)$ equals⁸

$$S_Q(f) = \sum_{\tau=-\infty}^{\infty} R_Q(\tau)e^{-i2\pi\tau f}, \text{ for } |f\tau| < 1/2, \text{ or } |f| < 0.5 \text{ MHz}. \quad (2)$$

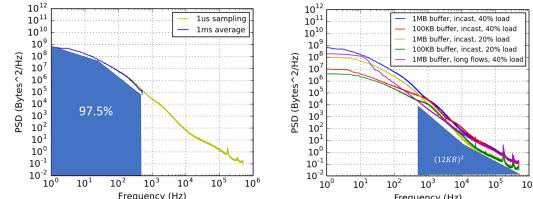
The PSD shows the amount of power in $Q_i(t)$ at different frequencies and is symmetric for positive and negative values of f in the range over which it is defined ($-0.5 \text{ MHz} < f < 0.5 \text{ MHz}$ in this case).

Plotting the PSD as a function of the frequency in Hz for $0 < f \leq 0.5 \text{ MHz}$, we get the yellow curve in Figure 4a. By computing the PSD similarly for $\bar{Q}_i(t)$ and plotting it, we get the blue curve in Figure 4a. The blue curve exactly coincides with the yellow for $-500 \text{ Hz} < f < 500 \text{ Hz}$, which means that the power at those frequencies in both $Q_i(t)$ and $\bar{Q}_i(t)$ are *identical*. Further, $\bar{Q}_i(t)$ has zero power in frequencies higher than 500 Hz; that is, averaging only removes the high frequency “noise” and preserves most of the power, 97.5% to be precise, of the microsecond-level signal $Q_i(t)$.

Remark. It is worth noting that the percentage of preserved signal power depends on the strength of the signal itself. The 97.5% number is obtained at 40% load with 1 MB switch buffers. Figure 4b shows the PSD measured with different network load and switch buffer size combinations. As can be seen, although the amount of power preserved after averaging (the low frequency signal power) varies, the power of the removed high frequency component remains constant: at $(12\text{KB})^2$ over all frequencies higher than 500 Hz.⁹

⁸Note that τ represents a 1 us interval; hence $1/\tau$ is 1 MHz.

⁹At the moment we do not have an explanation about this empirical ob-



(a) In a network with 1MB buffer, 40% incast load
(b) In different networks and with incast and long flows workload

Figure 4: PSD of queue depths in 10Gbps networks

As empirical evidence from a real deployment, Figure 14 compares the PSD of the queue size process measured using NetFPGA in a 1 Gbps testbed with shallow-buffered switches. As discussed below, the averaging interval is a function of only the line rate and for a 1 Gbps line it is 10 ms. In Figure 14 we again see that at 40% load the PSD of the averaged queue process coincides with the PSD of the queue size process. Again the power of the portion that is filtered out, the “noise”, is $(12KB)^2$.

Averaging: signal processing explanation. $\bar{Q}_i(t)$ is obtained precisely by passing $Q_i(t)$ through a low-pass filter with the pass band equal to $[-500 \text{ Hz}, 500 \text{ Hz}]$ [41]. This is easy to understand from Figure 5: $\bar{Q}_i(t)$ is obtained by (i) computing a running average of $Q_i(t)$ over a 1 ms window, and then (ii) down sampling the running average to 1 sample per ms. The frequency domain transfer functions ($H(f)$) and ($X(f)$) corresponding to (i) and (ii) are shown in the same figure. Essentially, (i) and (ii) precisely amount to removing the frequency components in $Q_i(t)$ over 500 Hz and preserving the rest.

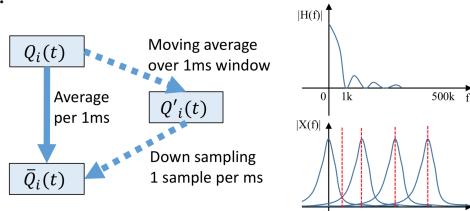
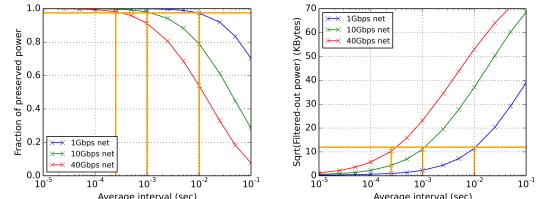


Figure 5: The millisecond-averaging process in time and frequency domains

Upshot. As a result of the above discussion, we shall reconstruct the averaged queue process $\bar{Q}_i(t)$ for each i rather than the packet-time-sampled queue process $Q_i(t)$. It remains to specify the “reconstruction interval”; we proceed to do this next.

Determining the reconstruction interval. The proper reconstruction interval for a given DCN turns to depend only on the *highest line rate* in the network, even when there are

servation that the removed noise power equals $(12KB)^2$. In the rest of the paper we use this $(12KB)^2$ to define the acceptable signal power loss caused by averaging and determine the averaging interval so as to achieve no more than this loss.



(a) The fraction of queue size power preserved
(b) The “noise” power or the power “filtered-out”

Figure 6: Power of queue size and noise processes at different reconstruction intervals

mixed link speeds. Intuitively, the faster the link speeds, the quicker the queues vary (or the higher the noise frequency), and the smaller is the reconstruction interval.

Figure 6a shows the percentage of power preserved by using different reconstruction intervals in networks with different link speeds at 40% load. As shown, to preserve more than 97.5% of the power, the reconstruction interval is 10 ms for 1Gbps links, 1 ms for 10 Gbps links and 250 μ s for 40Gbps links. In other words, the reconstruction interval is inversely proportional to the maximum link speed of the network. Figure 6b shows the square root of the power filtered-out at different reconstruction intervals. Remarkably, as seen in the figure, this value is a constant at 12KB *at all link speeds* when the reconstruction interval equals 10 ms, 1 ms and 0.25 ms for 1G, 10G and 40G links, respectively. In other words, the “noise” power is $(12KB)^2$.

In summary, we shall aim to reconstruct the average queuing times for all the queues in the switches for each reconstruction interval, which is determined by the maximum link speed of the network. The reconstructed average queues removes the high frequency noise and preserves most of the relevant information in the original queue signal.

3 The reconstruction algorithm

We now describe SIMON; specifically, we describe

- (1) a system that uses the individual records of probes to reconstruct the averaged queue or wait time processes using LASSO, and
- (2) uses the above and the individual records of data packets to determine link utilizations as well as queue and link compositions.

Preliminaries. Given a DCN, the first steps are to set up a probe mesh and to choose a reconstruction interval, I . Consider all the probes sent by all the servers in the reconstruction interval I (recall that we assume the clocks are all accurately synchronized). Let t_i^p , $i = 1, \dots, N^p$ be the ordered sequence of transmit timestamps of all the probes (regardless of source) sent in interval I , where N^p is the total number of probes transmitted in I . Let r_i^p be the receive timestamp of the probe transmitted at t_i^p . Note that some r_i^p may fall out-

side I ; this is fine.¹⁰ Set $D_i^p = r_i^p - t_i^p$ to be the one way delay of probe i . Let t_j^d, r_j^d, D_j^d and N^d be the corresponding quantities for data packets transmitted in interval I . Denote by D^p and D^d the $N^p \times 1$ and $N^d \times 1$ the vectors of one way delays of all the probe and data packets transmitted in interval I , respectively.

By the output-queue assumption, preceding each link in the network we imagine there is a queue in each direction of the link. Recall that there are a total of N^q queues (counting both directions at each link). Let $q_k(I)$ denote the average queue size process at queue numbered k in interval I . Since we've fixed I , we abbreviate $q_k(I)$ to q_k . Similarly let w_k be the waiting time of a packet in queue k in interval I (recall $w_k = q_k/L$, where L is the line rate in Bytes/sec). Let Q and W denote the $N^q \times 1$ vectors of queue sizes and wait times in interval I , respectively.

Recall that we have assumed that the path taken by a probe or a data packet is knowable from its header information, hence each probe or data packet visits a particular sequence of queues as it traverses the DCN. This gives rise to the following linear equations:

$$\begin{bmatrix} D^d \\ D^p \end{bmatrix} = \begin{bmatrix} A^d \\ A^p \end{bmatrix} W + \begin{bmatrix} Z^d \\ Z^p \end{bmatrix} \quad (3)$$

where A^d and A^p are, respectively, the $N^d \times N^q$ and the $N^p \times N^q$ 0-1-valued *incidence matrices*, identifying the set of queues visited by each probe and data packet; and Z^d and Z^p model noise due to various factors such as (i) faulty NIC timestamps, (ii) differences in propagation times on cables of different lengths, (iii) variations in the switching times at different switches, etc. The queuing times are typically at least a few microseconds and typically they are 10s or 100s of microseconds. By comparison, the noise is in the order of 10–100 nanoseconds.

Setting

$$D = \begin{bmatrix} D^d \\ D^p \end{bmatrix}, \quad A = \begin{bmatrix} A^d \\ A^p \end{bmatrix}, \quad \text{and} \quad Z = \begin{bmatrix} Z^d \\ Z^p \end{bmatrix},$$

equation (3) becomes

$$D = AW + Z. \quad (4)$$

The Reconstruction. Given the vector (D^d, D^q) , we are interested in getting an estimate, \hat{W} , of W . A natural criterion for the goodness of the estimate would be to minimize the mean squared error, $\mathbb{E}[(W - \hat{W})^2]$, where the expectation is over the noise. Typically, in a fat-tree network the number of queues with a positive delay is quite small even under high load; that is, W is typically a *sparse* vector (see [9]; we have also observed this in all our real-world experiments).

¹⁰Of course, probes may be dropped. We choose to ignore such probes and find that we obtain very good reconstruction results with the remaining probes, since these latter probes accurately capture large queue sizes and wait times are accurately captured from the other probes. Moreover, since probes are only 64 Bytes, they're much less likely to be dropped than data packets which are typically in the 500–1500 Bytes range. However, dropped probes convey valuable information about congestion and it is worthwhile to treat them specially. Due to a lack of space, we don't explore this aspect further in this paper.

It is not hard to show that any DCN with a multi-stage Clos topology interconnecting the servers has an adjacency matrix whose rank is less than the number of links, hence queues. This follows as a generalization of the following lemma, whose proof is simple and is omitted.

Lemma 1. Consider all equations in the $m + n$ variables A_1, \dots, A_m and B_1, \dots, B_n of the type $A_i + B_j$. This system of equations has a maximum rank equal to $m + n - 1$.

The previous discussion shows that the rank of the matrix A is less than N^q , hence equation (4) is underdetermined. The statistical procedure LASSO [53] is naturally suited for our problem. Thus, we seek

$$\hat{W} = \arg \min_W \|D - AW\|_2^2 + \alpha \|W\|_1, \quad (5)$$

where $\alpha > 0$ is a scalar multiplying the regularization term $\|\cdot\|_p$ is the standard L_p norm. We motivate the eventual solution by proceeding through the following simpler and instructive cases.

Case 1: Consider only the data packet timestamps. Disregard the probe packets and consider the equation $D^d = A^d W + Z^d$. We generate the data using the NS setup described in Section 2.1.

We solve

$$\hat{W} = \arg \min_W \|D^d - A^d W\|_2^2 + \alpha \|W\|_1$$

and compare the solution with the actual value of W (ground truth). The comparison is shown in Figures 7 and 8 below. The solid blue line is the ground truth and the red filling is the LASSO solution. Figure 7 shows a fairly accurate reconstruction at one queue. However, Figure 8 shows the LASSO algorithm has misattributed the queue sizes amongst the 3 queues shown in the figure. Essentially, what is going on is that the equations provided by the data packets yield an underdetermined linear system (not enough equations for the variables). Therefore, not all queues are correctly reconstructed.

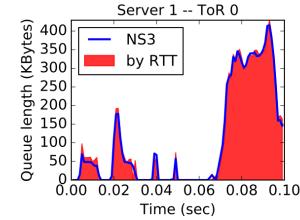


Figure 7: Reconstruction with data packets: good case

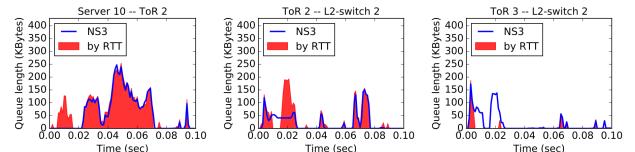


Figure 8: Reconstruction with data packets: bad case

Case 2: Consider only the probe packet timestamps. This time disregard data packets and consider the noisy linear sys-

tem

$$D^P = A^P W + Z^P. \quad (6)$$

We need to solve the inverse problem:

$$\hat{W} = \arg \min_W \|D^P - A^P W\|_2^2 + \alpha \|W\|_1.$$

The crucial difference between this case and Case 1 is that in Case 1 we cannot choose A^d but we can certainly choose A^P . We will say more about how to choose A^P in Section 3.1. For now, suppose each server probes to 10 ($K = 10$) other servers; thus, in every reconstruction interval, an average of $2 \times 2560 = 5120$ probes are sent.

As seen in Figure 9, the algorithm reconstructs all the queues very precisely with probe packet data. Because the probe mesh covers all the links fairly uniformly, the system of linear equations along with the sparsity constraint (captured in the L_1 regularizing term, $\|W\|_1$) determine the correct solution. Statistically, when compared with the ground truth, the algorithm achieves a root-mean-square error *across all the queues* of 5.2KB; that is, $RMS(\hat{Q} - Q) = 5.2KB$.¹¹

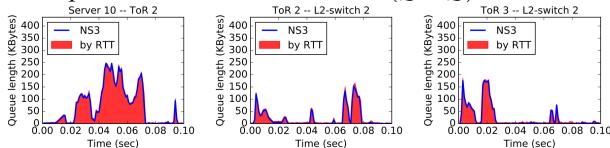


Figure 9: Network reconstruction with probe packets

3.1 Specifying the parameters of the probe mesh

As the previous reconstruction scenario has revealed, using a random graph for the probe mesh with $K = 10$ produces good reconstructions. A natural question is what is the right value of K to get a good reconstruction quality and does this depend on the DCN topology. We proceed by refining our definition of the probe mesh introduced earlier.

Definition: Probe graph. Let $G = (V, E)$ denote a graph with V equal to the servers in a given DCN, and $(i \rightarrow j) \in E$ if i probes j . G is called the *probe graph*.

Remark. Note that the above definition does not include the topology of the underlying DCN.

Definition: DCN probe mesh. Given the probe graph $G = (V, E)$ and a DCN, the *DCN probe mesh*, G_D , is the graph obtained by assigning DCN paths to probing edges $(i, j) \in E$. Note that the adjacency or incidence matrix of G_D is exactly equal to A^P defined by equation (3).

Observation 1. In order to reconstruct the waiting time or queue size of any queue in the DCN, a *necessary condition* is that queue is present on some path of G_D or, equally, that the column in A^P corresponding to that queue is not identically

¹¹Note that we use the standard definition of *RMS*; i.e., $RMS(Q) = \|Q\|_2$. We choose to show the reconstruction plots and report the errors for the queue size vector Q rather than for the delay vector W since we have found that the queue sizes are usually in the order of a few 100 KBs regardless of the line rate whereas the waiting times vary with the line rate.

zero. While this condition is necessary, it is not sufficient since it doesn't guarantee solvability of equation (6).

Observation 2. The maximal condition for the solvability of equation (6) is that the rank of A^P is at its maximum. Then the LASSO algorithm along with the sparsity constraint enforced by the L_1 regularizing term will produce a unique solution.

In order to understand conditions on K which give us an A^P with full rank, we simulate a variety of different DCNs and vary K to see how that affects the *RMS* error of queue size vector, Q . Figure 10 shows that the reconstruction accuracy greatly improves with K initially and then it tapers off. More importantly, the figure suggests that this relationship holds for several different network topologies and types of queue (the layer in the DCN the queue is at).

Clearly, even though higher values of K give better reconstruction quality, there is a penalty for setting K large: servers need to issue more probes, there is the overhead of timestamping, data collection and storage, and there is the effort of reconstruction. Quantitatively, a value of $K \in [10, 20]$ seems to achieve the best reconstruction quality while representing a small enough effort.

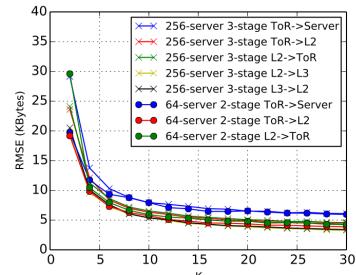


Figure 10: Reconstruction quality vs K

We provide an intuitive explanation of why a constant value of $K \in [10, 20]$ is sufficient to reconstruct the queues regardless of the size of the network, and give a probabilistic argument on the queue coverage probability in the appendix. As the size of network grows, the number of servers grows, hence the total number of probes will grow (for a fixed K), resulting in a constant sampling rate of each queue. To quantify the foregoing under a “full bisection bandwidth assumption” on the DCN, suppose we have N servers in the DCN. If we cut the network into two sets of servers, S_1 and S_2 , respectively with N_1 and N_2 servers (WLOG we assume $N_1 \leq N_2$), then the total number of links crossing the boundary of the two sections is N_1 (because of full bisection bandwidth). Since each server probes K servers chosen at random, the expected number of probes and echoes passing these N_1 links from S_1 to S_2 and from S_2 to S_1 each equal $\frac{2N_1 N_2 K}{N}$. Thus, every queue on the boundary is sampled on average by $\frac{2N_2 K}{N} \geq K$ probes. Therefore, as long as $K \geq 10$, we can guarantee each queue is sampled by at least 10 probes.

A final observation about Figure 10: the RMSE appears to drop like $\frac{1}{\sqrt{K}}$. Under the assumption that each probe only visits one congested queue (recall that DCN congestion is sparse), this can be explained as follows. If K probes pass through a queue with a reconstruction-interval-average size of Q and make independent noisy observations with variance σ^2 (where $\sigma^2 \approx (12KB)^2$), then the RMSE roughly equals σ/\sqrt{K} , which for $\sigma = 12KB$ and $K = 10$ approximately equals 4KB, as seen in Figure 10.

3.2 Link utilization and queue/link compositions

We're now interested in solving for all the performance measures, not just queue sizes or wait times. In particular, probes can only detect positive wait times; however, zero wait times don't imply absence of traffic! The link utilization may be too small to cause queueing delays. In order to determine link utilizations and the composition of queues and links (i.e., breaking down queue and link occupancies in a reconstruction interval according to flow ids), we have to use both the probes and data packets. We explain how to do this algorithmically, following the steps below.

Step 1. Solve Case 2 and obtain the (reconstruction interval average) queuing sizes and wait times, $(Q, W) = (q_k, w_k), 1 \leq k \leq N^q$.

Step 2. Use W to determine the position of a probe or data packet at each instant from transmission to reception. As an example, consider probe i , transmitted at time t_i^p and received at time r_i^p . Suppose this probe visited queues with delays $w_{k_1}, w_{k_2}, \dots, w_{k_L}$. Note that $r_i^p - t_i^p \approx w_{k_1} + w_{k_2} + \dots + w_{k_L} + P$, where P represents the total time probe i spent on all the links on the path. At time $t \in (t_i^p + w_{k_1}, t_i^p + w_{k_1} + w_{k_2})$ we assert that probe i is either in the second queue on its path or on the link connecting the first and second queues depending on how much larger than $t_i^p + w_{k_1}$ is t . We can also similarly determine the position of each data packet. Note that extra care must be taken to account for the time spent by a data packet in a queue or on a link since its length can be much larger than that of the probe packets.

In this manner we can decompose the contents of a queue into traffic segments of interest and specified by headers. We can similarly also obtain the link utilizations and link compositions.

To see how well the above procedure works, we again look at the NS simulation setup from Section 2.1. Figure 11 shows some examples of queue and link utilization compositions, comparing them to the ground truth taken from ns-3. There are three sets of plots, left, middle and right. Each set has two plots: a queue size plot on top and a link utilization plot on the bottom. The figure on the left shows the utilization of the link connecting L2-switch 8 to L3-switch 0 on the bottom and the size of the queue attached to this link on top. We see the reconstructed queue occupancies of the blue, yellow

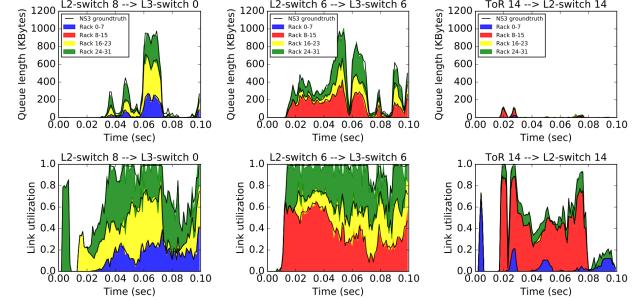


Figure 11: Reconstructing queue size and link utilization compositions

and green packets shown by the filling and the corresponding ground truth shown by the black lines. Here the blue, yellow and green packets are destined for servers in racks 0-7, 16-23 and 24-31, respectively. It is worth noting that when the link utilization nears 100%, the size of the corresponding queue shoots up; otherwise, when the link utilization is well below 100%, there is no queue buildup. The other two sets of plots show other links.

As can be seen, the SIMON algorithm does an excellent job of determining the queue size and link utilization compositions. Statistically, the reconstruction precisions (RMSE) of the queue size and link utilization composition for each class of traffic are 4.14KB and 1.01% (i.e. 0.101 Gbps in 10Gbps net), respectively.

Method	RMSE of composition		Storage space	Run time ¹²
	Queue	Link util		
Per packet	4.14KB	1.01%	2.84MB	4.00ms
100us count	4.12KB	0.98%	172.3KB	0.277ms
250us count	4.08KB	0.94%	98.4KB	0.173ms
500us count	4.19KB	1.01%	69.4KB	0.142ms
1ms count	4.71KB	1.81%	49.5KB	0.107ms

Table 1: Use byte counts to decompose queue and link utilization. Space and time are for 1 reconstruction interval.

The effort—space and computation time—needed for the link utilization and queue/link compositions can be vastly simplified if we count the number of Bytes sent by each 5-tuple in Step 2 of the algorithm rather than take the transmit timestamps for each data packet. We increase the interval over which the byte counts are taken from 100 us to 1 ms while tracking the reconstruction quality. The results are shown in Table 1. As can be seen in the table, using byte counts over a 500us interval, the space consumption is reduced by 41x and the run time is reduced by 28x with almost no change to the reconstruction quality.

¹²C++ single-thread program on an Intel Core i7 processor

4 Speeding up the implementation of SIMON

The LASSO algorithm optimizes a convex cost function over a large number of variables using gradient descent. Because it is iterative, for large problem instances, LASSO can take time. Even though reconstruction can be done offline and still be quite valuable, it is interesting to ask if reconstruction can be done in near-real time and, if so, how this would be possible. This would enable rapidly detecting bottlenecks and anomalies, raising alerts, as well as enabling sensitive A/B tests in near-real time.

We explore two ideas to accelerate SIMON in large networks. Section 4.1 decomposes the reconstruction problem to a hierarchy of subnets that can be solved individually and in parallel. Section 4.2 uses neural networks to function approximate LASSO and obtain significant speed improvements. These two methods allow us to scale SIMON to large data centers with tens of thousands of servers.

4.1 Hierarchical reconstruction

The fat-tree has become the de facto data center networking topology [5, 47]. The hierarchical nature of the fat-tree topology allows us to propose an algorithm that groups queues in layers and perform reconstruction from lower layers to upper layers. The reconstruction for each layer may contain a number of sub-reconstructions that can run in parallel. After queues at lower network layers are reconstructed, their sizes will be used to reconstruct higher-level queues.

As an example, we illustrate the hierarchical reconstruction algorithm using the topology in Figure 2. For notational simplicity, we use L_i blocks to denote the forest after truncating the tree at layer i ; that is, the connected components of the subgraph containing only switches of layer i or lower, and the servers. In Figure 2, there are 16 L1 blocks, 4 L2 blocks and 1 L3 block. We also define L_i probes to be the probes whose source and destination are within the same L_i block. The procedure is as follows:

1. We extract the equations given by the L1 probes. Each L1 block (rack) provides a system of linear equations, so there will be in total 16 of them. By solving the 16 linear systems, we reconstruct all queues in the L1 blocks (downlink of ToR switches).
2. We then reconstruct queues in the L2 block—the ToR uplink queues and the L2 switch downlink queues—using L2 probes. Each L2 probe passes through 3 queues: one ToR uplink, one L2 downlink and one ToR downlink. Since the ToR downlinks have been solved in the previous step, each L2 probe now provides an linear equation of the other two remaining queues. There are four linear systems for every L2 block. The tier 2 queues can then be reconstructed by solving the 4 linear systems.
3. Lastly, we reconstruct the L3 queues with L3 probes,

where we repeat the previous step. After solving L3 queues, all queues in the network would have been reconstructed.

#servers	#network layers	LASSO		Hierarchical LASSO	
		RMSE	Run Time	RMSE	Run Time
256	3	5.22KB	37.34ms	9.47KB	0.59ms
2048	4	5.76KB	532.60ms	10.3KB	1.25ms
4096	4	5.53KB	1147.22ms	10.69KB	1.75ms

Table 2: RMSE and Run Time Comparison

Performance. Table 2 show that LASSO can be sped up by several orders of magnitude, especially as the network size increases with a modest sacrifice of accuracy.

4.2 Accelerating reconstruction with neural networks

In this section we explore using the function approximation [29] capability of multi-layered neural networks to speed up LASSO and rapidly estimate the vector of queue sizes, \hat{Q} , from the vector of end-to-end probe delays, D . The goal is to function-approximate LASSO and learn the underlying function mapping D to \hat{Q} from the training examples.¹³ Since the neural network only involves simple arithmetic operations which can be implemented at high speed on modern hardware such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), we can potentially get tremendous speedups.

We take 60 seconds of data at 1 ms intervals from the NS scenario. The data are 60,000 (D, Q) pairs where each D is a 5120-dimensional vector and each Q is a 1280-dimensional vector (to be more specific, we consider D^p and not D^d ; for ease of notation, we shall denote D^p as D). We also use LASSO’s inferred queue sizes, \hat{Q} , following the reconstruction procedure in Section 3 and use 60,000 (D, \hat{Q}) pairs to train the neural network. This allows us to compare two training methods: one with the ground truth from ns-3 and the other is the output of LASSO.

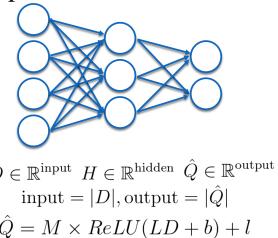


Figure 12: Neural network for network reconstruction

We use these two sets of data to train two instances of a ReLU (Rectified Linear Unit [39]) neural network with just 1 hidden layer, as shown in Figure 12. At the end of the training period, each neural network has “learned” the matrices

¹³Note that LASSO is a general statistical procedure; the fact that it may be function-approximable in this context is not an indication that it would work in other cases. Our estimation problem is special: we’re solving a system of noisy linear equations where the noise is additive. It may be harder to function approximate LASSO in nonlinear systems and/or with multiplicative noise.

L and M and the bias vectors b and l from their respective input-output pairs. We then test each neural network with new values of D to see how well the corresponding output agrees with the ground truth Q . The results are shown in Table 3. As can be seen, both neural networks perform very well when compared to LASSO; indeed, the neural network trained with the ground truth does better than LASSO.

RMSE (Bytes)	LASSO	Neural Net (trained w. GT)	Neural Net (trained w. LASSO)
256 server	5.22KB	4.15KB	5.34KB
2048 server, 4 stage	5.76KB	4.03KB	7.53KB
4096 server, 4 stage	5.53KB	3.95KB	7.08KB

Table 3: RMSE Comparison

Table 4 compares the speed of the neural networks with that of LASSO. The profiling of LASSO was run with Python Scikit-learn sparse LASSO package, on a server with 6-core Intel I7 processor, and the neural network profiling are done on an Nvidia GTX1080 GPU using Tensorflow¹⁴. The neural network is three magnitudes faster than LASSO during inference time due to the algebraic-simplicity of its forward pass and speed-up with hardware accelerators.

Run Time	LASSO (single core)	LASSO (6-core)	Neural Net
256 server	37.34ms	6.22ms	2.75us
2048 server, 4 stage	532.60ms	96.14ms	76.24us
4096 server, 4 stage	1147.22ms	208.25ms	235.97us

Table 4: Speed Comparison

There are two major points to make about using neural networks for data center measurement.

The first is to note that whereas LASSO requires the adjacency matrix A^q , the neural networks don't require it. This point and the abundance of operational data in modern data centers strongly suggest that neural networks can play a major role as a lightweight, very high speed system for detecting anomalies and even accurately estimating key performance measures.

For the second, consider the scenario where the network topology changes or the ECMP hash functions change (e.g., due to link failures) in the network. In modern DCNs that support software-defined networking (SDN), such changes can be known when the SDN controller receives corresponding events. In these cases, the D , Q relationship would change, which could make the already trained neural network obsolete. Thus, it may be required to re-train the neural network. However, naively employing neural networks for large data centers implies very long re-training time. Taking the 4096-server case as an example, even though training the neural network only requires 288 seconds worth of timestamp data, it takes 8 hours for the weights in the neural network to converge.

¹⁴We used the best off-the-shelf implementation we can find for both LASSO and NN. Our effort to speed up LASSO with GPUs using the SHOT-GUN algorithm did not result in much gain. LASSO iteratively optimizes a convex cost function using gradient descent, a task better-suited for CPUs than for GPUs.

In practice, we would use neural networks in combination with hierarchical reconstruction. This would not only speed up retraining (because subnets can be retrained in parallel), but it would also make the whole process more robust to failures (since only subnets affected by failure need to be re-trained). For example, retraining a 256-server subnet only requires ten seconds worth of (D, \hat{Q}) pairs and converges in 2.5 minutes on a Nvidia GTX1080 GPU.

5 Implementation

We implement the LASSO version¹⁵ of SIMON for Linux. This implementation has three components: the prober, the sensor and the reconstruction engine. The prober is implemented as a user space program. There is one prober sitting on each of the servers in the network. Each of the probers repeatedly probes K other random probers and echoes received probes. The sensor is implemented in the NIC driver and is in charge of collecting hardware transmit and receive timestamps for both data packets and probes. Then, upon reconstruction requests, the sensors will batch the time stamp data for the requested time range and ship the data to the reconstruction engine.

The reconstruction engine is a Spark Streaming [48] cluster. It takes timestamp data over a given time range as input and outputs the reconstructed queue lengths, link utilizations and their per-flow break downs for each reconstruction interval in that time range. The reconstruction engine exploits the fact that the data processing for different reconstruction intervals are independent. Upon the arrival of the timestamp data at the reconstruction engine, the data is originally sharded by server IDs since individual servers collected the data. Then the engine will re-shard the data into reconstruction intervals, and assign each interval to one of the servers in the cluster. Finally, after the reconstruction for each interval is done, the results are stitched together and returned to the user.

6 Deployment and Validation

This section uses three very different testbeds to verify that SIMON works with a wide variety of network configurations and under real-world complexities. The testbeds have link speeds ranging from 1 Gbps to 40 Gbps, contain 2 to 5 stages of switching fabric, and employ switches from different vendors and models. Furthermore, we discuss some pragmatic experience gained from these deployments, namely, how to still output meaningful reconstruction results with only partial probing coverage of the network. We use NetFPGAs to verify the accuracy of SIMON in the 1 Gbps testbed. Finally, we use cross-validation to verify and evaluate the correctness of the reconstruction results in the absence of ground

¹⁵Hierarchical reconstruction and neural networks are planned to be implemented for production environments in the future.

truth data.

We first specify the configuration of the three testbeds.

Testbed T-40G-3L. This is a 3-stage Clos network, all links are 40Gbps. It has 20 racks with roughly 12 servers per rack. There are 32 switches at both spine layers.

Testbed T-10G-40G-5L. This is a 5-stage Clos network containing ~ 500 racks, each of which has ~ 24 servers. Each server is connected to a ToR switch with two 10G aggregated links. The rest of the links are all 40G. We had access to 12 out of the 500 racks.

Testbed T-1G-2L. This is a 2-stage Clos network with all 1 Gbps links. It consists of 8 racks, each rack has 16 servers. There are 8 spine switches. The switches in the testbed are Cisco 2960s and Cisco 3560s.

6.1 Deployment experience

SIMON makes minimal assumptions about the network and only relies on data sensed at the NICs, hence its deployment has generally been smooth. SIMON needs two things: (i) hardware timestamping-capable NICs and (ii) the ability to know the paths taken by the packets. For (i), almost all current generation 10G or above NICs can timestamp all ingress and egress packets. In T-1G-2L, even the on-board 1G NICs support it. For (ii), since per-flow ECMP is the standard load-balancing scheme used by the industry, knowing the paths taken by the packets is not difficult. In T-40G-3L and T-10G-40G-5L, we ran traceroute [36] periodically to learn the paths taken by any targeted flows. In T-1G-2L, since we use 5-tuple-based source routing, we automatically know the paths taken by any packets basing on their 5-tuples.

Despite the overall smoothness of the deployments, one case that did require some extra care was when two or more queues cannot be further disambiguated by the probes. For example, in our deployment in T-10G-40G-5L, since we only had access to a small portion of the racks in a big network, our probes can only reach a subset of all the queues in the network. Within these reachable queues, some of them are always bundled together as seen by the probes. That is, a probe either traverses all of these queues or none of them. In this case, we treat the sum of these queues as a single virtual queue, and treat this virtual queue as a single variable in Equation 3.

6.2 Validating accuracy with NetFPGAs

We use NetFPGAs to validate the precision of SIMON in T-1G-2L. Figure 13 shows the setup. We configure the four ports in a NetFPGA, P_0 , P_1 , P_2 , P_3 , as two pass-through timestampers: P_0 and P_1 are connected back-to-back via the FPGA, and packets passing through these two ports will be timestamped according to a local counter in the FPGA; P_2 and P_3 are similar except the timestamp is inserted into the packet at a different offset. We connect the two timestampers to two different ports of a switch, S_0 and S_1 .

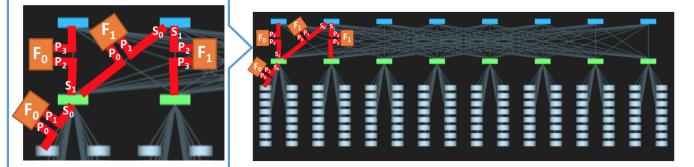


Figure 13: Setup of reconstruction validation in T-1G-2L. F_0 and F_1 are two NetFPGAs.

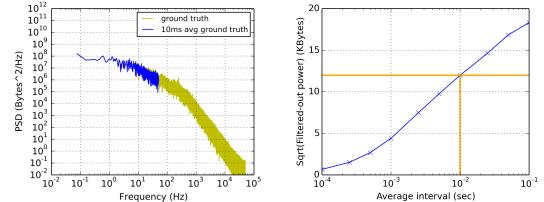


Figure 14: PSD of queues with NetFPGA ground truth at 40% load

pers to two different ports of the same switch, S_0 and S_1 . For all packets between S_0 and S_1 , we will know their overall switching and queueing delays in the switch from the NetFPGA timestamps. After subtracting the switching delay which can be measured when the network is idle, the remaining packet queueing delays can serve as ground truth for queueing delays of the output queues behind S_0 and S_1 . An incast load (same as in Section 3) is applied with loads between ToR switches and spine switches ranging from 10% to 80%. Each server probes 20 other servers ($K = 20$). The 10 ms average queue depths are reconstructed using probe timestamps with LASSO. Separately, we send extra probes through the queues enclosed by NetFPGAs to collect the ground truth waiting time. The reconstruction results are compared to the ground truth.

Figure 14 shows the PSDs of the observed queueing processes. It shows that the power of the filtered-out high frequency component is $(12KB)^2$, which matches the results in Section 3! Figure 15 shows the reconstruction results of SIMON verified against ground truth. We can see that SIMON accurately reconstructs the queue depths for different levels of switches at various network loads.

6.3 Cross-validation

It was not possible to get accurate ground truth in the testbeds T-40G-3L and T-10G-40G-5L. Most switches only support reading counter samples a few tens of times per second, and these samples are not taken frequently enough and cannot be otherwise used to get statistical quantities like millisecond-average queue sizes. We also did not have physical access to these data centers, hence NetFPGA-based evaluations were not possible.

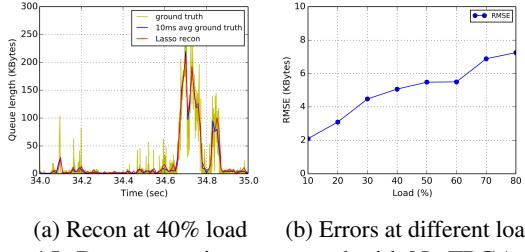


Figure 15: Reconstruction compared with NetFPGA ground truth

We proceed by using cross-validation to verify the accuracy of reconstruction. Consider two sets of probes, blue probes and red probes. No blue probing pair of servers is common to a red probing pair, nor is any end-to-end DCN path common to the blue and red probes: the two sets of probes are completely non-overlapping. Let the delay vectors and the incidence matrices for the blue and red probes be D_{blue} , D_{red} , A_{blue} and A_{red} , respectively. If, using the blue probes, we can get a reconstruction of the queueing delays \hat{W}_{blue} , and use this to accurately “predict” the delays of the red probes as $\hat{D}_{red} = A_{red}\hat{W}_{blue}$, then we can get a validation of the accuracy of our reconstruction.

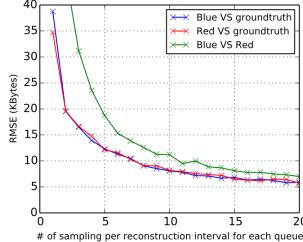


Figure 16: Blue red validation in simulations

Here we present a variant of the above method: instead of testing the agreement between \hat{D}_{red} and D_{red} , we test the agreement between \hat{Q}_{blue} and the reconstruction from red probes, \hat{Q}_{red} . We use this method because it is better for presentation (we get to plot the queue depths) and it helps bound the quality of the reconstruction: Denote the reconstruction error of the blue and red probes as $N_{blue} = \hat{Q}_{blue} - Q$ and $N_{red} = \hat{Q}_{red} - Q$ respectively. Assuming N_{blue} and N_{red} have zero mean and are independent, we have $\mathbb{E}[(\hat{Q}_{blue} - \hat{Q}_{red})^2] = \mathbb{E}[N_{blue}^2] + \mathbb{E}[N_{red}^2]$, which means the difference between the two reconstructions is bigger than the error of either reconstruction. Figure 16 illustrates this empirically with ns-3 simulations. As can be seen, at different queue sampling rates (probing rates), the error between the reconstructions obtained from the blue and red probes upper bonds their respective reconstruction errors (with respect to the ground truth).

This verification was performed on all three testbeds; we consider T-40G-3L. Each server in T-40G-3L blue probes to 10 random destination servers, and sends red probes to 10

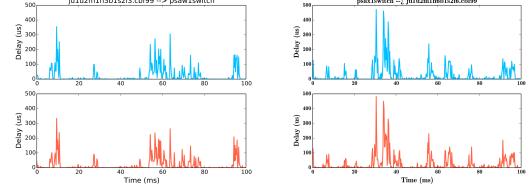


Figure 17: Blue-red validation examples in T-40G-3L

other random destination servers. Since the links are all 40G, the reconstruction interval is $250\mu s$. Figure 17 shows the reconstruction results at two different queues during a 100ms interval. Each sub-figure shows the blue-red comparison at one queue, with the upper half showing the blue reconstruction and the lower half showing the red probe one. There is excellent agreement between the blue and red reconstructions.

6.4 Example use cases of SIMON.

SIMON gives network administrators and application developers detailed visibility into the state and the dynamic evolution of the network. Thus, SIMON has many use cases in network regression testing, A/B testing, anomaly detection and bottleneck localization. We share a few of our experiences.

For the scenario in Figure 17, the normal maximum delay in a single queue is around $500\mu s$ in T-40G-3L. However, during an experiment, SIMON showed that the queueing delays in T-40G-3L suddenly increased to 5 to 10 milliseconds (shown in Figure 18), which is improbable in a 40G network as this implies very large buffers. This puzzling behavior was quickly resolved by observing the delays rose vertically in Figure 18 and that no packets were dropped. The only explanation is that the switches were configured to strict priority scheduling and our traffic was blocked by some higher priority traffic. This turned out to be the case. Thus, SIMON not only surfaced and explained the anomaly in the network, but also helped understand the precise way strict priority affects low-priority traffic.

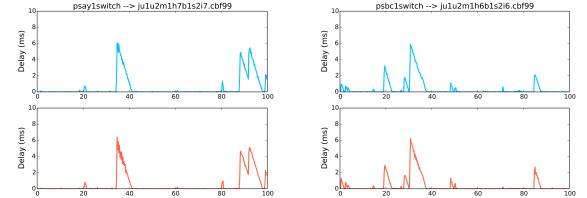


Figure 18: Reconstruction under strict priority packet scheduling in T-40G-3L

Another case: In T-1G-2L, SIMON discovered that one of the 256 1 Gbps ports in the testbed was mistakenly configured to 100 Mbps—a fact that is hard to determine without a

detailed link utilization plot! Moreover, in the same testbed, SIMON found that one application’s traffic was frequently causing other applications’ TCP flows to timeout. Many of these uses cases are beyond the reach of traditional network monitoring tools. We believe the visibility provided by SIMON can greatly increase the ability to understand and efficiently utilize the network.

7 Literature Survey

Switch-based methods. There are three main threads: (a) Obtain summaries and sketches of large, high-rate flows (called “heavy hitters”) as these are easy to identify and account for most of the bandwidth consumption and queue backlogs [23, 43, 35, 15]. (b) Capture detailed counts of *all packets and all flows*; for example, [46, 45] uses a hybrid SRAM-DRAM memory architecture and [34] uses Counter-Braids to “compress while counting”. These methods are harder to implement compared to those in (a) and entail offline processing to obtain the exact counts. (c) In-band network telemetry (INT) [58, 30, 32] uses switches to append telemetry data to packet headers as they pass through. The “postcards” approach [28] gathers telemetry data for all packets and sends it together with packet headers to servers. This approach takes advantage of the flexibility in P4 programmable switches and the storage and computation resources in servers, and is the focus of much current research.

Edge-based methods. Trumpet [38] uses flow event triggers to detect flow-level traffic spikes and congestion; for every packet dropped, 007 [8] attempts to identify the link responsible for the drop knowing the path taken by the packet (from traceroute); Pingmesh [27] uses RTT measurements of TCP probes to infer end-to-end latency distributions between any pair of hosts, including “packet black holes”; PathDump [51] uses switches to tag packets belonging to flows that satisfy a certain criterion and processes and stores the data at the edge; and SwitchPointer [52] goes further: it uses switches to store pointers to telemetry data relevant to that switch but held in end-hosts. PathDump and SwitchPointer follow the work of Everflow [59] which makes significant use of switches to “match” specific packets and “mirror” them to servers which can then trace the packets across the whole path. This enables Everflow to detect various faults such as silent packet drops, routing loops and load imbalance.

Tomography. As mentioned in the Introduction, reconstruction of network state variables in the wide area is not possible due to the long propagation times, unknown topology, and path information. Hence, researchers have obtained *distributions* of delays and backlogs of [42, 21, 19, 56, 55, 57, 27], topological relationships [17], or packet loss patterns at switch buffers [12, 11, 22, 54, 18, 26, 27].

In summary, switch-based methods are not easy to scale and can’t easily relate network bottlenecks to application performance since they lack application context. Current edge-

based methods obtain only a partial view of the network state and some may require non-trivial assistance from switches. Tomography results in the wide-area do not obtain near-exact reconstruction of network variables. To our knowledge, this is the first application of Network Tomography to data centers and for obtaining a near-exact reconstruction.

8 Conclusion

We introduced SIMON, a network tomography-based sensing, inference and measurement system for data centers. SIMON reconstructs network state variables, such as queueing times, link utilizations, and queue and link compositions, using timestamps of data packets and probes taken at NICs. By doing so, SIMON is able to connect bottlenecks at switches and network links to individual flows of the applications.

SIMON employed several techniques to simultaneously achieve precise reconstruction and scalability. First, a signal processing analysis suggested that reconstruction intervals that vary inversely as link speeds—10 ms for 1 Gbps links, 1 ms for 10 Gbps links, 250 μ s for 40 Gbps links, are appropriate for reconstructing queue sizes and wait times to an accuracy of over 97% of the power of the queue size or wait time process. Secondly, we used a mesh of probes to obtain extra information about network queue sizes and wait times, and described guidelines for picking the parameters of the probe mesh. We then showed that a LASSO-based reconstruction procedure accurately reconstructed the queue size, wait time, link utilization and queue/link composition processes. Two methods of simplifying the implementation of LASSO were presented: one method exploited the hierarchical structure of modern data center topologies to get a 1000x speedup of SIMON, and the other method used multi-layered neural networks and GPUs to accelerate LASSO by 5,000x–10,000x. These methods enable SIMON to run in near real-time.

We implemented SIMON on three testbeds with link speeds ranging from 1–40 Gbps and with up to 288 servers. The accuracy of the reconstruction is validated with NetFPGAs in the 1 Gbps testbed and by a cross validation technique in the other two testbeds. These implementations not only demonstrate the accuracy of SIMON’s reconstruction but also demonstrated its practicability. Since SIMON is agnostic of the switches in the network and only requires the timestamping capability readily available in most current generation NICs, it can be deployed in production data centers today.

References

- [1] Appdynamics, 2017. [Online; accessed 9-October-2017].
- [2] New relic, 2017. [Online; accessed 9-October-2017].
- [3] Splunk, 2017. [Online; accessed 9-October-2017].
- [4] Cisco tetration, 2018. [Online; accessed 14-February-2018].

- [5] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 63–74.
- [6] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *ACM SIGCOMM computer communication review* (2010), vol. 40, ACM, pp. 63–74.
- [7] ARISTA. Eos cloud networking technology - telemetry and analytics, 2018. [Online; accessed 9-January-2018].
- [8] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., LOO, B. T., AND OUTHRED, G. 007: Democratically finding the cause of packet drops. In *NSDI* (2018), USENIX.
- [9] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 267–280.
- [10] BROADCOM. High-performance data center soc with integrated netx-treme ethernet controller, 2018. [Online; accessed 9-October-2017].
- [11] BU, T., DUFFIELD, N., PRESTI, F. L., AND TOWSLEY, D. Network tomography on general topologies. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 21–30.
- [12] CÁCERES, R., DUFFIELD, N. G., HOROWITZ, J., AND TOWSLEY, D. F. Multicast-based inference of network-internal loss characteristics. *IEEE Transactions on Information theory* 45, 7 (1999), 2462–2480.
- [13] CASE, J. D., FEDOR, M., SCHOFFSTALL, M. L., AND DAVIN, J. Simple network management protocol (snmp). Tech. rep., 1990.
- [14] CAULFIELD, A. M., CHUNG, E. S., PUTNAM, A., ANGEPAT, H., FOWERS, J., HASELMAN, M., HEIL, S., HUMPHREY, M., KAUR, P., KIM, J.-Y., ET AL. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), IEEE, pp. 1–13.
- [15] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming* (2002), Springer, pp. 693–703.
- [16] CHUANG, S.-T., GOEL, A., MCKEOWN, N., AND PRABHAKAR, B. Matching output queueing with a combined input/output-queued switch. *IEEE Journal on Selected Areas in Communications* 17, 6 (1999), 1030–1039.
- [17] COATES, M., CASTRO, R., NOWAK, R., GADHIOK, M., KING, R., AND TSANG, Y. Maximum likelihood network topology identification from edge-based unicast measurements. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 11–20.
- [18] COATES, M. J., AND NOWAK, R. D. Network loss inference using unicast end-to-end measurement. In *ITC Conference on IP Traffic, Modeling and Management* (2000), pp. 28–41.
- [19] COATES, M. J., AND NOWAK, R. D. Network tomography for internal delay estimation. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on* (2001), vol. 6, IEEE, pp. 3409–3412.
- [20] CONTRIBUTORS, W. Syslog — wikipedia, the free encyclopedia, 2018. [Online; accessed 9-January-2018].
- [21] DUFFIELD, N. G., HOROWITZ, J., PRESTI, F. L., AND TOWSLEY, D. Network delay tomography from end-to-end unicast measurements. In *Thyrrhenian International Workshop on Digital Communications* (2001), Springer, pp. 576–595.
- [22] DUFFIELD, N. G., PRESTI, F. L., PAXSON, V., AND TOWSLEY, D. Inferring link loss using striped unicast probes. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2001), vol. 2, IEEE, pp. 915–923.
- [23] ESTAN, C., AND VARGHESE, G. *New directions in traffic measurement and accounting*, vol. 32. ACM, 2002.
- [24] GENG, Y., LIU, S., YIN, Z., NAIK, A., PRABHAKAR, B., ROSENBLUM, M., AND VAHDAT, A. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 81–94.
- [25] GHITA, D., KARAKUS, C., ARGYRAKI, K., AND THIRAN, P. Shifting network tomography toward a practical goal. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), ACM, p. 24.
- [26] GHITA, D., NGUYEN, H., KURANT, M., ARGYRAKI, K., AND THIRAN, P. Netscope: Practical network loss tomography. In *INFOCOM, 2010 Proceedings IEEE* (2010), IEEE, pp. 1–9.
- [27] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 139–152.
- [28] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI* (2014), vol. 14, pp. 71–85.
- [29] HAYKIN, S., AND NETWORK, N. A comprehensive foundation. *Neural networks* 2, 2004 (2004), 41.
- [30] HYUN, J., AND HONG, J. W.-K. Knowledge-defined networking using in-band network telemetry. In *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific* (2017), IEEE, pp. 54–57.
- [31] IEEE. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Standard 1588* (2008).
- [32] KIM, C., SIVARAMAN, A., KATTA, N., BAS, A., DIXIT, A., AND WOBKER, L. J. In-band network telemetry via programmable data-planes. In *ACM SIGCOMM* (2015).
- [33] LEE, K. S., WANG, H., SHRIVASTAV, V., AND WEATHERSPOON, H. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 454–467.
- [34] LU, Y., MONTANARI, A., PRABHAKAR, B., DHARMAPURIKAR, S., AND KABBANI, A. Counter braids: a novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (2008), 121–132.
- [35] LU, Y., PRABHAKAR, B., AND BONOMI, F. Elephanttrap: A low cost device for identifying large flows. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on* (2007), IEEE, pp. 99–108.
- [36] MANUAL, L. traceroute manual page, 2018. [Online; accessed 9-January-2018].
- [37] MELLANOX. Bluefield multicore system on chip, 2017. [Online; accessed 19-February-2018].
- [38] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 129–143.
- [39] NAIR, V., AND HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.
- [40] NETWORKS, J. Junos telemetry interface feature guide, 2018. [Online; accessed 9-January-2018].
- [41] OPPENHEIM, A. V. *Discrete-time signal processing*. Pearson Education India, 1999.

- [42] PRESTI, F. L., DUFFIELD, N. G., HOROWITZ, J., AND TOWSLEY, D. Multicast-based inference of network-internal delay distributions. *IEEE/ACM Transactions On Networking* 10, 6 (2002), 761–775.
- [43] RAMABHADRAN, S., AND VARGHESE, G. Efficient implementation of a statistics counter architecture. In *ACM SIGMETRICS Performance Evaluation Review* (2003), vol. 31, ACM, pp. 261–271.
- [44] sFLOW.ORG. sflow, 2018. [Online; accessed 9-January-2018].
- [45] SHAH, D., IYER, S., PRABHAKAR, B., AND MCKEOWN, N. Analysis of a statistics counter architecture. In *Hot Interconnects 9, 2001*. (2001), IEEE, pp. 107–111.
- [46] SHAH, D., IYER, S., PRAHHAKAR, B., AND MCKEOWN, N. Maintaining statistics counters in router line cards. *IEEE Micro* 22, 1 (2002), 76–81.
- [47] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 183–197.
- [48] SPARK. Spark: Lightning-fast cluster computing, 2018. [Online; accessed 9-January-2018].
- [49] SYSTEMS, C. Model-driven telemetry, 2018. [Online; accessed 9-January-2018].
- [50] SYSTEMS, C. Netflow, 2018. [Online; accessed 9-January-2018].
- [51] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *OSDI* (2016), USENIX.
- [52] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), USENIX.
- [53] TIBSHIRANI, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.
- [54] TSANG, Y., COATES, M., AND NOWAK, R. Passive network tomography using em algorithms. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP’01). 2001 IEEE International Conference on* (2001), vol. 3, IEEE, pp. 1469–1472.
- [55] TSANG, Y., COATES, M., AND NOWAK, R. Nonparametric internet tomography. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on* (2002), vol. 2, IEEE, pp. II–2045.
- [56] TSANG, Y., COATES, M., AND NOWAK, R. D. Network delay tomography. *IEEE Transactions on Signal Processing* 51, 8 (2003), 2125–2136.
- [57] TSANG, Y., YILDIZ, M., BARFORD, P., AND NOWAK, R. Network radar: tomography from round trip time measurements. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), ACM, pp. 175–180.
- [58] VAN TU, N., HYUN, J., AND HONG, J. W.-K. Towards onos-based sdn monitoring using in-band network telemetry. In *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific* (2017), IEEE, pp. 76–81.
- [59] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., ET AL. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 479–491.

9 Appendix

9.1 Queue Coverage Probability

Consider a network with l layers and N servers. Each server randomly probes K other servers. We show that without under-subscription, *i.e.*, the number of links at the spine layers is no greater than the number of servers, the probability that every queue in the network is covered by at least one probe packet is greater than $1 - (2l - 1)Ne^{-2K}$, where l is the number of layers, N is the number of servers and K is the number of random probes per server.

Proof:

A lower bound on the probability can be derived using a union bound. We consider queues at different levels, and bound the probability of covering all queues at every level. First, consider the lowest level queues, the queues on the ToR downlinks to the servers. There are exactly $N_1 = N$ such queues. Because the destinations of the probes are random, for a specific queue q and a specific probe p , the probability that p passes through q is

$$\frac{1}{N}$$

Since there are $2NK$ probes in total, the probability that q is not hit by any of the probes is

$$(1 - \frac{1}{N})^{2NK}$$

So the probability that there exists a level-1 queue, such that it is not hit by any probes can be bounded by

$$\mathbb{P}(\exists q \text{ not hit by any probe}) \leq \sum_{i=1}^N \mathbb{P}(q_i \text{ not hit by any probe})$$

$$= N(1 - \frac{1}{N})^{2NK} \approx Ne^{-2K}$$

For level-2 queues, namely the ToR uplink and L2 downlink queues, each has number N_2 . Conditioning on a probe reaching level 2, which queues it hits is uniformly random. So for a level-2 queue q and a probe p , we have

$$\mathbb{P}(q \text{ hit by } p | p \text{ traverse to level 2}) = \frac{1}{N_2}$$

Probe p goes to level 2 and above only if it is probing a server outside its own rack; this event has chance $\frac{N-R}{N}$ where R is the rack size. Combining with the previous equation we have

$$\mathbb{P}(q \text{ not hit by } p)$$

$$\begin{aligned} &= \mathbb{P}(q \text{ not hit by } p | p \text{ traverse to level 2}) \mathbb{P}(p \text{ traverse to level 2}) \\ &\quad + \mathbb{P}(q \text{ not hit by } p | p \text{ not traverse to level 2}) \mathbb{P}(p \text{ not traverse to level 2}) \\ &= (1 - \frac{1}{N_2})(1 - \frac{R}{N}) + 1 \cdot \frac{R}{N} \\ &= 1 - \frac{1}{N_2}(1 - \frac{R}{N}) \end{aligned}$$

So

$$\begin{aligned}\mathbb{P}(\exists q \text{ not hit by any probe}) &\leq \sum_{i=1}^{N_2} \mathbb{P}(q_i \text{ not hit by any probe}) \\ &= N_2 \left(1 - \frac{1}{N_2} \left(1 - \frac{R}{N}\right)\right)^{2NK} \\ &= N_2 e^{-2\frac{N-R}{N_2}K}\end{aligned}$$

This argument can be repeated for any level j . Assume the size of a level- j block is R_j , and the queues at level- j is N_j .

$$\begin{aligned}\mathbb{P}(q \text{ not hit by any probe}) &\leq \sum_{i=1}^{N_j} \mathbb{P}(q_i \text{ not hit by any probe}) \\ &= N_j \left(1 - \frac{1}{N_j} \left(1 - \frac{R_j}{N}\right)\right)^{2NK} \\ &= N_j e^{-2\frac{N-R_j}{N_j}K}\end{aligned}$$

Remarks:

- All the parameters N_i , R_i are properties of the network topology. In other words, the probability bounds are customized for the specific topology.
- The bounds are linear with respect to N_i , the number of queues, but exponential in K , the number of random probes per server. As an example, with the topology fixed, if $K_{new} = K + 2.3$, then the probability of not covered will decrease by 10 folds.
- Suppose there is no under-subscription (so $N_i \leq N$ for all i) and R_i is negligible compared with N . Then for any level, the probability of some queue not covered in that level is bounded by Ne^{-2K} . For a typical network with 10k servers with $K = 10$, this probability is roughly 2×10^{-5} . When there are l network layers, we have $2l - 1$ levels of queues. Applying a union bound across all levels give the desired upper bound on the coverage probability $1 - (2l - 1)Ne^{-2K}$.
- Over-subscription is good from a coverage-point of view, as there are fewer higher level paths to choose from. We note in reality, most networks are over-subscribed. At the same time, link speed sometimes are higher for upper layer switching. Both contribute to the fact that there are usually fewer links in the upper layers.

Is advance knowledge of flow sizes a plausible assumption?

Vojislav Đukić¹, Sangeetha Abdu Jyothi², Bojan Karlaš¹,
Muhsen Owaida¹, Ce Zhang¹, Ankit Singla¹

¹ETH Zurich ²University of Illinois at Urbana–Champaign

Abstract

Recent research has proposed several packet, flow, and coflow scheduling methods that could substantially improve data center network performance. Most of this work assumes advance knowledge of flow sizes. However, the lack of a clear path to obtaining such knowledge has also prompted some work on non-clairvoyant scheduling, albeit with more limited performance benefits.

We thus investigate whether flow sizes can be known in advance in practice, using both simple heuristics and learning methods. Our systematic and substantial efforts for estimating flow sizes indicate, unfortunately, that such knowledge is likely hard to obtain with high confidence across many settings of practical interest. Nevertheless, our prognosis is ultimately more positive: even simple heuristics can help estimate flow sizes for many flows, and this partial knowledge has utility in scheduling.

These results indicate that a presumed lack of advance knowledge of flow sizes is not necessarily prohibitive for highly efficient scheduling, and suggest further exploration in two directions: (a) scheduling under partial knowledge; and (b) evaluating the practical payoff and expense of obtaining more knowledge.

1 Introduction

Advance knowledge of future events in a dynamic system can often be leveraged to improve the system’s performance and efficiency. In data center networks, such knowledge could potentially benefit many problems, including routing and flow scheduling, circuit switching, packet scheduling in switch queues, and transport protocols. Indeed, past work on each of the above topics has explored this, and in some cases, claimed significant improvements [34, 21, 5, 4, 30].

Nevertheless, little of this work has achieved deployment. Modern deployments largely use techniques that do not depend on knowing traffic features in advance, such as shortest path routing with randomization, and first-in-first-out queuing. A significant barrier to the adoption of traffic-aware scheduling is that in practice, traffic features can be difficult to ascertain in a timely fashion with adequate accuracy.

We focus on the plausibility and utility of obtaining flow size information *a priori* for use in packet, flow, and coflow scheduling in data centers. We explore this problem in the context of four scheduling techniques from past work: pFabric [5], pHost [21], FastPass [34], and Sincronia [4]. Each of these is a clairvoyant scheduler, with advance knowledge of

the size of each flow at its start (but not necessarily the flow arrival times). For some problems, non-clairvoyant algorithms are also known, e.g., PIAS [7] for packet scheduling, and Aalo [13] for coflow scheduling. While such techniques outperform FIFO and fair-sharing baselines, there is still a substantial performance gap compared to clairvoyant algorithms (§3.2). Further, it is unclear if similar non-clairvoyant methods can be developed for scheduling problems such as FastPass [34], where absolute flow sizes are needed, rather than just the relative priorities leveraged by PIAS and Aalo.

We thus examine a wide array of possibilities for estimating flow sizes in advance, including modifications to the application-system interface for explicit signaling by the application, as well as more broadly applicable application-agnostic methods, ranging from simple heuristics like reading buffer occupancy and monitoring system calls, to sophisticated machine learning. We analyze the complexity, accuracy, and timeliness of different approaches, and the utility of the (often imprecise) flow size information gleaned by these methods across our four example scheduling techniques.

We find that even simple heuristics effectively estimate flow sizes for a large fraction of flows in many settings. But accurate estimation for *all* flows is likely intractable: for many scenarios of practical interest, each of the estimation approaches under consideration has limitations that prevent accurate flow size estimation. Superficially, this can be seen as a negative result for clairvoyant schedulers. However, this does not necessarily restrict us to non-clairvoyant scheduling — as recent work shows [9, 40], partial knowledge of flow sizes, coupled with heuristics to tackle the unknown flow sizes, can often provide an effective compromise. We pose a novel question by intersecting this past work with our results on the effectiveness and complexity of different methods of flow size estimation: how does investment in increasing the coverage of flow size estimation pay off? To the best of our knowledge, no prior work has tackled this issue — as we invest greater effort in flow size estimation, how does scheduling performance change?

We show, empirically and analytically, that for packet scheduling at switches, a simple approach¹ incentivizes greater efforts in estimating flow sizes. While this may seem obvious, somewhat surprisingly, we find that adding more flow size estimates does not always improve performance — for coflow scheduling, an intuitive scheduling scheme for

¹A simplification of Karuna [9] replacing discrete thresholds for priority queues with continuously degrading flow priority as more bytes are sent.

partial knowledge settings sometimes sees increased coflow completion times as more flow sizes are made known.

These first results call for more exploration in two directions: (a) schedulers explicitly designed for partial knowledge settings; and (b) the scheduling efficiency benefits of greater investments in learning flow sizes.

We believe this to be the first in-depth critical analysis of assumptions of (non-) clairvoyance in flow, coflow, and packet scheduling, together with considering the utility of partial knowledge, and various vectors for increasing this knowledge. We make the following contributions:

- We design **FLUX**, a framework for estimating flow sizes in advance, and tagging packets with this information.
- Using **FLUX**, we evaluate both simple heuristics using system calls and buffer sizes, as well as learning methods; identify which factors are effective in flow size estimation; and explain how these depend on applications.
- We implement **FLUX** in Linux², and demonstrate that even the learning method implemented incurs only small computational and latency overheads, which can be further reduced using specialized hardware.
- For each method of flow estimation, we also explore its limitations, concluding that for many practical scenarios, such estimation will remain challenging.
- We evaluate the utility of inferred (often imprecise) flow sizes across four scheduling techniques, finding significant benefits compared to flow-unaware scheduling.
- We analyze settings with some fraction of traffic permitting flow estimation, and show that the impact of increasing this fraction is not always positive.
- In a simplified model, we prove that for shortest remaining first packet scheduling [5], coupled with a simple heuristic for handling unknown flows, adding a flow’s size cannot worsen its completion time.

2 Background & motivation

Many scheduling techniques for data center networks have been proposed, promising substantial performance gains:

- PDQ [22] and D³ [43] schedule flows across the fabric, in a “shortest flow first” manner.
- pFabric [5] and EPN [29] schedule packets at switch queues using “least remaining flow” prioritization.
- pHHost [21], Homa [30], and FastPass [34] schedule sets of packets across the network.
- Orchestra [14], Varys [15], Sincronia [4], and Baraat [18] schedule coflows (app-level aggregates).
- c-Through [41], Helios [20], and several followup proposals schedule flows along time-varying circuits.

²Code and traces here: <https://github.com/vojislavdjukic/flux>.

All of these proposals are clairvoyant schedulers, *i.e.*, they assume that the size of a flow is known when it starts. Some of this work has made this assumption explicit:

“In many data center applications flow sizes or deadlines are known at initiation time and can be conveyed to the network stack (e.g., through a socket API) ...”

— Alizadeh et al. [5], 2013

“The sender must specify the size of a message when presenting its first byte to the transport ...”

— Montazeri et al. [30], 2018

There is not, however, consensus on the availability of such information; work in the years intervening the above two proposals has argued the opposite. For instance:

“... for many applications, such information is difficult to obtain, and may even be unavailable.”

— Bai et al. [7], 2015

Thus, we explore this question in depth: Is advance knowledge of flow sizes a plausible assumption? Further, what happens when only information for some flows is available? We design a framework for estimating flow size information and evaluate its utility for four example scheduling techniques in past work that depend on this information. We include here brief, simplified background on these techniques:

pFabric [5]: Each packet is tagged with a priority at the end-host, based on the remaining flow size. Switches then schedule packets in order of least remaining size. This results in near-optimal packet scheduling and can improve average flow completion time (FCT) by as much as 4× for certain workloads, compared to the oblivious FIFO scheme.

pHost [21]: pHHost uses distributed scheduling, with the source sending to the destination a “Request To Send” message carrying the number of pending bytes in the flow. The destination clears transmission for the flow with the least bytes. pHHost claims an average FCT reduction of 3×.

FastPass [34]: FastPass uses a centralized arbiter to schedule flows. When a host wants to send data, it asks the arbiter to assign it a data transmission time slot and path. The arbiter tries to make a decision based on the traffic demand (flow size) of all active flows. FastPass claims “near-zero queuing” on heavily loaded networks, cutting network RTT by 15×.

Sincronia [4]: Sincronia orders coflows using sizes of individual flows to find network bottlenecks and uses this ordering for priority scheduling. It achieves average coflow completion time (CCT) within 4× of the optimal.

If flow sizes were known *a priori*, such techniques could improve various performance metrics of interest by 3-15× compared to size-unaware ECMP-plus-FIFO scheduling. For some scheduling problems, where only relative flow priorities matter rather than absolute sizes, prior work has developed non-clairvoyant schedulers [7, 45, 13] that also beat the ECMP-FIFO baseline. But as we shall see later, their per-

formance improvements are often much more modest than clairvoyant schedulers. Thus, addressing the question of flow size estimation remains an interesting open problem.

3 Flow size estimation: design space

Before considering flow size estimation, it is necessary to define a flow. The primary goal of flow size estimation is to improve application performance using network scheduling. Where application messages directly translate to individual TCP connections, using TCP 5-tuples to define flows suffices. However, to avoid the overheads of connection setup and TCP slow start, it is common practice in many data centers to use persistent long-lived TCP connections (*e.g.*, between Web servers and caches) which carry multiple application messages over time. In these settings, it may be more appropriate to consider instead a series of packets between two hosts that are separated from other packet series by a suitable time gap. We note that this is an imprecise method, as system-level variability and workload effects can impair such identification of flows. For instance, multiple small cache responses sent out in quick succession could be mistakenly identified as one flow. This limitation applies to all application-oblivious methods — in some scenarios, mechanisms to identify packets that form an application-level message are inherently bound to be imprecise.

We next describe several approaches for obtaining flow sizes, and intuitively reason about their efficacy in various settings. Experimental evaluations of the quality of estimation and its impact on network scheduling are deferred to §6.

3.1 Exact sizes provided by application

Many applications can assess how much data they want to send, such as standard Web servers, RPC libraries, and file servers. Modifying such applications to notify the network of a message’s size is thus plausible. This would require a new interface between applications and the network stack, and is clearly doable, but not trivial. The replacement must be interruptible, like `send` in Linux, and it’s unclear how best to implement this – what happens when it gets interrupted after sending some bytes? When a new call is made to finish the transfer, how do we decide whether or not this is the same flow? Thus, this may also require introducing some notion of flow identifiers. While this can surely be done, we merely point out that it requires care.

Limitations: As discussed in some depth in prior work [7], there are several scenarios where the application itself is unaware of the final flow size when it starts sending out data, such as for chunked HTTP transfers, streaming applications, and database queries where partial results can be sent out before completion. Also, apart from introducing changes to the host network stack (which are not necessarily prohibitive for private data centers), this approach requires modifying a large number of applications to use the new API. In settings like the public cloud, this may not be feasible.

Still, this approach should not be casually dismissed; a few software packages dominate the vast majority of deployed applications, *e.g.*, a large fraction of Web servers use one of the three most popular server software packages, most data analytics tasks use one of a small number of leading frameworks, etc. Past work (*e.g.*, FlowProphet [42] and Hadoop-Watch [33]) has in fact explored the use framework-internals for gleaning flow sizes. Thus, this approach could make flow size information accessible to the network for many applications, provided the right APIs are developed.

3.2 Flow aging

A set of application-agnostic techniques have been proposed around the idea of using the number of bytes a flow has already sent as an estimator for its pending data. For instance, Least Attained Service [35] gives the highest priority to new flows and then decreases their priority as they send more data. Thus, flow priorities “age” over time. PIAS [7] explores a variant of this approach, coupling it with the use of a small number of discrete queues to fit commodity switches. Aalo [13] applies similar ideas to coflow scheduling.

Limitations: The most significant drawback is that this approach may not benefit scheduling techniques that require absolute flow sizes (as opposed to only relative priorities), such as Sincronia³, FastPass and optical circuit scheduling. Even where applicable, the effectiveness of such methods depends on flow size distribution. For instance, LAS does not work well when there are a large number of flows of similar size. In the limiting case, if all flows are the same size, older flows nearer to completion are deprioritized, which is the opposite of the desired scheduling. More sophisticated methods based on multi-level feedback queues [7] still depend on estimating a *stable* underlying flow size distribution⁴. Further, even in favorable settings, with stable heavy-tailed flow size distributions, the performance of such application-agnostic techniques can be substantially lower than clairvoyant ones. For instance, recent work [30] reports $\sim 2\times$ difference in 99th percentile slowdown between PIAS [7] and pFabric [5]. Similarly, Sincronia [4], the best-known clairvoyant coflow scheduler, claims a 2-8 \times advantage over Varys, and by extension, over CODA [45], the best-known non-clairvoyant coflow scheduler. (Note however, that the scheduling knowledge involved in CODA is not limited to flow sizes, but also classification of flows into coflows.)

3.3 TCP buffer occupancy

The occupancy of the TCP send buffer at the sending host can provide approximate information on flow sizes. When the buffer occupancy is small, the number of packets in the buffer may be the actual flow size of a small flow. When the

³Sincronia uses only relative priorities in the network, but for assigning these priorities, it computes the bottleneck port using sizes of all flows destined to each port. It is unclear if aging would be effective here.

⁴Alternatively, additional effort must be spent in continuously monitoring and following the changes in the underlying distribution [10].

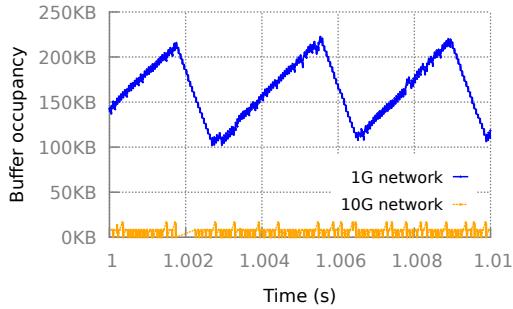


Figure 1: Buffer occupancy while transferring a 1GB static file from the hard disk over 1G and 10G connections. We show a representative 10 ms segment of the trace starting at 1 second.

buffer is fully occupied, *i.e.*, its draining rate is less than its filling rate, the flow may be categorized as a large flow. Mahout [17] and c-Through [41] used roughly this approach. ADS [31] also suggests (but does not evaluate) a similar mechanism, although it is unclear whether it uses system calls, or buffer occupancy, or both.

Limitations: Buffering reflects flow size only when the sender is network or destination limited. If the bottleneck is elsewhere, the buffer may not be filled even by a large flow. Consider a program that reads a large file from the disk and sends it over the network. The program reads data in chunks of a certain size (*e.g.*, 100 KB) and sends as follows:

```
while(...):
    read(file_desc, read_buffer, 100KB)
    write(socket_desc, read_buffer, 100KB)
```

Today’s SSDs achieve a read throughput of \sim 6 Gbps, while NIC bandwidths of 10-40 Gbps are common. This disparity implies that the buffer may remain sparsely populated most of the time. To illustrate this behavior, we ran a simple experiment. We transfer a 1GB static file served by a Web server over 1G and 10G connections (Fig.1). The file is stored on a regular 7200 RPM hard drive with the maximum read speed of \sim 1 Gbps. We see that for the faster connection, the buffer is almost empty. For slow connections, the buffer indicates the lower bound of the flow size, but when the buffer occupancy starts decreasing, it is unclear if that means that no additional data will be added. This presents a serious obstacle for flow size estimation.

3.4 Monitoring system calls

The write/send system calls from an application to the kernel provide information on the amount of data the application wants to send. The flow size is typically greater or equal to the number of bytes in the first system call of a flow. It is also interesting to notice that many applications have a standard system call length. For instance, Apache Tomcat by default transfers data in chunks of 8 KB. If it wants to send less than 8 KB, it issues a single system call which reflects the exact flow size. For larger flows, multiple calls are, of course, necessary. Other applications behave similarly;

MySQL uses chunks of 16 KB, Spark Standalone 100 KB, and YARN 262 KB. Thus, for identifying short flows, this is a reliable approach, and can directly enable algorithms like Homa [30]. Further, recent work from Facebook suggests that a substantial portion of flows is extremely small and most likely transferred over a single system call [46].

To test this approach, we run a simple experiment where we store 100,000 objects of sizes between 500 B to 1 KB using MySQL. Further, we execute three types of queries: fetch an object based on the key, fetch a range of 10 objects using a date index, and fetch 1000 objects (*e.g.*, to perform a join operation or backup). Since results for queries fetching 1 and 10 objects fit into the initial system call, we were, in fact, able to obtain their flow sizes accurately.

Limitations: The flow size information inferred from system calls may correspond to only a part of the flow rather than the whole flow, as in the above example for large queries. Increasing the size of the initial system call could work, but larger system calls require more buffer memory per connection. Thus, Web servers, databases, and other highly concurrent programs tend to keep system calls small.

3.5 Learning from past traces

We can also apply machine learning to infer flow size information from system traces. Ultimately, data sent out to the network trace causality to some data received, read from disk or memory, or generated through computation. Thus, traces of these activities may allow learning network flow sizes. Given that most jobs in data centers today are repetitive, there is a significant opportunity for such learning. For instance, in [25], the authors observed that more than 60% jobs in enterprise clusters are periodic with predictable resource usage patterns. Analysis of publicly available Google data center traces also confirms this finding: most of the resources are consumed by long-term workloads [37, 3, 36].

Unlike the simpler approaches above, the effectiveness and limitations of learning methods are hard to analyze without a serious attempt at building a learning system. A key challenge here is the short timescale: while past work [28, 44] has explored learning workload characteristics at timescales of minutes and hours, can we learn at the microsecond timescales necessary for flow size estimation? This represents a challenging leap across 8-10 *orders of magnitude* in timescales. We next detail our efforts towards building a learning system for flow size prediction.

4 Learning flow sizes

We explore the design of a learning-based approach for flow size estimation, addressing the following questions:

- Which methods can we use for flow size prediction?
- What prediction accuracy is achievable?

Learning task: We would like to learn flow sizes for outgoing flows in advance, using system traces. When a flow

f starts, are recent measurements of network, disk, memory I/O, CPU utilization, etc. predictive of f 's size?

We first present a superficial “black box” treatment of this question, going directly from training standard learning methods over traces we collected to the best accuracy we obtained. §5 shall delve into the details of how we made this approach work, and give intuition into its success. §6 will discuss how the accuracy results translate to network performance improvements, while §7 discusses its limitations.

4.1 Workloads

To explore what inputs are predictive of flow sizes, it is essential to gather job execution traces with as much detail as possible, across many instances of jobs with variable inputs and configurations. Unfortunately, publicly available data center traces do not contain enough information. Facebook’s traces [39], by sampling 1 per 30000 packets, provide no visibility at the flow granularity. Google’s traces [38] completely omit network data, focusing on CPU, memory utilization, etc. We thus collect traces for (a) a large range of synthetic workloads; and (b) machine learning applications running on our university clusters.

Our traces comprise 5 applications: PageRank, K-Means, and Stochastic Gradient Descent (SGD) implemented on Spark; training deep neural networks using TensorFlow; and a Web workload. The SGD and Tensorflow traces are from instrumented applications running on our university cluster.

Each of SGD, KMeans, and PageRank runs on a Spark cluster of 8 machines, each machine with 2 CPU sockets (4 cores each) and 24 GB DRAM. For SGD, the input sizes vary from 2-25 GB, with significant variation across the hyperparameters. We also impose large input variations for KMeans and PageRank: for PageRank, we randomly generate new graphs with 1-15 million nodes; and for KMeans, we generate datasets with 20-50 million points, while also varying K . We also vary the number of workers per job from 8 to 64.

The Tensorflow trace consists of one 25 minute long execution of distributed training of AlexNet [26] on the ImageNet dataset on 40 GPU machines.

For the synthetic Web workload, we use Apache Tomcat 7.0 to host a full Wikipedia mirror and fetch random pages.

Fig. 2 summarizes these workloads. Fig. 2(a) shows the job execution times across different executions for each algorithm. Execution times for KMeans, PageRank, and SGD vary by factors of as much as $2.6\times$, $1.5\times$, and $24.5\times$ respectively. Thus, there is substantial variation across executions. The main source of the variations across traces is the change in the size of the input data and the number of iterations. However, for many runs there were more resources in the cluster than required by the job, leading traces to further incorporate the influence of Spark’s scheduling decisions.

We also aggregate flow statistics across all jobs and executions for each application to give a sense of the traffic; Fig. 2(b) shows the flow size distributions, and Fig. 2(c)

shows the arrival rates (aggregated across the workers). The TensorFlow workload consists of many short flows with an average arrival rate of 8273 flows/second.

4.2 Machine learning models

We evaluate several ML models, but with only modest efforts to optimize these, because our goal is not to identify the best model or hyperparameters, but to show that a variety of methods could work (as we show with results on improvements in scheduling in §6) with reasonable effort, modulo the limitations of learning in this context, as discussed in §7.

Recurrent Neural Network with LSTM layers: All our traces are time series. Given the natural dependency between data points in the trace, we test a network that can keep state and learn these dependencies while processing the trace sequentially. For this purpose, we use an LSTM model [24]. Using Keras [12] and Tensorflow [2], we test varied LSTM models with different numbers of layers. The best configuration in our setting uses a single layer with 64 LSTM units.

Gradient Boosting Decision Trees: In many of our traces, simple conditionals reveal information about the flow size. For instance, if the first system call size is below a certain value, that can often reveal the flow’s size. Thus, we train GBDT models of different sizes (*i.e.*, numbers of trees) and find that using 50 trees (with maximum depth of 10 per tree) gives fast yet accurate results.

Feed-Forward Neural Network: The dependency between flow size and other system-level features should not strictly depend on the ML model we choose. Thus, we test a standard FFNN model [23] with various configurations for the number of layers and neurons, implemented using Keras [12]. We find that 2 layers (and the ReLU activation function) with 5 neurons each yield the best performance.

Results: We split the traces into 3 fixed sets – training, validation, and test. Table 1 compares the three tested models. We use the coefficient of determination (R^2) to measure accuracy. R^2 is very useful because it can be easily compared across different models: $R^2 = 1$ if the model produces perfect predictions, and $R^2 = 0$ if the model makes a prediction of zero value, always predicting the mean. GBDT and FFNN achieve comparable accuracy (Table 1), with the high values of R^2 implying highly accurate flow size predictions. For two workloads, LSTM gave inferior results and did not seem to capture the dependencies, particularly across traces where the underlying executions were very different (*e.g.*, test-set for SGD). With greater effort, for instance, specializing the model to these traces, it may be possible to overcome LSTM’s apparent deficiency. However, we wanted to use the same training and inference approach across traces.

GBDT’s accuracy, fast convergence, and fast inference motivate its choice for FLUX. The tradeoff is that the model updates in batch mode (not online); this should suffice, unless applications change at sub-second timescales.

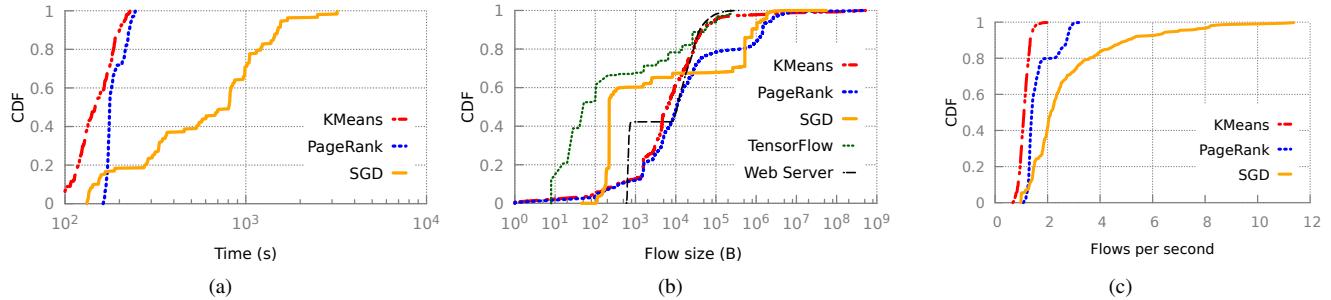


Figure 2: Workload diversity: (a) Execution time varies substantially across executions of the same job. We also show the distribution of (b) flow sizes and (c) flow arrival rate across our workloads.

	GBDT	FFNN	LSTM
Web server	94 96	92 94	73 74
TensorFlow	97 97	95 95	94 94
PageRank	85 83	84 84	83 83
Kmeans	88 90	88 95	88 93
SGD	58 79	54 72	46 10

Table 1: Prediction accuracy (shown for validation-set | test-set) across models and workloads in terms of R^2 percentage.

5 Opening the black box

What explains the high accuracy of our ML approach? We discuss the predictive power of various system-level measurements, and detail refinements that led from poor initial results to these high-accuracy predictions.

5.1 The treachery of time

We first tried what we considered a natural model for the data of our interest: time series. To generate time series data, during the execution of each workload, we sampled CPU and memory utilization, and disk and memory I/O, every 20 ms, and recorded headers of all incoming and outgoing packets. We then attempted to predict the next few time-steps for network traffic. However, this gave poor results due to low-level system effects that can have a significant impact on timing.

An alternative representation with a *flow-centric view* treats a job as a series of flows, with several attributes recorded per flow (Table 2). This is effective for flow size prediction, as it does not suffer from minor timing variations, and captures the relationship between (for instance) system calls and the volume of outgoing traffic. In addition, the measurements themselves serve as a “clock”, one that is more robust to system scheduling artifacts.

5.2 Why these features?

New flows are created either by reading data from the disk or memory, processing previously received flows, doing some computation to create data, etc. Thus, features that characterize each of these causal factors could help estimate flow size. For each type of system measurement, we track the total number of operations or bytes from the beginning of program execution. This enables the learning algorithm

Feature	Description
Start time, t_f	Start time of f relative to job start time
Flow gap	Time since the end of the previous flow
First Call	Size of the first system call t_f
Network In	Data received until t_f
Network Out	Data sent until t_f
Network In(d)	Data received at flow’s dest. d until t_f
Network Out(d)	Data sent by this host to d until t_f
CPU	CPU cycles used until t_f
Disk I/O	Total disk I/O until t_f
Memory I/O	Total memory I/O until t_f
Previous flows	Flow sizes for last k flows

Table 2: Features of a flow f . All network, memory, disk and CPU activity is cumulative until this flow’s start at t_f .

to know how many operations or bytes were processed between the last and the new flow. In our experiments, when we predict the flow size, we use features from Table 2 for last 5 flows. Thus, we try to catch dependencies between consecutive flows as well as resources that have been consumed.

As expected, the most predictive features vary across applications. For instance, some applications do not produce a lot of disk traffic, while others rely completely on the disk. The GBDT model provides a natural way of assessing feature importance: it uses a set of decision trees, with each attribute’s contribution measured in terms of “splits”, *i.e.*, in what percentage of branch conditions in the decision tree the attribute appears. Fig. 3 shows these splits for the aggregated flow-centric traces from a Spark environment. Interestingly, for these traces, we find that similar accuracy can be obtained by a model constrained to *not* use memory, disk, and CPU monitoring, relying only on network data and timing of flows. Web queries, on the other hand, have a different set of critical features where 66% of all splits use disk I/O.

Finding the *best* model and feature-set may require manual work, but effective solutions could be obtained automatically by comparing the accuracy of the largest model to the accuracy of candidate models limited to using only the first model’s most salient features.

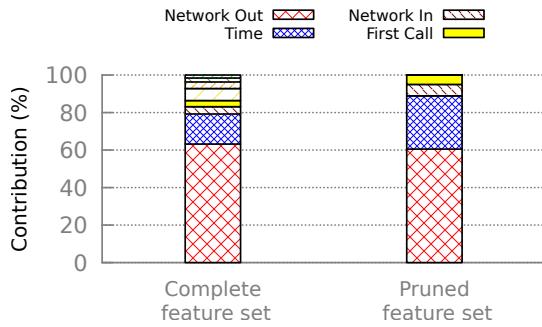


Figure 3: Feature contributions for 2 models for the Spark workloads, measured using GBDT splits. The figure omits labels for the less important features: memory and CPU utilization, and disk and memory I/O. Both models provide the same prediction accuracy. If we exclude any of the top 3 contributors, the accuracy decreases.

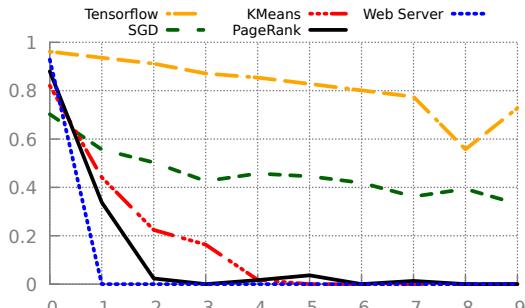


Figure 4: Prediction accuracy declines for more distant flows. $x = 0$ is the current flow, for which packets are starting to be sent out, while $x = 1$ is the next flow after this one, and so on.

5.3 Model accuracy

The accuracy of predictions obtained using learning depends on three main factors, which we discuss next.

How far the predicted future is: It is critical to make predictions within a time budget. Since most flows in data centers are small, this budget is commensurately small: if the inference takes too long, we might either block on the inference and slow down the flow, or allow packets to flow without having the result of the inference and without tagging them appropriately, resulting in sub-optimal performance.

There are two possibilities for overcoming this issue: (a) when a new flow starts, make a prediction in an extremely small time budget by engineering down the inference time; and (b) when a new flow starts, start inference for the *next* outgoing flow, or even more generally, for some future flow. This choice represents a trade-off: we can either get high-accuracy inference by incorporating the maximum information available for inference, but incurring a data path latency to do so (or use results late, as they become available); or get lower accuracy due to missing some relevant information from needing to predict a farther future.

Fig. 4 shows the dependence of prediction accuracy on this “future distance”, starting from trying to predict a flow’s size

immediately when it starts, through predicting the next several flows. For TensorFlow, predicting several flows into the future is possible with high accuracy, because flows are predictive of future flows. But as expected, for the Web server workload, it is only possible to accurately predict the flow starting now, because two consecutive flows share no relationship (because we are requesting random objects from a Wikipedia mirror) – in essence, each “job” is of size one.

Model size: Larger models often yield higher accuracy at the cost of more memory and computation, and consequently, and more crucially, higher latency for inference.

While details of the impact of model accuracy on scheduling performance are deferred to §6, we use flow completion times instead of R^2 to compare model sizes. For an example trace (pFabric scheduling for PageRank), when predicting the current (next) flow’s size, the average FCT using a smaller model with 20 trees is worse by 9% (10%) than the larger model with 50 trees. Interestingly, the larger model achieves better results than the smaller one, even when the larger model is impaired by having to predict the *next* flow, while the smaller model predicts the current flow.

Training dataset size: Obviously, the learning approach depends on having seen enough training data, but this “convergence time” varies across workloads. For the Web workload, the model only needs to observe ~ 50 requests to achieve $R^2 > 0.5$. To achieve nearly its maximum prediction accuracy, the model needs to observe ~ 500 requests. For a popular Web server, this is on the order of a few seconds. (Of course, our model for a Web server is extremely simple.)

The model for TensorFlow needs to see ~ 3000 flows to reach peak accuracy, but given its flow arrival rate of more than 8200 flows per second, convergence time is sub-second. This is negligible compared to the job duration itself (~ 25 min). The iterative nature of neural network training, with similar traffic across iterations, allows accurate prediction within a few iterations of monitoring a never-seen-before job. The Spark workloads show the highest variability, and this is reflected in their convergence time. Here we need traces from multiple executions of the same job type to achieve high accuracy; 10 executions suffice for each of our 3 test job types⁵. Fortunately, data processing frameworks are often run repetitively with many instances of the same job [25, 16] since the workloads often involve tasks like making daily reports, code builds, backups or re-indexing data structures.

Note that good results can be achieved even across repeat executions with very different underlying data and run configurations — the job instances over which we train and test exhibit such variations, as discussed in §4.1.

5.4 Fast-enough, deployable learning?

With data center round-trip times on the order of $10 \mu s$, our objective is to achieve inference in a fraction of this time.

⁵Each execution yields n traces, when n machines execute the job.

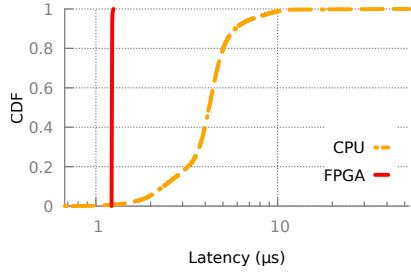


Figure 5: Inference latency across 100 measurements for GBDT with 50 trees implemented in-kernel on a CPU and using an FPGA.

CPU implementation: To efficiently implement GBDT, we use *treelite* [1], which takes an XGBoost model as input, and transforms it into a single C function, which is a long sequence of simple *if-else* statements. This approach incurs a minor slowdown when the model is updated (to generate C code), but improves inference performance by an order of magnitude in comparison to the original GBDT implementation in the XGBoost library [11]. This enables us to make inferences in $5\ \mu s$ within the typical case.

Accelerator implementation: The latency caused by the CPU implementation is small, but still comparable to data center RTTs. We investigate if offloading inference to specialized hardware could improve latency. Such logic could be built into NIC hardware, or deployed on the FPGAs already in use in some data centers [8].

We extend recent work on deploying GBDT on FPGA [32] to obtain our hardware implementation. Fig. 5 compares the inference latency for a 50-tree model on a CPU vs. using an FPGA. The mean latency is $4.3\ \mu s$ on the CPU, and $1.23\ \mu s$ on the FPGA. In each case, this includes the end-to-end time elapsed from when a new flow’s packet arrives in the kernel to when it has the result for packet tagging. The FPGA also eliminates the variance in software performance.

Thus, inference latency can be driven down to a fraction of the typical RTTs. Appendix A provides greater detail on our implementation for interested readers.

6 Improving network scheduling

Assessing accuracy of ML-based method in terms of mean error and R^2 is useful, but unsatisfactory — we ultimately want to understand the impact of errors on scheduling that uses the estimates. We thus quantify the performance of both flow-level (FastPass, pFabric, and pHHost) and coflow-level (Sincronia) schedulers with varying degrees of inaccuracy in flow sizes. Throughout this evaluation, we use the same traces used for our validation and testing results in §4.2.

6.1 Flow-level scheduling

We use the YAPS simulator [27]. We use the leaf-spine topology used in pFabric [5], with 4 spines, 9 racks, and 144 servers, with all network links being 10 Gbps. To measure the effect of inaccurate predictions on flow completion times

(FCT), we replay the network traces collected from our cluster in YAPS. Each experiment uses traces⁶ from one of the 5 job types. We run all our tests at 60% network utilization, mirroring the original pFabric and pHHost papers.

We compare network performance across the following flow estimators: (0) “Perfect”, an ideal predictor with zero error. (1) “Mean”, whereby every flow size is predicted to be the mean. (2) “GBDT”, the gradient-boosting decision tree learning approach with 50 trees. (3) Specifically for pFabric, we also evaluate the 0-knowledge LAS policy – “Aging” (§3.2). Today’s commonly deployed approach – FIFO scheduling at switches and ECMP forwarding – is also evaluated as a baseline (“Oblivious”).

Fig. 6 shows the average FCT across all 5 workloads, 3 flow-level scheduling techniques, and these flow estimators. Note that the Aging result is shown only for pFabric, because it can be easily modified to use LAS.

Oblivious often results in mean FCT more than $2\times$ that of Perfect, *e.g.*, compared to FastPass across all workloads, and compared to pFabric in Fig. 6(a) and 6(c); the largest gap is as large as $11.1\times$, vs. FastPass in Fig. 6(a). GBDT achieves mean FCT close to Perfect across all cases, with the largest gap being $1.21\times$, vs. FastPass in Fig. 6(e). Compared to Oblivious, improvements with GBDT range from $1.1\text{-}11.1\times$ across our experiments.

Understanding the performance of these schemes requires a closer look across the entire flow size distribution. Fig. 7 (left) shows the distribution for one example – pFabric scheduling over an SGD trace, *i.e.*, details behind the mean FCTs for pFabric in Fig. 6(a). Note that the logarithmic x -axis in Fig. 7 visually suppresses significant differences. Aging indeed achieves good results for the short flows for the SGD trace, but for longer flows, which share the same priority for a long time, its performance is worse than Oblivious, resulting in a larger mean FCT (Fig. 6(a)). The TensorFlow workload, with most flows being short, presents a difficult scenario for Aging – as noted in §3.2, for such workloads, Aging’s behavior is the opposite of desirable (Fig. 7 (right)).

In contrast to Aging, GBDT’s performance is similar to Perfect across the flow size spectrum for both workloads.

6.2 Coflow scheduling

We evaluate Sincronia, a recent proposal that leverages flow size information to provide near-optimal coflow completion time (CCT), with our imprecise flow size estimates.

We generate coflows from our traces by picking r consecutive flows grouped together to create a coflow. For each coflow, r is chosen uniformly at random from $\{1, 2, 3, \dots, 20\}$. For each of our five traces, we run 200 coflows with Sincronia’s offline simulator at 60% network load. We execute experiments for Perfect and GBDT,

⁶Note that, unfortunately, we cannot provide results for flow size distributions often used in data center research because we do not have the traces to produce the distribution of estimation error for them.

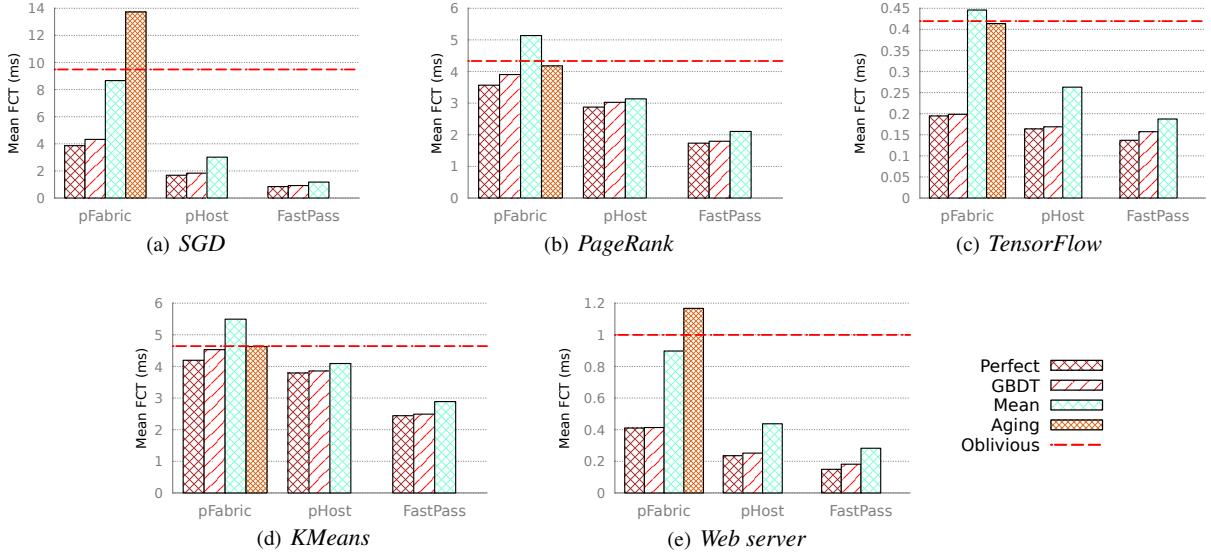


Figure 6: Mean FCT across 4 scheduling techniques, 5 workloads, and several flow size estimators.

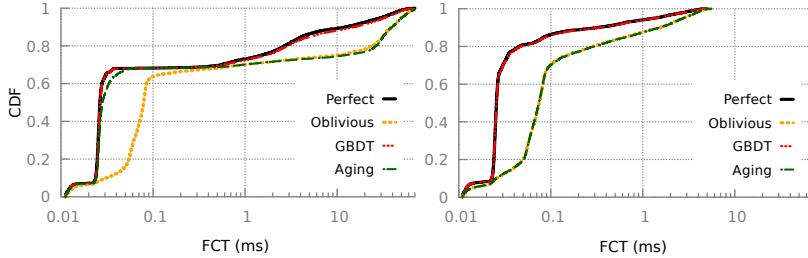


Figure 7: FCTs for pFabric for the SGD (left) and TensorFlow (right) workloads. Due to the log-scale, small visual differences are significant. On the right plot, Perfect and GBDT are visually indistinguishable, and so are Aging and Oblivious.

and record mean CCT. To measure the effect of inaccurate predictions, we define relative performance degradation as $GBDT\text{-}CCT / Perfect\text{-}CCT$.

Fig. 8 shows that the performance degradation for coflow scheduling for PageRank, KMeans and SGD, is substantially higher than for flow scheduling algorithms. That is because errors in estimates for individual flow sizes compound with coflows. This also explains why workloads with very high accuracy for individual flow sizes, such as Web server and TensorFlow, are only exposed to modest degradation.

7 Limitations of learning

It should be clear that the learning approach is not a panacea. There are several scenarios where it falls short. First and foremost, the prediction context should be clear, *i.e.*, the learning module has to identify the program that is responsible for sending a flow and monitor all features of interest for that flow, as described in §5.2. For Spark, the prediction context assumes knowing start time of a job as well as its ID. This is not unreasonable, as noted in §5.4.

However, for Web servers, we would have to tie disk and memory reads to particular requests. To demonstrate the ef-

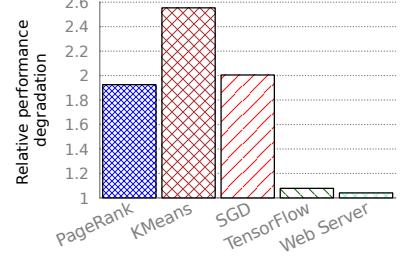


Figure 8: Relative performance degradation for Sincronia expressed as the ratio between mean CCT with imperfect estimates and perfect knowledge.

fect of missing context, we run Apache Tomcat, serving concurrent clients, so that it is not obvious how to match HTTP requests with corresponding disk reads and responses. In this case, disk reads become almost useless as an indicator, and we can only rely on system calls. The prediction accuracy can be made arbitrarily bad by tweaking the experiment parameters, so we omit a concrete accuracy number.

One possibility for obtaining such context is to apply *vertical context injection* [6], which is deployed in Google’s data centers; it tags system calls with application information for easier monitoring and debugging.

Further, for the execution of one-shot jobs without repetitive internal structure, there is clearly no learning potential. Likewise, for jobs where large, non-deterministic data volumes are generated (*e.g.*, computationally) for transmission, and there is little repetition across executions, it is unlikely that this approach can succeed.

Thus, for many workloads of practical interest, despite our best efforts, this approach will also be limited. We next discuss scenarios where learning or other heuristics can only estimate flow sizes for some fraction of the traffic.

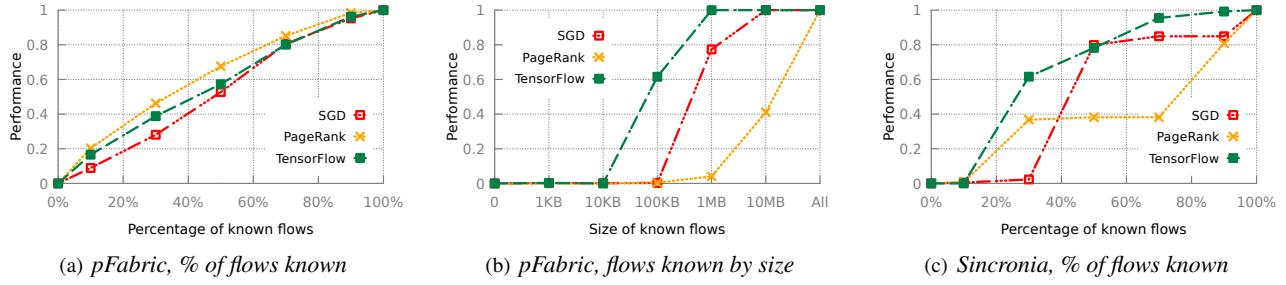


Figure 9: As more flow sizes are known, performance generally improves ...

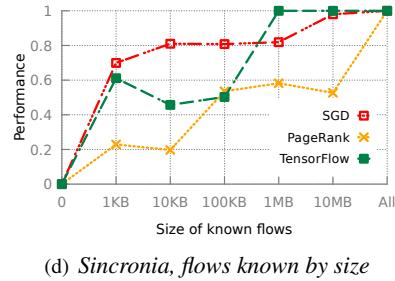


Figure 9: (continued) ... but for Sincronia, as larger and larger flows become known, performance sometimes degrades.

8 More knowledge \Rightarrow better performance?

While our exploration across several heuristics and an ML-based approach is promising, it is also clear that we will simply not get accurate flow size estimates for *all* applications. Thus, we advocate a pragmatic, two-fold approach: (a) Schedulers should tread a middle-ground – rather than giving up entirely on flow size estimation and operating in a non-clairvoyant manner, using estimates when they are available. (b) We should assess whether it is worth spending effort to expand the set of flows for which sizes can be estimated. We explore such pragmatism for flow- and coflow-level scheduling with pFabric and Sincronia.

To the best of our knowledge, past work has only touched on the first of these ideas. SOAP [40] and Karuna [9] have explored settings with a fixed proportion of flows of known and unknown sizes. Karuna combines pFabric’s shortest remaining first (SRF) approach for known flows with Aging⁷ for unknown flows. We refer to this policy as SRF-age, but consider a version with infinitely many priorities.

Instead of settings where we have size estimates for a fixed subset of flows, our interest is in examining what happens when we can invest in estimating a larger fraction of flows. In the following, we refer to flows with available sizes as “known” and other flows as “unknown”.

Knowing $x\%$ of all flows: If $x = 0$, SRF-age devolves to Aging, and if $x = 100$, it becomes Perfect (pFabric with full knowledge). We define performance with an arbitrary $x\%$ of flows known as the following normalization, where FCT_p is

⁷We are simplifying here; Karuna actually uses a multi-level feedback queue, with queue thresholds set based on the flow size distribution.

mean FCT with policy P :

$$Perf(x) = \frac{FCT_{SRF-age(x)} - FCT_{Aging}}{FCT_{Perfect} - FCT_{Aging}}$$

Fig. 9(a,b,c) shows the results on this normalized metric for a sample of workloads from §6. As more flow sizes become known, performance improves.

Knowing all flows of size up to x bytes: Given that some approaches, like using the initial system call, are more effective at estimating smaller flows, it is worth asking how much benefit knowledge of small flows gives. To evaluate this, we modify SRF-age as follows: We assign priorities to known flows following standard SRF, but for flows larger than x , we use $\max(age, x)$. This reflects our confidence that any unknown flow is larger than x bytes.

Fig. 9(b) shows that just knowing small flows will not improve performance drastically in terms of mean flow completion time, because they finish near the highest priority even in case of zero knowledge. However, their performance can improve if other, *larger* flows are known, and do not compete with small flows at the same high priority.

Coflow scheduling: Using Sincronia, we also explore the effects of having partial knowledge on coflow scheduling. We generate coflows in the same manner as in §6.2. We run Sincronia offline with 1000 coflows. Unfortunately, not having a packet-level implementation of Sincronia, coupled with the lack of a known or intuitive translation of Aging, limits our analysis. For unknown flows we thus assume that they are of the mean flow size of the whole trace, and refer to this policy as Sinc-mean.

We normalize performance with partial knowledge in the same manner as for pFabric, except using coflow completion times (CCT). The results with $x\%$ of flow sizes known and all flows smaller than x bytes known are shown in Fig. 9(c) and Fig. 9(d) respectively.

In some cases, knowing large fractions of flows does not improve CCT substantially. For instance, for the PageRank workload, knowing 70% of flows still gives more than 60% worse results than with perfect knowledge (Fig. 9(c)). This is due to unknown flows within a coflow acting like stragglers.

Fig. 9(d), oddly, indicates that sometimes adding knowledge decreases performance. We explain this with an exam-

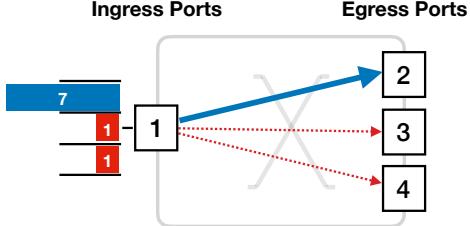


Figure 10: In this scenario, Sinc-mean scheduling policy leads to priority inversion and performance degradation when knowledge about certain flows is added to the system.

ple scenario, following a brief (simplified) overview of Sincronia. Sincronia finds a bottleneck port, *i.e.*, one with the largest number of bytes accumulative across flows; and then assigns the lowest priority to the largest coflow on that link. Flows within a coflow share priority.

Now consider a scenario with two coflows, with all their flows going from the same ingress port to different egress ports as shown in Fig. 10. Coflow c_1 contains only one flow with 7 packets, and coflow c_2 contains two flows of 1 packet each. The mean flow size is thus 3 packets. Regardless of which flows are un/known, with Sinc-mean, the ingress port would correctly be identified as the bottleneck. If all flows are unknown, Sinc-mean would consider all of them to be of the size 3. Sinc-mean would give c_1 higher priority, because its total estimated coflow size is 3 (compared to 6 for c_2), and, thus, finish c_1 first, within 7 time units. Now instead, say we had disclosed the size of c_1 's single constituent flow. This leads Sinc-mean to detect c_1 as the larger coflow (with size 7 for c_1 vs. an estimated 6 for c_2) and give higher priority to c_2 . In this case, c_1 finishes after c_2 with a coflow completion time of 9 time units. Thus, for c_1 , making its size known results in worse performance under Sinc-mean.

When does more knowledge help? Ideally, we would like the assurance that investing in learning about more flows only improves performance. Otherwise, there are limited incentives for data center operators and/or users to change their applications to expose flow size information or to deploy methods to estimate it.

This property clearly does not hold for Sinc-mean. It is as yet unclear to us how Aging could be incorporated into Sincronia, and whether a partial-knowledge variant can be developed that does not have the quirk of (sometimes) deteriorating when given additional knowledge. However, for the much simpler pFabric/SRF, we can prove a positive result in this direction, showing that for SRF-age, making a certain flow's size known can never deteriorate its performance, at least when interpreted in a worst-case manner.

Our simplified model assumes that all flows go through one link with unlimited output queuing. This output buffer queues packets in flow priority order. This implies that across different flows, packets leave the queue in priority order, but *within* flow packets leave in the same order as they arrive. At every timestep, either a packet leaves the queue,

or a new flow arrives. When flows arrive, all their packets are immediately added to this priority queue in their respective positions, with priority ties being broken randomly. To tackle this randomness, we define *worst-case scheduling* for a particular flow f_x as the schedule where any and all ties for f_x 's packets break against f_x .

For some flows, their flow sizes are known, and for others, they are not. For flows with unknown sizes, each packet uses the flow's age so far as its priority. The first packet of such a flow has the priority set to zero (highest), with successive packets seeing increments in priority value (*i.e.*, decreasing priority with more packets sent). (For brevity, we omit the distinction between packets and bytes and assume all packets are the same size.) In line with SRF, for known flows, the priority value for their last packet is zero (highest). If the size of a flow f is known, we denote it with f^k ; otherwise with f^u . We define priorities such that if $P(p)$ and $P(q)$ are the priorities of packets p and q , then $P(p) > P(q)$ implies p has higher priority, and is scheduled before q .

Theorem 8.1. All else fixed, with SRF-age, learning the flow size of a particular flow f_x cannot deteriorate its worst-case completion time, *i.e.*, $FCT(f_x^k) \leq FCT(f_x^u)$.

Proof. To prove the result, we shall show that every packet of any other flow that is scheduled before the end of f_x^k would have also been scheduled before the end of f_x^u , assuming worst-case scheduling for either. It is easy to see that this would imply that the FCT for f_x^k in a worst-case schedule cannot be worse than the FCT of f_x^u .

Suppose a packet r of some other flow is scheduled before a packet p_x^k in f_x^k , given worst-case scheduling for f_x^k . This scheduling implies r has priority higher than or equal to p_x^k , *i.e.*, $P(r) \geq P(p_x^k)$.

Now, say the last packet of f_x^u is l_x^u . Notice that this last packet of f_x^u must have priority lower than or equal to *all* packets of f_x^k , including p_x^k , *i.e.*, $P(l_x^u) \leq P(p_x^k)$. This follows from the definition of SRF-age. If the size of a flow f is $|f|$ packets, then the last packet of f^u (per aging) has priority value $|f| - 1$. The n^{th} packet of f^k has priority value $|f| - n$.

Putting the above two inequalities together yields $P(r) \geq P(l_x^u)$. Thus, r would also be scheduled before l_x^u (which is the end of f_x^u), at least in the worst-case schedule for f_x^u . \square

A few remarks about this result are in order:

- The theorem and the proof specify worst-case tie-breaking for the flow under consideration. It is easy to produce counterexamples to the theorem statement without the worst-case addendum.
- The definition of SRF-age is central to the result, and it is easy to produce counterexamples for an analogous statement for SRF-mean.
- For systems with a limited number of priority queues, like Karuna or PIAS, the theorem still holds if both known and unknown flows share the same priority thresholds.

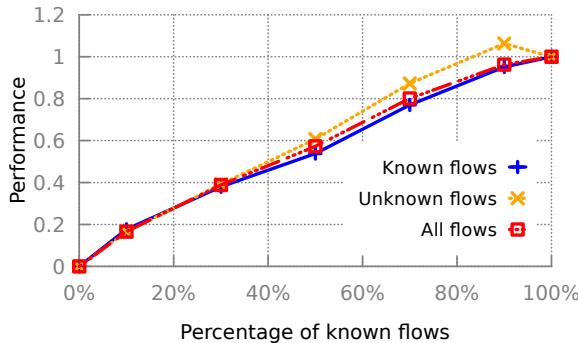


Figure 11: For the TensorFlow workload, with TCP, unknown flows finish faster under SRF-age.

- The result can appear counter-intuitive; after all, large unknown flows benefit from high priority in the beginning, which they wouldn’t if they were known. While this is true, unknown flows keep slowing down with aging, while known flows keep speeding up with SRF. The proof formalizes this idea.
- Our model, like past work, assumes that scheduling does not change packet inputs to the queue. This is *not true* for TCP flows entering a finite queue.

The impact of the last issue above has not been explored deeply in prior work on packet scheduling. To illustrate its impact in practice, we take a closer look at the TensorFlow trace in Fig. 9(a), separating out the FCTs for known and unknown flows. As Fig. 11 shows, unknown flows finish somewhat faster. This apparent deviation from our theorem’s result stems from our simple model which ignores TCP dynamics, assuming instead that all packets of a flow are available for scheduling at its arrival time. The TensorFlow workload comprises nearly 90% flows of sizes smaller than 100 KB. For unknown flows of this type, Aging results in higher priority in the beginning, allowing TCP’s exponential slow-start to grow such flows faster than flows with known sizes. Incorporating TCP dynamics into our model to potentially bound the disadvantage that known flows can suffer will require substantial additional effort, which is left to future work. While this discrepancy and its impact on scheduling results should be examined in greater detail, this does not take away from our results on incremental benefits from having greater knowledge with SRF-age scheduling overall.

Mean completion time across all co/flows: Although we have shown that SRF-age cannot deteriorate the performance of a particular flow when its size is made known, it is easy to produce examples where it hurts mean flow completion time across the set of all flows⁸. While our empirical results show improved mean FCT with SRF-age (and an overall trend for improvement even for mean CCT with Sinc-mean), a fuller analysis of this issue is left to future work.

⁸This is also true of Sinc-mean for mean coflow completion time.

9 Related work

We have described relevant past work in context throughout, discussing past efforts on using various types of heuristics for estimating flow sizes in §3, learning-based efforts that operate at slower timescales in §3.5, and work on non-clairvoyant methods that do not use flow sizes in §2 and §3.2.

CODA [45] merits mention as a coflow scheduler that acknowledges imprecision in scheduling inputs, and explicitly handles such imprecision. However, CODA’s focus is on clustering flows into coflows, rather than on flow size information, for which it also relies on PIAS-like techniques.

We also discussed Karuna [9] and SOAP [40] in §8, but as the closest prior efforts considering scheduling in a mixed setting with flow sizes (or deadlines, job sizes, etc.) available and not available, we highlight our contributions in comparison to these here. Both Karuna and SOAP only explore scheduling in a mixed setting with a fixed set of known and unknown flows, while our work (a) systematically examines ways of expanding the set of flows for which size estimates are available; (b) evaluates the utility of imprecise estimates across multiple scheduling approaches; and (c) takes first steps towards assessing how the incremental addition of knowledge about flows impacts scheduling, with interesting results for both flow and coflow scheduling.

10 Conclusion

While clairvoyant scheduling promises large performance benefits across a variety of network scheduling problems, its assumption of advance knowledge of flow sizes is, at best, optimistic. Our analysis of how such information may be obtained reveals several settings where even our best efforts are bound to fail. Superficially, this would suggest focusing on non-clairvoyant scheduling, but we argue that such absolutism is unnecessary – we should be using flow size information where available, and examining whether estimating it for more flows yields additional improvements in scheduling. Along these lines, we present several heuristics and a practically implementable learning-based approach to expand the scenarios where flow size knowledge is available. We further show empirically and analytically, that incrementally adding such knowledge is helpful for SRF packet scheduling. For coflow scheduling, we find that small errors in flow size estimation get compounded, leaving a sizable performance gap compared with fully clairvoyant coflow scheduling. We also find that for at least some intuitive policies for scheduling with partial information, additional information can *deteriorate* scheduling, thus necessitating deeper examination of this issue in future work.

Acknowledgments

We are grateful to Kai Chen, Mosharaf Chowdhury, Rachit Agarwal, and the anonymous NSDI reviewers, for their feedback; and to George Porter for shepherding our paper.

References

- [1] Treelite : toolbox for decision tree deployment. <http://treelite.readthedocs.io/en/latest/>, 2017.
- [2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.
- [3] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *IEEE CloudCom*, 2014.
- [4] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat. Sincronia: near-optimal network design for coflows. In *ACM SIGCOMM*, 2018.
- [5] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, 2013.
- [6] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *USENIX NSDI*, 2018.
- [7] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-agnostic flow scheduling for commodity data centers. In *USENIX NSDI*, 2015.
- [8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *IEEE/ACM MICRO*, 2016.
- [9] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling mix-flows in commodity datacenters with Karuna. In *ACM SIGCOMM*, 2016.
- [10] L. Chen, J. Lingys, K. Chen, and F. Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *ACM SIGCOMM*, 2018.
- [11] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *ACM SIGKDD*, 2016.
- [12] F. Chollet et al. Keras. <https://github.com/keras-team/keras>, 2015.
- [13] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM*, 2015.
- [14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *ACM SIGCOMM*, 2011.
- [15] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *ACM SIGCOMM*, 2014.
- [16] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SOSP*, 2017.
- [17] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *IEEE INFOCOM*, 2011.
- [18] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM*, 2014.
- [19] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *IEEE HPCA*, 2010.
- [20] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM*, 2010.
- [21] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *ACM CoNEXT*, 2015.
- [22] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, 2012.
- [23] K. Hornik, M. Stinchcombe, and H. White. Multi-layer feedforward networks are universal approximators. *Neural Netw.*, 2(5), 1989.
- [24] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1999.
- [25] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayananmurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *USENIX OSDI*, 2016.
- [26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

- [27] G. Kumar, A. Narayan, and P. Gao. YAPS network simulator. <https://github.com/NetSys/simulator>, 2015.
- [28] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *USENIX HotCloud*, 2014.
- [29] Y. Lu, G. Chen, L. Luo, K. Tan, Y. Xiong, X. Wang, and E. Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM*, 2017.
- [30] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, 2018.
- [31] A. Mushtaq, R. Mittal, J. McCauley, M. Alizadeh, S. Ratnasamy, and S. Shenker. Datacenter congestion control: Identifying what is essential and making it practical. <https://people.eecs.berkeley.edu/~radhika/adsrpt.pdf>, 2017.
- [32] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. In *FPL*, 2017.
- [33] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu. Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing. In *IEEE INFOCOM*, 2014.
- [34] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, 2014.
- [35] I. A. Rai, G. Urvoy-Keller, M. K. Vernon, and E. W. Biersack. Performance analysis of LAS-based scheduling disciplines in a packet switched network. In *ACM SIGMETRICS*, 2004.
- [36] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*, 2012.
- [37] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2014-11-17 for version 2.1. Posted at <https://github.com/google/cluster-data>.
- [38] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. *Google Inc., White Paper*, pages 1–14, 2011.
- [39] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network’s (datacenter) network. In *ACM SIGCOMM*, 2015.
- [40] Z. Scully, M. Harchol-Balter, and A. Scheller-Wolf. SOAP: One clean analysis of all age-based scheduling policies. In *ACM SIGMETRICS*, 2018.
- [41] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. c-through: Part-time optics in data centers. In *ACM SIGCOMM*, 2010.
- [42] H. Wang, L. Chen, K. Chen, Z. Li, Y. Zhang, H. Guan, Z. Qi, D. Li, and Y. Geng. FlowProphet: Generic and accurate traffic prediction for data-parallel cluster computing. In *IEEE ICDCS*, 2015.
- [43] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM*, 2011.
- [44] D. Xie, N. Ding, Y. C. Hu, and R. Komella. The only constant is change: Incorporating time-varying network reservations in data centers. In *ACM SIGCOMM*, 2012.
- [45] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng. CODA: Toward automatically identifying and scheduling coflows in the dark. In *ACM SIGCOMM*, 2016.
- [46] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM IMC*, 2017.

A Deployment architecture

Different deployment options expose different flexibility-performance tradeoffs. We first discuss FLUX’s placement in the app-kernel interface in a controlled environment, and then usage in virtualized environments. For illustration, without loss of generality, we consider the pFabric use case, tagging packets with the remaining flow size.

A.1 Where does FLUX operate?

There are three possibilities for partitioning FLUX’s components across user- and kernel-space, shown in Fig. 13.

The data collector must have visibility of app I/O calls, so it must be implemented as either a library to intercepts these, or within the kernel. The packet tagger must sit in the kernel to efficiently manipulate packets. Model training operates off-datatype, and can sit essentially anywhere. The key question is: where is inference implemented?

Fig. 13(a) – as a separate process: Inference is a standalone process, serving requests from the syscall interceptor whenever a new flow starts. This approach makes changing or updating the prediction model trivial. Since the model is a single C function, it can be compiled to a shared library, and loaded dynamically by the inference module.

When an app issues a *send*, the data collector requests inference; a response for which is sent to the tagger. The inference path is much longer than the data path, and packets will arrive for tagging before the inference. This blocking time distribution is shown in Fig. 12(a) across 1000 flow starts. The median (95th percentile) latency is 67.4 μ s (720 μ s).

Fig. 13(b) – as an interposed library: To reduce inter-process communication, inference runs in the same library as the collector. The *send* is intercepted and then issued to the kernel *after* inference finishes, with the predicted flow size. The application perceives this process as a long system call. The median (95th percentile) latency is 4.97 μ s (16.7 μ s).

Fig. 13(c) – as a kernel module: All components except learning can be implemented in the kernel. This kernel module itself runs inference for new flows, and communication with the tagger uses shared kernel memory. The latency comprises entirely of inference, and is shown in Fig. 12(b) for different model sizes, with an average of 4.3 μ s for the 50-tree model. This approach is quite inflexible: given that different apps could need different inference models, a new model must be inserted in the kernel for each new app.

A.2 Virtualization and offload

Virtualization: For containers, operation similar to a non-virtualized environment works. For a guest OS, FLUX must be interposed in the guest-host interface. Ultimately, the guest is an “application” on the host, and the network interface is similar, so this does much in terms of performance and accuracy, except in cases where some networking functionality is additionally offloaded to hardware.

Hardware offload: Parts of the network stack may be implemented in hardware, or VMs may interact directly with hardware, such as with SR-IOV [19]. Nevertheless, these environments still must expose a similar *send*, *recv* API to underlying layers, which FLUX can intercept. However, in these settings, FLUX must be implemented as part of a smart NIC. **This is fundamental to any method of using such packet tagging** for network scheduling, because the hypervisor may not touch individual packets at all.

RDMA stacks: RDMA has a significantly different API than TCP. However, even for RDMA networking stacks, the API exposes similar information about sent and received data, which FLUX can exploit.

A.3 Inference speed

Fig. 12 shows how inference latency depends on the size of GBDT models on CPU and on FPGA. For the 50-tree model, with which we obtain reasonable accuracy on our traces, the FPGA can achieve latency under 1.3 μ s. Such low latency is possible because the FPGA is connected to the CPU through the Intel’s QPI interconnect which guarantees very short FPGA-CPU round trip. On the other hand, a PCIe-attached FPGA exhibits a slower round trip (on the order of 2.5 μ s).

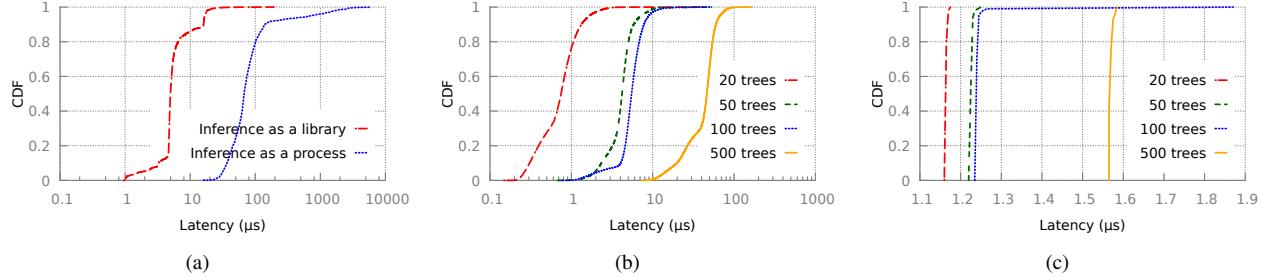


Figure 12: *Inference latency:* (a) 50 tree GBDT implemented in a library vs. in a process; as a function of model size (b) in-kernel on a CPU; and (c) using an FPGA.

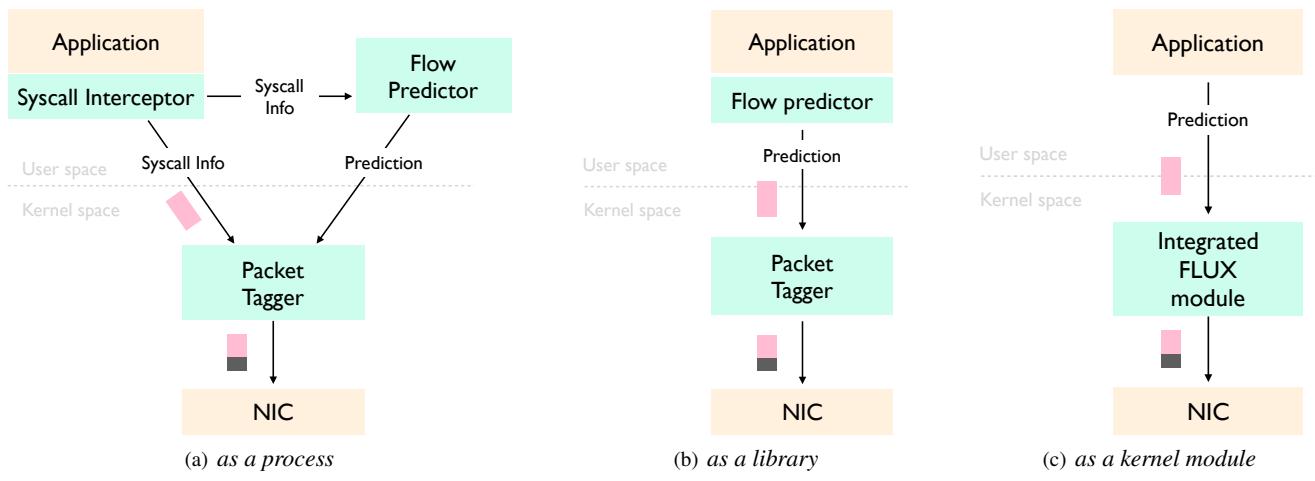


Figure 13: *Three possibilities for partitioning FLUX's components across user- and kernel-space.*

Stable and Practical AS Relationship Inference with ProbLink

Yuchen Jin[†]

Colin Scott[‡]

Amogh Dhamdhere[◊]

Vasileios Giotsas^{*}

Arvind Krishnamurthy[†]

Scott Shenker[‡]

[†]*University of Washington* [‡]*UC Berkeley and ICSI* [◊]*CAIDA* ^{*}*University of Lancaster*

Abstract

Knowledge of the business relationships between Autonomous Systems (ASes) is essential to understanding the behavior of the Internet routing system. Despite significant progress in the development of relationship inference algorithms, the resulting inferences are impractical for many critical real-world applications, cannot offer adequate predictability in the configuration of routing policies, and suffer from inference oscillations. To achieve more practical and stable relationship inference, we first illuminate the root causes of the contradiction between these shortcomings and the near-perfect validation results for AS-Rank, the state-of-the-art relationship inference algorithm. Using a “naive” inference approach as a benchmark, we find that available validation datasets over-represent AS links with easier inference requirements. We identify which types of links are harder to infer and develop appropriate validation subsets to enable more representative evaluation.

We then develop a probabilistic algorithm, ProbLink, to overcome the challenges in inferring hard links, such as non-valley-free routing, limited visibility, and non-conventional peering practices. ProbLink reveals key AS-interconnection features derived from stochastically informative signals. Compared to AS-Rank, our approach reduces the error rate for all links by $1.6\times$ and, importantly, by up to $6.1\times$ for various types of *hard* links. We demonstrate the practical significance of our improvements by evaluating their impact on three applications. Compared to the current state-of-the-art, ProbLink increases the precision and recall of route leak detection by $4.1\times$ and $3.4\times$ respectively, reveals 27% more complex relationships, and increases the precision of predicting the impact of selective advertisements by 34%.

1 Introduction

The Internet, often referred to as a “network of networks”, is composed of more than 60,000 Autonomous Systems (ASes). These ASes co-operate via the Border Gateway Protocol (BGP) to exchange routing information and obtain global reachability. The connections between ASes are shaped by business contracts that determine the economics and technical aspects of traffic exchange. For over 15 years, researchers have studied the problem of inferring the different types of relationships between ASes from publicly available BGP routing data. Relationship inferences are used for a wide range of applications and areas of research, such as detecting network congestion [13], identifying malicious ASes [36, 12], deploying incentive-compatible BGP security

mechanisms [23, 11], protecting the integrity of anonymization [47, 34], optimizing video streaming [38, 18, 30], and understanding Internet governance and the ramifications of public policy proposals [39, 40, 30].

In this paper, we revisit the AS relationship inference problem. We find, as others have, that available relationship inference algorithms perform poorly in many critical applications [44, 6, 47, 43, 57]. We seek to understand why state-of-the-art algorithms are insufficient, despite extensive validation that indicates an error rate as low as 1%. In particular, we consider the sophisticated AS-Rank technique [41] which is carefully crafted using eleven deterministic heuristics. As a first step in assessing the performance of AS-Rank, we create a baseline benchmark algorithm, CoreToLeaf, that consists of three simple steps and only assumes valley-free paths through a core set of transit-free ASes. In spite of its simplicity, CoreToLeaf achieves accuracy that is almost as high as that of AS-Rank. At the same time, we evaluate CoreToLeaf and AS-Rank in practical applications—detection of route leaks and the analysis of selective advertisements—which reveals that the performance of both algorithms falls short of the needs of those applications.

CoreToLeaf’s high accuracy against the validation datasets implies that the majority of AS-links in the validation datasets are relatively easy to infer. Yet the sub-optimal performance of both algorithms in practical applications indicates that the small minority of AS-links that are difficult to infer are crucial for those applications. We select subsets of the validation dataset that contain AS links that we consider *hard*, and find that both the CoreToLeaf and AS-Rank techniques have substantially lower accuracy on these validation subsets (confirming where these current algorithms fall short).

We next examine the challenges in developing a more accurate AS relationship inference algorithm. We observe first that the attributes of a link (and those of the paths that traverse the link) that might be used by an AS-relationship inference algorithm are noisy and often have only a weak correlation with the link’s relationship type. Second, many links appear in paths that likely violate the valley-free assumption made by existing algorithms. Third, existing algorithms are sensitive to the locations of the vantage points and the order in which the link relationships are inferred. An AS relationship inference technique must address the above challenges if it is to achieve higher accuracy for hard links.

We develop a *probabilistic AS relationship inference* algorithm, ProbLink, to address the above issues.

ProbLink provides a framework that allows for easy integration of many noisy but useful attributes into the relationship inference algorithm. ProbLink enables us to identify a set of link attributes that take into account not only observed paths but also information gleaned from the fact that certain paths are *not* observed. ProbLink allows for links to appear in paths that violate the valley-free property but attributes a lower probability to such occurrences. ProbLink uses an iterative algorithm that repeatedly infers link types based on statistical distributions of link attributes until the inferences reach a fixed point.

Our evaluation of ProbLink show that it achieves an error rate that is better than that of AS-Rank overall by $1.7\times$, and achieves $1.8\text{-}6.1\times$ better error rate for various categories of *hard* links. We find that even the small improvement in overall accuracy brought by ProbLink has a significant impact when applied to real-world applications. Compared to the current state of the art, ProbLink increases the precision and recall of route leak detection by $4.1\times$ and $3.4\times$ respectively, reveals 27% more complex relationships, and increases the precision of predicting the impact of selective advertisements by 34%.

2 Background and Related Work

The Border Gateway Protocol (BGP) is the mechanism used by ASes to exchange reachability information. A BGP AS path is a sequence of ASes denoting the routing path that the first AS in the path prefers to reach a destination prefix. The last AS in the path is referred to as the “origin AS” of the prefix. Each AS uses a complex decision process to select the most preferred path toward each destination prefix [9]. BGP route collection infrastructure is operated by Routeviws [4] and RIPE NCC [3]. This consists of routers that peer with ASes that volunteer to provide their BGP routing advertisement feeds for research or operational reasons. A route collector is configured to obtain the *best paths* from ASes it peers with, the most preferred available path towards each destination starting from that AS. Route collectors typically peer with several ASes, and thus obtain multiple best paths to each destination prefix.

AS relationships fall into two broad categories: customer-provider (c2p) and settlement-free peering (p2p). In a c2p relationship, the customer AS pays the provider AS for reachability to/from the rest of the Internet. In a p2p relationship, two networks agree to exchange traffic destined to prefixes they or their customers own without an associated fee. In practice, AS relationships can span a spectrum of types between c2p and p2p. These *hybrid or complex relationships* can occur when two ASes have multiple contractual agreements, one for each geographical region where an interconnection exists [27]. *Sibling* relationships exist between distinct ASes that are owned by the same organization and can exchange traffic without any cost or routing restrictions.

The *customer cone* of an AS X is the set of ASes that X can

reach using only p2c links. The size of the customer cone is an indication of the market power of an AS. A clique of *Tier-1 ASes* at the top of the Internet AS hierarchy are “transit-free”, meaning that they have routes to all other networks on the Internet through customer or peering links without the need to pay for transit.

2.1 AS Relationship Inference Techniques

Beginning with the seminal work by Gao [21], most AS-relationship inference algorithms are based on the assumption that valid BGP paths are *valley-free*, i.e., a path consists of zero or more c2p links, followed by zero or one peering link, followed by zero or more p2c links. This assumption captures the economic incentives that (at least partially) determine traffic exchange between ASes: an AS should not intentionally advertise routes learned from a peer or provider to another peer or provider, since this “free transit” increases infrastructure costs but provides no remuneration.

Another observation made by Gao and others [55, 14, 15, 61] is that providers usually have a higher *node degree* (i.e., the number of ASes to which an AS node directly connects to) than customers, while peers usually have similar degrees. *Node degree* is the number of neighbors an AS directly connects to, irrespective of whether the neighbors are providers, peers, or customers. Willinger et al. have shown that node degree is significantly biased by the fact that the available topological data reveals only a subset of the complete Internet topology, due to limited placement of vantage points adjacent to peer-to-peer AS links [60].

The state-of-the-art AS relationship inference technique, called the “AS-Rank” algorithm [41], makes three generally accepted assumptions: 1) there is a clique of large transit providers at the top of the hierarchy, 2) most customers purchase transit in order to be globally reachable, and 3) there are no cycles of p2c links. The AS-Rank algorithm takes 11 intricate steps to label each link as customer-provider (abbreviated as c2p or p2c depending on the directionality of the relationship) or peer-to-peer (p2p). An abbreviated version of the AS-Rank algorithm is available in Appendix A for ease of reference.

It is worth noting a few properties of the AS-Rank algorithm. First, AS-Rank uses the *transit degree* attribute as one of the main sources of information in determining relationship labels. *Transit degree* is the number of ASes that appear on either side of an AS in adjacent links of BGP paths, but it does not count neighbors for which the given AS does not transit traffic. Transit connectivity is easily observable by Route Collectors (except for backup or partial-transit links [29]), therefore it provides a more robust metric to describe an AS’s prominence than node degree. Second, AS-Rank considers ASes and links in a specific order, using the *transit degree* information in certain cases (step 5) and not in others (step 7).

3 Input Datasets

3.1 BGP Paths

We collect BGP paths towards IPv4 prefixes from RouteViews [4] and RIPE RIS [3]. In September 2018, both projects operated 22 collectors, which in total connect with more than 1,000 vantage points (VP) worldwide. Each RouteViews and RIPE RIS collector dumps a snapshot of their Adj-RIB-out tables every 2 hours and every 8 hours respectively. For the purpose of evaluating the various algorithms over longitudinal data (as discussed in §4 and §7), we consider snapshots of BGP paths on the first day of April, August, and December (i.e., every four months) since 2006.

After collecting BGP paths, we parse them to remove duplicated ASes that result from BGP path *prepending*. We also filter out paths with AS loops, i.e., when an ASN appears more than once and is separated by at least one other ASN. We also sanitize the BGP paths by removing paths containing reserved ASes [51]. Loops and reserved ASes showing up in a path are artifacts of route poisoning [8, 35].

3.2 Sibling Relationships

We use CAIDA’s AS-to-organization mapping dataset [10], which is derived from WHOIS data, to identify sibling links. This dataset provides quarterly information starting from 2009. We infer links between ASes that are operated by the same organization as sibling relationships.

3.3 IXP List

ASes often establish p2p relationships over shared switching fabric provided by IXPs. To facilitate dense peering connectivity, IXPs provide BGP Route Servers over which ASes establish many-to-many (multilateral) interconnections. To enable layer-3 connectivity Route Servers typically have their own ASN, but according to best practices it should be filtered-out from the AS path since the Route Server does not participate in the routing decision process [33]. However, for debugging reasons, some IXP members append the Route Server ASN in the BGP path. We sanitize BGP paths to remove Route Server ASNs since essentially the peering links are between the IXP members, and not between the IXP and ASes. To collect a list of AS Numbers (ASNs) used by IXP Route Servers, we query PeeringDB [2] for networks of type “Route Server” and extract the ASN. We augment this list by consulting the Euro-IX IXP Service Matrix [1] and extracting the Peering LAN ASN and Route Server ASN for each IXP. There were 172 IXP ASes in this list on 12/01/2017.

3.4 Validation Dataset

AS operators frequently encode the relationship type with their neighbors directly in their prefix advertisements using BGP Communities, an optional transitive BGP attribute used to attach metadata on BGP paths. While the use of communities attribute is not standardized, many ASes publicly document the meaning of their BGP communities on websites

Date (MM/DD/YYYY)	# links in validation set	# links in total	Percentage
04/01/2012	7,833	117,872	6.6%
04/01/2013	11,644	133,459	8.7%
04/01/2014	44,875	159,678	28.1%
04/01/2015	47,036	176,791	26.6%
04/01/2016	52,931	204,309	25.9%
04/01/2017	56,326	213,441	26.4%

Table 1: Size of the validation dataset vs. all links observed from all VPs.

and in IRR databases, enabling us to assemble a dictionary of BGP communities that denote relationship type. We used a dictionary of 1286 community values from 224 different ASes to construct a set of relationships from BGP data starting quarterly from April 2006 to April 2017. Similar to prior work [41], we treat this dataset as “best-effort” validation to evaluate existing inference techniques and our proposed approaches.

Table 1 shows the size of this validation dataset over the past 6 years. The coverage of our validation dataset increased from 6.6% in 2012 to about 26% of the observed links in recent years, due to the increasing popularity of BGP communities and the deployment of additional VPs that allow more communities to propagate to BGP collectors. As prior work has pointed out, links involving *Tier-1* ASes and VP ASes are over-represented, because public data on BGP communities mostly comes from large ASes [41], while communities from non-VP ASes may be stripped out during the propagation of BGP routes. However, unlike prior work, we take these biases into consideration during our evaluation.

As noted above, the use of BGP communities has become increasingly popular [24], raising the question of whether we can eventually exclusively rely on communities to extract relationships without the need for an inference algorithm. Even with prevalent use of BGP communities however, we would face two important limitations. While communities are by default a transitive attribute, in practice operators often strip out community tags before propagating advertisements to neighbors. Indeed, if all the communities in our dictionary were transitively propagated to our BGP collectors, our validation dataset should have over 58% coverage of the visible AS links. Instead, as shown in Table 1 our coverage is less than 30% for the past 6 years. A second limitation with communities is partial availability of publicly available documentation of those attributes. Despite our best efforts to maximize the number of interpretable community values via automated web scraping and text processing tools, we are only able to find authoritative documentation on the meaning of 35% of visible community values.

4 Establishing a Benchmark for Hard Links

As explained in the previous section, the evaluation dataset is extensive but biased toward specific types of links. It is important to understand if the links over-represented in the val-

idation dataset are easier to infer correctly, compared to the under-represented links, which may skew the overall evaluation results. To this end we develop CoreToLeaf, a very simple algorithm that allows us to understand which links are easy to infer. CoreToLeaf uses only the *valley-free* assumption and the list of Tier-1 ASes to infer relationships. We show that the inference accuracy of this algorithm is almost as high as that of the more sophisticated AS-Rank algorithm elaborated in §2.1, while the accuracy of both algorithms suffer for certain categories of links. Our findings reveal that indeed certain types of under-represented links in the evaluation dataset are harder to infer, possibly inflating the overall accuracy of past work. We address this issue by constructing distinct validation sub-datasets as benchmarks for hard links.

4.1 The CoreToLeaf Algorithm

CoreToLeaf starts by inferring a clique of Tier-1 ASes using the same inference method as AS-Rank. For each path that traverses a Tier-1, we skip the first link after the Tier-1 and label all succeeding links as p2c. For example, if AS_2 is a clique member in a BGP path “ $AS_1, AS_2, AS_3, AS_4, AS_5, AS_6$ ”, we infer links $\langle AS_4 - AS_5 \rangle$ and $\langle AS_5 - AS_6 \rangle$ as p2c. We skip inferring the relationship for $\langle AS_2 - AS_3 \rangle$ because it could either be a p2c or a p2p, but all subsequent links need to be p2c assuming that the path is *valley-free*. (Note that if AS_1 is a clique member, we would have labeled $\langle AS_2 - AS_3 \rangle$ also as a p2c link.) Finally, we label all remaining unclassified links as p2p.

In the step of labeling p2c links, a link could be labeled more than once if it shows up in multiple paths. In some cases, a link could be labeled as a p2c in some path and as a c2p when traversing a different path. We label this link as a “conflict” link when we encounter such an inconsistency.

Note that CoreToLeaf does not take into account degree or transit degree information, nor does it use paths that do not go through Tier-1s. This is in contrast to other traditional algorithms; for example, Gao’s algorithm [21] considers all paths, identifies the AS with the highest node degree in each AS path and treats it as the top provider, and then labels AS pairs before it as c2p or sibling and AS pairs behind it as p2c or sibling. The rationale behind CoreToLeaf is simply that there is greater certainty that it is customer routes that are being transitively exposed to Tier-1s and that there is less likelihood of paths being exported to Tier-1s due to complex peering mechanisms.

4.2 Evaluation

We evaluate this extremely simple algorithm against our validation dataset on 04/01/2017, which contains 23,528 p2p links and 32,798 p2c links (corresponding to 26.4% of the visible topology). Table 2 compares the *precision* (true positives / (true positives + false positives)) and *recall* (true positives / (true positives + false negatives)) of CoreToLeaf and AS-Rank. Surprisingly, CoreToLeaf achieves high preci-

Algorithm	p2c		p2p		Conflict (%)
	Precision (%)	Recall (%)	Precision (%)	Recall (%)	
CoreToLeaf	98.9	95.8	95.0	98.8	0.12
AS-Rank	97.8	97.5	98.8	98.9	0

Table 2: Precision and recall of CoreToLeaf and AS-Rank.

sion and recall for both p2c and p2p links (comparable to AS-Rank), with higher *precision* on p2c relationships (98.9% compared to 97.8%), and a small fraction of links labeled as ‘conflict’.

The 1.1% mistakenly inferred p2c links and the links which CoreToLeaf labels as ‘conflict’ are due to *valley-free* violation, which we quantify later in §5.2. Since the step of labeling p2c links uses just paths through Tier-1s, it fails to capture 4.2% (95.8% *recall*) of the actual p2c links. Consequently, these links are inferred as p2p in the third step and results in a 5.0% error rate for links labeled as p2p.

The accuracy of CoreToLeaf and AS-Rank seem quite high, but they perform sub-optimally when applied to real-world applications. Route leaks constitute a type of prevalent routing incident that can cause significant disruptions to Internet routing [54, 28]. In §8, we describe how we can use inferred relationships to detect route leaks and evaluate the effectiveness of AS-Rank inferences. Only 19% of the route leaks detected using AS-Rank were real route leaks, and almost 80% of the real leaks were missed. We observe relatively poor performance for two more applications we tested, as discussed in detail in §8. The high application-level error rates illustrate that a better AS relationship algorithm is needed for real-world applications.

4.3 Identifying Hard Links

The surprisingly high accuracy obtained by CoreToLeaf has many implications. First, it indicates that simple techniques might suffice for inferring the types of many of the links in the validation dataset. Second, it underscores the need for more comprehensive validation datasets that would be more representative of AS links beyond those associated with Tier-1 and VP ASes. Third, in the absence of more comprehensive validation datasets, one way to make progress on improving and evaluating AS relationship inference algorithms is to identify specific types of links for which the current algorithms do not work well. We therefore now attempt to extract collections of *hard* links from the overall validation dataset based on the inference performance of CoreToLeaf and AS-Rank.

We feed a large set of features of every link in the validation dataset along with information on whether a link was labeled as “inferred correctly” and “inferred incorrectly” by CoreToLeaf and AS-Rank into a *gradient boosted decision tree* [20], and calculate the feature importance for accurate predictions for the two algorithms. The feature importance calculation and results are presented in Appendix C. We extract the following five categories of “hard” links suggested

by the feature importance analysis and the CoreToLeaf algorithm.

1) Links with max node degrees smaller than 100. The feature importance analysis shows that CoreToLeaf and AS-Rank do not have high accuracy for links whose endpoint ASes both have small node degrees.

2) Links observed by more than 50 but less than 100 VPs. The feature importance analysis also reveals that links observed by at least 50 VPs but not more than 100 VPs are hard to infer correctly. The reason is that p2p links are often observed by few VPs and transit links are often observed by many VPs, so it is hard to distinguish the link types for the range in the middle.

3) Non-VP and non-Tier1 links. In general, a link that is directly connected to a VP or a Tier-1 is likely to appear in many BGP paths, and the AS inference algorithm is likely to have access to more information regarding the link. Moreover, most of our validation dataset are links that are connected to a VP or Tier-1 AS, so we want to specifically analyze the performance of inference algorithms on the “underrepresented” links in our validation dataset.

4) Unlabeled stub-clique links in CoreToLeaf. A stub AS connects with only one other AS through which it gains access to the entire Internet. A stub-clique link is a link whose one endpoint is a stub AS and the other endpoint is in the Tier-1 clique. In other words, the clique member is the only AS to which the stub AS connects. These links typically have very high transit degree difference, which is an important feature as shown by the feature importance analysis.

In CoreToLeaf, a stub-clique link $\langle X, Y \rangle$ (where X is a stub AS and Y is a clique AS) is inferred as a c2p *iff* there is a path containing an AS triplet “Z, Y, X” where Z is also a clique AS. We call the set of stub-clique links that are not inferred as c2p in the second step of CoreToLeaf (i.e., they are inferred as p2p in the later step) as “unlabeled stub-clique links”.

In step 9 of AS-Rank, stub-clique links are classified as c2p by default based on the assumption that stub networks are extremely unlikely to meet the peering requirements of clique members. We believe this assumption should be revisited with the trend of “Internet flattening”, as peering relationships between high-tier ASes and low-tier ASes are becoming more prevalent [22].

5) Conflicts in CoreToLeaf. Recall that CoreToLeaf labels some links as “conflicts”. These links appear to behave as p2c on some paths and c2p on others, and the main reason for this is violations of *valley-free* routing. We believe that this set of links is difficult to analyze because the two endpoints are likely to have unconventional routing policies.

Table 3 shows the error rates of inferences made by CoreToLeaf and AS-Rank on each category of hard links on 04/01/2016. We observe both algorithms yield more errors than their inferences on normal links, especially on unlabeled stub-clique links. Furthermore, the fraction of every

Category	CoreToLeaf (%)	AS-Rank (%)	Fraction all links	Fraction validation
Max node degree <100	13.7	8.6	16.1%	1.7%
Observed by 50-100 VPs	4.7	9.3	9.9%	8.1%
Non-VP & Non-Tier1	5.3	9.0	24.2%	11.6%
Unlabeled Stub-clique	95.5	33.4	0.3%	0.1%
Conflict	100.0	8.1	0.24%	0.16%

Table 3: Error rates of CoreToLeaf and AS-Rank on hard links on 04/01/2016. The fraction of each category of hard links that is in overall links vs. in the validation dataset shows that hard links are underrepresented in the validation dataset.

category of hard links in the validation dataset is less than that in the overall links, especially for the “Max node degree < 100” category. This indicates that the validation dataset is skewed to *easy* links. In addition to the entire validation dataset, we will use these more specific datasets for evaluating the AS inference algorithms in the subsequent sections.

5 Challenges With AS Relationship Inference

In this section, we identify three main challenges with AS relationship inference, and describe how they hamper existing inference techniques. This analysis helps inform the design of a probabilistic algorithm for AS relationship inference.

5.1 Degree Inversion

An AS inference algorithm can use any observed attribute associated with a link, its two endpoint ASes, AS links, and end-to-end paths that traverse the link in order to determine the link type. However, most attributes have stochastic information value, as we will illustrate below for AS degree.

Many existing techniques for inferring AS relationships make three assumptions: highest-degree ASes sit at the top of the routing hierarchy; peering ASes have similar degrees; and providers have larger degree than customers [21, 41, 16].

Over the past four years, the top two nodes with the largest transit and node degrees (as observed through BGP feeds from available VPs) have consistently been AS6939 (Hurricane Electric) and AS174 (Cogent Communications). However, both of these ASes are not Tier-1 ASes [59], so the assumption that the ASes with the highest degrees sit on top of the routing hierarchy is not universally valid. This fact influences the accuracy of some inference approaches since a key step in these approaches is identifying a clique of Tier-1 ASes at the top of the hierarchy [21, 41].

Figure 1a plots a CDF of the absolute transit degree differences of different link types. Transit degrees of ASes are computed from BGP paths observed on 04/01/2017, and link types are derived from the validation dataset on 04/01/2017 as described in §3.4. The validation dataset includes 55,016 links, 30,859 of which are p2c links and 24,157 are p2p links. We see that even though p2c links usually have larger degree differences than p2p links, over 14% of the p2p links have absolute transit degree differences larger than 1000, making many p2p links indistinguishable from p2c links in terms of transit/node degree difference.

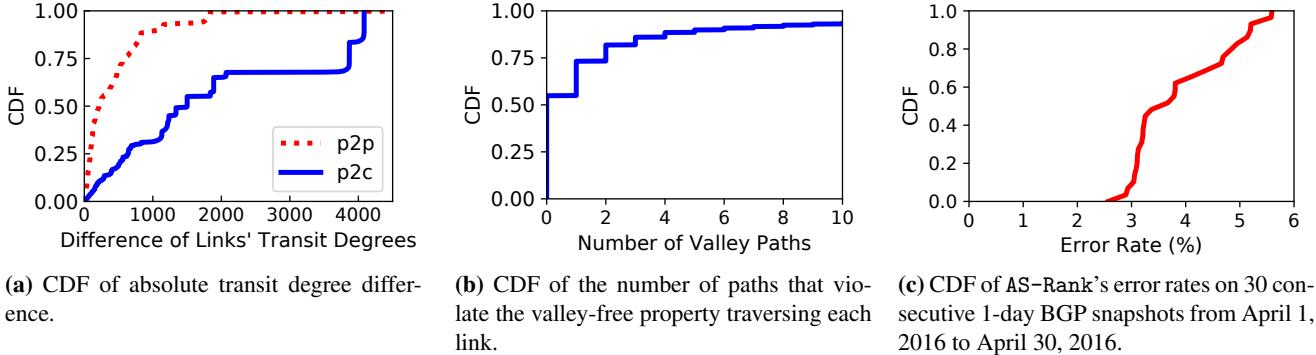


Figure 1: Analysis of transit degree difference, valley-free violations, and error rates of AS-Rank.

According to this observation, the existence of substantial differences in node/transit degrees between peering ASes is common. This phenomenon is explained in part by the fact that, during recent years, large content providers such as Google, Akamai, and Microsoft, which usually have high degrees, are more willing to peer with large numbers of lower tier ASes to get free and more efficient traffic exchange [56, 31]. This trend is referred to as the “flattening” of the Internet [22], and it significantly influences the AS relationship inference techniques that differentiate peers from providers or customers based on transit/node degree differences, or rank ASes in decreasing order by degrees and label links based on the order in which ASes are considered (as is the case with AS-Rank).

5.2 Violation of Valley-Free Property

Next, we study the prevalence of *valley-freeness*, which is the culprit behind mistakenly inferred p2c links and ‘conflict’ links in CoreToLeaf.

3% of the BGP paths violate *valley-freeness* in the AS-Rank inference on 04/01/2012. We find this level of *valley-free* violations is persistent over the various snapshots in our study. Figure 1b shows a CDF of the number of paths that violate the valley-free property for the links in BGP paths on 04/01/2012. 47% of the links in the AS topology are traversed by paths that violate the valley-free property. This statistic is consistent with prior work that analyzes the prevalence of valley-free violations, and it is a result of the deliberate BGP policies of ASes that use unconventional economic models [26].

The existence of these violations has certain implications for AS relationship inference. First, a robust inference algorithm has to take into account the structure of all paths traversing a given link. Second, it might have to revisit and update the inference made for a given link after inferring the types of neighboring links.

5.3 Current Techniques are Sensitive to VP and Snapshot Selection

We observe high variation in accuracy when applying the AS-Rank algorithm to consecutive snapshots of BGP paths.

Figure 1c plots a CDF of AS-Rank’s error rates ($1 - \text{accuracy}$) on 30 consecutive 1-day BGP snapshots in April, 2016. As shown in Appendix D.1, AS-Rank’s accuracy is also sensitive to the VP selections.

The reason for the AS-Rank algorithm’s sensitivity to snapshot and VP selections lies in the first step of its inference algorithm that identifies the Tier-1 clique and the subsequent steps that label links in a particular order starting with the Tier-1 ASes. AS-Rank first finds the biggest clique from the AS-links involving the largest ten ASes by transit degree, then visits the rest of the ASes top-to-bottom, and adds an AS to the clique if it connects with all the members in the current clique. It then labels p2c links using path segments that radiate from the Tier-1 clique. Errors that creep into the clique determination step have a significant impact on the order in which AS links are analyzed and labeled. See Appendix D.2 for detailed discussions.

6 Probabilistic AS Relationship Inference

In this section, we present a new AS relationship inference algorithm, ProbLink, that is designed to address the challenges discussed above. First, ProbLink is a probabilistic algorithm that enables the use of link attributes with stochastic information value. Second, in determining a link’s type, ProbLink simultaneously takes into account all information regarding the links and the paths that traverse it, and provides a framework for integrating conflicting information (e.g., paths that violate the valley-free property). Third, ProbLink does not prescribe a specific order in which ASes and links are considered, but rather continually updates the link type inferences and iterates till it reaches a fixed point in terms of the underlying stochastic distributions.

Crucially, our algorithm provides a framework for integrating various link attributes that might help infer a link’s type. We therefore first design a set of link features or attributes that provide noisy but still informative signals regarding the AS relationships. In particular, we design features that capture routing behavior in terms of both observed and unobserved routes as well as integrate information regarding a link’s endpoints. We note that many of the features used in our algorithm are distinct from that of prior tech-

niques, which mostly use only “valley-freeness” or “node/transit degree” features. We then describe how we use these features to build a probabilistic inference model.

6.1 Overview

Our algorithm starts with an initial classification of links based on the inference result of CoreToLeaf, so each link has deterministic relationship probabilities at the beginning. More concretely, if CoreToLeaf labels L as a p2p link, we will convert it to $P(L = p2p) = 1.0$, $P(L = p2c) = 0.0$, $P(L = c2p) = 0.0$ and provide that as the input to our algorithm. Note that ProbLink is essentially a meta-inference algorithm that can be bootstrapped by outcomes of any algorithm. Its performance is independent on the bootstrapping algorithm we choose, which we evaluate later in §7.1.

For each feature, ProbLink computes the conditional probability distribution based on observed data and the initial set of relationship types attributed to links. In each iteration, we update the probabilities of each link’s types ($P(L = p2p)$, $P(L = p2c)$, $P(L = c2p)$) by running our probabilistic algorithm described in §6.4, and recompute the distributions of features using the updated probability values of each link. We repeat this process until convergence, i.e., the percentage of links that change labels between each iteration drops below a small threshold.

6.2 Clique Inference

We first attempt to infer the ASes that are at the top of the hierarchy, namely Tier-1 ASes, because it is used to derive features employed by ProbLink. Tier-1 ASes should have the largest *customer cones* [41], so estimating the customer cones is the core of doing clique detection.

First, we find top N ASes in terms of transit degree, denoted as D .¹ These ASes are either Tier-1 or Tier-2 ASes because of the large number of neighbours to which they provide transit. Then, we estimate the customer cone size of each AS in the graph by determining the average number of destination ASes (last hops) for which an element of D uses this AS as part of a route. This is an effective way of estimating the customer cone size because, irrespective of whether a node $d \in D$ is a Tier-1 or a Tier-2 AS, if it reaches a destination t through an AS $x \in D$, then t is likely to be in x ’s customer cone.

Second, we find the maximal clique C with largest estimated customer cone size sum in D . Then, we test every other AS in order by estimated customer cone size to add members to C . An AS is added to C if it has links with every other AS in C . If there are three consecutive members (X-Y-Z) in C showing up in paths, disconnect the edge between X and Z even though X and Z are connected in some paths, because no AS path should have three consecutive clique ASes. Finally, we find the maximal clique in C as the inferred clique.

¹Any value of N between 10–40 does not affect the final result.

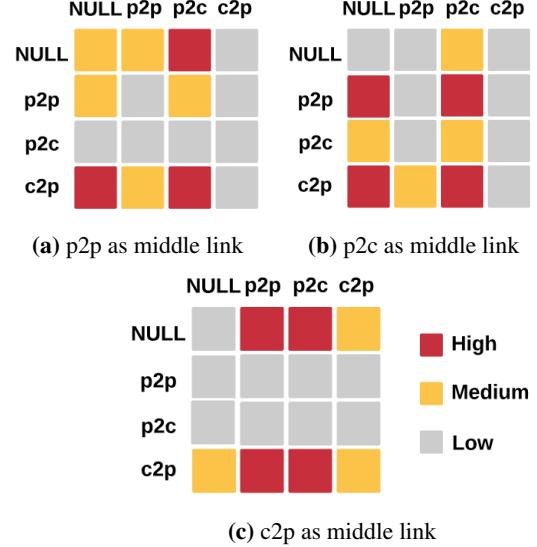


Figure 2: Conditional probability distribution for the triplet feature describing $P(\text{previous}, \text{next} | \text{middle})$. Probability values in the ranges of > 0.1 , $[0.01, 0.1]$, and < 0.01 , are categorized as high, medium, and low respectively in the figure.

6.3 Feature Design

An AS link can be characterized by the following three attributes: (A) The structure of paths that use the link; (B) The structure of paths that *do not* use the link; (C) Properties of the ASes on each side of the link. We carefully design six features that correspond to these three types of attributes.

Triplet feature (Type A). The triplet feature considers link triplets that appear in paths and attributes probabilistic values for the relationships of the first and the last links given the relationship of the middle link. Suppose three consecutive links “L1 - L - L2” show up in a BGP path, where L1, L, L2 are three links (AS pairs). “L1 - L - L2” is called a link triplet. We break down each BGP path in a snapshot into link triplets, and, for the first and the last links in each path, we insert a “NULL” link in front of and behind it. For example: a BGP path “8793 6939 1103 198499” is decomposed into 3 link triplets: “NULL - <8793, 6939> - <6939, 1103>”, “<8793, 6939> - <6939, 1103> - <1103, 198499>”, and “<6939, 1103> - <1103, 198499> - NULL”. As a consequence, each link in the BGP paths appears as a middle link in at least one link triplet. We take into account sibling relationships (described in §3.2) by skipping sibling links, i.e. treating the two ASes connected by a sibling link as a single AS, when constructing triplets.

The goal of the triplet feature is to model *valley-freeness* in a probabilistic way. For each middle link type, we compute the probability of the link type of its adjacent previous link and next link. If we put the link type of the previous link along the y-axis and the link type of the next link along the x-axis, we get a matrix view as shown in Figure 2, which is computed from CoreToLeaf initial labels. Each cell in

the matrix represents a probability that is categorized as high, medium, and low depending on it being in the range of > 0.1 , $[0.01, 0.1]$, or < 0.01 . For example, we can see from Figure 2a that when the middle link is of type p2p, the previous and next links are most likely to be $\langle \text{NULL}, \text{p2c} \rangle$, $\langle \text{c2p}, \text{NULL} \rangle$ and $\langle \text{c2p}, \text{p2c} \rangle$, but its previous link is unlikely to be p2c no matter what its next link's type is.

Non-path feature (Type B). In addition to observed routes, unobserved routes also provide some information regarding AS relationships. The *non-path feature* describes the probability of how many adjacent p2p or p2c links a link has, but none of them appear before this link on any of the paths. This feature is designed to capture the property that a link is unlikely to be a p2c link if it has many adjacent p2p/p2c links and none of them appear as a previous link on any of the paths containing the link.

Similar to the triplet feature, the non-path feature also models *valley-freeness* in a probabilistic way. The non-path feature is not necessarily applicable to all links. When a link does follow a p2p or p2c link or if a link does not have any adjacent p2p or p2c links, the non-path feature does not play a role in inferring the link's type.

Distance to clique feature (Type C). The *distance to clique* feature can be used to capture the observations that high-tier ASes are closer in distance (AS hops) to clique ASes than low-tier ASes, and that ASes in the same tier are likely to be peers, while high-tier ASes tend to be providers of low-tier ASes.

We first create an undirected graph by adding AS links as edges, and then compute the shortest path from each AS towards each clique member using Dijkstra's algorithm. For each AS, we compute its average distance to each member in the clique set, round it to a multiple of 0.1, and denote this value as $\text{dist}(\text{AS})$. We represent each link $\langle \text{AS}_1, \text{AS}_2 \rangle$ in the graph by a distance to clique tuple “ $\text{dist}(\text{AS1}), \text{dist}(\text{AS2})$ ”.

Vantage point feature (Type C). The number of VPs observing a link also suggests the link type. The vantage point feature captures the likelihood of a certain number of VPs with at least one path traversing a particular link given its link type. This feature naturally folds in the following intuition: p2c links are more likely to be seen by more VPs compared to p2p and c2p links. This feature considers path directions. For example, let's consider a link L ($\langle \text{AS1}, \text{AS2} \rangle$) where AS1 is the provider of AS2. ProbLink computes probabilities separately for both directions by counting how many paths traverse L in the direction of $\langle \text{AS1}, \text{AS2} \rangle$, and how many paths traverse L in the direction of $\langle \text{AS2}, \text{AS1} \rangle$.

To evaluate the informational value of this feature, we analyze the number of VPs that observe a given link and correlate that with the link's type computed by CoreToLeaf. Figure 3 shows the CDF of the number of VPs that observe a link for each link type. We observe that 93% of p2p links and 90% of c2p links are observed by ≤ 10 VPs, while 98% of p2c links are seen by more than 10 VPs.

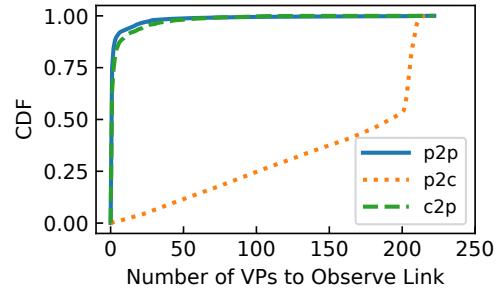


Figure 3: The visibility of each link type derived from CoreToLeaf inference results on 04/01/2017 BGP paths.

Co-located IXP and co-located private peering facility feature (Type C). The *co-located IXP* and *co-located peering facility* information is extracted from PeeringDB [2]. These features are based on the intuition that the more IXPs or facilities two ASes are co-located in, the more likely they are peering with each other. Based on the validation data, 90% of transit links do not have any co-located IXPs or facilities, while more than 70% p2p links have at least one co-located IXP or facility.

6.4 Inference Algorithm

We begin by reviewing the Naïve Bayes classifier. Given a link type variable C (which can be p2p, p2c, c2p) and a feature vector f_1 through f_n , Bayes' theorem states the following relationship:

$$P(C | f_1, \dots, f_n) = \frac{P(C, f_1, \dots, f_n)}{P(f_1, \dots, f_n)} \quad (1)$$

By assuming that each feature f_i is conditionally independent of every other feature:

$$P(f_i | C, f_1, \dots, f_{i-1}, f_{i+1}, \dots, f_n) = P(f_i | C) \quad (2)$$

Using the *chain rule* to rewrite the numerator of Eq. 1:

$$P(C, f_1, \dots, f_n) = P(C) \prod_{i=1}^n P(f_i | C) \quad (3)$$

So,

$$P(C | f_1, \dots, f_n) = \frac{P(C) \prod_{i=1}^n P(f_i | C)}{P(f_1, \dots, f_n)} \quad (4)$$

Since the denominator $P(f_1, \dots, f_n)$ does not depend on the class C , the Naïve Bayes classifier assigns a link being a type \hat{C} by the following function:

$$\hat{C} = \arg \max_C P(C) \prod_{i=1}^n P(f_i | C) \quad (5)$$

The inputs to ProbLink are BGP paths, link triplets extracted from these BGP paths, and initial relationship labels for each link as inferred by a bootstrapping algorithm. Algorithm 1 shows the pseudocode of ProbLink.

Algorithm 1: ProbLink: probabilistic AS relationship inference algorithm based on Naive Bayes

```

Input : 1) BGP paths → link triplets
           2) Initial AS relationships R
           3) Feature vector f = [Triplet, Non-path, Distance to clique, VP, Co-located IXP, Co-located facility]
Output: Inferred probabilities of each link being p2p, p2c, c2p
 $\text{/* Loop until convergence} \quad */$ 
1 while  $R - \text{last}(R) > \epsilon$  do
     $\text{/* Compute conditional distribution of each feature} \quad */$ 
    2 foreach feature  $f_i$  in feature vector f do
        3  $P(f_i | C) = \frac{P(f_i, C)}{P(C)} = \frac{N(f_i, C) + \alpha}{N(C) + \alpha \delta}$ 
    4 foreach link L do
        5  $all \leftarrow N(p2p) + N(p2c) + N(c2p)$ 
        6  $P(L = p2p) \leftarrow P(p2p) = \frac{N(p2p)}{N(all)}$ 
        7  $P(L = p2c) \leftarrow P(p2c) = \frac{N(p2c)}{N(all)}$ 
        8  $P(L = c2p) \leftarrow P(c2p) = \frac{N(c2p)}{N(all)}$ 
        9 foreach feature  $f_i$  in feature vector f do
            10  $P(L = p2p) *= P(f_i | p2p)$ 
            11  $P(L = p2c) *= P(f_i | p2c)$ 
            12  $P(L = c2p) *= P(f_i | c2p)$ 
        13 sum =  $P(L = p2p) + P(L = p2c) + P(L = c2p)$ 
        14  $P(L = p2p) \leftarrow P(L = p2p) / \text{sum}$ 
        15  $P(L = p2c) \leftarrow P(L = p2c) / \text{sum}$ 
        16  $P(L = c2p) \leftarrow P(L = c2p) / \text{sum}$ 
     $\text{/* Update link's type} \quad */$ 
    17  $R = \arg \max_C P(L = C)$ 

```

First, the algorithm calculates probabilities for each feature, conditional on the link type C (C in $\{p2p, p2c, c2p\}$) by accumulating probability values (line 2-3 in Algorithm 1). The parameter α is a smoothing parameter, which prevents a feature with examples in only one class from forcing the probability estimate to be 0 or 1. In our implementation, we use *Laplace (Add-1) Smoothing* [42], which sets the smoothing parameter to 1. The algorithm then assigns probability that link L is of each type by the prior probability distribution $P(C)$, which is the proportion of each link type in the data (line 5-8 in Algorithm 1). Then, it goes through each feature and multiplies the probability that link L is of each type by the conditional probability of the feature given each link type (line 9-12). In the end, the final probability of the link L of being each type is calculated by the fraction of each type's probability over the sum of probabilities of all possible link types (lines 14-16 in Algorithm 1). We then update L 's type by picking the link type with the largest probability (line 17). We repeat this process of link type inference and updating probability distributions of features until convergence, i.e., the percentage of links that change labels between each iteration drops below a small threshold. The algorithm usually converges within four iterations.

Algorithm Design Choice: We considered alternative approaches for prediction, such as supervised learning based on a training set of labeled link types from community attributes (say using boosted trees). However, since the size of the ground truth from community attributes has not increased in recent years and since our analysis in §4.3 shows that the validation dataset is skewed and is only partially representative of the overall Internet, a supervised learning approach would be affected by the biases in the training set.

We instead adopted an unsupervised approach that uses EM. In particular, ProbLink falls in the category of techniques that use the *expectation maximization algorithm for parameter estimation in Naïve Bayes classifiers* [46]. In particular, Expectation Maximization (EM) is the iterative technique used to separate out classes from a mixture, and Naïve Bayes is the classification technique used in each iteration. EM is suitable when there are hidden classes, and the observed feature distributions are a mixture of the feature distributions of different classes. In each iteration of EM, the algorithm groups together elements that are classified together and derives the feature parameters of each class. Recall that in our setting, the feature parameters are estimates of the probability of different types of links given a particular topological feature, namely, the probability of the middle link type given previous/next link type, the probability of a certain number of VPs observing a link given each link type, and so on.

Our approach and the underlying techniques have the following implications. First, there is no ground truth in any stage of the algorithm. Second, EM does not work when the classification technique is a black-box or a non-parametric technique (such as a neural network or a boosted decision tree) since it is hard for EM to converge to a stable set of black-box parameters. In fact, in the case of decision trees, convergence would mean that not only the values in the tree nodes do not change, but also the structure of the trees remains stable across iterations. This convergence requirement is hard to satisfy, and hence non-parametric techniques such as decision trees are ill-suited for EM in spite of their high classification accuracy. Naïve Bayes, on the other hand, is a parametric technique that has been shown to work well even when there is correlation between the features used for prediction. Crucially, when Naïve Bayes is used as the classification technique, EM can converge and attribute a stable set of parameters to be used by Naïve Bayes.

It is worth noting that Naïve Bayes makes the assumption that all features are conditionally independent. This independence assumption rarely holds in practical situations, including our own. Nevertheless, despite violating the independence assumption, the classification decisions made by Naïve Bayes are often of high quality [62, 53, 63, 58]. Moreover, in our context, what is needed is a parametric classification technique as opposed to a prediction technique that attributes precise probabilities for the different classes. The

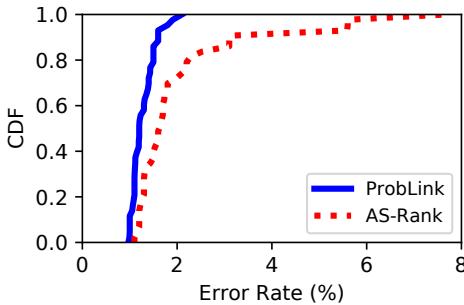


Figure 4: CDF of error rates of ProbLink and AS-Rank on the snapshots of BGP paths in the past 6 years.

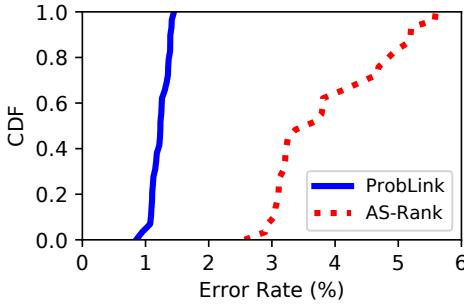


Figure 5: CDF of error rates of ProbLink and AS-Rank on 30 consecutive 1-day snapshots from April 1, 2016 to April 10, 2016.

Naïve Bayes classifier is appropriate for such settings, but the lack of conditional independence does have the downside that we cannot use the derived probability values as a confidence measure [7].

7 Evaluation

We now evaluate our probabilistic inference algorithm, ProbLink, from three aspects and show that:

- ProbLink consistently achieves low error rates across many years, reducing the average error rate of AS-Rank for all links by 1.7 \times , and attaining 1.8-6.1 \times better error rates for the different categories of *hard* links.
- ProbLink is not dependent on its bootstrapping algorithm, and it is stable with respect to snapshot and VP selection.
- Each feature used in ProbLink is meaningful and eliminating any of them harms the overall inference accuracy.

7.1 Accuracy

To evaluate the accuracy of ProbLink, we assemble daily snapshots of BGP paths on the first five days of April, August, and December (i.e., every four months) over the past 6 years. We apply our algorithm against these snapshots and compare it with the AS-Rank algorithm over this time period.

Figure 4 compares the error rates of inferences made by ProbLink and those made by AS-Rank. Our probabilistic inference algorithm consistently yields a low error rate smaller than 2%, reducing the average error rate of AS-Rank

Category	AS-Rank(%)	ProbLink(%)
Observed by 50-100 VPs	8.8	1.5
Non-VP & non-Tier1	4.4	1.7
Unlabeled Stub-clique	33.6	5.5
Conflict	6.8	3.8

Table 4: Average error rates on *hard* links. ProbLink achieves 5.9 \times , 2.6 \times , 6.1 \times , and 1.8 \times better error rate for the links observed by between 50 and 100 VPs, non-VP & non-Tier1 links, unlabeled stub-clique, and conflict than AS-Rank respectively.

Feature excluded	Error rate
None	1.5%
Triplet	2.4%
Non-path	1.7%
Distance to Clique	1.7%
VP	4.3%
Co-located IXP and peering facility	1.8%

Table 5: Error rates of ProbLink with all features turning on and without each feature in turn against 04/01/2017 BGP paths. for all links from 2.1% to 1.2%.

Figure 5 shows a comparison between error rates of ProbLink and AS-Rank on 30 consecutive snapshots of BGP paths during April 2016. The max and average error rates across these days for ProbLink are 1.4% and 1.2%, while the error rate of AS-Rank ranges from 2.6% to 5.6%, with an average error rate of 3.9%. ProbLink is not sensitive to the specific set of paths used in a snapshot, and it achieves uniformly low error rates in spite of clique inference inaccuracies that adversely impact AS-Rank.

Figure 6 plots the CDFs of error rates of inferences made by ProbLink and AS-Rank on the four categories of *hard* links identified in §4.3. Not only does our algorithm yield much smaller error rates, but it also has less variation than AS-Rank. Table 4 lists the average error rate of our algorithm and AS-Rank for links *observed by 50 to 100 VPs*, *non-VP and non-Tier1* links, *stub-clique* links, and *conflict* links. Our probabilistic algorithm reduces the error rate on the four categories by a factor of 5.9, 2.6, 6.1, and 1.8 respectively compared to AS-Rank.

ProbLink is not dependent on the initial labels provided by the bootstrapping algorithm. Bootstrapping with CoreToLeaf and AS-Rank only results in 0.15% overall accuracy difference on average. Completely random initial assignment would not work well because our algorithm attempts to separate mixture distributions with some underlying properties. Our algorithm should, however, be robust to various types of initial label assignments as long as there is some weak correlation between the initial labeling and the actual assignments.

7.2 Feature Importance Analysis

Eliminating any of the features used by ProbLink results in lower accuracy. We next run ProbLink with each feature excluded in order to show that each feature adds value. Ta-

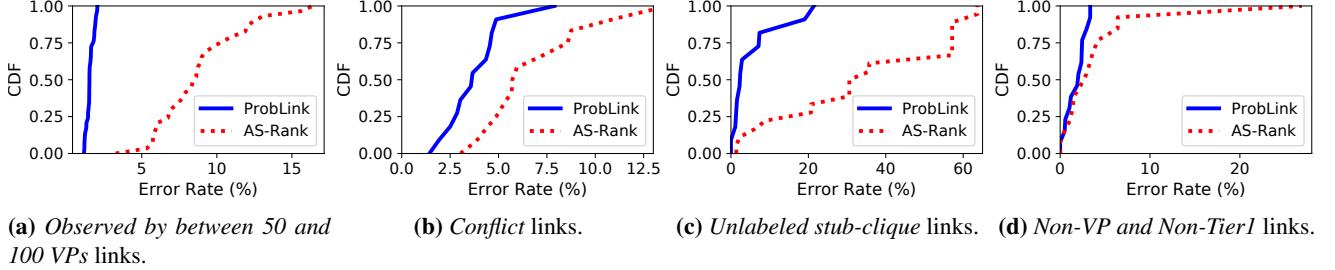


Figure 6: CDFs of error rates of ProbLink vs. CoreToLeaf on hard links in a period of 30 days in April 2016.

ble 5 lists the error rates of ProbLink after turning off each feature one-by-one against the 04/01/2017 snapshot of BGP paths.

Excluding any feature in ProbLink results in a higher error rate, which suggests that each feature adds some value. Among all the features, excluding the VP feature harms the overall accuracy the most, indicating that the visibility of BGP paths from many vantage points is a crucial attribute for inferring link types. We believe that integrating more features can further improve ProbLink’s accuracy.

8 Practical Applications

8.1 Route Leak Detection

Route leaks are a class of common routing incidents that can cause large Internet service disruptions [54]. They are caused by violations of the policies among the ASes involved. For instance, on November 5, 2012, a Google peer Moratel (AS23947) improperly advertised Google routes to its provider, causing Moratel’s providers to select the leaked routes as the preferred ones destined to Google. As Moratel could not handle such large traffic volumes, Google’s services went offline in parts of Asia for half an hour [28].

A conventional method for detecting route leaks is through checking valley-free violations in BGP paths. Mauch built a routing leak detection system based on this intuition by searching for valley paths containing three or more major networks with known relationships [32].

In the same spirit, we build a route leak detection system by detecting valley-free violations in paths based on the link relationship inference results of ProbLink. It is worth noting that a large fraction (more than 50%) of the valley-free violations do not result from route leaks but intended policies from ASes that are research/educational or IXPs [26]. Such ASes often establish a special type of AS relationship called indirect peering, where an AS functions as an intermediate link between two other ASes who wish to peer but go through intermediate ASes. Therefore, we ignore a path if it contains research/educational or IXP ASes when detecting route leaks.

To evaluate the performance of ProbLink and other AS relationship inference techniques on route leak detection, we use only those links for which we have validation data in BGP paths. For example, suppose a path has link relation-

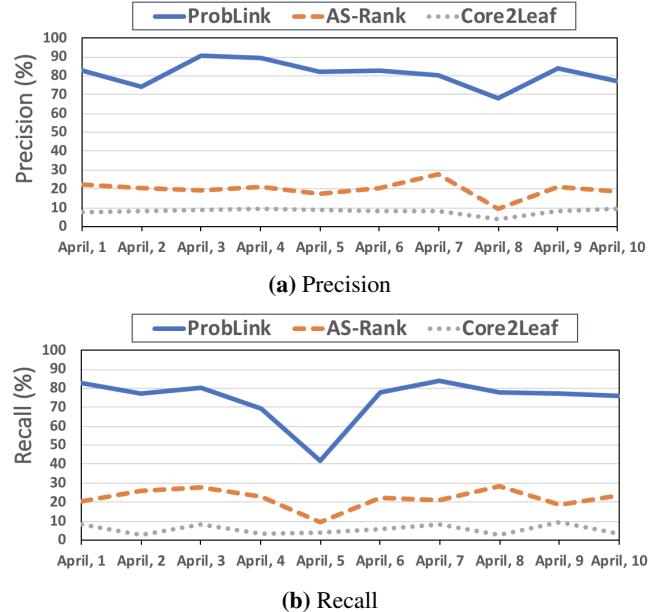


Figure 7: Evaluation of route leak detection across 10 days.

ships “* * p2p * c2p”, where * is unknown in the validation dataset. Even though a relationship inference algorithm should have predictions on the unknown links, we just detect route leaks by its predictions on the two known links in order to compare against the validation dataset. Figure 7 compares the precision and recall of ProbLink, AS-Rank, and Core2Leaf, against the real route leaks implied by the validation dataset across 10 days in April 2016. The average precision for ProbLink, AS-Rank, and Core2Leaf is 81.1%, 19.8%, and 8.1%, respectively; and the average recall for ProbLink, AS-Rank, and Core2Leaf is 76.2%, 22.1%, and 5.6%, respectively.

While ProbLink significantly improves the state-of-the-art in route leak identification, the number of false-positive inferences is still relatively high (almost 20%) when used as a stand-alone inference heuristic. That said, ProbLink shows that valley-free violations are a strong signal for the occurrence of route leaks. This indicator facilitates post processing and can be used as part of a composite detection mechanism that combines multiple sources of information, such as the detection of abrupt changes in prefix traffic levels [5].

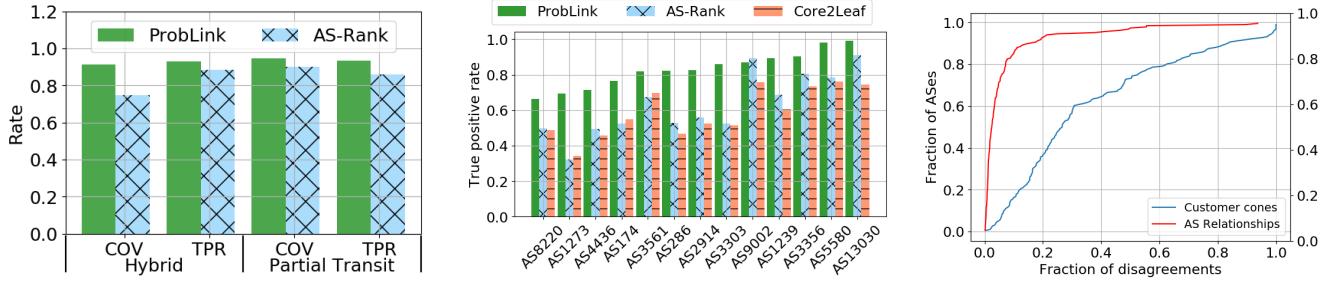


Figure 8: Evaluation of complex relationships inference and the prediction of path changes due to selective prefix advertisements.

8.2 Inference of Complex Relationships

AS relationships may be more complex than the traditional p2c/p2p model. Such complex agreements may take the form of a *hybrid* relationship with different relationship type for different Points-of-Presence (PoPs) or a *partial-transit* relationship, in which a provider offers transit only toward its peers and customers, but not its providers, or restricts transit to a specific geographic region [15, 19]. The state-of-the-art algorithm for inferring complex relationships (CR algorithm) takes as input a set of conventional relationships and iteratively refines them by combining active traceroute measurements with geolocation data to discover the PoP-level propagation patterns of inter-domain paths [25]. Due to the high measurement cost in terms of traceroute queries required to infer complex relationships, CR utilizes customer cones to optimize the allocation of queries to traceroute probes and maximize the discovery of hybrid relationships within the limited querying budgets used by platforms such as RIPE Atlas [52]. Therefore, the quality of the p2p/p2c relationships can affect the precision, accuracy, and coverage of complex relationship inference.

To test the performance of ProbLink for complex relationship inference, we implemented the CR algorithm and executed two 2-day measurement campaigns over the RIPE Atlas platform, on 2018/09/06 and 2018/09/08 using ProbLink and AS-Rank respectively. For each measurement round, we allocated the maximum permissible number of measurement credits, which resulted in 125,529 traceroute queries from 7,870 Atlas probes. CR+ProbLink inferred 1,308 hybrid relationships and 3,163 partial transit links, while CR+AS-Rank inferred 1,029 hybrid relationships and 3,009 partial transit links. We evaluated these inferences against our validation dataset, which includes 346 hybrid links and 402 partial-transit links. As shown in Figure 8a, combining CR with ProbLink not only improves the True Positive Rate (TPR) of the algorithm both for hybrid and partial transit relationships, but, importantly, we significantly expand its coverage (COV) by capturing 91% of the hybrid and 95% of the partial transit relationships, compared to 76% and 90%, respectively, for CR+AS-Rank. Overall, CR+ProbLink dis-

covers 27% more hybrid relationships than CR+AS-Rank.

8.3 Predicting the Impact of Selective Advertisements

The ability to predict the impact of traffic engineering policies on the active BGP paths can be valuable to network operators, as it would limit the need for trial-and-error experimentation, allow the configuration of more predictable and stable routing policies, and minimize the risk of propagating unintended routes [45]. However, past works have shown that the existing AS relationship datasets have poor predictive capabilities, making them impractical for such purposes. In this section, we evaluate the impact of ProbLink’s improved relationship inferences in predicting the outcome of a selective advertisement. Selective advertisement is a popular traffic engineering technique used by AS operators to achieve traffic load balancing, by advertising certain routes only to a subset of their inter-domain neighbors [48].

To predict the impact of selective advertisement “in the wild”, we first need to explicitly capture the activation and the scope of such policies. We detect selectively advertised prefixes by utilizing route redistribution BGP Communities, which are increasingly utilized to implement selective prefix advertisement [49, 17]. In particular, many providers define an array of Community values that can be set by their customers, to allow them to control whether the provider should propagate or not a route to a specific peer or group of peers. For instance, if AS9002 (RETN) receives a prefix advertisement from a customer annotated with the BGP Community 9002:65535, then RETN will propagate this route only to its customers, but not its peers or providers [50]. Redistribution Communities can further limit the scope of the prefix advertisement by determining a location for which the redistribution policy will be applied. For instance, when the Community 286:49 is applied on a prefix, AS286 (KPN) will not advertise this prefix to its US peers [37]. By parsing WHOIS records and NOC websites, we compile a dictionary of Community values that define one of the following types of selective route redistribution:

- Do not announce route to neighbors of type *R*.
- Do not announce route to neighbors of type *R* at *L*.

- Announce route only to neighbors of type R .
- Announce route only to neighbors of type R at L .

R indicates relationship type (customer, provider, peer) and L indicates a city-level or country-level location identifier. In total, we extracted 644 Community values from 152 ASes.

After compiling our Communities dictionary, we monitor the BGP messages of the corresponding ASes to capture BGP Updates annotated with one of the redistribution Communities. Let us assume we observe a BGP Update for a destination prefix d annotated with a BGP Community C , which instructs AS_C to propagate p *only* to its neighbors of type R . We calculate which ASes will have to change their paths as follows: We first parse the BGP paths right before C was applied on the prefix d , and we collect all the paths P_{ALL} to d that traversed AS_C . Then, based on the inferred relationships, we find the paths $P_{R'} \subseteq P_{ALL}$ in which AS_C advertises the route toward d to a neighbor with relationship type $R' \neq R$. Since the Community C allows the prefix announcement only to neighbors with relationship type R , we infer that the paths $P_{R'}$ will be withdrawn, and the corresponding ASes in these paths will choose a different path. When C also defines a geographic scope for the prefix advertisement in addition to the relationship type, we use the techniques described in [25] to map the city-level location of AS interconnections, and calculate the affected paths in a similar manner. We validate our inferences by observing the withdrawn paths after C was applied on the path. We consider as false positive any AS $a \in P_{R'}$ that did not withdraw its path 15 minutes after we observed the BGP Update with C . We do not consider false negatives, as an AS may change its path to d for different reasons, and this change may simply coincide with the application of the Community C on the same prefix.

Figure 8b shows our validation results after executing the above experiment for the first week of April 2016. During that period, we found 480 prefixes tagged with redistribution communities defined by 13 ASes. Overall, 83% of ProbLink’s predictions were correct, compared to 62% for AS-Rank and 59% for CoreToLeaf. ProbLink outperformed AS-Rank for every AS except AS9002, and in some cases (e.g. AS1273), the true positive rate was 2x higher compared to the other algorithms. These results are surprising given that less than 4% of the relationship inferences differ between ProbLink and AS-Rank. To understand the significant improvement achieved by ProbLink, we investigate the impact of the relationship disagreements between the two algorithms on the customer cones obtained using the *Provider/Peer Observed* methodology proposed in [41]. We focus on the ASes with at least 100 ASes in their downstream path. For each of these ASes we calculate the fraction of their relationships and the fraction of their customer cones that disagree between ProbLink and AS-Rank. As shown in Figure 8c, while less than 10% of the ASes had more than 20% relationship mismatches, over 60% of the ASes had at

least 20% difference in their customer cones. This finding highlights the fact that even a few incorrect relationship inferences can lead to significant differences in properties of the resulting downstream paths and substantial deviations in the predictive capabilities of ProbLink and AS-Rank.

9 Conclusion

We revisit the AS relationship problem and inference techniques. We first develop a simple inference algorithm that achieves accuracy comparable to that of the state-of-the-art inference technique, AS-Rank, indicating that the types of most links in validation datasets are relatively easy to infer. We then construct different subsets of the validation dataset that might be considered *hard* and use these as benchmarks for evaluating improvements in AS relationship inference. Further, we observe that many of the features that can be used by inference techniques are of a stochastic nature, so we present a probabilistic AS relationship inference algorithm that provides a framework for easy integration of many noisy but useful attributes into the relationship inference algorithm. We show that this probabilistic algorithm is more accurate and less sensitive to the locations of vantage points and BGP paths compared to the state-of-the-art algorithms.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Andreas Haeberlen for their valuable feedback. This research was partially supported by the National Science Foundation under Grants CNS-1614717 and CNS-1513847.

References

- [1] IXP Service Matrix. <https://www.euro-ix.net/en/tools/ixp-service-matrix/>.
- [2] PeeringDB. <https://www.peeringdb.com/>.
- [3] RIPE (RIS). <http://www.ripe.net/ris/>.
- [4] U. Oregon Route Views Project. <http://www.routeviews.org/>.
- [5] B. Al-Musawi, P. Branch, and G. Armitage. BGP anomaly detection techniques: A survey. *IEEE Communications Surveys Tutorials*, 19(1):377–396, Firstquarter 2017.
- [6] R. Anwar, H. Niaz, D. Choffnes, Í. Cunha, P. Gill, and E. Katz-Bassett. Investigating interdomain routing policies in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, pages 71–77. ACM, 2015.
- [7] P. Bennett. Assessing the calibration of Naive Bayes’ posterior estimates. Technical report, September 2000. Computer Science Department, School of Computer Science, Carnegie Mellon University.

- [8] R. Bush, O. Maennel, M. Roughan, and S. Uhlig. Internet optometry: assessing the broken glasses in internet reachability. IMC '09.
- [9] M. Caesar and J. Rexford. BGP routing policies in ISP networks. *Netwkr. Mag. of Global Internetwkg.*, 19(6):5–11, Nov. 2005.
- [10] CAIDA. Inferred AS to organization mapping dataset. <http://www.caida.org/data/as-organizations/>.
- [11] A. Cohen, Y. Gilad, A. Herzberg, and M. Schapira. Jumpstarting BGP security with path-end validation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 342–355. ACM, 2016.
- [12] G. Comarella, E. Terzi, and M. Crovella. Detecting unusually-routed ASes: methods and applications. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 445–459, New York, NY, USA, 2016. ACM.
- [13] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. P. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 1–15, New York, NY, USA, 2018. ACM.
- [14] G. Di Battista, M. Patrignani, and M. Pizzonia. Computing the types of the relationships between autonomous systems. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 156–165. IEEE, 2003.
- [15] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, G. Riley, et al. AS relationships: Inference and validation. SIGCOMM '07, 2007.
- [16] X. Dimitropoulos, D. Krioukov, B. Huffaker, G. Riley, et al. Inferring AS relationships: Dead end or lively beginning? In *International Workshop on Experimental and Efficient Algorithms*, pages 113–125. Springer, 2005.
- [17] B. Donnet and O. Bonaventure. On BGP communities. *ACM SIGCOMM Computer Communication Review*, 38(2):55–59, 2008.
- [18] Q. Fan, H. Yin, G. Min, P. Yang, Y. Luo, Y. Lyu, H. Huang, and L. Jiao. Video delivery networks: Challenges, solutions and future directions. *Computers & Electrical Engineering*, 66:332–341, 2018.
- [19] P. Faratin, D. D. Clark, S. Bauer, W. Lehr, P. W. Gilmore, and A. Berger. The growing complexity of internet interconnection. *Communications & Strategies*, (72):51, 2008.
- [20] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.
- [21] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking*, 9(6):733–745, Dec 2001.
- [22] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. The flattening internet topology: Natural evolution, unsightly barnacles or contrived collapse? In *International Conference on Passive and Active Network Measurement*, pages 1–10. Springer, 2008.
- [23] P. Gill, M. Schapira, and S. Goldberg. Let the market drive deployment: A strategy for transitioning to BGP security. In *SIGCOMM '11*, volume 41, pages 14–25. ACM, 2011.
- [24] V. Giotsas, C. Dietzel, G. Smaragdakis, A. Feldmann, A. Berger, and E. Aben. Detecting peering infrastructure outages in the wild. In *SIGCOMM '17*, pages 446–459, 2017.
- [25] V. Giotsas, M. Luckie, B. Huffaker, and k. claffy. Inferring complex AS relationships. In *Internet Measurement Conference (IMC)*, pages 23–30, Nov 2014.
- [26] V. Giotsas and S. Zhou. Valley-free violation in internet routinganalysis based on BGP community data. In *Communications (ICC), 2012 IEEE International Conference on*, pages 1193–1197. IEEE, 2012.
- [27] V. Giotsas, S. Zhou, M. Luckie, and k. claffy. Inferring multilateral peering. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 247–258, New York, NY, USA, 2013. ACM.
- [28] S. Goldberg. Why is it taking so long to secure Internet routing? *Communications of the ACM*, 57(10):56–63, 2014.
- [29] Y. He, G. Siganos, M. Faloutsos, and S. Krishnamurthy. Lord of the links: a framework for discovering missing links in the Internet topology. *IEEE/ACM Transactions on Networking (ToN)*, 17(2):391–404, 2009.
- [30] G. Huston. The death of transit and the future internet. Second ITU Workshop on Network 2030, December 2018.
- [31] G. Inc. Google peering policy. <https://peering.google.com>.

- [32] M. Jared. BGP routing leak detection system. <https:////puck.nether.net/bgp/leakinfo.cgi>.
- [33] E. Jasinska, N. Hilliard, R. Raszuk, and B. N. RFC 7947: Internet exchange BGP route server. <https:////tools.ietf.org/html/rfc7947>.
- [34] J. Juen, A. Johnson, A. Das, N. Borisov, and M. Caesar. Defending tor from network adversaries: A case study of network path prediction. *Proceedings on Privacy Enhancing Technologies*, 2015(2):171–187, 2015.
- [35] E. Katz-Bassett, D. R. Choffnes, Í. Cunha, C. Scott, T. Anderson, and A. Krishnamurthy. Machiavellian routing: improving Internet availability with BGP poisoning. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 11. ACM, 2011.
- [36] M. Konte, R. Perdisci, and N. Feamster. Aswatch: An AS reputation system to expose bulletproof hosting ases. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 625–638. ACM, 2015.
- [37] KPN NOC. BGP Communities For AS286. <https:////as286.net/AS286-communities.html>, September 2018.
- [38] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici. Unikernels everywhere: The case for elastic CDNs. *SIGPLAN Not.*, 52(7):15–29, Apr. 2017.
- [39] P. Laskowski and J. Chuang. Network monitors and contracting systems: Competition and innovation. *SIGCOMM '06*, pages 183–194, 2006.
- [40] A. H. Lodhi. *The economics of Internet peering interconnections*. PhD thesis, Georgia Institute of Technology, 2014.
- [41] M. Luckie, B. Huffaker, A. Dhamdhere, V. Giotsas, et al. AS relationships, customer cones, and validation. IMC '13.
- [42] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, New York, NY, USA, 2008.
- [43] R. Mazloum, M.-O. Buob, J. Auge, B. Baynat, D. Rossi, and T. Friedman. Violation of interdomain routing assumptions. In *International Conference on Passive and Active Network Measurement*, pages 173–182. Springer, 2014.
- [44] A. Mitseva, A. Panchenko, and T. Engel. The state of affairs in BGP security: A survey of attacks and defenses. *Computer Communications*, 124:45 – 60, 2018.
- [45] W. Mühlbauer, A. Feldmann, O. Maennel, M. Roughan, and S. Uhlig. Building an AS-topology model that captures route diversity. *SIGCOMM '06*, pages 195–206, 2006.
- [46] K. Nigam, A. K. McCallum, S. Thrun, and T. Mitchell. Text classification from labeled and unlabeled documents using EM. *Machine learning*, 39(2-3):103–134, 2000.
- [47] R. Nithyanand, O. Starov, P. Gill, A. Zair, and M. Schapira. Measuring and mitigating AS-level adversaries against Tor. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.
- [48] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig. Interdomain traffic engineering with BGP. *IEEE Communications Magazine*, 41(5):122–128, May 2003.
- [49] B. Quoitin, S. Tandel, S. Uhlig, and O. Bonaventure. Interdomain traffic engineering with redistribution communities. *Computer Communications*, 27(4):355–363, 2004.
- [50] RETN NOC. BGP communities For AS9002. <http:////retn.net/support/bgp-communities/>, September 2018.
- [51] RFC. RFC 6996: Autonomous System (AS) reservation for private use. <https:////tools.ietf.org/html/rfc6996>.
- [52] RIPE Atlas. User-defined measurements - rate limits. <https://atlas.ripe.net/docs/udm/#rate-limits>.
- [53] I. Rish et al. An empirical study of the Naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.
- [54] K. Sriram, D. Montgomery, D. McPherson, E. Osterweil, and B. Dickson. Problem definition and classification of BGP route leaks. <https:////tools.ietf.org/html/rfc7908>.
- [55] L. Subramanian, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the Internet hierarchy from multiple vantage points. IEEE, 2002.
- [56] A. Technologies. Akamai Network partnerships. <https://www.akamai.com/us/en/products/network-operator/akamai-network-partnerships.jsp>.

- [57] J. S. Varghese and L. Ruan. Computing customer cones of peering networks. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 35–37. ACM, 2016.
- [58] Q. Wang, G. M. Garrity, J. M. Tiedje, and J. R. Cole. Naive Bayesian classifier for rapid assignment of rRNA sequences into the new bacterial taxonomy. *Applied and environmental microbiology*, 73(16):5261–5267, 2007.
- [59] Wikipedia. List of Tier-2 ASes. https://en.wikipedia.org/wiki/Tier_2_network.
- [60] W. Willinger and M. Roughan. Internet topology research redux. *ACM SIGCOMM eBook: Recent Advances in Networking*, 2013.
- [61] J. Xia and L. Gao. On the evaluation of AS relationship inferences. In *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, volume 3, pages 1373–1377. IEEE, 2004.
- [62] H. Zhang. The optimality of Naive Bayes. In V. Barr and Z. Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*. AAAI Press, 2004.
- [63] H. Zhang and J. Su. Naive Bayesian classifiers for ranking. In *European conference on machine learning*, pages 501–512. Springer, 2004.

A AS-Rank Algorithm

The AS-Rank algorithm takes the following 11 steps to infer the relationship of each link.

1. Discard or sanitize paths with artifacts.
2. Sort ASes in decreasing order of computed transit degree, then node degree.
3. Infer a transit-free clique (i.e., Tier-1) ASes at top of AS hierarchy and label the links between every pair of ASes in the clique as p2p links.
4. Discard poisoned paths.
5. Visit ASes in order of the ranking in (2), and label a link as c2p if its previous link in a BGP path is composed of two clique members, or if its previous link in a BGP path is already labeled as c2p.
6. Infer c2p relationships from VPs inferred to be announcing no provider routes.
7. Infer c2p relationships for ASes where the customer has a larger transit degree.
8. Infer customers for ASes with no providers.
9. Infer c2p relationships between stub ASes and clique ASes.
10. Infer c2p relationships where adjacent links have no relationship inferred.
11. Infer all other links left as p2p.

Feature label	Meaning
f1	Number of VPs which observe a link
f2	Max distance to Tier-1
f3	Min distance to Tier-1
f4	Max node degree
f5	Min node degree
f6	Node degree difference
f7	Max transit degree
f8	Min transit degree
f9	Transit degree difference

Table 6: Features fed into GBDT

B AS-Rank Clique Inference

The clique inference algorithm in AS-Rank works as the following:

1. Find the top 10 ASes by transit degree.
2. If there are three consecutive members (X-Y-Z) in the top 10 ASes showing up in paths, and there are more than 5 ASes downstream from X Y Z (to make sure that the paths containing three consecutive members are not poisoned), disconnect the edge between X and Z even though X and Z are connected in some paths.
3. Find the largest clique in terms of transit degree sum among the top 10 ASes, denoted as C .
4. Visit the rest ASes top to down by transit degree, add an AS Z to C if Z has links with all members in C .
5. Similar to Step 2: If there are three consecutive members (X-Y-Z) in C showing up in paths, and there are more than 5 ASes downstream from X Y Z, disconnect the edge between X and Z.
6. Find the largest clique in C in terms of transit degree sum as the final inferred clique.

C Feature Importance Computation

Gradient boosting [20] is a widely used machine learning technique for solving classification problems. In gradient boosting, *GBDT* (Gradient boosting decision trees) produces a prediction model in the form of an ensemble of multiple decision trees. It is straightforward to retrieve importance scores for features when constructing *GBDT*. An importance score (F score) describes the number of times a feature is used to split the data across all trees. The more a feature is used to make key decisions with decision trees, the higher its importance score.

To decide what features can distinguish *hard* links from *easy* links in the Internet, we first split the validation dataset into two halves. The set of links which *CoreToLeaf* or AS-Rank infers incorrectly are labeled as “hard”, while those which are inferred correctly are labeled with “easy”. Then, we feed the features listed in Table 6 of links along with their labels into the *GBDT* and calculate the importance score corresponding to each feature.

Figure 9 plots the importance score of each feature divided by the sum of all features’ scores. We can tell the features

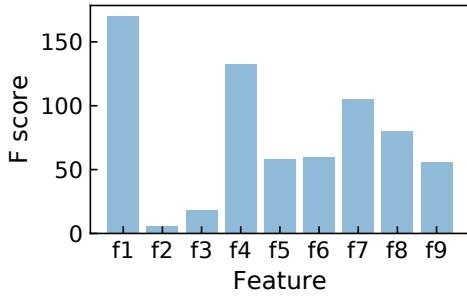


Figure 9: Feature importance scores provided by gradient boosting tree.

f_1, f_4, f_7, f_8, f_9 are the most important ones, so we translate them into the various categories of features to characterize “hard” links in §4.3.

D AS-Rank Sensitivity Analysis

D.1 Sensitivity to VP Selection

Each vantage point provides its own view of the Internet AS-level topology and the flow of traffic from the VP to rest of the Internet. VPs are located in different places, belong to different tiers, and they themselves have different import and export policies.

Even though the number of VPs have been growing over time, VPs are free to join or leave the set of public collectors, so the selection of VPs we have access to is arbitrary, biased, and under flux. A good AS relationship inference algorithm should not be sensitive to the selection of these VPs.

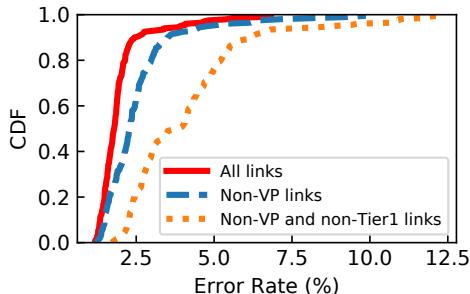


Figure 10: CDF of AS-Rank’s error rates on paths seen from 200 different half-VP sets.

We run the AS-Rank algorithm repeatedly, 200 times, against the 04/01/2017 BGP snapshot. For each of these 200 executions, we choose a random VP subset consisting of half of all available VPs (which we denote as V below) and give as input to the AS-Rank algorithm only the BGP paths visible to the VPs in that subset. Figure 10 plots CDFs of AS-Rank’s error rates using paths from these VP subsets. In the plot, we examine all links in the ground-truth dataset, links in the ground-truth dataset except links that directly connect with V (i.e., non-VP links), and the links in

the ground-truth dataset except V ’s links as well as Tier-1 links (i.e., non-VP and non-Tier1 links). The inference error rates on overall links range from 1.2% to 6.9%, and the error rates on *non-VP and non-Tier1* links range from 1.8% to 12.3%. AS-Rank’s accuracy is thus quite sensitive to the VP selections, especially for links which are relatively difficult to infer (i.e., not VP or Tier1 links).

D.2 Root Cause Analysis of AS-Rank Sensitivity

To illustrate the issue described in §5.3, let’s consider two VP sets, V_1 and V_2 , drawn from our 200 executions. AS-Rank’s inference accuracy from V_1 is low, while its inference accuracy from V_2 is high. The largest ten ASes differ for different sets of VPs because the transit degrees of ASes are determined by paths observed by the VPs. For example, the 9th largest AS (AS2914) observed by V_2 is the 12th largest AS observed by V_1 , so AS2914, which is a real Tier-1 AS, shows up in the clique chosen from the top 10 ASes using V_2 ’s paths, but it does not show up in the clique chosen using V_1 ’s paths.

For V_1 , the AS-Rank algorithm determines the maximum clique with the largest transit degree (from the top 10 ASes) to be “AS3356, AS6939, AS8220, AS9002, AS43531”. AS43531 is not considered for V_2 due to a relatively lower measurement of its transit degree, and it is not a higher-tier AS in reality. This affects the subsequent expansion of the clique, wherein ASes are considered in order by degree and added to the clique if they have connections to all members of the clique. So, in V_1 ’s execution, all added members are required to have a direct link with AS43531, and AS1764, AS8767, AS12389, AS12552, AS20485, AS25091, AS33891, AS43531, AS57724 are therefore all added to the clique, even though they are all low-tier ASes.

In a nutshell, the clique inference of AS-Rank algorithm is sensitive to the top 10 largest ASes ranked by transit degrees, which are determined by the selection of VPs and the selection of snapshots. Further, the clique membership determines the order in which links are analyzed by AS-Rank, impacts the computation of *customer cones* for each clique member (i.e., the set of ASes that a clique AS can reach using p2c links), and impacts the overall accuracy of the algorithm.

NetBouncer: Active Device and Link Failure Localization in Data Center Networks

Cheng Tan¹, Ze Jin², Chuanxiong Guo³, Tianrong Zhang⁴, Haitao Wu⁵, Karl Deng⁴, Dongming Bi⁴, and Dong Xiang⁴

¹New York University, ²Cornell University, ³Bytedance, ⁴Microsoft, ⁵Google

Abstract

The availability of data center services is jeopardized by various network incidents. One of the biggest challenges for network incident handling is to accurately localize the failures, among millions of servers and tens of thousands of network devices. In this paper, we propose NetBouncer, a failure localization system that leverages the IP-in-IP technique to actively probe paths in a data center network. NetBouncer provides a complete failure localization framework which is capable of detecting both device and link failures. It further introduces an algorithm for high accuracy link failure inference that is resilient to real-world data inconsistency by integrating both our troubleshooting domain knowledge and machine learning techniques. NetBouncer has been deployed in Microsoft Azure’s data centers for three years. And in practice, it produced no false positives and only a few false negatives so far.

1 Introduction

As many critical services have been hosted on the cloud (e.g., search, IaaS-based VMs, and databases), enormous data centers have been built which contain millions of machines and tens of thousands of network devices. In such a large-scale data center network, failures and incidents are inevitable, including routing misconfigurations, link flaps, network device hardware failures, and network device software bugs [19, 20, 22, 23, 28, 45]. As the foundation of network troubleshooting, failure localization becomes essential for maintaining a highly available data center.

Localizing network failures in a large-scale data center is challenging. Given that nowadays data centers have highly duplicated paths between any two end-hosts, it is unclear to the end-hosts which links or devices should be blamed when a failure happens (e.g., TCP retransmits). And, because of the Equal-Cost Multi-Path (ECMP) routing protocol, even routers are unaware of the whole routing path of a packet.

Moreover, recent research [23, 27, 28] reveals that *gray failures*, which are partial or subtle malfunctions, are prevalent within data centers. They cause the major availability breakdowns and performance anomalies in cloud environments [28]. Different from *fail-stop failures*, gray failures drop packets probabilistically, and hence cannot be detected by simply evaluating connectivity.

In order to localize failures and be readily deployable in a *production* data center, a failure localization system

needs to satisfy three key requirements, which previous systems [1, 16, 18, 23, 37, 44, 46] fail to meet simultaneously.

First, as for detecting gray failures, the failure localization system needs an end-host’s perspective. Gray failures have been characterized as “differential observability” [28], meaning that the failures are perceived differently by end-hosts and other network entities (e.g., switches). Therefore, traditional monitoring systems, which query switches for packet loss (e.g., SNMP, NetFlow), are unable to observe gray failures.

Second, to be readily deployable in practice, the monitoring system should be compatible with commodity hardware, the existing software stack and networking protocols. Previous systems which need special hardware support [37], substantial modification on the hypervisor [40] or tweak standard bits on network packets [46] are unable to be readily deployed in production data centers.

Third, localizing failures should be precise and accurate, in terms of pinpointing failures in fine-granularity (i.e., towards links and devices) and incurring few false positives or negatives. Some prior systems, like Pingmesh [23] and NetNORAD [1], can only pinpoint failures in a region, which needs extra efforts to discover the actual errors. And others [16, 18, 44] incur numerous false positives and false negatives when exposed to gray failures and real-world data inconsistency.

In this paper, we introduce NetBouncer, an active probing system that detects device failures and infers link failures from end-to-end probing data. NetBouncer satisfies the previous requirements by actively sending probing packets from the servers, which doesn’t need any modification in the network or underlying software stack. In order to localize failures accurately, NetBouncer provides a complete failure localization framework targeting data center networks, which incorporates real-world observations and troubleshooting domain knowledge. In particular, NetBouncer introduces:

- *An efficient and compatible path probing method* (§3). We design a probing method called *packet bouncing* to probe a designated path. It is built on top of the IP-in-IP technique [7, 47], which has been implemented in ASIC of modern commodity switches. Hence, packet bouncing is compatible to current data center networks and efficient without consuming switch CPU cycles.
- *A probing plan which is able to distinguish device failures and is proved to be link-identifiable* (§4). A probing plan is a set of paths which will be probed. Based on an observation that the vast majority of the network is

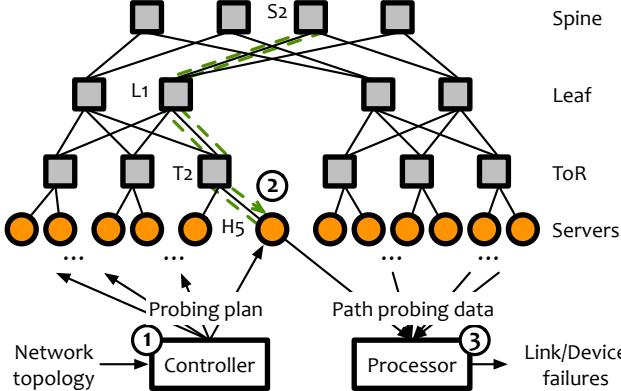


Figure 1: NetBouncer’s workflow: ①, the controller designs a probing plan and sends it to all the probing servers. ②, the servers follow the plan to probe the paths in the network. ③, the processor collects the probing data and infers the faulty devices and links.

healthy, we conceive a probing plan which reveals the device failures. And, by separating the faulty devices from the network, we *prove* that the remaining network is link-identifiable, meaning that the status of each link can be uniquely identified from the end-to-end path probing.

- *A link failure inference algorithm against real-world data inconsistency* (§5). A link-identifiable probing is not sufficient for pinpointing failures due to real-world data inconsistency. We formulate an optimization problem which incorporates our troubleshooting domain knowledge, and thus is resilient to the data inconsistency. And, by leveraging the characteristic of the network, we propose an efficient algorithm to infer link failures by solving this optimization problem.

NetBouncer has been implemented and deployed (§8) in Microsoft Azure for three years, and it has detected many network failures overlooked by traditional monitoring systems.

2 NetBouncer overview

NetBouncer is an active probing system which infers the device and link failures from the path probing data. NetBouncer’s workflow is depicted in Figure 1, which is divided into three phases as follows.

Probing plan design (① in Figure 1). NetBouncer has one central controller which produces a probing plan based on the network topology. A probing plan is a set of paths that would be probed within one probing epoch. Usually, the probing plan remains unchanged. Yet, for cases such as topology changes or probing frequency adjustments, the controller would update the probing plan.

An eligible probing plan should be link-identifiable, meaning that the probing paths should cover all links and

more importantly, provide enough information to determine the status of every single link. However, the constraints in developing real-world systems make it challenging to design a proper probing plan (§4.2).

Based on an observation that the vast majority of the links in a network is healthy, we prove the *sufficient probing theorem* (§4.3) which guarantees that NetBouncer’s probing plan is link-identifiable in a Clos network when at least one healthy path crosses each switch.

Efficient path probing via IP-in-IP (② in Figure 1). Based on the probing plan, the servers send probing packets through the network. The packet’s routing path (e.g., $H_5 \rightarrow T_2 \rightarrow L_1 \rightarrow S_2 \rightarrow L_1 \rightarrow T_2 \rightarrow H_5$ in Figure 1) is designated using the IP-in-IP technique (§3.1). After each probing epoch, the servers upload their probing data (i.e., the number of packets sent and received on each path) to NetBouncer’s processor.

NetBouncer needs a path probing scheme that can explicitly identify paths and imposes negligible overheads. Because the data center network is a performance-sensitive environment, even a small throughput decrease or latency increase can be a problem [30]. NetBouncer leverages the hardware feature in modern switches – the IP-in-IP [7, 47] technique – to explicitly probe paths with low cost.

Failure inference from path measurements (③ in Figure 1). The processor collects the probing data and runs an algorithm to infer the device and link failures (§4.4, §5.2). The results are then sent to the operators for further troubleshooting and failure processing.

The main challenge of inferring failures comes from the data inconsistency in the data center environment (§5.1). We’ve analyzed some real-world cases and encoded our troubleshooting domain knowledge into a specialized quadratic regularization term (§5.2). On top of that, we develop an efficient algorithm based on coordinate descent (CD) [53] which leverages the sparse characteristic of links in all paths. And the algorithm is more than one order of magnitude faster than off-the-shelf SGD solutions.

NetBouncer’s targets and limitations. NetBouncer targets non-transient (no shorter than the interval between two probing), packet-loss network incidents. Though its expertise is on detecting gray failures which would be overlooked by traditional monitoring systems, any other packet-loss related incidents are also under its radar.

Admittedly, there are cases where NetBouncer fails to detect (see false negatives in §8). We discuss NetBouncer’s limitations in more details in §9.

3 Path probing via packet bouncing

In a data center environment, the probing scheme of a troubleshooting system needs to satisfy two main requirements: first, the probing scheme should be able to pinpoint the routing path of probing packets, because a data center network provides many duplicated paths between two end-hosts.

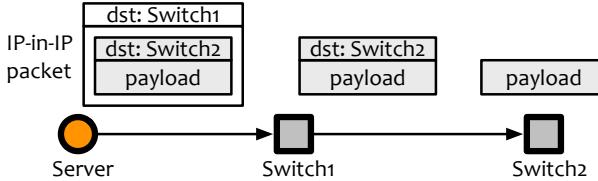
Second, the probing scheme should consume little network resources, in terms of switch CPUs and network bandwidth. This is especially important under heavy workloads when failures are more likely to happen.

Conventional probing tools fail to meet both requirements simultaneously. In short, ping-based probing is unable to pinpoint the routing path; Tracert consumes switch CPUs, which might adversely impact the reliability and manageability of the network.

NetBouncer designs and implements an approach called *packet bouncing*, which takes advantage of the IP-in-IP technique to accomplish both requirements. Other source routing schemes [15, 21, 26, 29] might also be plausible, but require much more deployment effort. NetBouncer uses probes from end-hosts. With programmable switches, it is possible to deploy probing agents at switches and probe each link individually. NetBouncer chooses end-host based approach as most of the switches in our data centers are still non-programmable. Nonetheless, NetBouncer’s failure localization algorithm applies to switch-based approaches as well.

3.1 IP-in-IP basics

IP-in-IP [7, 47] is an IP tunneling protocol that encapsulates one IP packet in another IP packet. This protocol has been implemented in the modern commodity switches (in ASIC) which allows devising a specific probing path without involving the CPUs of switches.



NetBouncer utilizes this IP-in-IP technique to explicitly probe one path by encapsulating the desired destination in the nested IP packets. In the above abstract example, NetBouncer is able to probe a certain path (Server → Switch₁ → Switch₂) by encapsulating the final destination (Switch₂) in the inner IP header and the intermediate hop (Switch₁) in the outer IP header. The switch that receives the IP-in-IP packets (i.e., Switch₁) would decapsulate the outer IP header and forward the packet to its next destination.

Indeed, some legacy or low-end switches might not support IP-in-IP in hardware. We do consider this challenge and design NetBouncer as only requiring the top-layer switches (i.e., core switches) having such support (details in §4.3). We believe that the core switches in a modern data center would be high-end with such support.

3.2 Packet bouncing

On top of the IP-in-IP technique, NetBouncer adopts a path probing strategy called packet bouncing. Namely, the

probing server chooses a switch as its destination and inquires the switch to bounce the packet back. As an example in Figure 1, a server (e.g., H₅) sends a probing packet to a switch (e.g., S₂). The probing path contains the route from the server to the switch (H₅ → T₂ → L₁ → S₂) and its “bouncing back” route (S₂ → L₁ → T₂ → H₅).

In NetBouncer’s target network, the Clos network [2, 48], packet bouncing simplifies NetBouncer’s model, design and implementation, due to the following three reasons.

(1) It minimizes the number of IP-in-IP headers NetBouncer has to prepare. The packet bouncing only needs to prepare one IP-in-IP header which leads to a simple implementation. Given a Clos network, *only one path* exists from a server to a upper-layer switch (also observed by [46]). Hence, preparing an IP-in-IP packet (to a upper-layer switch) only needs one outer IP header (with its destination as that switch), which remarkably simplifies the implementation.

(2) Links are evaluated bidirectionally which leads to a simpler model. When packet bouncing is in use, all the links are evaluated bidirectionally. This simplifies NetBouncer’s model, allowing the network graph to be undirected (§ 4.1). Indeed, this bidirectional evaluation cannot differentiate which direction of a link is dropping packets. However, in practice, this is not an issue because a link is problematic whichever direction drops packets.

(3) The sender and receiver are on the same server, which makes NetBouncer robust against server failures. Because of bouncing, the probing data for a certain path is preserved by one server, which is both the sender and the receiver. Thus, the probing data are “all or nothing”. Otherwise, if the senders and receivers are different servers, NetBouncer has to consider the failures of senders (fewer sent packets, causing false negatives) or receivers (fewer received packets, causing false positives) or both, which makes the failure handling more complicated, especially in a large-scale distributed system.

4 Probing plan and device failure detection

This section proposes NetBouncer’s failure localization model (§4.1) and introduces the challenges of probing path selection (§4.2) which motivates the probing plan design (§4.3) and device failure detection algorithm (§4.4).

We assume in this section that the success probability for each link is stable (i.e., remain the same among different measurements) which will be relaxed in the next section (§5).

4.1 Underlying model

We define a data center network as an undirected graph whose vertices are devices (e.g., servers, switches and routers) and edges are physical links. Each link has a *success probability*, which is denoted by x_i for the i^{th} link ($link_i$).

A path is a finite sequence of links which connect a sequence of devices. In NetBouncer, a probing path is

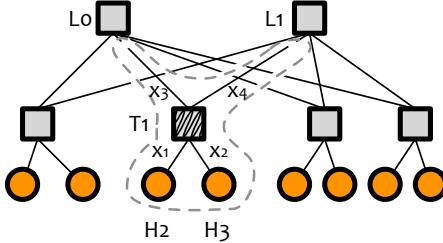


Figure 2: An unsolvable example. Switch T_1 cannot bounce packets. And, x_1, x_2, x_3, x_4 represent the success probabilities of link H_2-T_1 , H_3-T_1 , T_1-L_0 and T_1-L_1 respectively.

the sequence of links traversed by a probing packet from its sender to receiver. A *path success probability* is the probability of successfully delivering one packet through all links within this path. We use y_j to indicate the success probability of the j^{th} path (path $_j$).

NetBouncer’s model *assumes* that dropping packets on different links are independent events, which has been justified in earlier work [17, 18, 42] (also see §9 for more discussion). Thus, the probability of successfully delivering one packet through a path can be described as

$$y_j = \prod_{i: \text{link}_i \in \text{path}_j} x_i, \forall j, \quad (1)$$

where the success probability of path $_j$ is the product of its link success probabilities.

In the context of failure localization, the path success probabilities (y_j s) can be measured by sending probing packets through the paths, and our ultimate goal is to pinpoint the faulty links (whose success probabilities x_i s are below a certain threshold) and faulty devices (whose associated links are faulty).

4.2 Real-world challenges for path selection

In order to localize failures, the first question we need to answer is: *which paths should be probed so that all the links are identifiable?* This link identifiability problem can be formalized as follows.

Given a network graph G and all its possible paths U , how to construct a set $A \subseteq U$, so that the set of equations $\{y_j = \prod_{i: \text{link}_i \in \text{path}_j} x_i \mid \text{path}_j \in A\}$ has a unique solution for all x_i s.

Whether the above equations have a unique solution has been well-studied in the literature of linear algebra (by taking logarithm at both sides of Equation 1, it becomes linear). However, in reality, not all paths can be probed. The probing path must start and end at servers, since most switches cannot issue packets (most of the switches are non-programmable). Moreover, the bouncing scheme further restricts the sender and receiver to be the same server (§3.2).

Under such constraints, we notice that if *any* switch cannot “bounce” packets (i.e., doesn’t support IP-in-IP), there is no

unique solution. As an example, Figure 2 depicts a simple two-tier Clos network with switch T_1 (the shaded switch) unable to bounce packets. As a result, there doesn’t exist a unique solution in the circled subgraph, which is illustrated as follows.

Suppose we probe all the possible paths in the circled subgraph (i.e., $H_2-T_1-L_0$, $H_2-T_1-L_1$, $H_3-T_1-L_0$ and $H_3-T_1-L_1$) and obtain four equations as

$$\begin{aligned} y_{\{H_2-T_1-L_0\}} &= x_1 \times x_3, & y_{\{H_2-T_1-L_1\}} &= x_1 \times x_4, \\ y_{\{H_3-T_1-L_0\}} &= x_2 \times x_3, & y_{\{H_3-T_1-L_1\}} &= x_2 \times x_4. \end{aligned}$$

Intuitively, since one of the four equations is redundant ($y_{\{H_2-T_1-L_0\}} \times y_{\{H_3-T_1-L_1\}} = y_{\{H_2-T_1-L_1\}} \times y_{\{H_3-T_1-L_0\}}$), the number of effective equations is smaller than the number of variables. Thus, there doesn’t exist a unique solution *in general*.

Unfortunately, cases similar to the above example occur in a data center network for many reasons. On the one hand, some switches (especially ToR switches) may not support the IP-in-IP forwarding, so that they cannot bounce packets; On the other hand, delayed uploading and failures are common in a large-scale system. Within one epoch, the probing data from a certain switch may fail to be uploaded. More importantly, bouncing every single switch is expensive and thus not favorable in terms of the huge number of probing paths.

4.3 Link-identifiable probing plan

In view of the challenges when choosing the paths, finding a probing plan that has a unique solution is generally difficult. However, in the real-world scenario, we observe that the vast majority of the links in a network are well-behaved and thus most of the paths are healthy.

Motivated by this observation, we come up with the *sufficient probing theorem*, which proves that when the network is healthy (at least one healthy path passes each switch), a simple probing plan (probing all paths from the servers to the top-layer switches) is link-identifiable. By link-identifiable, we mean that this probing plan can guarantee a *unique* solution (i.e., a set of x_i s) to the path selection problem (§4.2) which is consistent with our measurements (i.e., all the y_j s). Therefore, this plan is used as NetBouncer’s probing plan.

Theorem 1. (*sufficient probing theorem*). *In a Clos network with k layers of switches ($k \geq 1$), by probing all paths from the servers to the top-layer switches, we can uniquely infer the link success probabilities from the measured path success probabilities, if and only if at least one path with success probability 1 passes each switch.*

The intuition behind the proof of this theorem (see full version proof in appendix §A) is that if the success probability of a path is 1, all the links included by this path should also have success probabilities 1, considering the constraint $x_i \in [0, 1], \forall i$.

Furthermore, from the proof, we can see that this theorem can be easily extended to all the *layered* networks. In fact, the Clos network is a special case of a general layered network, where switches on layer n only connect to switches on layers $n - 1$ and $n + 1$, switches on the same layer do not connect to each other, and servers connect only to the first-layer switches.

Most of the probing plan designs in the literature [11, 12, 36, 39, 44, 54] target how to minimize the number of probing paths. Reducing the probing path number, however, is not a goal of NetBouncer. In fact, redundant paths through one link can be considered as validations to each other. These validations in turn increase NetBouncer’s accuracy.

4.4 Device failure detection

Using NetBouncer’s probing plan, Theorem 1 provides a sufficient and necessary condition (i.e., *at least one path with success probability 1 passes each switch*) for the existence of a unique solution. By checking whether the above condition holds for each switch, we can split a Clos network into a solvable part (having a unique solution) and an unsolvable part (no unique solution).

The unsolvable part would be a collection of switches which fail to have even one good path across it. Since NetBouncer probes many paths (usually hundreds to thousands) across each switch, one switch is highly suspicious if it doesn’t even have one good path through it. Hence, NetBouncer reports these switches as faulty devices to the operators.

Theoretically, the reported device can be a false positive if it happens to be surrounded by bad devices. However, this case is extremely rare since one switch usually connects to many devices. Thus, we are highly confident that the reported devices are faulty.

To sum up, the servers first probe paths based on the probing plan in §4.3. Then the processor collects all the probing data from the servers, and extracts the faulty devices (unsolvable part) from the network. Based on Theorem 1, the remaining subgraph has a unique solution for each link’s success probability. Next the processor runs the link failure inference algorithm described in the next section (§5.2), and infers the faulty links. Finally, the processor reports the faulty devices and links to the operators. The algorithm running on NetBouncer’s processor is depicted in Figure 3.

5 Link failure inference

The previous section describes NetBouncer’s probing plan and algorithm for localizing device failures. Yet, the last jigsaw piece of NetBouncer’s algorithm (Figure 3, line 6) is still missing: *how can one infer the link probabilities x_{is} from the end-to-end path measurements y_{js} ?*

Define:

$devs$: all devices

$Y : path \rightarrow [0,1]$ // a map from a path to its success probability

```

1: procedure PROCESSOR()
2:   (1) Collect probing data from agents as  $Y$ 
3:   (2)  $badDev \leftarrow \text{DETECTBADDEVICES}(Y)$  // line 9
4:   // eliminate the unsolvable subgraph
5:   (3)  $Y \leftarrow Y \setminus \{\text{path}_r \mid \text{path}_r \text{ passes any device in } badDev\}$ 
6:   (4)  $badLink \leftarrow \text{DETECTBADLINKS}(Y)$  // in Figure 5, §5.2
7:   return  $badDev, badLink$ 
8:
9: procedure DETECTBADDEVICES( $Y$ )
10:    $badDev \leftarrow \{\}$ 
11:   for  $dev_p$  in  $devs$  :
12:      $goodPath \leftarrow \text{False}$ 
13:     for all  $\text{path}_q$  passes  $dev_p$  :
14:        $If Y[\text{path}_q] = 1$  then  $goodPath \leftarrow \text{True}$ ; break
15:     If not  $goodPath$  then  $badDev += dev_p$ 
16:   return  $badDev$ 

```

Figure 3: Algorithm running on NetBouncer processor.

In practice, the above inference problem cannot be resolved simply using linear algebra or least squares, because of the real-world data inconsistency.

5.1 Data inconsistency

In the real-world data center environment, the measurement data are usually inconsistent or even conflicting. Such data inconsistency derives from two main reasons:

- *Imperfect measurement.* The data center network is huge and its state changes constantly. Due to its gigantic size, all the paths cannot be probed simultaneously. Thus, different path probings may reflect different instantaneous states of the network. Moreover, as the probing sample size is limited (hundreds of packets per path), the measurements on each path are coarse-grained.
- *Accidental packet loss.* In a large-scale network, accidental errors are inevitable, which can happen on any layer (e.g., hypervisor, OS, TCP/IP library) of the execution stack as a result of bugs, misconfigurations, and failures.

These two reasons lead to inconsistency in the path probing data and further to misreporting (mostly false positives, reporting a well-behaved link as a faulty one). The reason why *accidental packet loss* introduces false positives is straightforward. As it incurs dropping packets which no link or device should be responsible for, such packets might be attributed to the non-lossy links which produces false positives.

As for the *imperfect measurement*, the reason why it causes false positives is that the inference results might *overfit* the imperfect measurements. We demonstrate this problem by a real-world example (Figure 4).

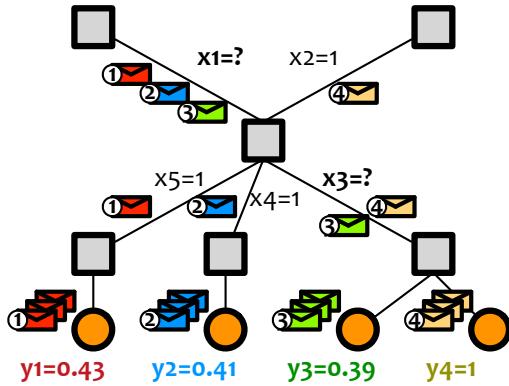


Figure 4: A false positive example of the least square solution overfitting imperfect measurement data. The circled number on the probing packets indicates which path this packet passes.

In Figure 4, we have priori knowledge that some links are good ($x_2 = x_4 = x_5 = 1$), and we want to infer the link success probabilities x_1 and x_3 from observed path success probabilities ($y_1 = 0.43$, $y_2 = 0.41$, $y_3 = 0.39$, and $y_4 = 1$). Using the least squares approach we obtain the estimates $x_1 = 0.406$ and $x_3 = 0.965$, which indicates that both links are dropping packets. However, the faulty link with respect to x_3 , unfortunately, is a false positive. Such false positive is caused by the imperfect measurements of y_1, y_2, y_3 as their observed success probabilities are slightly different. In this case, the least square results overfit the imperfect data when minimizing the fitting error.

To mitigate the above false positives, we introduce a specialized regularization term and propose a regularized estimator for the latent factor model to resolve the failure localization problem, which is described in the next section.

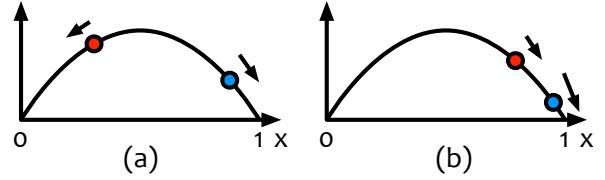
5.2 NetBouncer’s latent factor model

We have formulated a latent factor model for the link failure inference problem. Under the constraint $x_i \in [0, 1], \forall i$, the objective function to be minimized when estimating the latent link probabilities x_i s is the sum of squared errors plus a regularization term as

$$\begin{aligned} & \text{minimize}_{x_i} \sum_j (y_j - \prod_{i: \text{link}_i \in \text{path}_j} x_i)^2 + \lambda \sum_i x_i(1-x_i) \\ & \text{subject to } 0 \leq x_i \leq 1, \forall i \end{aligned} \quad (2)$$

Specialized regularization. In the model, we have designed a specialized regularization term $\sum_i x_i(1-x_i)$ which incorporates our troubleshooting domain knowledge to reduce false positives described in §5.1.

There are two desired characteristics of this regularization term: (a), it has a two-direction penalty; (b), because of the quadratic term, the closer to 1 the greater the slope.



The characteristic (a) separates the bad links and the good links, as it tends to move the link probability toward 0 or 1. The insight behind this is that the regularization term tends to “pull” the good links to be better, while “push” the bad links to be worse, while the product of link probabilities will stay approximately the same. It helps resolve the false positive cases (e.g., Figure 4 in §5.1) where the imperfect measurement involves a bad link (x_1) and a good link (x_3).

The characteristic (b) mitigates the accidental packet loss and noisy measurements, which helps endorse most links (good) and assign the blame to only a small number of links (bad). The intuition of this characteristic is that (i) most of the links are good, and (ii) the larger the success probability (x_i closer to 1) the more likely the loss is an accidental loss or an inaccurate/noisy measurement. In response, when one x_i is closer to 1, the regularization term provides stronger strength (greater slope) to “pull” this x_i to be 1 (i.e., a good link).

As for the standard penalties, some (e.g., L_1 and L_2) only promote one-direction shrinkage; Other two-direction penalties (e.g., entropy) are inefficient in terms of analytical solution and numerical optimization (our regularization term leads to a neat analytical solution and an associated efficient minimization algorithm).

Non-convex representation. In our model (Equation 2), we use a non-convex representation which, in practice, has better performance than its corresponding convex representation.

From the theoretical perspective, convexity is a desired property that guarantees the convergence to the global optimal solution. The convex representation can be obtained by applying a logarithm transformation to Equation 2 (similar model used by Netscope[18]). It converts the multiplication equations to linear equations and results in a convex problem.

However, our experiments (§6.4) show that the non-convex representation has better performance. The reason is that the convex representation suffers from a scale change and skewed log function (e.g., machine epsilon, no $\log(0)$ exists), and thus does not work well numerically in practice.

5.3 Algorithm for link failure inference

Given the above optimization problem, we adopt coordinate descent (CD), an optimization method, to solve the link failure inference problem. This algorithm is depicted in Figure 5 (the pseudocode and complexity analysis are in appendix §B).

Coordinate descent leverages the sparse characteristic of the real-world network which results in an efficient algorithm. By sparsity, we mean that, in a data center network, each link is included by only a few paths comparing to the whole path

Define:

$$X \leftarrow \text{all } x_i, \quad Y \leftarrow \text{all } y_i$$

$$f(X, Y) \leftarrow \sum_j (y_j - \prod_{i: \text{link}_i \in \text{path}_j} x_i)^2 + \lambda \sum_i x_i (1 - x_i)$$

```

1: procedure DETECTBADLINKS(Y)
2:    $X \leftarrow \text{INITLINKPROBABILITY}(Y)$  // line 12
3:    $L_0 \leftarrow f(X, Y)$  // initial value for target function  $f$ 
4:   for iteration  $k = 1, \dots, \text{MaxLoop}$ :
5:     for each  $x_i$  in  $X$ :
6:        $x_i \leftarrow \underset{x_i}{\operatorname{argmin}} f(X, Y)$ 
7:       project  $x_i$  to  $[0, 1]$ 
8:        $L_k \leftarrow f(X, Y)$ 
9:       If  $L_{k-1} - L_k < \varepsilon$  then break the loop
10:      return  $\{(i, x_i) | x_i \leq \text{bad link threshold}\}$ 
11:
12: procedure INITLINKPROBABILITY(Y)
13:    $X \leftarrow \{\}$ 
14:   for linki in links:
15:     // initialize link success probability
16:      $x_i \leftarrow \text{avg}(\{y_j | \text{link}_i \in \text{path}_j\})$ 
17:   return  $X$ 

```

Figure 5: Coordinate Descent for regularized least squares with constraints. x_i is the success probability of link_i; y_j is the success probability of path_j; ε is the threshold of path error; λ is the tuning parameter of regularization. X is the set of all x_i s which changes when x_i s are updated.

set. Consequently, for updating the success probability of a link, it is more efficient to leverage the information of the paths relevant to this specific link, which is exactly how CD works.

Why not SGD? Stochastic Gradient Descent (SGD) [52] is a classical and universal optimization algorithm. Nevertheless, it fails to meet our throughput requirement due to the huge amount of data (hundreds GB per hour) generated by the real-world data centers (we’ve tried to improve SGD to our best, such as lazy-update optimization [8, 32, 38]). By unable to meet our requirement, we mean that SGD cannot produce the failure links within a certain time budget (in practice, 5min for one probing epoch, see §7.2). Our experiments show that CD converges more than one order of magnitude faster than SGD (§6.4).

There are two reasons why SGD is much slower. First, SGD iterates over all the paths, and during each loop for a particular path, all link probabilities are updated simultaneously due to the regularization term. Such updating does not take advantage of the path’s sparse representation (only a few links are related to one path) and includes all the links no matter whether they are on the paths or not, and thus is not efficient.

Second, SGD suffers from the boundary constraints. When the link probability is supposed to cross the boundary 0 or 1 according to its current update, we have to clip it and consequently lose information about its gradient.

6 Simulation studies

We run simulations to demonstrate the following: that NetBouncer’s probing plan is sufficient (§6.2); that NetBouncer’s device failure detection is effective (§6.3); that NetBouncer’s design choices are justified (§6.4), and that NetBouncer performs well comparing with other designs (§6.5).

6.1 Simulation setup

We build our simulation on top of a python trace generator whose outputs are path probing data consumed by NetBouncer’s processor. The trace generator simulates the path probings on a standard three-layer Clos network [2, 48] including 2.8 thousand switches (48-port switches), 27.6 thousand servers and 82.9 thousand links.

We follow the settings from previous work [18, 41], while change the specific loss rates to fit the data center environment¹: for faulty links/devices, the packet drop probabilities are between 0.02 and 1; and for non-lossy links, the packet drop probabilities are between 0 and 0.001 (to simulate noise). In addition, we randomly choose 10 devices as faulty devices.

For each probing path, 100 packets are sent. Whether a packet will be successfully transmitted is determined by generating a random number uniformly between 0 and 1. NetBouncer considers a link good if its estimated success probability is greater than 0.999 (because the noise rate is 0.001). All the experiment results are the averages of 10 executions.

6.2 Probing plan

We perform both NetBouncer’s probing plan (§4.3) and a *hop-by-hop probing plan* in this experiment. Hop-by-hop probing plan is a probing plan that sends packets from every server to every relevant switch. Because of its exhausted probing, hop-by-hop probing plan is able to identify all the links, but with very high cost.

The results of hop-by-hop probing plan and NetBouncer’s probing plan are listed in columns “Hop-by-hop” and “NetBouncer” of Figure 6 respectively. NetBouncer’s probing plan achieves the same performance as hop-by-hop probing (there are minor differences with 10% faulty links, which come from the randomness of faulty link selection), while it remarkably reduces the number of paths to be probed.

6.3 Device failure detection

In §4.2, we demonstrate that if all the paths through a specific device are dropping packets, we cannot uniquely infer the link success probabilities (no unique solution exists), which motivates our device failure detection algorithm (§4.4). To

¹ We change the packet drop probability of any faulty link from $[0.05, 1]$ to $[0.02, 1]$, and that of any non-lossy link from $[0.0002, 1]$ to $[0, 0.001]$, which makes the failure detection more challenging.

Faulty link%	Hop-by-hop		w/o DFD		Convex		$L_1 (\lambda=0.5)$			$L_1 (\lambda=1)$			$L_1 (\lambda=2)$			NetBouncer ($\lambda=1$)		
	#FN	#FP	#FN	#FP	#FN	#FP	#FN	#FP	Err	#FN	#FP	Err	#FN	#FP	Err	#FN	#FP	Err
0.1%	0	0	135.3	0	0	46.9	0	48.5	0.01	0	0	0.03	0.3	0	0.14	0	0	0.01
1%	0	0	164.0	0	1.9	522.7	0	81.1	0.07	0	0	0.32	1.1	0	1.41	0	0	0.11
10%	0.6	0	123.3	0	257.6	4.1k	0	695.7	0.91	0.1	0.6	3.80	25.6	0	15.88	0.3	0.2	1.43

Figure 6: Simulation experiment results on variants of NetBouncer with setup in §6.1. “Faulty link%” indicates the proportion of faulty links over all links. “#FN” and “#FP” indicates the number of false negatives and false positives. “Err” is the estimation error (the smaller the better, see the definition in §6.4). As for the results, “Hop-by-hop” represents the experiment using hop-by-hop probing (§6.2); “w/o DFD” represents the results without faulty device detection (§6.3); “Convex” indicates the experiment on convex representation; “ L_1 ” represents the experiments using standard regularization with different parameters (§6.4). And, the final column “NetBouncer” is the performance of NetBouncer.

understand the necessity of device failure detection, we evaluate NetBouncer without its device failure detection (DFD) under different faulty link ratios. Column “w/o DFD” in Figure 6 shows the results.

Without the help of faulty device detection, NetBouncer produces many false negatives. There are two main reasons: First, without faulty device detection, the links associated with the faulty devices have an infinite number of solutions (§4.2). Hence, NetBouncer may end up with an arbitrary solution. Second, the regularization in NetBouncer tends to aggregate the path failures and assign the blames to a small number of links, which is reasonable for link failure detection, but not for faulty device cases, because all the links connected to a faulty device are faulty and should be blamed.

6.4 NetBouncer design choices

To verify NetBouncer’s design choices in §5.2 and §5.3, we compare NetBouncer to its variants with alternative choices.

Convex vs. non-convex. We implement the convex version of NetBouncer with similar regularization by applying a logarithmic transformation to Equation 2, and design an algorithm similar² to Figure 5 to solve this convex problem.

From the results in column “Convex” of Figure 6, we can see that the convex representation has both false positives and false negatives, which mainly derive from its skewed log scale and boundary clipping. For example, when $x=0$, $\log(x)$ is invalid, so we have to assume $x=1e-8$ as an approximation.

L_1 vs. specialized regularization. We now evaluate the effectiveness of NetBouncer’s specialized regularization by comparing it with a standard regularization L_1 . The L_1 regularization term is defined as $-\sum_i x_i$ [18]. In order to compare the estimation accuracy, we use the squared estimation error of link probabilities as a metric, which is defined as $\sum_i (x_i - x'_i)^2$ (x_i is the success probability from “ground truth” and x'_i is its estimate). The smaller the error metric (“Err” in Figure 6), the more accurate the estimation.

The results using the L_1 regularization with different λ s are presented in Figure 6. From the experiments, we find that the

²There are slight differences for the boundary handling due to the log scale of the convex model.

OptMethod	Learning rate	#round	Time(s)
CD	–	4	14.8
SGD-lazy	0.001	145	513.3
SGD-lazy	0.005	45	157.5
SGD-lazy	0.01	161	569.9

Figure 7: Comparison of CD and SGD. The faulty link% of the workload is 0.1% and λ is 1. To help SGD converge, we have relaxed the convergence condition for SGD ($\epsilon = 0.0001$ for CD, $\epsilon = 0.001$ for SGD). “#round” is the number of rounds to converge. “Time(s)” is the total time of execution in seconds.

specialized regularization obtains similar numbers of false discoveries to L_1 under the best tuning parameter λ in terms of failure diagnosis, while it achieves much lower estimation error when fitting link probabilities.

SGD vs. CD. Although SGD is broadly used as an off-the-shelf method, we adopt CD, a more efficient method for NetBouncer based on the sparse structure of our problem. To make a fair comparison, we implement an SGD with the lazy-update optimization [8, 32, 38], which is more efficient than a naive implementation.

Figure 7 summarizes the performance of CD and SGD. It shows that CD converges much faster than SGD. The time spent on each round of CD and SGD are similar, but CD converges in significantly fewer rounds, which validates our analysis.

Regularization parameter λ . The results of NetBouncer are affected by the tuning parameter λ in the regularization affects. Intuitively, λ balances the fitting error and false discoveries. A large λ may trigger false negatives, while a small λ may leave false positives.

To illustrate how the results change as λ varies, we run NetBouncer with different λ values on the network with 1% faulty links. The numbers of false discoveries are shown in Figure 8 where the x-axis is in log-scale. The results demonstrate the trade-off between false positives and false negatives from choosing λ .

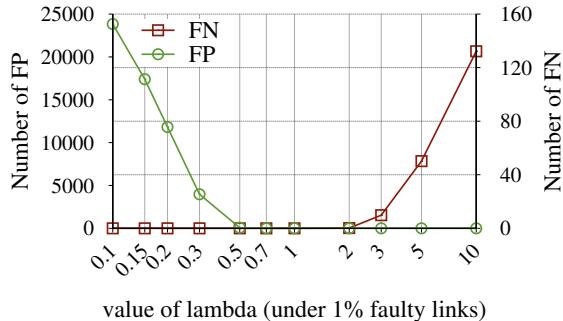


Figure 8: Number of false positives and false negatives for different λ .

Faulty link%	0.1%		1%		10%	
	#FN	#FP	#FN	#FP	#FN	#FP
NetBouncer	0	0	0	0	0.3	0.2
deTector (0.6)	187.5	6.0k	215.5	7.2k	204.0	22.8k
deTector (0.9)	204.5	0.7	191.5	0.4	208.0	21.7
NetScope (0.1)	0	9.1k	3.0	10.8k	167.5	12.6k
NetScope (1)	0.3	43.7	10.2	395.5	319.5	3.8k
NetScope (10)	28.7	6.3	291.5	86.7	2.4k	1.2k
KDD14	7.8	21.0	76.6	433.2	213.8	3.0k

Figure 9: Compare NetBouncer with existing schemes. The number in parentheses of “deTector” is a tuning parameter to filter false positives (*hit ratio* in paper[44]). The number in parentheses of “NetScope” is its regularization parameter (ω in paper[18]).

6.5 Comparison with existing systems

In this section, we compare NetBouncer with three existing systems: deTector [44], NetScope [18], and KDD14 [25]. Note that we only compare the failure inference part on the processor (step ③ in Figure 1), that is inferring failures from the path measurement data. To the best of our knowledge, deTector is the state-of-the-art heuristic algorithm after a long line of work [16, 31, 43]. NetScope improves the previous work [41, 49] and takes data inconsistency into account. KDD14 applies statistical techniques to data center failure localization. They are the most relevant network diagnosis algorithms, and thus the most appropriate benchmarks for NetBouncer.

The original NetScope is designed for troubleshooting on Internet which is an underdetermined system, whereas data center network is an overdetermined system. As a result, we extend the NetScope algorithm following its core idea – “ L_1 -norm minimization with non-negativity constraints” (§III.D in [18]), and apply it to the logarithmic transformation of Equation 1. For KDD14, we assume that the routing path of each packet is known, which is an improvement over the original version.

Using the same setup in §6.1, we run experiments on NetBouncer as well as deTector, NetScope and KDD14 with various faulty link ratios, and present the results in Figure 9.

As a greedy algorithm, deTector is designed to blame the smallest number of links, so that it incurs false negatives

when there are faulty devices. On the other hand, deTector also incurs false positives due to noise in the data. Lastly, it uses a tuning parameter (i.e., hit ratio in [44]) to filter false positives, which may result in false negatives as well. This is a trade-off requiring domain knowledge and experiences from the network operators.

Similar to the implementation of NetBouncer’s convex representation, NetScope encounters numerical problems mainly due to its logarithmic representation. KDD14 produces both false positives and false negatives resulting from its assumption that there is at most one faulty link among all the paths between two servers, which is unrealistic for a large-scale data center network.

7 Implementation and evaluation

7.1 Implementation

Controller. The NetBouncer Controller is a central place to decide how the agents probe the whole network. It takes the network topology as input and generates a probing plan for servers. The controller has multiple replicas for fault tolerance.

Agent. The NetBouncer Agent runs on servers. It fetches its probing plan from the Controller, which contains the paths to be probed. For each path, the probing plan contains the related parameters including the number of packets to send, the packet size, the UDP source destination port range, the probe frequency, the TTL and ToS values, etc. For each probed path, the Agent generates a record which contains the path, the packet length, the total number of packets sent, the number of packet drops, the RTTs at different percentiles. The CPU and traffic overhead of the agents are both negligible in practice.

Processor. The NetBouncer Processor has two parts. A front-end which is responsible for collecting the records from the NetBouncer Agents, and a back-end Data processor which runs the algorithm. The front-end and back-end run on the same physical server. For load-balance and geo fault tolerance considerations, we run multiple NetBouncer Processors. Each Processor is responsible for a set of data center regions. Within a geolocation, we run four NetBouncer Processor instances behind a software load-balancer VIP. One instance acts as the master and runs the NetBouncer algorithm, the other three are slaves. The VIP is configured in a way so that only the master receives records from the agents.

Result verification and visualization. After localizing failures in the network, NetBouncer provides a result verification tool which the operators can use to issue probing packets on-demand. These on-demand verification packets are sent from the same Agents as the NetBouncer service.

This tool also shows the packet drop history of links for visualization. One failure example is illustrated in Figure 10. It reveals the packet drop history of a link detected by NetBouncer. Users can click the “Quick probe” button to

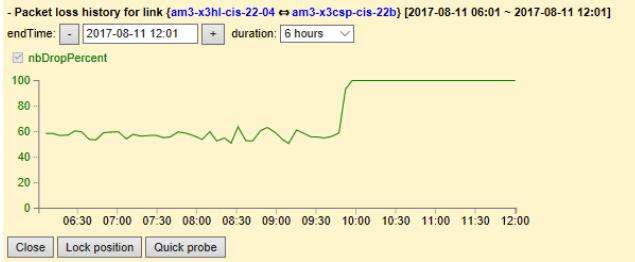


Figure 10: Result verification and visualization tool.

launch probes on-demand for verification. As the figure shows, the packet drop rate of the link changed from 60% to 100% at around 10:00. This is because the NetBouncer report triggered our network repairing service, which in turn shutdown this link to mitigate the incident. The link will be manually inspected, cleaned up, and restored later.

With the help of this verification tool, we gain more confidence in the performance of NetBouncer. For the cases we have verified, we did not experience any false positives. However, NetBouncer is not false negative-free, as discussed in § 9. **Probing epoch.** Probing epoch, the interval that NetBouncer does one run of failure inference algorithm, is a critical parameter of the system design. One deceptive intuition is that a shorter probing epoch always leads to faster failure detection. However, in reality, too frequent failure inferences may result in either less confident results or bursts of probing traffic.

From our experiences, determining the probing epoch is a three-way trade-off among inference accuracy, probing resource cost, and failure localization speed. On one hand, with fixed probing frequency, inferring failures too often ends up with less probing data for each inference, and hence may harm the accuracy. On the other hand, by increasing the probing frequency on servers, the accuracy is able to keep unchanged. Nevertheless, it may cause probing traffic bursts, which may introduce network congestion, ramping up the risk of instability of the whole network.

Considering the previous tradeoffs, NetBouncer chooses 5 minutes as one probing epoch. Yet, it is plausible to sacrifice the other two factors to significantly shorten the probing time.

7.2 Data processor runtime evaluation

In this section, we evaluate the performance of Data Processor, which is executed on a machine having Intel Xeon E5 2.4GHz CPU with 24 cores (48 logical cores) and 128GB memory. The operating system is Windows Server 2016.

We have around 30 regions in total. The Data Processor uses one thread to process the data for one region. Data processing in the Data Processor is therefore naturally parallelized.

We run NetBouncer on one-hour real-world data trace on November 5, 2016, which is 130GB in size and covers tens of global data centers (other hours have similar results). In production, NetBouncer detects the faulty links in the

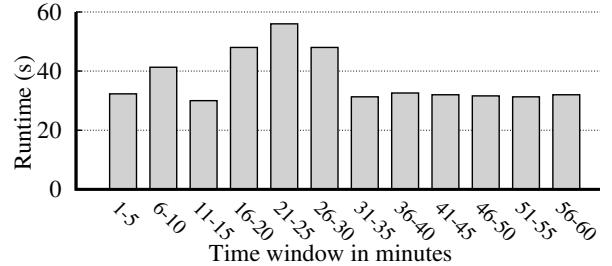


Figure 11: NetBouncer running time on real-world data.

hopping window of every 5 minutes. We follow the same detection frequency in this experiment.

As we show in Figure 11, the min, average, and max running times are 30.0s, 37.3s, and 56.0s, respectively. The max processing time happened at window 21-25, in which our detection algorithm converged after 9 iterations, whereas most of the rest windows finished in 4 iterations. We did the investigation and found that the additional iterations are caused by a few “hard-to-solve” faulty links, when the faulty links appear in the same local sub-graph. In that case, our algorithm needs to go back and forth with several more iterations to converge. In all the cases, the time-to-detection (TTD) for failures is within 60 seconds.

We also study the time spent on each of the stages. On average, NetBouncer’s algorithm with regularization (§5.2) costs 54.9% of the CPU time; other stages – faulty device detection (§4.4), data cleaning and data structure setup – take 12.4%, 23.8% and 8.9%, respectively. NetBouncer consumes about 15-20 GB memory during the processing which is only a small portion of the 128G memory of the server.

Overall, a single NetBouncer Processor instance can handle multiple regions with tens of thousands of switches and more than one million network links, with spare capacity for future growth.

8 Deployment experiences

NetBouncer has been running in Microsoft Azure for three years. In this section, we’re going to share our deployment experiences and some representative failures NetBouncer detected.

NetBouncer deployment, before vs. after. Before NetBouncer was deployed, gray failures could last hours to days. Since no clue was provided, operators had to pinpoint the failure via trial and error in a large region of the network based on their troubleshooting experiences.

After NetBouncer went online, it has successfully reduced the detection time from hours to minutes, and further shortening is also possible (see the probing epoch in §7.1). Moreover, NetBouncer greatly deepened our understanding of the reasons why packet drops happen, including silent packet drops, packet blackholes, link congestion, link flapping, BGP routing flapping, switch unplanned reboot, etc. For example, it

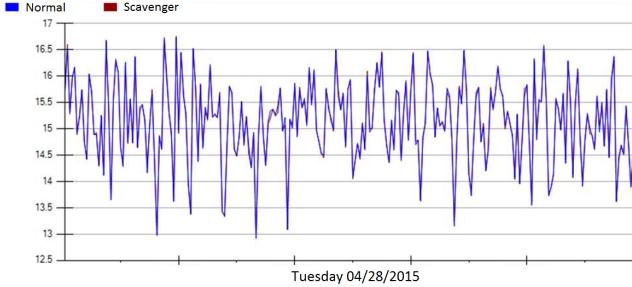


Figure 12: The packet drop probability detected by NetBouncer. This is a silent packet drop case.

once caught a case where a switch was periodically changing its state between normal and 100% packet drops. It turned out that the switch was rebooted continuously by a buggy firmware upgrade procedure.

Next, we present three representative cases in production detected by NetBouncer, which otherwise would be extremely difficult or tricky to locate.

Case 1: spine router gray failure. The first case is one of the most challenging scenarios that motivated the development of NetBouncer: gray failures, where switches silently drop packets without any signals. We had an incident where a spine switch was dropping packets silently, because of an issue in one of this switch’s linecard hardware. Many of our customers experienced packet drops and latency increases. The wide impact was also detected by our end-to-end latency service Pingmesh [23]. It was clear that one or more spine switches were dropping packets. But we could not tell which one.

We obtained the link loss rate history of all the spine links using NetBouncer. Figure 12 shows the packet drop history of the lossy link. We found that this link was constantly dropping packets with around 15% packet drop probability. It dropped packets without differentiation, as both the “Normal” and “Scavenger” traffic encountered the same dropping rate (the two curves in Figure 12 overlap with each other). In our network, Normal traffic is given priority over Scavenger traffic.

Case 2: polarized traffic. In the second case, NetBouncer caught a network congestion scenario and also helped identify the root-cause of the incident: a switch firmware bug, which polarized the traffic load onto a single link. Figure 13 shows the packet drop probability measured by NetBouncer. Among a huge number of packet drops, we observed that the packets on Scavenger traffic was dropped at a probability around 35%, but the Normal traffic was not affected. Since NetBouncer directly told us the congested link, detection became trivial. We then mitigated this issue by rekeying the ECMP hash function and solved the problem.

Case 3: miscounting TTL. Time to live (TTL) in an IP packet is supposed to be decremented by one through each switch. However, NetBouncer has discovered that when an IP packet passes through a certain set of switches, its TTL is decremented by two. This issue manifests as a “false positive” by misclassifying affected good links as bad links, which is in fact caused

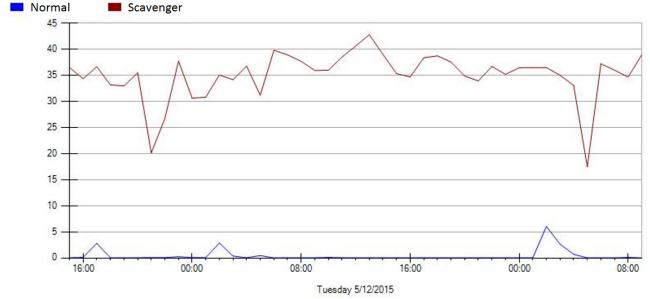


Figure 13: The packet drop probability detected by NetBouncer, caused by link polarization.

by an internal switch fireware bug. Though this miscounting TTL bug hasn’t caused harmful consequences on the actual data traffic yet. Nevertheless, such a hidden issue will raise severe latent risk for service reliability and cause huge confusion for troubleshooting.

False negatives and false positives. In practice, we ran into several false negative cases. In one case, we once ran into a DHCP booting failure, in which some servers could send out the DHCP DISCOVER packets, but could not receive the responding DHCP OFFER packets from the DHCP server. NetBouncer did not detect such DHCP packet drops. Resorting to packet capturing, we could identify that the switches did not drop the DHCP OFFER packets, and this problem was caused by the NIC.

In another case, we encountered an incident due to a misconfigured switch ACL, which resulted in packet drops for a very limited set of IP addresses. Since NetBouncer scanned a wide range of IP addresses, so the signal detected by NetBouncer was weak. Similarly, NetBouncer cannot help when some firewall rules were (wrongly) applied to certain applications.

NetBouncer in theory is not false positive-free. However, NetBouncer did not produce false positives in production so far, because of our specialized regularization (§5.2) and a strict criteria (1% packet drops) for reducing false alarms.

9 Discussions

NetBouncer’s limitations. NetBouncer has two major limitations. First, NetBouncer makes an assumption that the probing packets experience the same failures as real applications, which may not hold in all cases (as we shown in §8). Our future work is to systematically investigate the false negative conditions and improve the coverage of our probing packets.

Second, theoretically, NetBouncer cannot guarantee zero false positives or zero false negatives. Nevertheless, this is ubiquitous to all real-world monitoring systems, since the measurement data cannot be perfectly accurate. In practice, NetBouncer has produced no false positives and only a few false negatives (both confirmed by the network operators) so far.

Theory vs. practice. Theoretically, NetBouncer’s probing plan is proved to be link-identifiable (§4.3). However, in practice, such nice theory property does not guarantee the results to be false positive free or false negative free, which drove us to seek help from machine learning approaches (latent factor model, regularization and CD).

Yet, we argue that the theory result, though not sufficient, is necessary for solving our problem in reality. Without the guidance of the *sufficient probing theorem*, a chosen probing plan might not be link-identifiable, thus the outputs of the machine learning approach can be arbitrary.

Does the independent assumption hold? In NetBouncer’s model (§4.1), we assume that the failures are independent. Our experiences reveal that this assumption holds for a large number of scenarios including random packet drops due to cable/fiber bit errors, infrequently hardware gray failures e.g., bit flips in hardware memory, and packet drops caused by software Heisenbugs. Those scenarios share the same characteristic that they are hard to detect and localize. Once detected, they are typically easy to mitigate (by shutting down the problematic links or rebooting the faulty devices).

How does NetBouncer handle congestion packet loss? As stated in overview (§2), NetBouncer targets *non-transient* failures. NetBouncer treats persistent congestion as failure because persistent congestion affects users. NetBouncer filters out transient congestion since it uses minute-level failure detection interval.

10 Related work

Network tomography. Compared with original network tomography approaches [5, 6, 9, 13, 14, 17, 18, 51] which target the Internet-like networks, NetBouncer has different challenges. First, the topology of a data center network is known, but it requires to design a link-identifiable probing plan. Second, the standard of a well-behaving link in the Internet (failure probability < 2% in [18]) is way lower than that in a data center network (usually, failure probability < 0.1%).

As for the link failure inference algorithm, Tomo [16] and deTector [44] use heuristic algorithms for failure inference. However, these approaches may generate false results due to their heuristic nature. NetScope [18] (as well as NetQuest [49] and LIA [41]) takes data inconsistency into account. NetBouncer also uses a similar approach (i.e., regularization), but in addition we encode our troubleshooting domain knowledge into the model which results in better performance.

What differentiates NetBouncer from other tomography systems is that NetBouncer provides a complete framework targeting data center networks, including probing plan design (§4.3), device failure detection (§4.4) and link failure inference (§5.2) against real-world data inconsistency.

Other failure localization approaches. SNMP and OpenFlow are widely used in nowadays data centers. However,

recent research [23, 28] shows that these tools cannot detect gray failures, a type of failure that causes availability breakdowns and performance anomalies in cloud environment.

Herodotou et al. [25] use statistical data mining techniques for data center failure localization. They propose a probabilistic path model since it does not control the path of a probing packet. Whereas, NetBouncer controls the routing paths of the probing packets. Furthermore, they assume there is only one link failure in a path, which leads to false positives and false negatives as we have shown in §6.5.

Sherlock [4] assumes that only few failures exist in the system. It then enumerates all the possible combinations to find the best match. The running time grows exponentially along with number of failures, which is unacceptable for a large-scale network.

Pingmesh [23], NetSonar [55] and NetNORAD [1] use a TCP or UDP agent for end-to-end reachability and traceroute variants (Tcptraceroute or ftraceroute) for path probes. However, the ICMP packets generated from the probes need to be handled by the switch CPUs, hence need to be carefully managed. In addition, NetSonar uses Sherlock [4] algorithm for failure detection, which cannot support many simultaneous failures.

Passive probing [46] uses core switches to tag IDs in the DSCP or TTL field of IP header for path pinpointing. However, DSCP and TTL fields, which are commonly used for QoS, might not be available.

NetPoirot [3] and [10] leverage decision trees for failure diagnosis. The data sources are from end-host application and TCP logs. These approaches can tell whether the problem is from the network or not, but they do not work for our scenario as they do not differentiate ECMP paths.

Network troubleshooting. Several systems [24, 50, 56] have been proposed for network troubleshooting and debugging. These systems typically need to capture packets or collect packet summaries, which are complementary to NetBouncer.

Other troubleshooting systems need either non-trivial modification on software stack [33, 34, 35, 40], or hardware support [37], which cannot be transparently applied to current data centers in production.

11 Conclusion

In this paper, we propose the design and implementation of NetBouncer, an active probing system which infers the device and link failures from the path probing data. We demonstrate that NetBouncer’s probing plan design, device failure detection, and link failure inference perform well in practice. NetBouncer has been running in Microsoft Azure’s data centers for three years, and has helped mitigate numerous network incidents.

References

- [1] ADAMS, A., LAPUKHOV, P., AND ZENG, J. H. Netnorad: Troubleshooting networks via end-to-end probing. <https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/>, February 2016.
- [2] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM* (2008).
- [3] ARZANI, B., CIRACI, S., LOO, B. T., SCHUSTER, A., AND OUTHRED, G. Taking the blame game out of data centers operations with netpoirot. In *ACM SIGCOMM* (2016).
- [4] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM* (2007).
- [5] BATSAKIS, A., MALIK, T., AND TERZIS, A. Practical passive lossy link inference. In *Passive and Active Measurement Workshop* (2005).
- [6] CÁCERES, R., DUFIELD, N. G., HOROWITZ, J., AND TOWSLEY, D. Multicast-based inference of network-internal loss characteristics. *IEEE Trans. Inform. Theory* 45 (November 1999).
- [7] CAO, J., XIA, R., YANG, P., GUO, C., LU, G., YUAN, L., ZHENG, Y., WU, H., XIONG, Y., AND MALTZ, D. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2013).
- [8] CARPENTER, B. Lazy sparse stochastic gradient descent for regularized multinomial logistic regression. *Alias-i, Inc., Tech. Rep* (2008), 1–20.
- [9] CASTRO, R., COATES, M., LIANG, G., NOWAK, R., AND YU, B. Network tomography: Recent developments. *Statistical Science* 19 (August 2004).
- [10] CHEN, M., ZHENG, A. X., LLOYD, J., JORDAN, M. I., AND BREWER, E. Failure diagnosis using decision trees. In *Proceedings of the First International Conference on Autonomic Computing* (2004).
- [11] CHEN, Y., BINDEL, D., AND KATZ, R. H. Tomography-based overlay network monitoring. In *ACM IMC* (2003).
- [12] CHUA, D. B., KOLACZYK, E. D., AND CROVELLA, M. Efficient monitoring of end-to-end network properties. In *IEEE International Conference on Computer Communications (INFOCOM)* (2005).
- [13] COATES, M., AND NOWAK, R. Network Loss Inference Using Unicast End-to-End Measurement. In *Proc. ITC Conf IP Traffic, Modeling and Management* (2000).
- [14] CUNHA, I., TEIXEIRA, R., FEAMSTER, N., AND DIOT, C. Measurement methods for fast and accurate blackhole identification with binary tomography. In *ACM IMC* (2009).
- [15] DE GHEIN, L. *MPLS fundamentals*. Cisco Press, 2016.
- [16] DHAMDHERE, A., TEIXEIRA, R., DOVROLIS, C., AND DIOT, C. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2007).
- [17] DUFFIELD, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory* 52 (Dec 2006).
- [18] GHITA, D., NGUYEN, H., KURANT, M., ARGYRAKI, K., AND THIRAN, P. Netscope: Practical network loss tomography. In *IEEE International Conference on Computer Communications (INFOCOM)* (2010).
- [19] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM* (2011).
- [20] GOVINDAN, R., MINEI, I., KALLAHALL, M., KOLEY, B., AND VAHDAT, A. Evolve or die: High-availability design principles drawn from google’s network infrastructure. In *ACM SIGCOMM* (2016).
- [21] GUILBAUD, N., AND CARTLIDGE, R. Localizing packet loss in a large complex network (ppt). <https://www.nanog.org/meetings/nanog57/presentations/Tuesday/tues.general.GuilbaudCartlidge.Topology.7.pdf>.
- [22] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing? lessons from hundreds of service outages. In *SoCC* (2016).
- [23] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM* (2015).
- [24] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).
- [25] HERODOTOU, H., DING, B., BALAKRISHNAN, S., OUTHRED, G., AND FITTER, P. Scalable near real-time failure localization of data center networks. In *KDD* (2014).
- [26] HU, S., CHEN, K., WU, H., BAI, W., LAN, C., WANG, H., ZHAO, H., AND GUO, C. Explicit path control in commodity data centers: Design and applications. *IEEE/ACM Transactions on Networking* (2016).
- [27] HUANG, P., GUO, C., LORCH, J. R., ZHOU, L., AND DANG, Y. Capturing and enhancing in situ system observability for failure detection. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2018).
- [28] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray failure: The Achilles’ heel of cloud-scale systems. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2017).
- [29] JYOTHI, S. A., DONG, M., AND GODFREY, P. Towards a flexible data center fabric with source routing. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015).
- [30] KOHAVI, R., AND LONGBOTHAM, R. Online experiments: Lessons learned. *IEEE Computer* (September 2007).
- [31] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. Ip fault localization via risk modeling. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2005).
- [32] LANGFORD, J., LI, L., AND ZHANG, T. Sparse online learning via truncated gradient. *Journal of Machine Learning Research* 10, Mar (2009), 777–801.
- [33] LENERS, J. B., GUPTA, T., AGUILERA, M. K., AND WALFISH, M. Improving availability in distributed systems with failure informers. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2013).
- [34] LENERS, J. B., GUPTA, T., AGUILERA, M. K., AND WALFISH, M. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)* (2015).
- [35] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the falcon spy network. In *ACM Symposium on Operating Systems Principles (SOSP)* (2011).
- [36] LI, H., GAO, Y., DONG, W., AND CHEN, C. Taming both predictable and unpredictable link failures for network tomography. In *Proceedings of the ACM Turing 50th Celebration Conference-China* (2017).
- [37] LI, Y., MIAO, R., KIM, C., AND YU, M. Lossradar: Fast detection of lost packets in data center networks. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2016).

- [38] LIPTON, Z. C., AND ELKAN, C. Efficient elastic net regularization for sparse linear models. *arXiv preprint arXiv:1505.06449* (2015).
- [39] MA, L., HE, T., LEUNG, K. K., SWAMI, A., AND TOWSLEY, D. Identifiability of link metrics based on end-to-end path measurements. In *Proceedings of the 2013 conference on Internet measurement conference* (2013).
- [40] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM* (2016).
- [41] NGUYEN, H. X., AND THIRAN, P. Network loss inference with second order statistics of end-to-end flows. In *ACM SIGCOMM* (2007).
- [42] PADMANBHA, V., QIU, L., AND WANG, H. Server-based inference of internet performance. In *In Proc. of IEEE INFOCOM* (2003).
- [43] PATI, Y. C., REZAIIFAR, R., AND KRISHNAPRASAD, P. S. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on* (1993).
- [44] PENG, Y., YANG, J., WU, C., GUO, C., HU, C., AND LI, Z. detector: a topology-aware monitoring system for data center networks. In *USENIX Annual Technical Conference* (2017).
- [45] POTHARAJU, R., AND JAIN, N. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *SoCC* (2013).
- [46] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive real-time datacenter fault detection and localization. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2017).
- [47] SIMPSON, W. IP in IP Tunneling, 1995. RFC 1853.
- [48] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network. In *ACM SIGCOMM* (2015).
- [49] SONG, H. H., QIU, L., AND ZHANG, Y. Netquest: a flexible framework for large-scale network measurement. In *ACM SIGMETRICS Performance Evaluation Review* (2006).
- [50] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2016).
- [51] VARDI, Y. Netowrk tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association* 91 (March 1996).
- [52] WIKIPEDIA. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.
- [53] WRIGHT, S. J. Coordinate descent algorithms. *Mathematical Programming* 151, 1 (2015), 3–34.
- [54] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2012).
- [55] ZENG, H., MAHAJAN, R., MCKEOWN, N., VARGHESE, G., YUAN, L., AND ZHANG, M. Measuring and troubleshooting large operational multipath networks with gray box testing. Tech. Rep. MSR-TR-2015-55, Microsoft Research, 2015.
- [56] ZHU, Y., AND ET AL. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015).

A Proof of sufficient probing theorem

Proof. Sufficient condition: For any link with its success probability x_0 , we consider the only two possibilities:

(1) If a path with success probability 1 includes this link, then $x_0 = 1$.

(2) Otherwise, the condition guarantees that at least one link with success probability 1 connects the upper (lower) node of this link to a node in the upper (lower) layer. According to the way we probe, at least one path includes this link so that all the other links in this path have success probability 1. Thus, we will find an equation $x_0 = y_0$ in the path success probability equations.

Necessary condition: Assume the condition does not hold, then there exists at least one node so that all of its links have success probability not 1. Therefore, there exists a subgraph including this node connected with n nodes in the upper layer and m nodes in the lower layer, which is separable from the rest of the whole graph.

Given that the success probabilities of all the other links in the rest of the whole graph are known, we try to solve this subgraph and consider the only three possibilities:

(1) If all subpath probabilities are not 0 in this subgraph, then we can transform the path success probability equations of this subgraph to the linear equations

$$\log y_{i,n+j} = \log x_i + \log x_{n+j}, 1 \leq i \leq n; 1 \leq j \leq m$$

where the rank of corresponding matrix is $n + m - 1$, less than the number of linear equations. As a result, no unique solution is available.

(2) If some but not all subpath probabilities are 0 in this subgraph, we only focus on the nonzero equations, which do not have a unique solution due to the redundancy either.

(3) If all subpath probabilities are 0 in this subgraph, we simply cannot distinguish the solution where all links are 0 or just some are 0.

To sum up, the solution is not unique in the subgraph even when the rest of the whole graph is solved. Therefore, it is impossible to get a unique solution for the whole graph. \square

B Failure inference algorithm and complexity analysis

Algorithm 1 describes the full version of NetBouncer’s failure inference algorithm, Coordinate Descent for regularized least squares with constraints.

We analyze the complexity of Algorithm 1 as follows. Assume that each link is included by C paths on average. The time complexity is $O(CM)$ for link initialization. In each iteration, the time complexity is $O(CM)$ for link updating, and $O(N + M)$ for convergence checking. Suppose there are K iterations until convergence, the total time complexity is $O(CM + KCM + KN + KM) = O(KCM + KN)$. The space complexity is $O(N)$ for all path rates, and $O(M)$ for all link rates, and $O(N)$ for the mapping from the paths to the links they include, and $O(CM)$ for the mapping from the links to the paths including them. Then the total space complexity is $O(N + M + N + CM) = O(N + CM)$.

Algorithm 1 Coordinate Descent for regularized least squares with constraints.

Require: $N = \text{number of paths}$, $M = \text{number of links}$, $y_j = \text{sample success probability of path } j$, $n_j = \text{sample size of path } j$, $K = \text{maximal number of iterations}$, $\epsilon = \text{threshold of path error}$, $\lambda = \text{tuning parameter of regularization}$.

```

1: initialize  $x_i^{(0)} = \sum_{j:x_i \in y_j} (n_j y_j) / \sum_{j:x_i \in y_j} n_j, i \in [1, M]$ 
2: for iteration  $k = 1, \dots, K$  do
3:    $x_i^{(k)} = x_i^{(k-1)}, i \in [1, M]$ 
4:   for link  $i = 1, \dots, M$  do
5:      $R_i^{(k)} = 2 \sum_{j:x_i \in y_j} \left( \prod_{\ell: \ell \neq i, x_\ell \in y_j} x_\ell^{(k)} \right)^2 - 2\lambda$ 
6:      $S_i^{(k)} = 2 \sum_{j:x_i \in y_j} (y_j \prod_{\ell: \ell \neq i, x_\ell \in y_j} x_\ell^{(k)}) - \lambda$ 
7:      $T_i^{(k)} = S_i^{(k)} / R_i^{(k)}$ 
8:     if  $R_i^{(k)} = 0$  then
9:       if  $S_i^{(k)} > 0$  then  $x_i^{(k)} = 1$ 
10:      else  $x_i^{(k)} = 0$ 
11:    else if  $R_i^{(k)} > 0$  then
12:      if  $T_i^{(k)} > 1$  then  $x_i^{(k)} = 1$ 
13:      else if  $T_i^{(k)} < 0$  then  $x_i^{(k)} = 0$ 
14:      else  $x_i^{(k)} = T_i^{(k)}$ 
15:    else
16:      if  $T_i^{(k)} > 1/2$  then  $x_i^{(k)} = 0$ 
17:      else  $x_i^{(k)} = 1$ 
18:     $L^{(k)} = \sum_j (y_j - \prod_{i:x_i \in y_j} x_i^{(k)})^2 + \lambda \sum_i x_i^{(k)} (1 - x_i^{(k)})$ 
19:    If  $L^{(k-1)} - L^{(k)} < \epsilon$  then break the loop
return  $\{x_i^{(k)} | i \in [1, M]\}$ 

```

C Acknowledgement

We thank Jain Shalabh Jain and Pradeepkumar Mani for their contributions to the early NetBouncer system.

Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services

Frank Wang
MIT CSAIL

Ronny Ko, James Mickens
Harvard University

Abstract

Riverbed is a new framework for building privacy-respecting web services. Using a simple policy language, users define restrictions on how a remote service can process and store sensitive data. A transparent Riverbed proxy sits between a user’s front-end client (e.g., a web browser) and the back-end server code. The back-end code remotely attests to the proxy, demonstrating that the code respects user policies; in particular, the server code attests that it executes within a Riverbed-compatible managed runtime that uses IFC to enforce user policies. If attestation succeeds, the proxy releases the user’s data, tagging it with the user-defined policies. On the server-side, the Riverbed runtime places all data with compatible policies into the same universe (i.e., the same isolated instance of the full web service). The universe mechanism allows Riverbed to work with unmodified, legacy software; unlike prior IFC systems, Riverbed does not require developers to reason about security lattices, or manually annotate code with labels. Riverbed imposes only modest performance overheads, with worst-case slowdowns of 10% for several real applications.

1 INTRODUCTION

In a web service, a client like a desktop browser or smartphone app interacts with datacenter machines. Although smartphones and web browsers provide rich platforms for computation, the core application state typically resides in cloud storage. This state accrues much of its value from server-side computations that involve no participation (or explicit consent) from end-user devices.

By running the bulk of an application atop VMs in a commodity cloud, developers receive two benefits. First, developers shift the burden of server administration to professional datacenter operators. Second, developers gain access to scale-out resources that vastly exceed those that are available to a single user device. Scale-out storage allows developers to co-locate large amounts of data from multiple users; scale-out computation allows developers to process the co-located data for the benefit of users (e.g., by providing tailored search results) and the benefit of the application (e.g., by providing targeted advertising).

1.1 A Loss of User Control

Unfortunately, there is a disadvantage to migrating application code and user data from a user’s local machine to a remote datacenter server: the user loses control over where her data is stored, how it is computed upon, and how the data (and its derivatives) are shared with other services. Users are increasingly aware of the risks associated with unauthorized data leakage [11, 62, 82], and some governments have begun to mandate that online services provide users with more control over how their data is processed. For example, in 2016, the EU passed the General Data Protection Regulation [28]. Articles 6, 7, and 8 of the GDPR state that users must give consent for their data to be accessed. Article 17 defines a user’s right to request her data to be deleted; Article 32 requires a company to implement “appropriate” security measures for data-handling pipelines. Unfortunately, requirements like these *lack strong definitions and enforcement mechanisms at the systems level*. Laws like GDPR provide little technical guidance to a developer who wants to comply with the laws while still providing the sophisticated applications that users enjoy.

The research community has proposed information flow control (IFC) as a way to constrain how sensitive data spreads throughout a complex system [35, 42]. IFC assigns labels to program variables or OS-level resources like processes and pipes; given a partial ordering which defines the desired security relationships between labels, an IFC system can enforce rich properties involving data secrecy and integrity. Unfortunately, traditional IFC is too burdensome to use in modern, large-scale web services. The reason is that creating and maintaining a partial ordering of labels is too difficult—the average programmer or end-user struggles to reason about data safety via the abstraction of fine-grained label hierarchies. As a result, no popular, large-scale web service uses IFC to restrict how sensitive data is processed and shared.

1.2 Our Solution: Riverbed

In this paper, we introduce Riverbed, a distributed web platform for safeguarding the privacy of user data. Riverbed provides benefits to both web developers and end users. To web developers, Riverbed provides a *practical* IFC system which

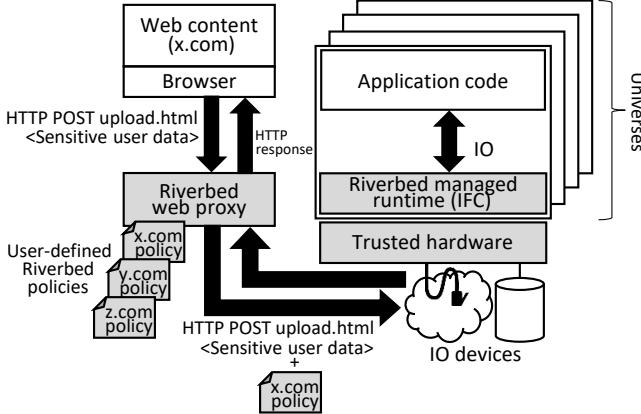


Figure 1: Riverbed’s architecture. The user’s client device is on the left, and the web service is on the right. Unmodified components are white; modified or new components are grey.

allows developers to easily “bolt on” stronger security policies for complex applications written in standard managed languages. To end users, Riverbed provides a straightforward mechanism to verify that server-side code is running within a privacy-preserving environment.

Figure 1 describes Riverbed’s architecture. For each Riverbed web service, a user defines an information flow policy using simple, human-understandable constraints like “do not save my data to persistent storage” or “my data may only be sent over the network to `x.com`.“ In the common case, users employ predefined, templated policy files that are designed by user advocacy groups like the EFF. When a user generates an HTTP request, a web proxy on the user’s device transparently adds the appropriate data flow policy as a special HTTP header.

Within a datacenter, Riverbed leverages the fact that many services run atop managed runtimes like Python, .NET, or the JVM. Riverbed modifies such a runtime to automatically taint incoming HTTP data with the associated user policies. As the application derives new data from tainted bytes, the runtime ensures that the new data is also marked as tainted. If the application tries to externalize data via the disk or the network, the externalization is only allowed if it is permitted by user policies. The Riverbed runtime terminates an application process which attempts a disallowed externalization.

In Riverbed, application code (i.e., the code which the managed runtime executes) is totally unaware that IFC is occurring. Application developers have no way to read, write, create, or destroy taints and data flow policies. The advantage of this scheme is that it makes Riverbed compatible with code that has not been explicitly annotated with traditional IFC labels. However, different end users will likely define incompatible data flow policies. As a result, policy-agnostic code would quickly generate a policy violation for some subset of users; Riverbed would then terminate the application. To avoid this problem, Riverbed spawns multiple, lightweight copies of the back-end service, one for each set of users

who share the same data flow policies. We call each copy a *universe*. Since users in the same universe allow the same types of data manipulations, any policy violations indicate true problems with the application (e.g., the application tried to transmit sensitive data to a server that was not whitelisted by the inhabitants of the universe).

Before a user’s Riverbed proxy sends data to a server, the proxy employs remote attestation [9, 15] to verify that the server is running an IFC-enforcing Riverbed runtime. Importantly, a trusted server will perform *next-hop attestation*—the server will not transmit sensitive data to another network endpoint unless that endpoint is an attested Riverbed runtime whose TLS certificate name is explicitly whitelisted by the user’s data flow policy. In this manner, Riverbed enables controlled data sharing between machines that span different domains.

1.3 Our Contributions

To the best of our knowledge, Riverbed is the first distributed IFC framework which is practical enough to support large-scale, feature-rich web services that are written in general-purpose managed languages. Riverbed preserves the traditional advantages of cloud-based applications, allowing developers to offload administrative tasks and leverage scale-out resources. However, Riverbed’s universe mechanism, coupled with a simple policy language, provides users with understandable, enforceable abstractions for controlling how datacenters manipulate sensitive data. Riverbed makes it easier for developers to comply with laws like GDPR—users give explicit consent for data access via Riverbed policies, with server-side universes constraining how user data may be processed, and where its derivatives can be stored.

We have ported several non-trivial applications to Riverbed, and written data flow policies for those applications. Our experiments show that Riverbed enforces policies with worst-case end-to-end overheads of 10%. Riverbed also supports legacy code with little or no developer intervention, making it easy for well-intentioned (but average-skill) developers to write services that respect user privacy.

2 RELATED WORK

In this section, we compare Riverbed to representative instances of prior IFC systems. At a high level, Riverbed’s innovation is the leveraging of universes and human-understandable, user-defined policies to enforce data flow constraints in IFC-unaware programs. Riverbed enforces these constraints without requiring developers to add security annotations to source code.

2.1 Explicit Labeling

In a classic IFC system, developers explicitly label program state, and construct a lattice which defines the ways in which differently-labeled state can interact. Roughly speaking, a program is composed of assignment statements; the IFC system only allows a particular assignment if all of the policies

involving righthand objects are compatible with the policies of the lefthand side.

IFC-visible assignments can be defined at various levels of granularity. For example, Jif [48], Fabric [45], and similar frameworks [14, 29, 79, 80] modify the compiler and runtime for a managed language, tracking information flow at the granularity of individual program variables. In contrast, frameworks like Thoth [25], Flume [39], Camflow [50], and DStar [81] modify the OS, associating labels with processes, IO channels, and OS-visible objects like files. Taint can be tracked at even high levels of abstraction, e.g., at the granularity of inputs and outputs to MapReduce tasks [66].

All of these approaches require developers to reason about a complex security lattice which captures relationships between a large number of privileges and privilege-using entities like users and groups. Porting a complex legacy application to such a framework would be prohibitively expensive, and to the best of our knowledge, there is no large-scale, deployed system that was written from scratch using IFC with explicit labeling. Developer-specified labels are also a poor fit for our problem domain of *user-specified* access policies.

Tracking data flows at too-high levels of abstraction can introduce problems of overtainting—to avoid false negatives, systems must often use pessimistic assumptions about how outputs should be tainted. For different reasons, overtainting is also a challenge for ISA-level taint tracking [27, 65]. For example, if taint is accidentally assigned to %ebp or %esp, then taint will rapidly propagate throughout the system, yielding many false positives [67]. To avoid these problems, Riverbed taints at the managed runtime level, a level which does not expose raw pointers, and defines data types with less ambiguous tainting semantics.

In Jeeves [7, 78], a developer explicitly associates each sensitive data object with a high-confidentiality value, a low-confidentiality value, and a policy which describes the contexts in which a particular value should be exposed. An object’s value is symbolic until the object is passed to an output sink, at which point Jeeves uses the context of the sink to assign a concrete value to the object. Riverbed avoids the need for developers to label objects with policies or concrete values with different fidelities; via the universe mechanism, Riverbed applications always compute on high-fidelity data while satisfying user-defined constraints on data propagation.

2.2 Implicit Labeling

Some IFC systems use predefined taint sources and IFC policies. For example, TaintDroid [26] uses a modified JVM to track information flows in Android applications. TaintDroid predefines a group of sensors and databases that generate sensitive data; examples of these sources include a smartphone’s GPS unit and SMS database. The only sink of interest is the network, because TaintDroid’s only goal is to prevent sensitive information from leaking via the network. Because TaintDroid uses a fixed, application-agnostic set of IFC rules,

TaintDroid works on unmodified applications. Riverbed also works on unmodified applications. However, TaintDroid operates on a single-user device, whereas Riverbed targets a web service that has many users, each of whom may have unique preferences for how their data should be used. Thus, Riverbed requires users (but not developers or applications) to explicitly define information flow policies. Riverbed also requires the universe mechanism (§4.4) to prevent the mingling of data from users with incompatible flow policies.

2.3 Formal Verification

IronClad [33] servers, like Riverbed servers, use remote attestation to inform clients about the server-side software stack. In Ironclad, server-side code is written in Dafny [41], a language that is amenable to static verification of functional correctness. Nothing prevents Riverbed from executing formally-verified programs; however, Riverbed’s emphasis on running complex code in arbitrary managed languages means that Riverbed is generally unable to provide formal assurances about server-side code.

3 THREAT MODEL

Riverbed assumes that developers want to enforce user-defined privacy policies, but are loathe to refactor code to do so. Thus, Riverbed assumes that server-side code is weakly adversarial: poorly-designed applications may unintentionally try to leak data via explicit flows, but developers will not intentionally write code that attempts to surreptitiously leak data, e.g., via implicit flows, or via targeted attacks on the taint-tracking managed runtime. Riverbed is compatible with mechanisms for tracking implicit flows [5, 6, 10, 61], but our Riverbed prototype does not track them for several reasons. One reason is that the punitive aspects of laws like the GDPR disincentivize companies from writing code that intentionally subverts compliance mechanisms like Riverbed. Furthermore, in many common programming languages, mechanisms for detecting implicit flows have undesirable properties like flagging some well-behaved programs as malicious [5], or requiring annotations from developers [6]. Riverbed strives for compatibility with legacy, non-annotated code written in popular languages.

A datacenter operator has physical access to servers, which enables direct manipulation of server RAM. So, our current Riverbed prototype assumes that datacenter operators are trusted. To ease this assumption, Riverbed could leverage a hardware-enforced isolation mechanism like SGX [20, 37]. However, SGX places limits on the memory size of secure applications. SGX also requires the applications to run in ring 3, forcing the code to rely on an untrusted OS in ring 0 to perform IO; the result is a large number of context switches for applications that perform many IOs [8]. Riverbed strives to be compatible with complex applications that issue frequent IOs. Thus, our Riverbed prototype eschews mechanisms like SGX, and must be content with not protecting against actively-

malicious datacenter operators. To implement remote attestation, Riverbed does rely on tamper-resistant, server-side TPM hardware (§4.3), but TPMs do not protect against physical attacks on the rest of the server hardware.

Riverbed assumes that the entire client-side is trusted, with the exception of the web content in a particular page. Buggy or malicious content may try to disclose too much information to a server. However, Riverbed ensures that whatever data is sent will be properly tagged. Since Riverbed uses TLS to authenticate network endpoints, the HTTPS certificate infrastructure must be trusted.

On a server, Riverbed’s TCB consists of a taint-tracking managed runtime, a reverse proxy that forwards requests to the appropriate universes (§4.4), the TPM hardware that provides the root of trust for attestation, and a daemon which servers use to attest to clients. We make standard cryptographic assumptions about the strength of the ciphers and hash functions used by the attestation protocol. Between the TPM hardware and the managed runtime are a boot loader, a hypervisor, and other systems software. Each end-user can choose a different collection of intermediate software to trust. A user’s preferences are expressed in her policies (§4.2), so that Riverbed’s client-side proxy can refuse to disclose data to untrusted server-side systems code.

4 DESIGN

Figure 1 provides a high-level overview of the Riverbed architecture. In this section, we provide more details on how users specify their policies, and how Riverbed enforces those policies on the server-side.

4.1 Riverbed-amenable Services

Riverbed is best-suited for certain types of web services.

- **Services with per-user silos for application state, and no cross-user sharing:** Examples include back-up services like Ionic [51], and private note-taking apps like Turtl [47]. Riverbed prevents information leakage between per-user silos (although an individual silo may span multiple server-side hostnames and cloud providers).
- **Services that silo user data according to explicitly-defined group affinities:** For example, a social networking site can create a universe for the state belonging to a corporation’s private group. The corporation’s users map to the same Riverbed user (§4.2), with no data flows between different corporations. Financial analysis sites and email services can use this decomposition to isolate data belonging to a particular business or social group.
- **Services which aggregate unaffiliated users by shared policies:** For example, in a news site, users can define policies that impact whether the site may aggregate user data for targeted advertising. Riverbed places users with equivalent policies into the same universe, ensuring that the site respects each user’s preferences.

```

USER-ID: ALICE
AGGREGATION: False
PERSISTENT-STORAGE: True
ALLOW-TO-NETWORK: x.com
ALLOW-TO-NETWORK: y.com
TRUSTED-SERVER-STACK: {
    83145c082bbf608989f05e85c3c211f83,
    c8cd7ac93cab2b94f65a5b2de5709767f,
    ...
    590f01d8d18b1141988ee1975b3ce3b30
}

```

Figure 2: An example of a Riverbed policy. For simplicity, we elide graph-based contextual attestation predicates (§4.3).

Child policies (§4.2) can whitelist communication between server-side endpoints with otherwise incompatible policies. However, such whitelisting is easier when the server-side application consists of small, well-defined components, so that whitelisting individual components has well-understood security implications.

4.2 Expressing Policies

Figure 2 provides an example of a Riverbed policy. A policy consists of several parts, as described below.

The `USER-ID` field describes the owner of the policy. User ids only need to be unique within the context of a particular web service. Riverbed is agnostic about the mechanism that a service uses to authenticate users and log them into the service. However, Riverbed’s server-side reverse proxy must know who owns the policy that is associated with each user request, so that the proxy can forward the request to the appropriate universe (§4.4).

Since Riverbed is agnostic about a service’s login mechanism, a `USER-ID` field could actually be bound to a group of users. In this scenario, the users in the group would have different service-specific usernames, but share the same `USER-ID` field in their Riverbed policies. From Riverbed’s perspective, the sensitive data of each individual user would all belong to a single logical Riverbed user.

The `AGGREGATION` flag specifies whether a user’s data can be involved in server computations that include the data of other users. For example, suppose that a server wishes to add two numbers, each of which was derived from the data of a different user. If both users allow aggregation, Riverbed can execute the addition in the same universe. If one or both users disallow aggregation, then Riverbed must create separate universes for the two users. The `AGGREGATION` field specifies a yes/no policy—either arbitrary aggregation is allowed, or all aggregation is disallowed.

The binary `PERSISTENT-STORAGE` flag indicates whether server-side code can write a user’s data to persistent storage. If so, the user expects that when the data is read again by the server-side application, the application will treat the data as tainted. A Riverbed managed runtime terminates

applications that try to write tainted data to persistent storage, but lack the appropriate permissions.

A policy can optionally include an email address that belongs to the policy owner. If a Riverbed managed runtime must terminate policy-violating code, Riverbed can email the policy owner, informing the user about the thwarted policy breach. The user can then complain to the service operator, or take another corrective action.

The ALLOW-TO-NETWORK field is optional, and allows a user to whitelist network endpoints to which user data may flow. Endpoints are represented by hostnames; each whitelisted hostname is expected to have a valid X.509 certificate, e.g., as used by HTTPS. Before a Riverbed managed runtime allows tainted data to externalize via a socket, the runtime will check whether the remote endpoint is whitelisted by the tainted data’s policy. If so, the runtime forces the remote endpoint to attest its software stack. If that stack is whitelisted by the policy, the runtime allows the transfer to complete. Otherwise, the runtime terminates the application. Note that Riverbed allows untainted data to be sent to arbitrary remote servers.

The final item in a policy is typically one or more TRUSTED-SERVER-STACK entries. Each trusted stack is represented by a list of hash values; see Section 4.3 for more details about how these hash values are generated by servers, and later consumed by the attestation protocol.

As discussed in Sections 4.3 and 4.6, a client-side proxy leverages attestation to validate the server-side software stack up to, but not including, the application-defined managed code. Once the proxy determines that Riverbed’s taint-tracking managed runtime is executing on the server, the proxy will trust the runtime to enforce the policies described earlier in this section. However, the policies from earlier in this section only enable aggregation at a binary granularity (i.e., “allowed” or “disallowed”); a universe which disallows aggregation can never permit data to flow to a universe which *does* allow aggregation. This restriction prevents several useful types of selective aggregation. For example, two email servers in separate no-aggregate universes could ideally send emails to a trusted spam filter application which trains across all inboxes, and then returns a filter to each universe. To allow such aggregation by explicitly trusted components, Riverbed policies can decorate an ALLOW-TO-NETWORK field with a child policy. The child policy can override settings in the parent policy, allowing aggregation to occur at the endpoint. The child policy must specify a full-stack attestation record, to allow Riverbed to verify the identity of a *particular type* of trusted application-level code (e.g., SpamAssassin [2]). Data received from a trusted aggregator is marked with the taint descriptor of the receiving universe.

Riverbed allows a user to define her own policy for each web service that she uses. However, some policies may be fundamentally incompatible with certain services. As a trivial example, a Dropbox-like service that provides online storage is

intrinsically incompatible with a PERSISTENT-STORAGE: False policy.¹ In the common case, we expect users to rely on trusted outside authorities, called *policy generators*, to define reasonable policies for sites. For example, consider a web site that wants to deliver targeted advertising via a third-party ad network evil-ads.com. A consumer advocacy group can advise users to avoid policies that whitelist evil-ads.com. Consumer advocacy groups can also publish suggested policy files for particular sites, based on research about what reasonable permissions for those sites should be.

Note that modern web browsing is already influenced by a variety of curated policies. For example, Google maintains a set of known-malicious URLs; multiple browser types consult this list to prevent accidental user navigation to attacker-controlled pages [31]. As another example, ad blockers [18] interpose on a page load, blocking content from sites deemed objectionable by the creators of the ad blocker. Riverbed introduces a new kind of web policy, but does not shatter prior expectations that web browsing must be an unmediated experience.

4.3 Server Attestation

The client-side proxy shepherds the interactions between the client and server portions of a Riverbed application. In this section, we describe the proxy in the context of a traditional web service whose client/server protocol is HTTP. Proxies are easily written for other protocols like SMTP (§4.5). We assume that the reader understands the basics of remote attestation, but readers who lack this knowledge can refer to the appendix for the necessary background material.

A user configures her browser to use the Riverbed proxy to connect to the Internet. At start-up time, the proxy searches a well-known directory for the user’s policy files; the proxy assumes that each filename corresponds to the hostname in a server-side X.509 certificate (e.g., x.com). When the proxy receives an HTTP request that is destined for x.com, the proxy opens a TLS connection to x.com’s server, and forces that server to remotely attest its software stack. If the attestation succeeds, the proxy issues the HTTP request that triggered the attestation. Later, upon receiving a response from the server, the proxy forwards the response to the browser. By default, the proxy assumes that an attestation is valid for one day before a new attestation is necessary.

Riverbed strives to be practical, but traditional remote attestation [9, 15] has some unfortunate practical limitations. Consider the following challenges.

Server-side ambiguity: In traditional attestation, servers establish trust with clients by providing an explicit list of server-side software components. However, servers may not wish to share a perfectly-accurate view of their local software environment. For example, servers might be concerned that a

¹...unless the service is intentionally exporting a RAM-only storage abstraction.

malicious client will launch zero-day attacks against vulnerable (and precisely-identified) server components.

Potentially safe code: A server-side component may be intrinsically secure, but currently unvetted by the creator of a user’s Riverbed policies. Alternatively, a server-side executable might be intrinsically insecure, but perfectly safe to run if launched within a sandboxed environment like a virtual machine. Traditional attestation protocols are ill-suited to handle cases like these, since trust decisions are binary—a hash value in an attestation message corresponds to a categorically trusted component, or a categorically untrusted component.

Policy updating: A virtuous server administrator will be diligent about applying the latest patches to server-side code. If the user’s policy generator is not as diligent, then users will reject legitimately trustworthy stacks as suspicious. Similarly, if users are more aggressive about updating policies than a server administrator, then out-of-date server-side stacks will be legitimately rejected as untrustworthy, but the server administrator will lack an immediate explanation for why. Traditional attestation protocols focus on the cryptographic aspects of client-server communication, but cannot resolve these kinds of policy disputes.

Riverbed uses the Cobweb attestation system [74] to handle these practical concerns. In traditional attestation, the attestor sends a TPM-signed PCR [10] value, and a list of $\langle \text{filename}, \text{filehash} \rangle$ tuples representing the objects that are covered by the cumulative hash in PCR [10]. Cobweb allows the attestor to augment the traditional attestation report with a *contextual graph* that provides additional information about the attestor’s software stack. For example, a contextual graph might represent a process tree, where each vertex is a process and each edge represents a parent/child `fork()` relationship. An edge could also represent a dynamic information flow, e.g., indicating that two processes have communicated via IPC. Attestation verifiers specify policies as graph predicates that look for desired structural properties in the contextual graph or the regular attestation list of $\langle \text{filename}, \text{filehash} \rangle$ tuples.

Riverbed uses contextual graphs, and policy specification via graph predicates, to eliminate some of the practical difficulties with traditional attestation. For example:

- If attestation fails (i.e., if a client-side Riverbed proxy discovers that a graph predicate cannot be satisfied), the proxy sends the failed predicate to the server. The server can then initiate concrete remediating steps, e.g., by updating software packages, or removing a blacklisted application.
- A Riverbed server can also dispute the failure of a graph predicate. For example, if a user’s proxy believes that a particular server-side component is out-of-date, the server can respond with a list of signed, vendor-supplied updates for which the user’s proxy may be unaware. The proxy can then ask the user’s policy generator for a new policy.

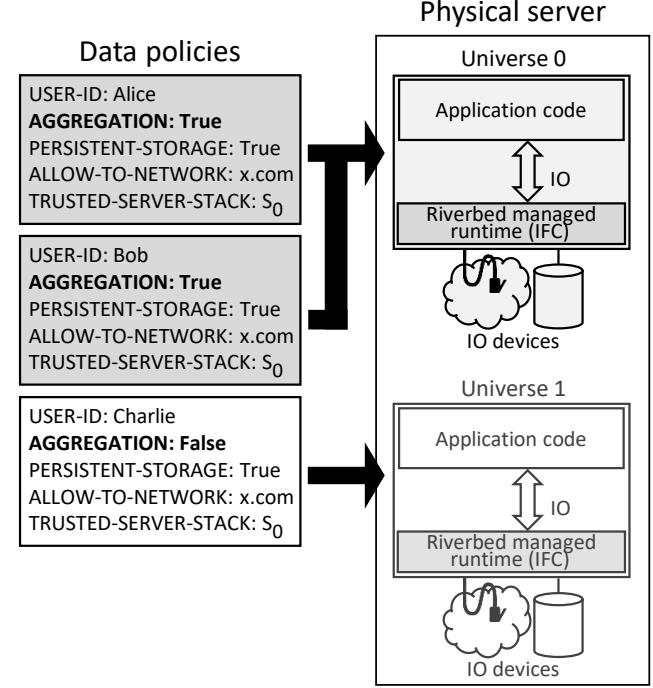


Figure 3: Alice and Bob have compatible policies, so Riverbed maps them to the same universe. Charlie has an incompatible policy because he disallows aggregation. Thus, Charlie must receive his own universe.

- A user’s Riverbed policy can tolerate an unknown or normally untrusted server binary if that binary is launched within a sandbox that isolates the component from other components which the user does require to be trusted. To provide confidence in the sandbox, the server’s contextual graph should contain the `fork()` / `exec()` history for the server, as well as the configuration files for the sandbox environment. As a concrete example, suppose that a server needs to run a `telnet` daemon to communicate with a legacy internal service. The `telnet` protocol is known to be insecure, but a Riverbed proxy can trust the server’s Apache instance if the server uses a virtual machine or a Docker container to isolate `telnetd`.

Riverbed also leverages Cobweb’s support for server-side software ambiguity, but we refer the reader to the Cobweb paper [74] for a discussion of how Cobweb implements this feature.

4.4 Universes

Consider Alice, Bob, and Charlie, three Riverbed users whose policies are shown in Figure 3. The policies of these users are *almost* the same—they differ only with respect to the AGGREGATION token. Alice and Bob allow aggregation, but Charlie does not. How should Riverbed handle the data of these users on the server-side?

Riverbed could optimistically assume that the server-side application code will never try to aggregate Charlie’s data with that of Alice or Bob. Riverbed executes the code atop a taint-tracking runtime (§4.5), so Riverbed could syn-

chronously detect attempted violations of Charlie’s policy. Unfortunately, attempted violations are likely, since Riverbed executes unmodified applications that are unaware of Riverbed policies. If a violation occurs, Riverbed would lack good options for moving forward. Riverbed could permanently terminate the application, which would prevent the disallowed aggregation of Charlie’s data. However, all three users would be locked out of the now-dead service. To avoid this outcome, Riverbed could try to synchronously clone the application at policy violation time, creating two different versions: one for Alice and Bob, and another for Charlie. However, determining which pieces of in-memory and on-disk state should belong in which clone is difficult without application-specific knowledge; a primary goal of Riverbed is to enforce security in a service-agnostic manner.

Riverbed’s solution emerges from the insight that Riverbed does not need to run any code to determine whether a set of policies might conflict. Instead, Riverbed can simply examine the policies themselves. For example, if a policy does not allow aggregation, then Riverbed can *preemptively* spawn a separate copy of the service for the policy’s owner. Riverbed can spawn this copy on-demand, upon receiving the first request from the owner. Now consider a policy P that allows aggregation and a particular set of storage and network permissions (e.g., PERSISTENT-STORAGE: True and ALLOW-TO-NETWORK: $x.com$). All users whose policies match P can be placed in the same copy of the service. Riverbed can spawn the copy upon receiving the first request that is tagged with P .

We call each service copy a *universe*. To implement the universe mechanism, Riverbed places a reverse proxy in front of the actual servers which run application code. Clients send their requests to the reverse proxy; the reverse proxy examines the policy in each request, spawns a new universe if necessary, and then forwards the request to the appropriate universe. Our Riverbed prototype instantiates each universe component inside of a Docker [21] instance that contains a taint-tracking runtime (§4.5) and the component-specific code and data. Docker containers are much smaller than traditional virtual machines since Docker virtualizes at the POSIX layer instead of the hardware layer. As a result, creating, destroying, and suspending universes is fast (§6.4). Docker runs each container atop a copy-on-write file system that belongs to the host [24]. Thus, universes share the storage that is associated with application code and other user-agnostic files.

Universes provide a final advantage: since all of the sensitive data in a universe has the same policy, a universe’s taint-tracking runtime only needs to associate a single logical bit of taint with each object (“tainted” or “untainted”). If data from all users resided in the same universe, the runtime would have to associate each object with a value that represented a specific taint pattern.

The relationship between the number of users and the number of universes is application-specific. Some web services

will specifically target a 1-1 mapping. For example, in a “private Dropbox” service that implements confidential online storage, users will naturally specify data policies that prevent aggregation (and thus require a universe per user). In contrast, social networking applications intrinsically derive their value from the sharing of raw user data, and the extraction of interesting cross-user patterns. For these applications, users must allow aggregation (although the scope of aggregation can be restricted using groups (§4.2)).

4.5 Taint Tracking

A managed language like Python, Go, or Java does not expose raw pointers to applications, or allow those applications to directly issue system calls. Instead, the language runtime acts as a mediation layer, controlling how a program interacts with the outside world. Like much of the prior work on dynamic tainting [26, 34, 46, 13], Riverbed enforces information flow control inside the managed runtime. Our Riverbed prototype modifies PyPy [53], a state-of-the-art Python interpreter, to extract Riverbed policies from incoming network data, and assign taint to derived information.

PyPy translates Python source files to bytecodes. Those bytecode are then interpreted. Riverbed adds taint-tracking instrumentation to the interpreter, injecting propagation rules that are similar to those of TaintDroid [26]. For example, in a binary operation like ADD, the lefthand side of the assignment receives the union of the taints of the righthand sides. Assigning a constant value to a variable clears the taint of the variable. If an array element is used as a righthand side, the lefthand side receives the taint of both the array and the index.

We call Riverbed’s modified Python runtime PyRB. If a Python application tries to send tainted data to remote host $x.com$, PyRB first checks whether externalization to $x.com$ is permitted by the tainted data’s policy. If so, PyRB forces $x.com$ to remotely attest its software stack; in this scenario, PyRB acts as the client in the protocol from Section 4.3. If $x.com$ ’s stack is trusted by the tainted data’s policy, PyRB allows the data to flow to $x.com$. Otherwise, PyRB terminates the application. Riverbed provides a standalone attestation daemon that a server can use to respond to attestation requests.

PyRB must also taint incoming network data that was sent by end-user clients like web browsers. To do so without requiring modifications to legacy application code, PyRB assumes two things. First, clients are assumed to use standard network protocols like HTTP or SMTP. Second, PyRB assumes that when clients send requests using those protocols, clients embed Riverbed policies in a known way. Ensuring the second property is easy if unmodified clients run atop Riverbed proxies; for example, when an unmodified web browser sends requests through a client-side Riverbed proxy, the proxy will automatically embed Riverbed policies using the Riverbed-policy HTTP header. Similarly, a client-

side SMTP proxy can attach Riverbed policies using a custom SMTP command.

On the server-side, PyRB assumes that traffic intended for well-known ports uses the associated well-known protocol. Upon receiving a connection to such a port, PyRB reads the initial bytes from the socket before passing those bytes to the application. If the initial bytes cannot be parsed as the expected protocol, PyRB forcibly terminates the connection. Otherwise, if PyRB finds a Riverbed policy, PyRB taints the socket bytes and then hands the tainted bytes to the higher-level application code. If there is no policy attached to the bytes, PyRB hands untainted bytes to the higher-level code. Importantly, the application code is unaware of the tainting process, and cannot read or write the taint labels.

If policies allow server-side code to write to persistent storage, PyRB taints the files that the application writes. PyRB does whole-file tainting, storing taint information in per-file extended attributes [40]. PyRB prevents application code from reading or writing those attributes. Whole-file tainting minimizes the storage overhead for taints, but Riverbed is compatible with taint-aware storage layers (§6.2) that perform fine-grained tainting, e.g., at the level of individual database rows; the use of such storage layers will minimize the likelihood of overtainting.

When an application reads data from a tainted file, PyRB taints the incoming bytes, preventing the application from laundering taint through the file system. Note that, even though a policy contains multiple constraints (§4.2), all of the users within a universe share the same policy; thus, PyRB only needs to associate a single logical bit with each Python object (§4.4). PyRB does need to store one copy of the full policy, so that the policy can be consulted when tainted data reaches an output sink.

Managed languages sometimes offer “escape hatches” that allow an application to directly interact with the unmanaged world. For example, in Java, the JNI mechanism [49] enables applications to invoke code written in native languages like C. In Python, interfaces like `os.system()` and `subprocess.call()` allow managed code to spawn native binaries. A Riverbed runtime can use one of three strategies to handle a particular escape hatch.

- The runtime can disallow the escape hatch by fiat.
- Alternatively, the runtime can whitelist the binaries that can be launched by the escape hatch. Each whitelisted binary must have a pre-generated taint model attached to it [26], such that the runtime can determine whether the binary is safe to launch given a particular set of tainted inputs, and if so, how taint should be assigned in the managed world when the binary terminates.
- The runtime can track instruction-level information flows in binaries launched by an escape hatch. To do so, the runtime must execute the native instructions via emulation [54, 76]. Strictly speaking, the runtime only needs to emulate instructions that touch sensitive data;

the runtime can use page table permissions to detect when native code tries to access tainted data [36, 55, 56]. This optimization allows most native code to execute unemulated, i.e., directly atop the hardware.

PyRB could use any or all of these strategies. Our current PyRB prototype uses the first two. PyRB disallows C bindings by fiat, and only allows applications to spawn a child process if that process will be an instance of the PyRB interpreter (with the Python code to run in the child process specified as an argument to the child process). The parent and child PyRB interpreters will introspect on cross-process file descriptor communication, encapsulating the raw bytes within a custom protocol which ensures that taint is correctly propagated between the two runtimes.

4.6 Discussion

The necessity of IFC: A Riverbed server attests its systems software and its Riverbed managed runtime. However, the server does not attest the contents of higher-level code belonging to the web service. At first glance, this approach might seem odd: why not have the server attest all application code as well? If clients trust the attested application code, then server-side IFC might be unnecessary. However, in many cases, application code is not open source, e.g., because the code contains proprietary intellectual property that confers a competitive advantage to the web service owner. Code like this cannot be audited by a trusted third party, so end-users would gain little confidence from remote attestations of that code. Even if the server-side code were open source and publicly auditable, there are many more server applications than OSes and low-level systems software. Given a finite amount of resources that can be devoted to auditing, those resources are best spent inspecting the lowest levels of the stack. Indeed, if those levels are not secure, then even audited higher-level code will be untrustworthy. Also note that, even if the web service code has been audited, Riverbed provides security in depth, by catching any disallowed information flows that the audit may have missed.

Universe migration: Due to server-side load balancing or fail-over, a container belonging to a universe can migrate across different physical servers. From a user’s perspective, migration is transparent if a user-facing container is placed on a server with a trusted stack—attestation involving the new server and the user’s Riverbed proxy will succeed as expected. However, before migration occurs, the old server must force the new server to attest; in this fashion, the old server ensures that the new server runs a trusted Riverbed stack (and will therefore respect the data policies associated with the universe being migrated).

Preventing denial-of-service via spurious universe creation: Attackers might generate a large number of fake users, each of which has a policy that requires a separate universe; the attacker’s goal would be to force the application to exhaust resources trying to manage all of the universes. Fortunately,

in a given Riverbed application, each universe employs copy-on-write storage layered atop a base image. As a result, a new universe consumes essentially zero storage resources until the universe starts receiving actual client requests that write to storage. Riverbed also suspends cold universes to disk. Thus, a maliciously-created universe that is cold will consume no CPU cycles and no RAM space; storage overhead will be proportional to the write volume generated by client requests, but this overhead is no different than in a non-Riverbed application. Regardless, a Riverbed application should perform the same user verification [32, 73, 75] that a traditional web service performs.

Hostname management: Applications which use a microservice architecture will contain many small pieces of code that are executed by a potentially large number of hostnames. An application that uses elastic scaling may also dynamically bind service state to a large set of hostnames. User policies can employ wildcarded TLS hostnames [30] to avoid the need for a priori knowledge of all possible hostnames.

Taint relabeling: Consider a user named Alice. A Riverbed service assigns Alice to a universe upon receiving the first request from Alice (§4.4). What happens if Alice later wants to re-taint her data, i.e., assign a different policy to that data?

Suppose that Alice lives in a singleton universe that only contains herself. Further suppose that her policy modification keeps her in a singleton universe. In this scenario, re-tainting data is straightforward. If storage permissions were enabled but now are not, Riverbed deletes Alice’s data on persistent storage. If network permissions changed, then Riverbed will only allow tainted data to flow to the new set of whitelisted endpoints. Nothing special must be done to handle tainted memory in the managed runtime—since Alice still lives in a singleton universe, there is no way for the service to combine her in-memory data with the data of others. If Alice later wants to invoke her “right to be forgotten,” Riverbed just destroys Alice’s universe.

The preceding discussion assumed that Alice only has universe state in a single TLS domain (e.g., `x.com`). However, Alice’s singleton universe will span multiple domains if Alice’s original policy enabled cross-domain data transfers. In these scenarios, Riverbed must disseminate a policy modification request to all relevant domains. Doing so is mostly straightforward, since the relevant domains are explicitly enumerated in Alice’s original policy. Riverbed does need to pay special attention to wildcarded network sinks like `*.x.com`; such domains must expose a directory service that allows Riverbed to enumerate the concrete hostnames that are covered by the wildcard.

Now consider a different user Bob who wants to change his policy. If Bob lives in a universe that is shared with others, then re-tainting is harder, regardless of whether Bob wishes to transfer to a shared universe or a singleton one. The challenges are the same ones faced by a synchronous universe clone at policy-violation time (§4.4): since Riverbed is application-

agnostic, Riverbed has no easy way to cleanly splice a user’s data out of one universe and into another. Thus, if Bob lives in a shared universe and wishes to move to a different one, Riverbed must first use application-specific mechanisms to extract his data from his current universe. Then, Riverbed deletes Bob’s current universe. Finally, Riverbed must re-inject Bob’s data into the appropriate universe via application-specific requests. This migration process may be tedious, but importantly, *Riverbed narrows the scope of data finding and extraction*. When re-tainting must occur, the application only needs to look for Bob’s data within Bob’s original universe, not the full set of application resources belonging to all users. Before and after re-tainting, Riverbed ensures that Bob’s IFC policies are respected.

CDNs: Large-scale web services use CDNs to host static objects that many users will need to fetch. CDN servers do not run application logic, but they do see user cookies which may contain sensitive information. So, by default, client-side Riverbed proxies force CDN nodes to attest. However, a proxy can explicitly whitelist CDN domains that should not be forced to attest.

Policy creep: Traditional end-user license agreements represent a crude form of data consent. In a EULA, a service provider employs natural language to describe how a service will handle user data; a user can then decide whether to opt into the service. Riverbed tries to empower users by giving *users* the ability to define policies for data manipulation. However, Riverbed cannot force a service to regard a user-defined policy as acceptable. Furthermore, the history of traditional EULAs suggests that, in a Riverbed world, services will prefer less restrictive Riverbed policies. For example, a service may refuse to accept a user if the user’s Riverbed policy will not allow data flows to a particular advertising network. In this situation, the service can mandate that a less restrictive policy is the cost of admission to the service. Riverbed cannot prevent such behavior. However, Riverbed does force services to be more transparent about data promiscuity, because any service-suggested policy must be explicit about how data will be used. Riverbed also uses IFC to force services to adhere to policies.

Deployment considerations: Riverbed assumes that datacenter machines have TPM hardware. This assumption is reasonable, since TPMs are already present in many commodity servers.

In a complex, multi-tier application, components may span multiple administrative domains. The failure of some domains to run up-to-date stacks may lead to cascading problems with the overall application, as trusted stacks refuse to share data with unpatched ones. This behavior is actually desirable from the security perspective, and it incentivizes domains to keep their software up-to-date.

5 IMPLEMENTATION

The core of our Riverbed prototype consists of a client-side proxy (§4.3), a server-side reverse proxy (§4.4), and a taint-tracking Python runtime (§4.5). The two proxies, which are written in Python, share parts of their code bases, and comprise 773 lines in total, not counting external libraries to handle HTTP traffic [57] and manipulate Docker instances [23]. PyRB is a derivative of the PyPy interpreter [53], and contains roughly 500 lines of new or modified source code.

To implement remote attestation, servers used LG’s UEFI firmware, which implemented the TPM 2.0 specification [70]. At boot time, the firmware extended a PCR with a TPM-aware version of the GRUB2 bootloader [17]. GRUB2 then extended the PCR with a TPM-aware version of the Linux 4.8 kernel. The kernel used Linux’s Integrity Management Architecture [44] to automatically extend the PCR when loading kernel modules or user-mode binaries. Contextual attestation graphs were generated by Cobweb [74], with servers and client-side Riverbed proxies using the Cobweb library to implement the attestation protocol.

6 EVALUATION

In this section, we demonstrate that Riverbed induces only modest performance penalties, allowing Riverbed to be a practical security framework for realistic applications. In all experiments, server code ran on an Amazon c4 instance which had a 4-core Intel Xeon E5-2666 processor and 16 GB of RAM. The client was a 3.1 GHz Intel Core i7 laptop with 16 GB of RAM. The network latency between the client and the server was 14 ms.

6.1 Attestation Overhead

Before a client-side Riverbed proxy will send data to a server, the proxy will force the server to attest. We evaluated attestation performance under a variety of emulated network latencies and bandwidths. The client’s policy required the attesting server to run a trusted version of `/sbin/init`, as well as trusted versions of 31 low-level system binaries like `/bin/sh`. The policy also used a Cobweb graph predicate (§4.3) to validate the process tree belonging to the Docker subsystem, ensuring that the tree contained no extraneous or missing processes.

Due to space restrictions, we only provide a summary of the results. Attestations were small (112 KB), so attestation time was largely governed by network latency, the cost of the slow TPM `quote()` operation (which took 215 ms on our server hardware), and Cobweb overheads for graph serialization, deserialization, and predicate matching (which required 562 ms of aggregate compute time on the server and the client-side proxy). On a client/server network link with a 14 ms RTT, the client-perceived time needed to fetch and validate an attestation was 846 ms. Proxies cache attestation results (§4.3), so this attestation penalty is amortized.

6.2 Case Studies

To study Riverbed’s post-attestation overheads, we ported three Python applications to Riverbed.

- MiniTwit [59] is a Twitter clone that implements core Twitter features like posting messages and following users. Application code runs in Flask [58], a popular server-side web framework. MiniTwit uses a SQLite database to store persistent information. We defined a Riverbed policy which allowed user data aggregation, and allowed tainted data to be written to storage and to other network servers in our MiniTwit deployment.
- Ionic Backup [51] is a Dropbox clone that provides a user with online storage. Ionic allows a user to upload, download, list, and delete files on the storage server. The Ionic client uses HTTP to communicate with the server. For this application, we defined a Riverbed policy which allowed user data to be written to disk, but disallowed aggregation, and prevented user data from being sent to other network servers.
- Thrifty P2P [43] implements a peer-to-peer distributed hash table [60, 68]. The primary client-facing operations are `PUT(key, value)` and `GET(key)`. Internally, Thrifty peers issue their own traffic to detect failed hosts, route puts and gets to the appropriate peers, and so on. For this application, we defined a Riverbed policy which allowed aggregation and storage, but only allowed tainted data to be written to endpoints that resided in our test deployment of Thrifty servers.

Ionic required no modifications to run atop Riverbed. Thrifty peers used a custom network protocol to communicate; so, we had to build a proxy for the Thrift RPC layer [3] that injected Riverbed policies into outgoing messages, and tainted incoming data appropriately. MiniTwit’s core application logic required no changes, but, to reduce the likelihood of over-tainting, we did modify MiniTwit’s Python-based database engine to be natively taint-aware, e.g., so that each database row had an associated on-disk taint bit, and so that query results were tagged with the appropriate union taints, based on the items that were read and written to satisfy the query. Our modifications are hidden beneath a narrow abstraction layer, making it easy to integrate the Python-level MiniTwit logic with off-the-shelf taint-tracking databases [63, 64, 77].

Figure 4 depicts end-to-end performance results for MiniTwit, Ionic, and Thrifty. The results demonstrate that Riverbed imposes small client-perceived overheads (1.01x–1.10x). Figure 5 isolates Riverbed’s server-side computational penalties. For each request type, we compare server-side performance when using unmodified PyPy, PyRB in which no data is tainted, or PyRB in which data is tainted according to the policies that we described earlier in this section. For MiniTwit, Riverbed had overheads of 1.02x–1.15x. For Ionic, Riverbed imposed overheads of 1.04x–1.16x. For Thrifty, puts and gets had slowdowns of 1.18x and 1.26x respectively. Riverbed imposed the least overhead for Ionic’s “remove” and

Operation	Without Riverbed	With Riverbed
MiniTwit view timeline	229 ms	252 ms
Ionic download	82.5 ms	83.1 ms
Ionic ls	14.1 ms	14.2 ms
Thrift GET request	27.5 ms	28.0 ms

Figure 4: End-to-end response times for processing various user requests. For MiniTwit, the user viewed her timeline. For Ionic, the user downloaded a 300 KB file, or asked for a list of the contents of a server-side directory. For Thrifty, the client fetched a 20 byte value from a DHT that contained 2 nodes; the DHT was intentionally kept small to emphasize the computational overheads of Riverbed. The client/server network latency was 14 ms. Each result is the average of 50 trials.

Operation	Regular PyPy	PyRB (no taint)	PyRB (taint)
MiniTwit post message	14 ms	15 ms	15 ms
MiniTwit view timeline	4.1 ms	4.2 ms	4.2 ms
MiniTwit follow user	13 ms	15 ms	15 ms
Ionic upload	2.3 ms	2.5 ms	2.5 ms
Ionic download	4.8 ms	5.0 ms	5.0 ms
Ionic ls	0.43 ms	0.50 ms	0.50 ms
Thrift PUT request	0.16 ms	0.17 ms	0.19 ms
Thrift GET request	0.19 ms	0.24 ms	0.24 ms

Figure 5: Server-side overheads for processing various user requests. The workloads are a superset of the ones in Figure 4. Each result is the average of 50 trials.

“delete” operations, since PyRB could handle these operations merely by issuing file system calls, without handling much in-memory data that had to be checked for taint. In contrast, operations that involved reading or writing network data required PyRB to interpose on data processing code, even if no data was tainted, and perform extra work at data sources and sinks.

6.3 PyPy Benchmarks

For a wider perspective on PyRB’s performance, we used PyRB to run the benchmarking suite from the Performance project [52]. The suite focuses on real Python applications, downloading the necessary packages for those applications and then running the real application code. Figure 6 shows PyRB’s performance on a representative set of benchmarks. The benchmarks that are above the thin black line resemble applications that might run inside of a Riverbed universe; these benchmarks perform actions that are common to web services, like parsing HTML, responding to HTTP requests, and performing database queries. These benchmarks tend to be IO-heavy, with occasional CPU idling as code waits for IOs to complete. In contrast, the benchmarks beneath the thin black line are CPU-intensive. PyRB does not affect the speed of IOs, but does affect the speed of computation, so PyRB has slightly higher overhead for the bottom set of benchmarks. Overall, PyRB is at most 1.19x slower. These results overestimate PyRB’s overheads because clients and

Benchmark	Overhead
Django	1.14x
Render HTML table	1.16x
Code run in PyPy interpreter	1.08x
JSON parsing	1.13x
Python git operations	1.01x
SQL Alchemy	1.05x
Spitfire	1.19x
Twisted	1.17x
Fractal Generation	1.18x
Spectral Norm	1.10x
Raytracing	1.19x

Figure 6: PyRB’s performance on representative benchmarks from the Performance benchmark suite [52]. PyRB’s performance is normalized with respect to that of regular PyPy. No data was tainted in these experiments.

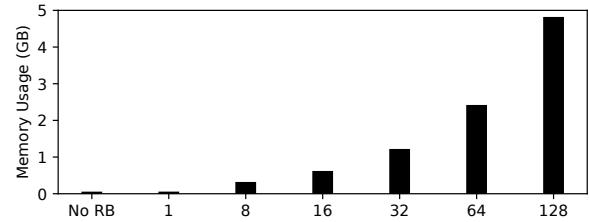


Figure 7: Physical memory pressure in MiniTwit when run without Riverbed, or with Riverbed using various numbers of universes. Note that in MiniTwit, each universe requires only one container. In each test configuration, we measured memory pressure after submitting 1000 requests to each MiniTwit instance that existed in the configuration.

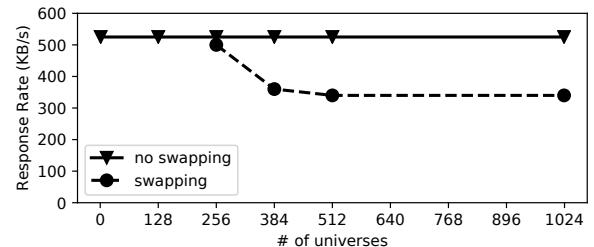


Figure 8: MiniTwit server response rate as a function of (1) the number of universes, and (2) whether the server had 60 GB of RAM or 16 GB of RAM. We used the Apache Benchmark tool [1] to simulate clients that requested MiniTwit timelines which had 100 messages. In each trial, we submitted 1000 requests, with 100 outstanding requests at any given time. For the server with 16 GB of RAM, swapping began with 256 universes.

servers resided on the same machine (and thus incurred zero network latency).

6.4 Universe Overhead

The size for a base Riverbed Docker image is 212 MB. The image contains the state that belongs to the PyRB runtime, and is similar in size to the official PyPy Docker image [22]. Each Riverbed service adds application-specific code and

data to the base Riverbed image. However, a live Docker instance uses copy-on-write storage, so multiple Riverbed universes share disk space (and in-memory page cache space) for common data.

We believe that for most Riverbed applications, the universe abstraction will not increase overall storage requirements; in other words, the space needed for per-universe data plus shared-universe data will be similar to the space needed for the non-Riverbed version of the application. For example, in MiniTwit, for a given number of timelines with a given amount of posts, the storage requirements are the same if the timelines are partitioned across multiple Riverbed universes, or kept inside a single, regular MiniTwit deployment. However, Docker’s copy-on-write file system does result in slower disk IOs. As a concrete example, we measured MiniTwit’s database throughput when MiniTwit ran directly atop ext4, and when MiniTwit ran inside a universe that used Docker’s overlayfs file system [24]. We examined database workloads with read/write ratios of 95/5 and 50/50, akin to the YCSB workloads A and B [16]. The targeted database rows were drawn from a Zipf distribution with $\beta = 0.53$, similar to the distribution observed in real-life web services [4, 72]. We found that, inside a Riverbed universe, transaction throughput slowed by 7.7% for the 95/5 workload, and by 17.3% for the 50/50 workload.

For our three sample applications, spawning a new Docker container required 260–280 ms on our test server. In Riverbed, the container creation penalty is rarely paid; the reverse proxy only has to create a new universe upon seeing a request with a policy that is incompatible with all pre-existing universes. Subsequent requests which are tagged with that policy will be routed to the pre-existing universe.

Creating new universes is rare, but pausing and unpausing old ones may not be. If an application has many universes, and memory pressure on a particular physical server is high, then temporarily-quiescent universes can be suspended to disk. On our test server with 512 live containers, pausing or unpauseing a single Docker instance took roughly 30 ms. However, recent empirical research has shown that in datacenters, a tenant’s resource requirements are often predictable [19]. Thus, universes can be assigned to physical servers in ways that reduce suspension/resumption costs.

Docker virtualizes at the POSIX level, so the processes inside of a Riverbed universe are just processes inside of the host OS. As a result, the RAM footprint for a Riverbed universe is just the memory that is associated with the host processes for the universe. Our Riverbed prototype was able to spawn up to 1023 live containers on a single server. This 1023 bound is a well-known limitation of the current Docker implementation. Docker associates a virtual network card with each instance, and attaches the virtual card to a Linux network bridge [69]; a Linux bridge can only accept 1023 interfaces. Regardless, the current bound of 1023 containers per machine does not imply that a single application can

have at most 1023 universes. The bound just means that, if an application has more than 1023 universes, then those universes must be spread across multiple servers. Riverbed’s reverse proxy (§4.4) considers server load when determining where to create or resurrect a universe; thus, the per-server container limit is not a concern in practice.

Figure 7 demonstrates that Riverbed’s memory pressure is linear in the number of active containers. As shown in Figure 8, a large number of universes has no impact on server throughput if all of the hot universes fit in memory. Unsurprisingly, throughput drops if active universes must be swapped between RAM and disk. However, a Docker container is just a set of Linux processes that are constrained using namespaces [38] and cgroups [12]; thus, the memory overhead for launching a Riverbed universe with N processes is similar to the memory overhead of scaling out a regular application by creating N regular processes. That being said, a Riverbed application does create processes more aggressively than a normal application. In Riverbed, incompatible policies require separate universes (and therefore separate processes), even if aggregate load across all universes is low.

7 CONCLUSION

Riverbed is a platform that simplifies the creation of web services that respect user-defined privacy policies. A Riverbed universe allows a web service to isolate the data that belongs to users with the same privacy policy; Riverbed’s taint tracking ensures that the data cannot flow to disallowed sinks. Riverbed’s client-side proxy will not divulge sensitive user data until servers have attested their trustworthiness. Riverbed is compatible with commodity managed languages, and does not force developers to annotate their source code or reason about security lattices. Experiments with real applications demonstrate that Riverbed imposes no more than a 10% performance degradation, while giving both users and developers more confidence that sensitive data is being handled correctly.

REFERENCES

- [1] Apache Software Foundation. Apache Benchmark. <https://httpd.apache.org/docs/2.4/programs/ab.html>.
- [2] Apache Software Foundation. Apache SpamAssassin: Open-source Spam Filter. <http://spamassassin.apache.org/>.
- [3] Apache Software Foundation. Apache Thrift. <https://thrift.apache.org/>.
- [4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of SIGMETRICS*, pages 53–64, 2012.
- [5] T. Austin and C. Flanagan. Efficient Purely-dynamic Information Flow Analysis. *ACM SIGPLAN Notices*, 44(8):20–31, 2009.

- [6] T. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of PLAS*, 2010.
- [7] T. Austin, J. Yang, and C. F. A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 15–26, 2013.
- [8] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of OSDI*, pages 267–283, 2014.
- [9] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of USENIX Security*, pages 305–320, 2006.
- [10] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit’s JavaScript Bytecode. In *International Conference on Principles of Security and Trust*, pages 159–178, 2014.
- [11] A. Booth. Charities Hit with Fines for Sharing Donors’ Data Without Consent, December 7, 2016. Sophos Naked Security Blog. <https://nakedsecurity.sophos.com/2016/12/07/charities-hit-with-fines-for-sharing-donors-data-without-consent/>.
- [12] N. Brown. Control groups, July 7, 2014. LWN. <https://lwn.net/Articles/604609/>.
- [13] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proceedings of the Computer Security Applications Conference*, pages 463–475, 2007.
- [14] S. Chong, K. Vikram, and A. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of USENIX Security*, pages 1–16, 2007.
- [15] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O’Hanlon, J. Ramsdell, A. Segall, and B. Sniffen. Principles of Remote Attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.
- [16] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SOCC*, pages 143–154, 2010.
- [17] CoreOS. GRand Unified Bootloader 2.0. <https://github.com/coreos/grub>.
- [18] J. Corpuz. Best Ad Blockers and Privacy Extensions: Chrome, Safari, Firefox, and IE. Tom’s Guide. <https://www.tomsguide.com/us/pictures-story/565-best-adblockers-privacy-extensions.html>.
- [19] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of SOSP*, pages 153–167, 2017.
- [20] V. Costan and S. Devadas. Intel SGX Explained, February 20, 2017. Cryptology ePrint Archive: Version 20170221:054353. <https://eprint.iacr.org/2016/086.pdf>.
- [21] Docker. Docker Home Page. <https://docker.com>.
- [22] Docker. Docker PyPy Images. https://hub.docker.com/_/pypy/.
- [23] Docker. Docker SDK for Python. <https://docker-py.readthedocs.io/en/stable/>.
- [24] Docker Docs. Using the OverlayFS storage driver. <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>.
- [25] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of USENIX Security*, pages 637–654, 2016.
- [26] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Real-time Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2), 2014.
- [27] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Towards Practical Taint Tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010.
- [28] EU Parliament. GDPR Portal, 2017. <http://www.eugdpr.org/eugdpr.org.html>.
- [29] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of OSDI*, pages 47–60, 2012.
- [30] GoDaddy. What is a Wildcard SSL certificate? <https://www.godaddy.com/help/what-is-a-wildcard-ssl-certificate-567>.
- [31] Google. What is Safe Browsing? <https://developers.google.com/safe-browsing/>.
- [32] S. Gurajala, J. White, B. Hudson, and J. Matthews. Fake Twitter Accounts: Profile Characteristics Obtained Using an Activity-based Pattern Detection Approach. In *Proceedings of the International Conference on Social Media and Society*, 2015.
- [33] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of OSDI*, pages 165–181, 2014.
- [34] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JS-Flow: Tracking Information Flow in JavaScript and its APIs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1663–1671, 2014.
- [35] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Proceedings of the Marktoberdorf Summer School*, August 2011.
- [36] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection Using Demand Emulation. In *Proceedings of EuroSys*, pages 29–41, April 2006.
- [37] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of OSDI*, pages 533–549,

- 2016.
- [38] M. Kerrisk. Namespaces in Operation, Part 1: Namespaces Overview, January 4, 2013. LWN. <https://lwn.net/Articles/531114/>.
- [39] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of SOSP*, 2007.
- [40] J. Layton. Extended File Attribute Rock!, June 29, 2011. <http://www.linux-mag.com/id/8741/>.
- [41] K. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370, 2010.
- [42] P. Li, Y. Mao, and S. Zdancewic. Information Integrity Policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*, September 2003.
- [43] A. Lindsay. Thrifty P2P. <https://github.com/atl/thrifty-p2p>.
- [44] Linux. Integrity Measurement Architecture. <https://sourceforge.net/p/linux-ima/wiki/Home/>.
- [45] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, and A. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of SOSP*, pages 321–334, October 2009.
- [46] B. Livshits. Dynamic taint tracking in managed runtimes. *Technical Report MSR-TR-2012-114, Microsoft*, 2012.
- [47] Lyon Brothers Enterprises. Turtl: Find Your Private Space. <https://turtlapp.com/>.
- [48] A. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
- [49] Oracle Corporation. Java Native Interface. <http://docs.oracle.com/javase/8/docs/technotes/guides/jni/>.
- [50] T. Pasquier, J. Singh, J. Bacon, and D. Eyers. Information Flow Audit for PaaS Clouds. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 42–51, 2016.
- [51] F. Primerano. Ionic Backup. <https://github.com/Max00355/IonicBackup>.
- [52] PyPy. PyPy Benchmarks. <https://bitbucket.org/pypy/benchmarks>.
- [53] PyPy. PyPy Home Page. <https://pypy.org/>.
- [54] C. Qian, X. Luo, Y. Shao, and A. Chan. On Tracking Information Flows Through JNI in Android Applications. In *Proceedings of DSN*, pages 180–191, June 2014.
- [55] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of MICRO*, pages 135–148, December 2006.
- [56] A. Razeen, A. Lebeck, D. Liu, A. Meijer, V. Pistol, and L. Cox. SandTrap: Tracking Information Flows On Demand with Parallel Permissions. In *Proceedings of MobiSys*, June 2018.
- [57] K. Reitz. Requests: HTTP for Humans. <http://docs.python-requests.org/en/master/>.
- [58] A. Ronacher. Flask. <http://flask.pocoo.org/>.
- [59] A. Ronacher. Minitwit. <https://github.com/pallets/flask/blob/master/examples/minitwit/>.
- [60] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350, 2001.
- [61] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of CSF*, pages 186–199, 2010.
- [62] P. Sayer. German Consumer Groups Sue WhatsApp Over Privacy Policy Changes, January 30, 2017. PCWorld. <http://www.pcworld.com/article/3163027/private-cloud/german-consumer-groups-sue-whatsapp-over-privacy-policy-changes.html>.
- [63] D. Schoepe, D. Hedin, and A. Sabelfeld. SeLINQ: Tracking Information Across Application-Database Boundaries. In *ACM SIGPLAN Notices*, volume 49, pages 25–38, 2014.
- [64] D. Schultz and B. Liskov. IFDB: Decentralized Information Flow Control for Databases. In *Proceedings of EuroSys*, pages 43–56, 2013.
- [65] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, 2010.
- [66] S. Sen, S. Guha, A. Datta, S. Rajamani, J. Tsai, and J. Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 327–342, 2014.
- [67] A. Slowinska and H. Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of EuroSys*, pages 61–74, 2009.
- [68] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.
- [69] The Linux Foundation. Linux Network Bridge. <https://wiki.linuxfoundation.org/networking/bridge>.
- [70] Trusted Computing Group. TPM 2.0 Library Specification. <https://trustedcomputinggroup.org/tpm-library-specification/>.
- [71] Trusted Computing Group. Trusted Platform Module (TPM) Summary. <https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/>.
- [72] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *International Journal of Computer and Telecommunications*

- Networking*, 53(11):1830–1845, 2009.
- [73] E. van der Walt and J. Ellof. Using Machine Learning to Detect Fake Identities: Bots vs Humans. *IEEE Access*, 6:6540–6549, January 2018.
- [74] F. Wang, Y. Joung, and J. Mickens. Cobweb: Practical Remote Attestation Using Contextual Graphs. In *Proceedings of SysTEX*, 2017.
- [75] C. Xiao, D. Freeman, and T. Hwa. Detecting Clusters of Fake Accounts in Online Social Networks. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, pages 91–101, 2015.
- [76] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *Proceedings of USENIX Security*, pages 289–306, August 2017.
- [77] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of PLDI*, pages 631–647, 2016.
- [78] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *ACM SIGPLAN Notices*, volume 47, pages 85–96, 2012.
- [79] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *Proceedings of SOSP*, pages 291–304, 2009.
- [80] A. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of SOSP*, pages 1–14, 2001.
- [81] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing Distributed Systems with Information Flow Control. In *Proceedings of NSDI*, pages 293–308, 2008.
- [82] K. Zetter. Hackers Finally Post Stolen Ashley Madison Data, August 18, 2015. *Wired*. <https://www.wired.com/2015/08/happened-hackers-posted-stolen-ashley-madison-data/>.

APPENDIX: OVERVIEW OF ATTESTATION

In this section, we give a slightly simplified description of the classic attestation protocol. We explain how an *attestor* (i.e., a potentially untrustworthy machine) securely describes its software stack to a remote *verifier* machine. For more details, we refer the interested reader to other work [15, 20, 9].

Setup: The attestor’s trusted hardware (called a TPM chip [71]) possesses a unique public/private key pair that is burned into the hardware. The private key is never exposed to the rest of the machine. The attestor also has a certificate, signed by the manufacturer of the TPM, that binds the attestor to its public key. Thus, the hardware manufacturer acts as a certificate authority (CA). Before the remote attestation protocol begins, the verifier must download the public key of the CA.

A TPM contains a small number of platform configuration registers (PCRs). Each PCR is made of tamper-resistant, non-volatile RAM that only the TPM can access.

At boot time, the TPM resets each PCR to a well-known value. The TPM’s `extend(index, value)` is the only way that entities external to the TPM can update a PCR. An extension sets $\text{PCR}[\text{index}] = \text{SHA1}(\text{PCR}[\text{index}] \parallel \text{value})$. During the boot process, the BIOS automatically extends $\text{PCR}[10]$ with a `value` equal to the SHA1 hash of the BIOS code. The BIOS then reads the bootloader from the disk, extends $\text{PCR}[10]$ with the hash of the bootloader, and jumps to the first instruction of the bootloader. The bootloader reads the kernel binary into RAM, extends $\text{PCR}[10]$ with the hash of the kernel image, and then jumps to the first instruction of the kernel. These PCR extensions continue as the OS loads additional kernel modules and user-level system binaries. Thus, the attestor’s $\text{PCR}[10]$ register will contain a cumulative hash of the local software stack.

Remote attestation: The verifier generates a random nonce and sends it to the attestor. The attestor asks its local TPM to generate a signature over the nonce and the value of $\text{PCR}[10]$; this signature, which is called a “quote” in TPM parlance, uses the attestor’s unique private key (whose corresponding public key is validated by a certificate from the CA). The attestor returns the following information to the verifier:

- the attestor’s certificate,
- the quote,
- the value of $\text{PCR}[10]$ that is attested by the quote,
- a list of the SHA1 hashes that were used to extend $\text{PCR}[10]$, and
- optionally, a mapping from each hash to the server-side file name representing the content that was hashed.

The verifier checks the validity of the attestor’s public key using the certificate. The verifier then checks the validity of the quote signature, and confirms that cumulatively extending $\text{PCR}[10]$ with the attestor-reported hash list results in the attestor-reported $\text{PCR}[10]$ value. If these checks succeed, the verifier sees whether the hash list corresponds to a trusted ordering of trusted system components. If so, the remote attestation succeeds.

Attesting VMs: The traditional attestation protocol can be extended to cover the software stack inside of a VM [9]. During the initial boot sequence of a physical server, the physical $\text{PCR}[10]$ will be extended with the bootloader code, the hypervisor binary, and other low-level software. The hypervisor will then extend $\text{PCR}[10]$ using content associated with the virtual TPM manager; this content includes the binary of the manager itself, as well as certificates needed to vouch for the signatures produced by a VM’s virtual TPM. When a VM launches, the manager initializes the VM’s virtual PCRs using the values in the physical PCRs. The VM then boots, extending (virtual) PCRs as usual. In this manner, the attestation produced by a virtual TPM will be linked to a non-virtualized root of trust (i.e., the physical TPM of the server).

Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs

Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park*
Geoff Langdale[†], Jiayu Hu, and Heqing Zhu

Intel Corporation *KAIST [†]*branchfree.org*

Abstract

Regular expression matching serves as a key functionality of modern network security applications. Unfortunately, it often becomes the performance bottleneck as it involves compute-intensive scan of every byte of packet payload. With trends towards increasing network bandwidth and a large ruleset of complex patterns, the performance requirement gets ever more demanding.

In this paper, we present Hyperscan, a high performance regular expression matcher for commodity server machines. Hyperscan employs two core techniques for efficient pattern matching. First, it exploits graph decomposition that translates regular expression matching into a series of string and finite automata matching. Unlike existing solutions, string matching becomes a part of regular expression matching, eliminating duplicate operations. Decomposed regular expression components also increase the chance of fast DFA matching as they tend to be smaller than the original pattern. Second, Hyperscan accelerates both string and finite automata matching using SIMD operations, which brings substantial throughput improvement. Our evaluation shows that Hyperscan improves the performance of Snort by a factor of 8.7 for a real traffic trace.

1 Introduction

Deep packet inspection (DPI) provides the fundamental functionality for many middlebox applications that deal with L7 protocols, such as intrusion detection systems (IDS) [9, 10, 28], application identification systems [4], and web application firewalls (WAFs) [3]. Today’s DPI employs regular expression (regex) as a standard tool for pattern description as it flexibly represents various attack signatures in a concise form. Not surprisingly, numerous research works [16, 18, 32, 38, 39, 41, 42] have proposed efficient regex matching as its performance often dominates that of an entire DPI application.

Despite continued efforts, the performance of regex matching on a commodity server still remains impractical to directly serve today’s large network bandwidth. Instead, the de-facto best practice of high-performance DPI generally employs multi-string pattern matching as a pre-condition for expensive regex matching. This hybrid approach (or prefiltering) is attractive as multi-string matching is known to outperform multi-regex matching by two orders of magnitude [21], and most input traffic is innocent, making it more efficient to defer a rigorous check. For example, popular IDSes like Snort [9] and Suricata [10] specify a string pattern per each regex for prefiltering, and launch the corresponding regex matching only if the string is found in the input stream.

However, the current prefilter-based matching has a number of limitations. First, string keywords are often defined manually by humans¹. Manual choice does not scale as the ruleset expands over time, and improper keywords would waste CPU cycles on redundant regex matching. Second, string matching and regex matching are executed as two *separate* tasks, with the former leveraged only as a trigger for the latter. This results in duplicate matching of the string keywords when the corresponding regex matching is executed. Third, current regex matching typically translates an entire regex into a single finite automaton (FA). If the number of deterministic finite automaton (DFA) states becomes too large, one must resort to a slower non-deterministic finite automaton (NFA) for matching of the whole regex.

In this paper, we present Hyperscan, a high performance regex matching system that exploits regex decomposition as the first principle. Regex decomposition splits a regex pattern into a series of disjoint string and FA components². This translates regex matching into a sequence

¹The content option in Snort and Suricata are determined by humans with domain knowledge.

²We refer to a subregex that contains regex meta-characters or quantifiers, which has to be translated into either a DFA or an NFA for

of decomposed subregex matching whose execution and matching order is controlled by fast string matching. This design brings a number of benefits. First, our regex decomposition identifies string components automatically by performing rigorous structural analyses on the NFA graph of a regex. Our algorithm ensures that the extracted strings are pre-requisite for the rest of regex matching. Second, string matching is run as a part of regex matching rather than being employed only as a trigger. Unlike the prefilter-based design, Hyperscan keeps track of the state of string matching throughout regex matching and avoids any redundant operations. Third, FA component matching is executed only when all relevant string and FA components are matched. This eliminates unnecessary FA component matching, which allows efficient CPU utilization. Finally, most decomposed FA components tend to be small, so they are more likely to be able to be converted to a DFA and benefit from fast DFA matching.

Beyond the benefits of regex decomposition, Hyperscan also brings a significant performance boost with single-instruction-multiple-data (SIMD)-accelerated pattern matching algorithms. For string matching, we extend the shift-or algorithm [13] to support efficient multi-string matching with bit-level SIMD operations. For FA matching, we represent a state with a bit position while we implement state transitions and successor state-set calculation with SIMD instructions on a large bitmap. We find that our SIMD-accelerated string matching outperforms state-of-the-art multi-string matching by a factor of 1.3 to 2.5. We also find that our SIMD-accelerated regex matching achieves 24.8x to 40.1x performance improvement over PCRE [6] widely adopted by DPI middleboxes such as Snort and Suricata.

In summary, we make the following contributions:

- We present a novel regex matching strategy that exploits regex decomposition. Regex decomposition performs rigorous graph analysis algorithms that extract key strings of a regex for efficient matching, and drives the order of pattern matching by fast string matching. This drastically improves the performance.
- We develop SIMD-accelerated pattern matching algorithms for both string matching and FA matching to leverage CPU's compute capability on data parallelism.
- Our evaluation shows Hyperscan greatly helps improve the performance of real-world DPI applications. It improves the performance of Snort by 8.7x for a real traffic trace.
- We share our experience with developing Hyperscan and present lessons learned through commercialization.

matching as an FA component.

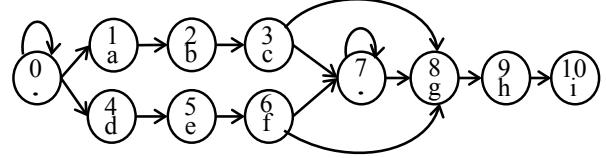


Figure 1: Glushkov NFA for $(abc|def)^*ghi$

2 Background and Motivation

DPI is a common functionality in many security middleboxes, and its performance has been mainly driven by that of regex matching [19, 41]. There has been a large body of research that improves the performance of regex matching. Due to space constraint, we briefly review only a few, categorizing them by their approach.

String matching is a subset of regex matching, which requires specialized algorithms [12, 24, 29] to achieve high performance. The most popular one is the Aho-Corasick (AC) algorithm [12] that uses a variant of DFA for fast multi-string matching. It runs in $O(n)$ time complexity where n is the number of input bytes. Unfortunately, AC suffers from frequent cache misses due to large memory footprint and random memory access pattern, which significantly impairs the performance. In addition, the model of processing one byte at a time creates a sequential data dependency that stalls instruction pipelines of modern processors. DFC [21] employs a set of small bitmap filters that quickly pass out innocent traffic by checking the first few bytes of string patterns against the input stream. Each matched input moves onto the verification stage for full pattern comparison. DFC substantially reduces memory accesses and cache misses by using small and cache-friendly data structures, which outperforms AC by 2 to 3.6 times. The string matcher of Hyperscan takes the two-stage matching similar to DFC, but its bucket-based shift-or algorithm benefits from SIMD instructions, which further improves the performance beyond that of DFC.

An NFA implements a space-efficient state machine even for complex regexes. Despite its small memory footprint, the execution is typically slow as each input character triggers $O(m)$ memory lookups ($m = \#$ of current states). For this reason, a DFA is preferred to an NFA whenever a regex can be translated into the former. One place where NFA might be preferred is a logic-based design that maps automata to hardware accelerators such as FPGA [14, 22, 23, 34, 35, 40]. An FPGA-based design can exploit parallelism by running multiple finite automata simultaneously and does not suffer from sequential state transition table lookups. On the down side, it is limited to a small ruleset due to its hardware constraints. Also, it

suffers from the DMA overhead of moving data between the CPU and the FPGA device. This overhead can impose prohibitive latency, especially when input data is not large (as would be the case for scanning of small packets).

We use Glushkov NFA [27] for Hyperscan, which is widely used due to its two useful properties. First, it does not have epsilon transitions, which simplifies state transition implementation. Second, all transitions into a given state are triggered by the same input symbol. Figure 1 shows an example of a Glushkov NFA. Each circle represents a state whose id is shown as a number, and each character represents the input symbol by which any previous state transitions into this state. For example, one can transition into state 8 from state 3, 6, or 7 only for an input symbol, ‘g’. The second property implies that the total number of states is bounded by the number of input symbols and a few special states – a start state, states with ‘.’, etc.

A **DFA** achieves high performance as it runs in $O(1)$ per each input character. Its main disadvantage, however, is a large memory footprint and a potential of state explosion at transforming an NFA to a DFA. Thus, most works on DFA focus on memory footprint reduction [15, 17, 18, 20, 26, 31, 32, 33]. D²FA [32] compresses the memory space by sharing multiple transitions of states with a similar transition table and by establishing a default transition between them. A-DFA [18] presents useful features such as alphabet reduction that classifies alphabets into smaller groups, indirect addressing that reduces memory bandwidth by translating unique state identifiers to memory address, and multi-stride structure that processes multiple characters at a time.

An **extended FA** is a proposal that restructures the state-of-the-art FA to address state explosion. XFA [38, 39] associates update functions with states and transitions by having a scratch memory that compresses the space. HFA [16] presents a hybrid-FA that achieves comparable space to that of an NFA by making head DFAs and trailing NFA or DFAs. The theory behind it is to discover boundary states so that one can conduct partial conversion of an NFA to a DFA to avoid exponential state explosion from a full conversion.

Prefilter-based approaches are the most popular way to scale performance of regex matching in practice. Both Snort and Suricata extract string keywords from regex rules and perform unified multi-string matching with the Aho-Corasick algorithm. Expensive regex matching is only needed if AC detects literal strings in the input. SplitScreen [36] applies a similar approach to ClamAV [30], a widely-used anti-malware application, and achieves a 2x speedup compared to original ClamAV.

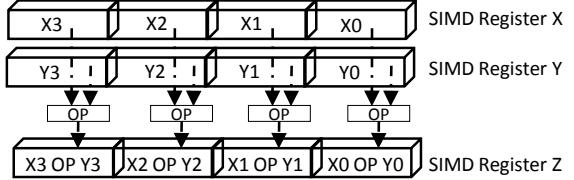


Figure 2: Typical two-operand SIMD operation

A **SIMD** instruction executes the same operation on multiple data in parallel. As shown in Figure 2, a SIMD operation is performed on multiple lanes of two SIMD registers independently, and the results are stored in the third register. Modern CPU supports a number of SIMD instructions that can work on specialized vector registers (SSE, AVX, etc.). The latest AVX512 instructions support up to 512-bit operations simultaneously.

Despite its great potential, few research works have exploited SIMD instructions for regex matching. Sitaridi et al. propose a SIMD-based regex matching design [37] for database, which uses a gather instruction to traverse DFA for multiple inputs simultaneously. However, it cannot be applied to our case as regex matching for DPI is typically performed on a single input stream.

Summary and our approach. Most prior works on regex matching attempt to build a special FA that performs as well as a DFA while its memory footprint is as small as an NFA. However, one common problem in all these works is that FA restructuring inevitably imposes extra performance overhead compared to the original DFA. For example, XFA takes multiple tens to hundreds of CPU cycles per input byte, which is slower than a normal DFA by one or two orders of magnitude. In contrast, the prefilter-based approach looks attractive as it benefits from multi-string matching most time, which is faster than multi-regex matching by a few orders of magnitude. However, it is still suboptimal as it must perform duplicate string matching during regex matching, and wrong choice of string patterns would trigger redundant regex matching (as shown in Section 6.2). To avoid the inefficiency, we take a fresh approach that divides a regex pattern into multiple components, and leverages fast string matching to coordinate the order of component matching. This would minimize the waste of CPU cycles on redundant matching and thus improves the performance. In addition, we develop our own multi-string matching and FA matching algorithms carefully tailored to exploit SIMD operations.

3 Regular Expression Decomposition

In this section, we present the concept of regex decomposition, and explain how Hyperscan matches a sequence of regex components against the input. Then, we introduce

graph-based decomposition whose graph analysis techniques reliably identify the strings in regex patterns most desirable for string matching.

3.1 Matching with Regex Decomposition

The key idea of Hyperscan is to decompose each regex pattern into a disjoint set of string and subregex (or FA) components, and to match each component until it finds a complete match. A string component consists of a stream of literals (or input symbols). Subregex components are the remaining parts of a regex after all string components are removed. They may include one or more meta-characters or quantifiers in regex syntax (like ‘^’, ‘\$’, ‘*’, ‘?’, etc.) that need to be translated into an FA for matching. Thus, we refer to it as an **FA** component.

Linear regex. We start with a simple regex where each component is concatenated without alternation. We call it a linear regex. Formally, a linear regex pattern that contains at least one string can be represented as the following production rules:

1. $\text{regex} \rightarrow \text{left str FA}$
2. $\text{left} \rightarrow \text{left str FA} \mid \text{FA}$

where **str** and **FA** are both indivisible components, and **FA** can be empty. A linear regex without any string is implemented as a single DFA or NFA. In practice, however, we find that 87% to 94% of the regex rules in IDSeS have at least one extractable string, so a majority of real-world regexes would benefit from decomposition. The production rules imply that if we find the rightmost string in a linear regex pattern, we can recursively apply the same algorithm to decompose the rest of the pattern. One complication lies in a subregex with a repetition operator such as $(R)?$, $(R)*$, $(R)^+$, and $(R)\{m,n\}$, where R is arbitrarily complex. Hyperscan treats $(R)?$ and $(R)*$ as a single FA since R is optional while it converts $(R)^+ = (R)(R)*$, and $(R)\{m,n\} = (R)...(R)(R)\{0,n-m\}$ ((R) appears m times)). Then, it decomposes their prefixes and treats the suffix as an FA.

In general, a decomposable linear regex can be expressed as $/FA_n str_n FA_{n-1} str_{n-1} \dots str_2 FA_1 str_1 FA_0/$. For any successful match of the original regex, all strings must be matched in the same order as they appear. Based on the observation, Hyperscan applies the following three rules for regex matching.

1. String matching is the first step. It scans the entire input to find all **strs**. Each successful match of **str** may trigger the execution of its neighbor **FA** matching.
2. Each **FA** has its own switch for execution. It is off by default except for the leftmost **FA** components.
3. For a generalized form like */left FA str right/* where “left” or “right” is an arbitrary set of decomposed com-

ponents including an empty character. Only if all components of “left” are matched successfully, the switch of **FA** is turned on. Only if **str** is matched successfully and the **FA** switch is on, **FA** matching is executed. Finally, only if **FA** is matched successfully, the leftmost **FA** of “right” is turned on.

Let’s take one example regex, */.*foo[^X]barY+/*, and consider two input cases. The regex pattern is decomposed into */FA₂ str₂ FA₁ str₁ FA₀/*, where $FA_2 = “.*”$, $str_2 = “foo”$, $FA_1 = “[^X]”$, $str_1 = “bar”$, $FA_0 = “Y+”$.

- **Input="XfooZbarY":** This is overall a successful match. First, the string matcher finds str_2 (“foo”), and triggers matching of FA_2 (“.”*) against “X” since the leftmost **FA** switch is always on. Then, the switch of FA_1 (“[^X]”) is turned on. After that, the matcher finds str_1 (“bar”), which triggers matching of FA_1 against “Z”, and its success turns on the the switch of FA_0 (“Y+”). Since FA_0 is the rightmost **FA**, it is executed against the remaining input, “Y”.
- **Input="XfoZbarY":** This is overall a failed match. First, the string matcher finds str_1 (“bar”), and sees if it can trigger matching of FA_1 (“[X]”). Then, it figures out that the switch of FA_1 is off since str_2 (“foo”) was not found, and thus none of FA_2 and str_2 was a successful match. So, the matching of regex FA_1 terminates, ensuring no waste of CPU resources.

Our implementation tracks of the input offsets of matched strings and the state of matching individual components, which allows invoking appropriate **FA** matching with a correct byte range of the input.

Regex with alternation. If a regex includes an alternation like $(A|B)$, we expand the alternation into two regexes only if both A and B are decomposed into **str** and **FA** components (decomposable). If not, $(A|B)$ is treated as a single FA. In case A or B itself has an alternation, we need to recursively apply the same rule until there is no alternation in their subregexes. Then, each expanded regex would become a linear regex, which would benefit from the same rules for decomposition and matching as before.

Pattern matching with regex decomposition presents its main benefit – it minimizes the waste of CPU cycles on unnecessary **FA** matching because **FA** matching is executed only when it is needed. Also, it increases the chance of fast DFA matching as each decomposed FA is smaller, so it is more likely to be converted into a DFA. In contrast, the prefiltering approach has to execute matching of the entire regex even when it is not needed (e.g., matching “bar” in the example above would trigger regex matching even if “foo” is not found), and regex matching must re-match the string already found in string

matching. Furthermore, conversion of a whole regex into a single FA is not only more complex, but often ends up with a slower NFA to avoid state explosion. In terms of correctness, pattern matching with regex decomposition produces the same result as the original regex, but we leave its formal proof as our future work.

3.2 Rationale and Guidelines

In practice, performing regex decomposition on its textual form is often tricky as some string segments might look hidden behind special regex syntax. We provide several such examples below:

- Character class (or character-set). $/b[i|l]l\s\{0,10\}/$ includes a character class that can be expanded to three strings ("bil", "bll" and "b1l") while naïve textual extraction might find only 'b' and 'l'.
- Alternation. The alternation sequence in $(.*\x2d(h|H)(t|T)(t|T)(p|P))$ makes it harder to discover "http" sequences from textual extraction.
- Bounded repeats. From the perspective of text, the strings with a minimum length of 32 are hidden from bounded repeats in $/[\x40\x90]\{32,\}/$.

To reliably find these strings, we perform regex decomposition on the Glushkov NFA graph [27], which would benefit from structural graph analyses. We describe useful guidelines for finding the strings effective for regex matching.

1. Find the string that would divide the graph into two subgraphs, with the start state in one subgraph, and the set of accept states in the other. Matching such a string is a necessary condition for any successful match on the entire regex. If the start and accept states happen to be in the same subgraph, the corresponding FA will always have to run regardless of a string match.
2. Avoid short strings. Short strings are prone to match more frequently, and are likely to trigger expensive FA matching that fails.³
3. Expand small character-sets⁴ to multiple strings to facilitate decomposition. This would not only increase the chance of successful decomposition but also lead to a longer string if a character-set intercepts a string sequence (i.e. "document[\x22\x27]object").
4. Avoid having too many strings. Having too many strings for matching would overload the matcher and degrade the entire performance. So, it is important to find a small set of "good" strings effective for regex matching.

³Our current limit is 2 to 3 characters.

⁴Our current implementation treats a character-set that expands to 11 or smaller strings as a small character-set.

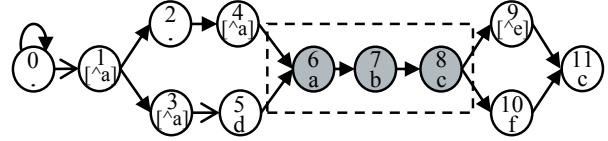


Figure 3: Dominant path analysis

3.3 Graph-based String Extraction

We develop three graph analysis techniques that discover strings in the critical path for matching. We describe the key idea of each algorithm below, and provide more detailed algorithms in an appendix.

Dominant path analysis. A vertex u is called a *dominator* of vertex v if every path from the start state to vertex v must go through vertex u . A dominant path of vertex v is defined as a set of vertices W in a graph, where each vertex in W is a dominator of v and the vertices form a trace of a single path. Dominant path analysis finds the longest common string that exists in all dominant paths of any accept state. For example, Figure 3 shows the string on the dominant path of the accept state (vertex 11).

The string selected by the analysis is highly desirable for matching as it clearly divides start and accept states into two separate subgraphs, satisfying the first guideline. The algorithm calculates the dominant path per each accept state, and finds the longest common prefix of all dominant paths. Then, it extracts the string on the chosen path. If a vertex on the path is a small character-set, we expand it and obtain multiple strings.

Dominant region analysis. If the dominant path analysis fails to extract a string, we perform dominant region analysis. It finds a region of vertices that partition the start state into one graph and all accept states into the other. More formally, a dominant region is defined as a subset of vertices in a graph such that (a) the set of all edges that enter and exit the region constitute a cut-set of the graph, (b) for every in-edge (u, v) to the region, there exist edges (u, w) for all w in $\{w : w \text{ is in the region and } w \text{ has an in-edge}\}$, where (u, v) refers to an edge from vertex u to v in the graph, and (c) for every out-edge (u, v) from the region, there exist edges (w, v) for all w in $\{w : w \text{ is in the region and } w \text{ has an out-edge}\}$.

If a discovered region consists of only string or small character-set vertices, we transform the region into a set of strings. Since these strings connect two disjoint subgraphs of the original graph, any match of the whole regex must match one of these strings. Figure 4 shows one example of a dominant region with 9 vertices. Vertices 5, 6, and 7 are the entry points with the same predecessors and vertices

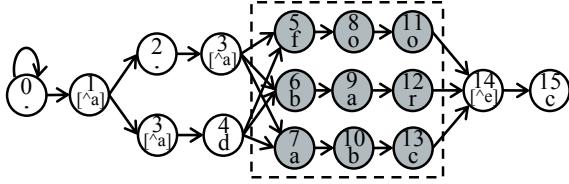


Figure 4: Dominant region analysis

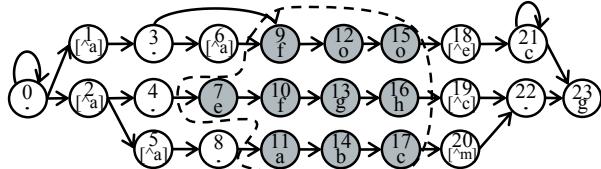


Figure 5: Network flow analysis

11, 12, and 13 are the exits with the same successor. We can extract strings, "foo", "bar", and "abc", as a result of dominant region analysis.

The algorithm for dominant region analysis first creates a directed acyclic graph (DAG) from the origin graph to avoid any interference from back edges. Then, it performs topological sort on the DAG, and iterates each vertex to see if it is added to the current candidate region, its boundary edges form a valid cut-set. We repeat this to discover all regions in the graph. Since we only analyze the DAG, the back edges of the original graph might affect the correctness. Thus, for each back edge, if its source and target vertices are in different regions, we merge them (and all intervening regions) into a single region. Finally, we extract the strings from the dominant region.

Network flow analysis. Since dominant path and dominant region analyses depend on a special graph structure, they may not be always successful. Thus, we run *network flow analysis* for generic graphs. For each edge, the analysis finds a string (or multiple strings) that ends at the edge. Then, the edge is assigned a score inversely proportional to the length of the string(s) ending there. The longer the string is, the smaller the score gets. With a score per edge, the analysis runs “max-flow min-cut” algorithm [25] to find a minimum cut-set that splits the graph into two that separate the start state from all accept states. Then, the “max-flow min-cut” algorithm discovers a cut-set of edges that generate the longest strings from this graph.

Figure 5 shows a result of network flow analysis, extracting a string set of "foo", "efgh", and "abc" that would divide the whole graph into two parts.

Effectiveness of graph analysis. Our graph analysis effectively produces “good” strings for most of real-world rules. Table 1 shows that 97.2% to 99.2% of decomposable real-world regex rules benefit from dominant path

Ruleset	Total	Decomp	D-Path	D-Reg	N-flow
S-V	1,663	1,563	1,551	32	16
S-E	7,564	6,756	6,575	100	203
Suri	7,430	6,501	6,318	94	201

Table 1: Effectiveness of graph analysis on real-world rulesets. S-V: Snort Talos (May 2015), S-E: Snort ET-Open 2.9.0, Suri: Suricata rulesets 4.0.4. D-Path, D-Reg, and N-flow refer to dominant path, dominant region, and network flow analysis, respectively. Decomps is the total number of decomposable rules. Note that one regex could benefit from multiple graph analyses, so the sum of graph analyses is larger than the Decomp fields.

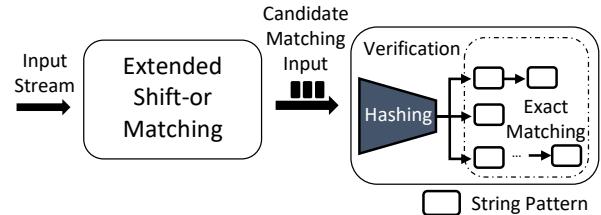


Figure 6: Two-stage matching with FDR

analysis while remaining patterns exploit dominant region and network flow analysis. These strings prove to be highly beneficial for reducing the number of regex matching invocations, as shown in Section 6.2.

4 SIMD-accelerated Pattern Matching

In this section, we present the design of multi-string and FA matching algorithms that leverage SIMD operations of modern CPU.

4.1 Multi-string Pattern Matching

We introduce an efficient multi-string matcher called FDR.⁵ The key idea of FDR is to quickly filter out innocent traffic by fast input scanning. As shown in Figure 6, FDR performs extended *shift-or matching* [13] to find candidate input strings that are likely to match some string pattern. Then, it verifies them to confirm an exact match.

Shift-or matching. We first provide a brief background of shift-or matching that serves as the base algorithm of FDR. The shift-or algorithm finds all occurrences of a string pattern in the input bytestream by performing bit-wise *shift* and *or* operations, as shown in Figure 7. It uses two data structures – a *shift-or mask* for each character c in the symbol set, ($\text{sh-mask}(c)$), and a *state mask* (st-mask) for matching operation. $\text{sh-mask}(c)$ zeros all bits whose bit position corresponds to the byte position of c in the string pattern while all other bits are set to 1. The bit position in a sh-mask is counted from the *rightmost* bit while the byte position in a pattern is counted from the *leftmost* byte. For example, for a string pattern, "aphp",

⁵It is named after the 32nd President of the U.S.

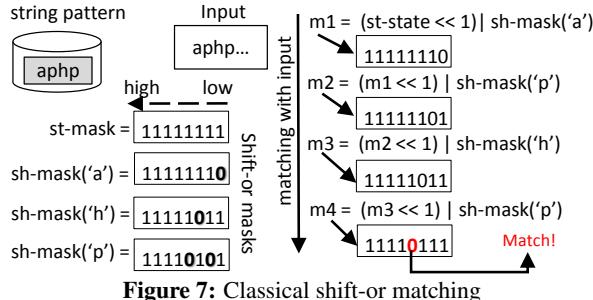


Figure 7: Classical shift-or matching

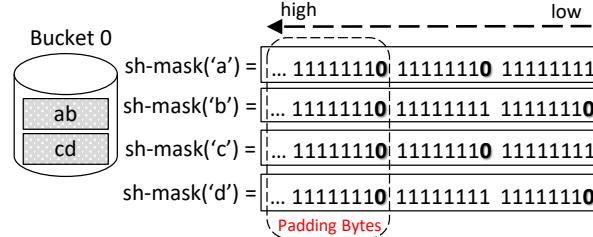


Figure 8: Example shift-or masks with two patterns at bucket 0. No other buckets contain ‘a’, ‘b’, ‘c’, ‘d’ in their patterns.

$\text{sh-mask}('p') = 11110101$ as ‘p’ appears at the second and the fourth position in the pattern. If a character is unused in the pattern, all bits of its sh-mask are set to 1. The algorithm keeps a st-mask whose size is equal to the length of a sh-mask. Initially, all bits of the st-mask are set to 1. The algorithm updates the st-mask for each input character, ‘x’ as $\text{st-mask} = ((\text{st-mask} \ll 1) | \text{sh-mask}('x'))$. For each matching input character, 0 is propagated to the left by one bit. If the zero bit position becomes the length of the pattern, it indicates that the pattern string is found.

The original shift-or algorithm runs fast with high space efficiency, but it leaves two limitations. First, it supports only a single string pattern. Second, although it consists of bit-level operations, the implementation cannot benefit from SIMD instructions except for very long patterns. We tackle these problems as below.

Multi-string shift-or matching. To support multi-string patterns, we update our data structures. First, we divide the set of string patterns into n distinct buckets where each bucket has an id from 0 to $n-1$. For now, assume that each string pattern belongs to one of n buckets as we will discuss how we divide the patterns later (‘pattern grouping’). Second, we increase the size of sh-mask and st-mask by n times so that a group of n bits in $\text{sh-mask}('x')$ record all the buckets that have ‘x’ in some of their patterns. More precisely, the k -th n bits of $\text{sh-mask}('x')$ encode the ids of all buckets that hold at least one pattern which has ‘x’ at the k -th byte position. One difference from the original algorithm is that the byte position in a pattern is counted from the *rightmost* byte. This enables parallel execution of multiple CPU instructions per cycle as explained later (‘SIMD acceleration’). For efficient implementation, we

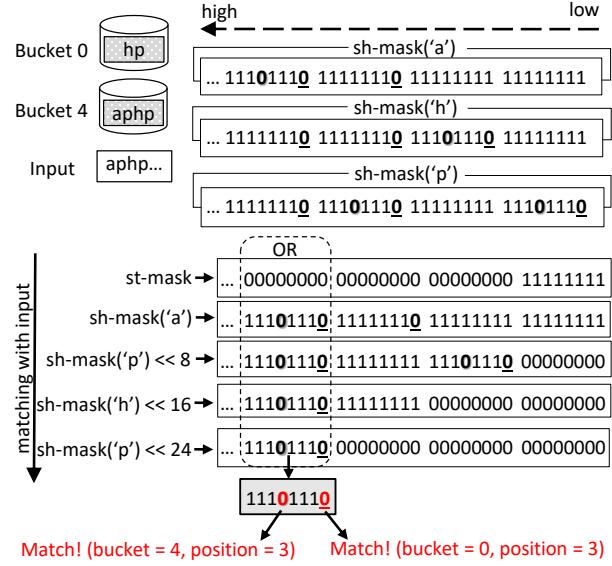


Figure 9: FDR’s extended shift-or matching

set n to 8 so that the byte position in a sh-mask matches the same position in a pattern. This implies that the length of a sh-mask should be no smaller than the longest pattern.

Figure 8 shows an example. Bucket 0 has two string patterns, “ab” and “cd”. Since ‘a’ appears at the second byte of only “ab” in bucket 0, sh-mask(‘a’) zeros only the first bit (= bucket 0) of the second byte. The i -th bit within each byte of a sh-mask indicates the bucket id, $i-1$. If the byte position of a sh-mask exceeds the longest pattern of a certain bucket (called ‘padding byte’), we encode the bucket id in the padding byte. This ensures matching correctness by carrying a match at a lower input byte along in the shift process. Examples of this are shown in the padding bytes in Figure 8, and in the sh-masks in Figure 9 that zero the first bit in the third and fourth bytes.

The pattern matching process is similar to the original algorithm except that sh-masks are shifted left instead of the st-mask. The st-mask is initially 0 except for the byte positions smaller than the shortest pattern. This avoids a false-positive match at a position smaller than the shortest pattern. Now, we proceed with input characters. The matcher keeps k , the number of characters processed so far modulo n . For an input character, ‘x’, $\text{st-mask} |= (\text{sh-mask}('x') \ll (k \text{ bytes}))$. The matcher repeats this for n input characters, and checks if the st-mask has any zero bits. Zero bits represent a possible match at the corresponding bucket. For example, Figure 9 shows that bucket 0 and 4 have a potential match at input byte position 3. The verification stage illustrated later checks whether they are a real match or a false positive.

Pattern grouping. The strategy for grouping patterns into each bucket affects matching performance. A good

strategy would distribute the patterns well such that most innocent traffic would pass with a low false positive rate. Towards the goal, we design our algorithm based on two guidelines. First, we group the patterns of a similar length into the same bucket. This is to minimize the information loss of longer patterns as the input characters match only up to the length of the shortest pattern in a bucket for matching correctness. Second, we avoid grouping too many short patterns into one bucket. In general, shorter patterns are more likely to increase false positives. To meet these requirements, we sort the patterns in the ascending order of their length, and assign an id of 0 to $(s-1)$ to each pattern by the sorted order. Then, we run the following algorithm that calculates the minimum cost of grouping the patterns into n buckets using dynamic programming. The algorithm is summarized by the two equations below:

1. $t[i][j] = \min_{k=i+1}^{s-1} (cost_{ik} + t[k+1][j-1])$, where s is the number of patterns and $t[i][j]$ stores the minimum cost of grouping the patterns i to $(s-1)$ into $(j+1)$ buckets.
2. $cost_{ik} = (k-i+1)^\alpha / length_i^\beta$, where $cost_{ik}$ is the cost of grouping patterns i to k into one bucket, $length_i$ is for pattern i , α and β are constant parameters.

$t[i][j]$ is calculated as the minimum of the sum of the cost of grouping patterns i to k into one bucket ($cost_{ik}$) and the minimum cost of grouping remaining patterns $(k+1)$ to $(s-1)$ into j buckets ($t[k+1][j-1]$). $cost_{ik}$ gets smaller as the bucket has a longer pattern, which allows more patterns in the bucket. It gets larger as the bucket has a shorter pattern, limiting the number of such patterns. Our implementation currently uses $\alpha = 1.05$ and $\beta = 3$ towards this goal, and computes $t[0][7]$ to divide all string patterns into 8 buckets, and records the bucket id per each pattern in the process. In practice, we find that the algorithm works well, automatically reaching the sweetspot that minimizes the total cost.

Super Characters. One problem with the bucket-based matching is that it produces false positives with patterns in the same bucket. For example, if a bucket has $/ab/$ and $/cd/$, the algorithm not only matches the correct patterns but also matches false positives, $/ad/$ and $/cb/$. To suppress them, we use an m -bit ($m > 8$) super character (instead of an 8-bit ASCII character) to build and index the sh-masks. An m -bit super character consists of a normal (8-bit) character in the lower 8 bits and low-order ($m-8$) bits of the next character in the upper bits. If it is the last character of a pattern (or in the input), we use a null character (0) as the next character. The key idea is to reflect some information of the next character in a pattern into building the sh-mask for the current character.

Only if the same two characters appear in the input⁶, we declare a match at that input byte position. This would significantly reduce false positives at the cost of a slightly large memory for sh-masks.

In practice, m should be between 9 and 15. Let's say $m = 12$ bits. For a pattern, $/ab/$, we see two 12-bit super characters, $\alpha = ((\text{low-order 4 bits of } 'b' \ll 8) | 'a')$, and $\beta = 'b'$. Then, we build sh-masks for α and β , respectively. When the input arrives, we construct a 12-bit super character based on the current input byte offset, and use it as an index to fetch its sh-mask. We advance the input one byte at a time as before. For example, if the input is $'ad'$, it first constructs $\gamma = ((\text{low-order 4 bits of } 'd' \ll 8) | 'a')$, fetches sh-mask(γ), and performs "shift" and "or" operations as before. Then, it advances to the next byte and constructs $\delta = 'd'$. So, the input $'ad'$ will not match even if a bucket contains $/ab/$ and $/cd/$.

SIMD acceleration. Our implementation of FDR heavily exploits SIMD operations and instruction-level parallelism. First, it uses 128-bit sh-masks so that it employs 128-bit SIMD instructions (e.g., *pslldq* for "left shift" and *por* for "or" in the Intel x86-64 instruction set) to update the masks. As "shift" and "or" are the most frequent operations in FDR, it enjoys a substantial performance boost with the SIMD instructions. Second, it exploits parallel execution with multiple CPU execution ports. In the original shift-or matching, the execution of "shift" and "or" operations is completely serialized as they have dependency on the previous result. This under-utilizes modern CPU even if it can issue multiple instructions per CPU cycle. In contrast, FDR exploits instruction-level parallelism by pre-shifting the sh-masks with multiple input characters in parallel. Note that this is made possible as we count the byte position differently from the original version. The parallel execution effectively increases instructions per cycle (IPC) and significantly improves the performance. To accommodate the parallel shifting, we limit the length of a pattern to 8 bytes and extract the lower 8 bytes from any pattern longer than 8 bytes. Because it requires minimum 8-byte masks and up to 7 bytes of shifting, a 128-bit mask would not lose high bit information during left shift. In actual matching, FDR handles 8 bytes of input at a time. To guarantee a contiguous matching across 8-byte input boundaries, the sh-mask of the previous iteration is shifted right by 8 bytes for the next iteration.

Verification. As our shift-or matching can still generate false positives, we need to verify if a candidate match is

⁶Of course, there is still a small chance of a false positive as we use partial bits of the next character, but the probability becomes fairly small as the pattern length grows.

Algorithm 1 FDR Multi-string Matcher

```
1: function MATCH
2:   n := number of bits of a super character
3:   R := startMask
4:   for each 8-byte V ∈ input do
5:     for i ∈ 0...7 do
6:       index = V[i*8..i*8+n-1]
7:       M[i] := shiftOrMask[index]
8:       S[i] := LSHIFT(M[i], i)
9:     end for
10:    for i ∈ 0...7 do
11:      R := OR128(R, S[i])
12:    end for
13:    for zero bit b in low 8 bytes of R do
14:      j := the byte position containing bit b
15:      l := length of string in bucket b
16:      h := HASH(V[j-l+1..j])
17:      if h is valid then
18:        Perform exact matching for each
19:        string in hash bucket [h]
20:      end if
21:    end for
22:    R := RSHIFT(R, 8))
23:  end for
24: end function
```

an exact match. This phase consists of hashing and exact string comparison. To minimize hash collisions, we build a separate hash table for each bucket. Then, we leverage the byte position of a match and compare the input with each string in the hash bucket to confirm a match. In practice, we find hashing filters out a large portion of false positives.

4.2 Finite Automata Pattern Matching

Successful string matching often triggers FA component matching, which is essentially the same as general regex matching. Our strategy is to use a DFA whenever is possible, but if the number of DFA states exceeds a threshold ⁷, we fall back to NFA-based matching. As the state-of-the-art DFA already delivers high performance, we introduce fast NFA-based matching with SIMD operations here.

In NFA-based matching, there can be multiple current states (current set) that are active at any time. A state transition with an input character is performed on every active state in the current set in parallel, which produces a set of successor states (successor set). A match is successful if any state reaches one of the accept states.

We develop *bit-based NFA* where each bit represents a state. We choose the bit-based representation as it outperforms traditional NFA representations that use byte arrays to store transitions, and look up a transition table for each current state in a serialized manner. Also, bit-based NFA leverages SIMD instructions to perform vectorized bit

operations to further accelerate the performance. Our scheme assigns an id to each state (i.e. each vertex in an n -node NFA graph) from 0 to $n-1$ by the topological order, and maintains a current set as a bit mask (called current-set mask) that sets a bit to 1 if its position matches a current state id. We define a *span of a state transition* as the id difference between the two states of the transition. Since state ids are sequentially assigned by the topological order, the span of a state transition is typically small. We exploit this fact to compactly represent the transitions below.

The bit-based NFA implements a state transition with an input character, ‘c’, in three steps. First, it calculates the successor set that can be transitioned to, from any state in the current set with *any* input character. Second, it computes the set of all states that can be transitioned to, from any state with ‘c’ (called reachable states by ‘c’). Third, it computes the intersection of the two sets. This produces a correct successor set as a Glushkov NFA graph guarantees that one can enter a specific state only with the same character or with the same character-set.

The challenge is to efficiently calculate the successor set. One can pre-calculate a successor set for every combination of current states, and look up the successor set for the current-set mask. While this is fast, it requires storing 2^n successor sets, which becomes intractable except for a small n . An alternative is to keep a successor-set mask per each individual state, and to combine the successor set of every state in the current set. This is more space-efficient but it costs up to n memory lookups and $(n-1)$ “or” operations. We implement the latter, but optimize it by minimizing the number of successor-set masks, which would save memory lookups. To achieve this, we keep a set of *shift-k masks* shared by all relevant states. A shift-k mask records all states with a forward transition of span k, where a forward transition moves from a smaller state id to a larger one, and a backward transition does the opposite. Figure 10 shows some examples of shift-k masks. Shift-1 mask sets every bit to 1 except for bit 7 since all states except state 7 has a forward transition of span 1.

We divide each state transition into two types – typical or exceptional. A *typical* transition is the one whose span is smaller or equal to a pre-defined shift limit. Given a shift limit, we build shift-k masks for every k ($0 \leq k \leq \text{limit}$) at initialization. These masks allow us to efficiently compute the successor set from the current set following typical transitions. If the current-set mask is *S*, then $((S \& \text{shift-}k \text{ mask}) \ll k)$ would represent all possible successor states with transitions of span k from *S*. If we combine successor sets for all k , we obtain the successor set reached by all typical transitions.

⁷We use 16,384 states as the threshold.

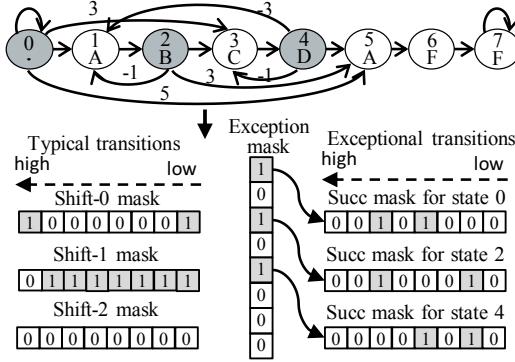


Figure 10: NFA representation for $(AB|CD)^*AFF^*$

We call all other transitions *exceptional*. These include forward transitions whose span exceeds the limit and any backward transitions.⁸ Any state that has at least one exceptional transition keeps its own successor mask. The successor mask records all states reached by exceptional transitions of its owner state. All exceptional states are maintained in an exception mask.

As you can see, the choice of the shift limit affects the performance. If it is too large, we would have too many shift-k masks representing rare transitions, and if it is too small, we would have to handle many exceptional states. Our current implementation uses 7 after performance tuning with real-world regex patterns.

Figure 10 shows an NFA graph for $(AB|CD)^*AFF^*$. We set the shift limit to 2 and mark exceptional edges with the difference of ids. State 0, 2, and 4 are highlighted as they have exceptional out-edge transitions. The exception mask holds all exceptional states, and each state points to its own successor mask. For example, successor mask for state 2 sets bits 1 and 5 as its exceptional transitions point to states 1 and 5.

Algorithm 2 shows our bit-based NFA matching. It combines the successor masks possibly reached by typical transitions ($SUCC_TYP$) and exceptional transitions ($SUCC_EX$). Then, it fetches the reachable state set with the current input character, c , ($reach[c]$) and perform a bitwise "and" operation with the combined successor mask ($SUCC$). The result is the final successor set, and we report a match if the successor set includes any accept state. Otherwise, it proceeds with the next input character. For each character, it runs in $O(l + e)$ where l is the shift limit, and e is the number of "exception" states. Our implemen-

⁸In our implementation, forward transitions that cross the 64-bit boundary of the state id space (e.g., from an id smaller than 64 to an id larger than 64) are also treated as exceptional. This is related to a specific SIMD instruction that we use, so we omit the detail here.

Algorithm 2 Bit-based NFA Matching

```

1: # SH_MSKS[i]      : shift-i masks for typical transitions
2: # SUCC_MSKS[i]   : successor mask for state i
3: # EX_MSK          : exception mask
4: # reach[k]        : reachable state set for character k
5: function RUNNFA( $S$ : current active state set)
6:    $SUCC\_TYP := 0$ ,  $SUCC\_EX := 0$ 
7:   for  $c \in \text{input}$  do
8:     if any state is active in  $S$  then
9:       for  $i := 0$  to  $shiftLimit$  do
10:          $R0 := AND(S, SH\_MSKS[i])$ 
11:          $R1 := LSHIFT(R0, i)$ 
12:          $SUCC\_TYP := OR(SUCC\_TYP, R1)$ 
13:     end for
14:      $S\_EX := AND(S, EX\_MSK)$ 
15:     for active state  $s$  in  $S\_EX$  do
16:        $SUCC\_EX :=$ 
17:        $OR(SUCC\_EX, SUCC\_MSKS[s])$ 
18:     end for
19:      $SUCC := OR(SUCC\_TYP, SUCC\_EX)$ 
20:      $S := AND(SUCC, reach[c])$ 
21:     Report accept states in  $S$ 
22:   end if
23:   end for
24: end function

```

Total Size	1 GBytes
Number of Packets	818,682
Number of TCP Packets	818,520
Percent of TCP Bytes	97.2%
Percent of HTTP Bytes	92.9%
Average Packet Size	1265 Bytes

Table 2: HTTP traffic trace from a cloud service provider.

tation uses a 128-bit mask (and extends it up to a 512-bit mask with four variables if needed), and employs 128-bit SIMD instructions for fast bitwise operations. In practice, we find that 512 states are enough for representing the NFA graph of most regexes.

5 Implementation

Hyperscan consists of compile and run time functions. Compile-time functions include regex-to-NFA graph conversion, graph decomposition, and matching components generation. The run time executes regex matching on the input stream. While we cover the core functions in Section 3 and 4, Hyperscan has a number of other subsystems and optimizations:

- Small string-set (<80) matching. This subsystem implements a shift-or algorithm using the SSSE3 "PSHUFBD" instruction applied over a small number (2-8) of 4-bit regions in the suffix of each string.
- NFA and DFA cyclic state acceleration. Where a state (in the case of the DFA) or a set of states (in the case of the NFA) can be shown to recur until some input is seen, we consider these cyclic states. In case where

# of regexes	Prefilter	Hyperscan	Reduction
500	2,971,652	645,326	4.6x
1000	93,595,304	714,582	131.0x
1500	110,122,972	791,017	139.2x
2000	139,804,519	780,665	179.1x
2500	156,332,187	857,100	182.4x

Table 3: Regex invocations of Snort’s ET-Open ruleset

current states in NFA or DFA are all cyclic states with a large reachable symbol set, there is a high probability of staying at current state(s) for many input characters. We have SIMD acceleration for searching the first exceptional input sequence (1-2 bytes) that leads to one or more transitions out of current states or switches off one of the current states.

- Small-size DFA matching. We design a SIMD-based algorithm for a small-size DFA (< 16 states) that outperforms the state-of-the-art DFA by utilizing the shuffle instruction for fast state transitions.
- Anchored pattern matching. When an anchored pattern consists of comparatively short acyclic sequences (i.e. no loops), the automata corresponding to them are both simple and short-lived. They are thus cheap to scan and scale well. We run DFA-based subsystems specialized to attempt to discover when anchored patterns are matched or rejected.
- Suppression of futile FA matching. We design a fast lookahead approach that peeks at inputs that are near the triggers for an FA before running it. This often allows us to discover that the FA either does not need to run at all or will have reached a dormant state before the triggers arrive. These checks are implemented as comparatively simple SIMD checks and can be done in parallel over a range of input characters and character classes. For example, in the regex fragment $/R \backslash d \backslash s\{4,5\}foo/$, where R is a complex regex, we can first detect that the digit and space character classes have matched with SIMD checks, and, if not, avoid or defer running a potentially expensive FA associated with R .

6 Evaluation

In this section, we evaluate the performance of Hyperscan to answer the following questions. (1) Does regex decomposition extract better strings than those by manual choice? (2) How well do multi-string matching and regex matching perform in comparison with the existing state-of-the-art? (3) How much performance improvement does Hyperscan bring to a real DPI application?

6.1 Experiment Setup

We use a server machine with Intel Xeon Platinum 8180 CPU @ 2.50GHz and 48 GB of memory, and compile the

# of regexes	Prefilter	Hyperscan	Reduction
700	699,622	18,164	38.5x
850	7,516,464	19,214	391.2x
1000	17,063,344	32,533	524.5x
1150	17,737,814	34,075	520.6x
1300	25,143,574	36,040	697.7x

Table 4: Regex invocations for Snort’s Talos ruleset

code with GCC 5.4. To separate the impact by network I/O, we evaluate the performance with a single CPU core by feeding the packets from memory. We test with packets of random content as well as a real-world Web traffic trace obtained at a cloud service provider as shown in Table 2. For all evaluation, we use the latest version of Hyperscan (v5.0) [2].

6.2 Effectiveness of Regex Decomposition

The primary goal of regex decomposition is to minimize unnecessary FA matching by extracting “good” strings from a set of regexes with rigorous graph analyses. To evaluate this point, we compare the number of regex matching invocations triggered by a prefilter-based DPI and by Hyperscan. We extract the content options and their associated regex from Snort rulesets, and count the number of regex matching invocations for a prefilter-based DPI. And then, we measure the same number for Hyperscan where Hyperscan automatically extracts the strings from regexes rather than using the keywords from the content option. For the ruleset, we use ET-Open 2.9.0 [8] and Talos 2.9.11.1 [11] against the real traffic trace, and confirm the correctness – both versions produce the identical output for the traffic.

Tables 3 and 4 show that Hyperscan dramatically reduces the number of regex matching invocations by over two orders of magnitude! As the number of regex rules increases, the reduction by Hyperscan grows, affirming that regex decomposition is the key contributor to efficient pattern matching. Close examination reveals that there are many single-character strings in the content option of the Snort rulesets, which invokes redundant regex matching too frequently. In practice, other rule options in Snort may mitigate the excessive regex invocations, but frequent string matching alone poses a severe overhead. In contrast, Hyperscan completely avoids this problem by triggering regex matching only if it is necessary.

6.3 Microbenchmarks

We evaluate the performance of FDR, our multi-string matcher, as well as that of regex matching of Hyperscan.

Multi-string pattern matching. We compare the performance of FDR with that of DFC and AC. We measure the performance over different numbers of string patterns ex-

Ruleset	PCRE	PCRE2	RE2-s	Hyperscan-s	RE2-m	Hyperscan-m
Talos	6,942	394	1,777	173	29	2.15
ET-Open	12,800	913	4,696	516	1,116	133

Table 5: Performance comparison with PCRE, PCRE2, RE2 and Hyperscan for Snort Talos (1,300 regexes) and Suricata (2,800 regexes) rulesets with the real Web traffic trace. Numbers are in seconds.

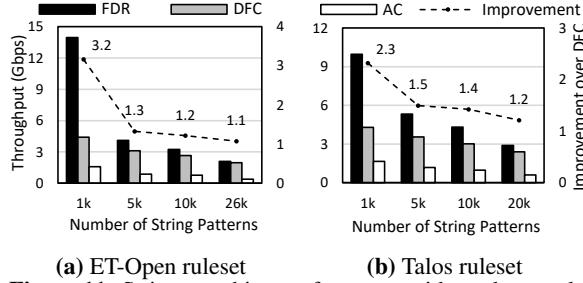


Figure 11: String matching performance with random packets

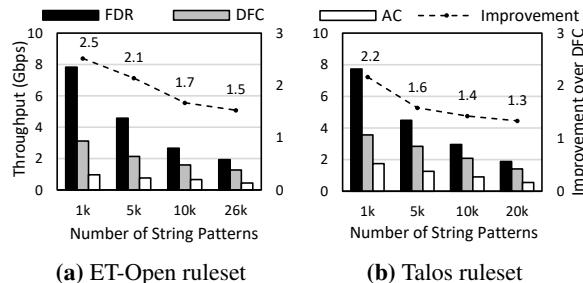


Figure 12: String matching performance with a real traffic trace

tracted from Snort ET-Open and Talos rulesets. Figure 11 and Figure 12 show that FDR outperforms the state-of-the-art matcher, DFC, by 1.1x to 3.2x (and AC by 4.2x to 8.8x) for packets of random content and by 1.3x to 2.5x (and AC by 3.2x to 8.2x) for the real traffic trace. We also evaluate it with the Suricata ruleset but we omit the result here since it exhibits the similar performance trend. When the number of string patterns is small, Hyperscan benefits from small CPU cache footprint and SIMD acceleration, but when the number of patterns grows, the performance becomes compatible with DFC due to increased cache usage, but it is still much better than AC.

Regex matching. We now evaluate the performance of regex matching of Hyperscan. We compare the performance with PCRE (v8.41) [6] as it is most widely used in network DPI applications, and PCRE2 (v10.32) [7], a more recent fork from PCRE with a new API. We enable the JIT option for both PCRE and PCRE2. We also compare with RE2 (v2018-09-01) [1], a fast, small memory-footprint regex matcher, developed by Google. Both Hyperscan and RE2 support multi-regex matching in parallel while PCRE matches one regex at a time. For fair comparison with PCRE and PCRE2, we measure the total time for matching all regexes in a serial manner (e.g.,

match against one regex at a time), which would require passing the entire input for each regex. For Hyperscan and RE2, we measure two numbers – one for matching one regex at a time (RE2-s, Hyperscan-s), and the other for matching all regexes in parallel (RE2-m, Hyperscan-m). For testing, we use 1,300 regexes from the Snort Talos ruleset and 2,800 regexes from the Suricata ruleset.

Table 5 shows the result. For the Snort Talos ruleset, Hyperscan-s outperforms PCRE, RE2-s, and PCRE2 by 40.1x, 10.3x, and 2.3x, respectively. Hyperscan-m is 13.5x faster than RE2-m while it reveals 183.3x performance improvement over PCRE2! For the Suricata ruleset, Hyperscan-s shows 24.8x, 9.1x, and 1.8x speedups over PCRE, RE2-s, and PCRE2. Hyperscan-m outperforms RE2-m and PCRE2 by 8.4x and 6.9x, respectively. We do not report DFA-based PCRE performance as we find it much slower than the default operation with NFA-based matching [5]. RE2-s uses a DFA for regex matching (and fails on a large regex that requires an NFA), but it needs to translate the whole regex into a single DFA. In contrast, Hyperscan splits a regex into strings and FAs, and benefits from fast string matching as well as smaller DFA matching of the FA components, which explains the performance boost.

6.4 Real-world DPI Application

We now evaluate how much performance improvement Hyperscan brings to a popular IDS like Snort. We compare the performance of stock Snort (ST-Snort) and Hyperscan-ported Snort (HS-Snort) that performs pattern matching with Hyperscan, both with a single CPU core. ST-Snort employs AC and PCRE for multi-string matching and regex matching, respectively. HS-Snort keeps the basic design of Snort but it replaces AC and PCRE with the multi-string and single-regex matchers of Hyperscan. It also replaces the Boyer-Moore algorithm in Snort with a fast single-literal matcher of Hyperscan. With the Snort Talos ruleset, ST-Snort achieves 113 Mbps on our real Web traffic. In contrast, HS-Snort produces 986 Mbps on the same traffic, a factor of 8.73 performance improvement. We find that the main contributor for performance improvement is the highly-efficient multi-string matcher of Hyperscan as shown in Figure 11. In practice, we expect a much higher performance improvement if we restructure Snort to use multi-regex matching in parallel.

7 Evolution, Experience, and Lessons

Hyperscan has been developed since 2008, and was first open-sourced in 2013. Hyperscan has been successfully adopted by over 40 commercial projects globally, and it is in production use by tens of thousands of cloud servers in data centers. In addition, Hyperscan has been integrated into 37 open-source projects, and it supports various operating systems such as Linux, FreeBSD, and Windows. Hyperscan APIs are initially developed in C, but there are public projects that provide bindings for other programming languages such as Java, Python, Go, Node.js, Ruby, Lua, and Rust. In this section, we briefly share our experience with developing Hyperscan and lay out its future direction.

7.1 Evolution of Hyperscan

Hyperscan was developed at a start-up company, Sensory Networks, after a move away from hardware matching, which was expensive in terms of material costs and development time. We investigated GPU-based regex matching, but it imposed unacceptable latency and system complexity. As CPU technology advances, we settled at CPU-based regex matching, which not only became cost-effective with high performance, but also made it simple to be employed by applications.

Version 1.0. The initial version was released in 2008, with the intent of providing a simple block-mode regex package that could handle large numbers of regexes. Like other popular solutions at that time, it used string-based prefiltering to avoid expensive regex matching. However, the initial version was algorithmically primitive and lacked streaming capability (e.g., pattern matching over streamed data). Also, it suffered from quadratic matching time as it had to re-scan the input from a matched literal for each match.

Version 1.0 did include a large-scale string matcher (a hash-based matcher called "hlm", akin to Rabin-Karp [29] with multiple hash tables for different length strings) as well as a bit-based implementation of Glushkov NFAs. The NFA implementation allowed support of a broad range of regexes that would suffer from combinatorial explosions if the DFA construction algorithm was used. The Glushkov construction mapped well to Intel SIMD instructions, allowing NFA states to be held in one or more SIMD registers.

Version 2.0. The algorithmic issues and the absence of a streaming capability led to major changes to version 1.0, which became Hyperscan version 2.0. First, it moved towards a Glushkov NFA-based internal representation (the "NFA Graph") that all transformations operated over, departing from ad-hoc optimizations on the regex syntax

tree. Second, it supported 'streaming' – scanning multiple blocks without retaining old data and with a fixed-at-pattern-compile-time amount of stream state. Support for efficient streaming was especially desirable for network traffic monitoring as patterns may spread over multiple packets. Third, it scanned patterns that used one or more strings, detected by a string matching pre-pass, followed by a series of NFA or DFA engines running over the input only when the required strings are found. This approach avoids the high risk of potential quadratic behavior of version 1.0, with the tradeoff of potentially bypassing some comparatively lesser optimizations if a regex could be quickly falsified at each string matching site.

Unfortunately, version 2.0 still had a number of limitations. First, we observed the adverse performance impact of prefiltering. Prefiltering did not reduce the size of the NFA or DFA engines even if a string factor completely separated a pattern into two smaller ones. This exacerbated the problem of a large regex that often needed to be converted into an NFA. As the system had a hard limit of 512 NFA states (dictated by the practicalities of a data-parallel SIMD implementation of the Glushkov NFA; more than 512 states resulted in extremely slow code), it often did not accommodate user-provided regexes when they were too large. Further, if prefiltering failed (i.e., when the string factors were all present), it ended up consuming more CPU cycles than naïvely performing the NFA engines over all the input.

Another serious limitation was that matches emerged from the system in an undefined order. Since the NFAs were run after string matching had finished, the matches from these NFAs would emerge based on the order of which NFAs were run first and no rigorous order was defined for when these matches would appear. Further confusing matters, the string matcher was capable of producing matches of plain strings ahead of the NFA executions. In fact, due to potential optimizations where NFA graphs might be split (for example, splitting into connected components to allow an unimplementable large NFA to be run as smaller components), it was even possible to receive duplicate matches for a given pattern at a given location. After an NFA is split into connected components and run in separate engines, no mechanism existed to detect whether these different components (which would be running at different times) might be sometimes producing matches with the same match id and location.

For example, a regex workload consisting of patterns `/foo/`, `/abc[xyz]/` and `/abc[xy].*def|abc.z.*de[fg]/` might first produce matches for the simple literal `/foo/`, then provide all the matches for the components of the pattern `/abc[xyz]/`, then provide matches for the two parts

of the alternation $abc[xy].*def$ and $abc.z.*de[fg]$ without removing duplicate matches for inputs that happened to match both parts of the pattern on the same offset (e.g. the input "abxxxxzdef").

Versions 2.1 and 3.0. (The version 2.1 release series of Hyperscan saw considerable development and in retrospect should have merited a full version number increment) The limitation of prefiltering spurred the development of an alternate matching subsystem called ‘Rose’. ‘Rose’ allowed both ordered matching, duplicate match avoidance, and pattern decomposition. This subsystem was maintained in parallel to the prefiltering system inherited from the original 2.0 design. Whenever it was possible to decompose patterns, the patterns were matched with the ‘Rose’ subsystem, which initially was not capable of handling all regular expressions.

FDR was developed during in the version 2.1 release series; it replaced the hash-based literal matcher ("hlm") with considerably performance improvements and reduction in memory footprint.

Eventually, by version 3.0, the old prefiltering system was entirely removed, as the Rose path was made fully general. Version 3.0 also marked an organizational change in that Intel Corporation had acquired Sensory Networks.

Version 4.0. Version 4.0 was released in October 2013 under an open-source BSD license to further increase the usage of Hyperscan, by removing barriers of cost and allowing customization. Many elements of Hyperscan’s design continued to evolve. For example, the initial Rose subsystem had a host of special-purpose matching mechanisms that identify the strings separated by various common regex constructs such as $.*$ or X^+ for some character class X . For example, it is frequently the case that strings in regexes might be separated by the $.*$ construct (i.e. $/foo.*bar/s$). This is usually implementable by requiring only that “*foo*” is seen before “*bar*” (usually, but not always: consider the expression $/foo.*oar/s$). The original version of Rose had many special-purpose engines to handle these type of subcases; during the evolution of the system, this special-purpose code was almost entirely replaced with generalized NFA and DFA mechanisms, amenable to analysis and optimization, and were needed for the general case of regex matching in any case.

Version 5.0. Version 5.0, which is the latest version as of writing this paper, mainly focused on enhancing the usability of Hyperscan. Two key added features are support for logical combinations of patterns and *Chimera*, a hybrid regex engine of Hyperscan and PCRE [6]. As the detection technology of malicious network traffic matures, it often requires evaluating a logical combination of a group of patterns beyond matching a single pattern. To

support this, the system now allows user-defined AND, OR, NOT along their patterns. For example, an AND operation between patterns `/foobar/` and `/teakettle/` required that both patterns are matched for input before reporting a match. Version 5.0 added Chimera, a hybrid matcher of Hyperscan and PCRE, brings the benefit of both worlds – support for full PCRE syntax while enjoying the high performance of Hyperscan. Lack of support of Hyperscan for full PCRE syntax (such as capturing and back-references) made it difficult to completely replace PCRE in adopted solutions. Chimera employs Hyperscan as a fast filter for input, and triggers the PCRE library functions to confirm a real match only when Hyperscan reports a match on a pattern that has features not supported by Hyperscan.

7.2 Lessons Learned

We summarize a few lessons that we learned in the course of development and commercialization of Hyperscan.

Release quickly and iterate, even with partial feature support. The difficulty of generalized regex matching often led Hyperscan to focus on the delivery of some capability in partial form. From a theoretical standpoint, it is unsatisfactory that Hyperscan still cannot support all regexes (even from the subset of ‘true’ regexes) and that the support of regexes with the ‘start of match’ flag turned on is even smaller. However, customers can still find the system practically useful despite these limitations. Despite the limited pattern support of version 2.0 and the problems of ordering and duplicated matches, there was immediate commercial use of the product even in that early form (use which made subsequent development possible). This lends support to the idea of releasing a “Minimal Viable Product” early, rather than developing a product with a long list of features that customers may or may not want.

Evolve new features over several versions, at the expense of maintaining multiple code paths. Academic systems are usually built for elegance and to illustrate a particular methodology. However, a commercial system must stay viable as a product while a new subsystem is built. For example, Hyperscan maintained both the ‘prefilter’ arrangement and the new ‘Rose-based’ decomposition arrangement in the same code base, resulting in considerable extra complexity. However, the benefits of having the new ‘ordered’ semantics (with additional powers of support for large patterns due to decomposition) outweighed the complexity cost. It was almost impossible that a small start-up could have managed to transition from one system to the other in a span of a single release, or to have simply not meaningfully updated the project for an extended time while working on a substantial update.

Commercial products may need to emphasize less interesting subcases of a task, or unusual corner cases. There was also considerable commercial pressure to be the best option at some comparatively degenerate subset of regex matching, or some relatively ‘hard case’. Customers often wanted Hyperscan to function as a string matcher - sometimes even a single-string-at-a-time matcher! Other customers wanted high performance despite very high regex match rates (for example, more than 1 match per character). Such demands often force special optimizations that lack deep ‘algorithmic interest’, but are necessary for commercial success.

Be cautious of cross-cutting complexity resulting from customer API requests. One illuminating experience in the delivery of a commercially viable regex matcher was that customer feature requests for new ‘modes’ or unusual calls at the API level resulted in cross-cutting complexity that made the code base considerably more complicated (due to a combinatorial explosion of interactions between features) while rarely being reused by other customers. Features added in the 2.0 or 3.0 release series over time were not carried forward to the 4.0 series; we found that frequently such features were only used by a single customer (despite being made available to all).

Examples of two such features were “precise alive” (the ability to tell at any given stream write boundary whether a pattern might still be able to match) and an ad-hoc stream state compression scheme that allowed some stream states to be discarded if no NFA engines had started running. These features were complicated and suppressed potential optimizations as well as interacting poorly with other parts of the system.

7.3 Future Directions

Hyperscan is performance-oriented; future development in Hyperscan will still focus on delivering the best possible performance, especially on upcoming Intel Architecture cores featuring new instruction set extensions such as Vector Byte Manipulation Instructions (VBMI). Improvement of scanning performance as well as reduction of overheads such as the size of bytecodes, size of stream states and time to compile the pattern matching bytecode are obvious next steps.

Beyond this, adding richer functionality, including support for currently unsupported constructs such as generalized “lookaround” asserts and possible some level of support for back-references would aid some users. There is a considerable amount of usage of the ‘capturing’ functionality of regexes, which Hyperscan does not support at all (an experimental subbranch of Hyperscan, not widely released, supported capturing functionality for a limited

set of expressions). Hyperscan could be extended to have enriched semantics to support capturing, which would allow portions of the regexes to ‘capture’ parts of the input that matched particular parts of the regular expression.

8 Conclusion

In this paper, we have presented Hyperscan, a high-performance regex matching system that suggests a new strategy for efficient pattern matching. We have shown that the existing prefilter-based DPI suffers from frequent executions of unnecessary regex matching. Even though Hyperscan started with the similar approach, it has evolved to address the limitation over time with novel regex decomposition based on rigorous graph analyses. Its performance advantage is further boosted by efficient multi-string matching and bit-based NFA implementation that effectively harnesses the capacity of modern CPU. Hyperscan is open sourced for wider use, and it is generally recognized as the state-of-the-art regex matcher adopted by many commercial systems around the world.

Acknowledgment

We appreciate valuable feedback by anonymous reviewers of USENIX NSDI’19 as well as our shepherd, Vyas Sekar. We acknowledge the code contributions over the years by the Sensory Networks team: Matt Barr, Alex Coyte and Justin Viiret. This work is in part supported by the ICT Research and Development Program of MSIP/IITP, South Korea, under projects [2018-0-00693, Development of an ultra-low latency user-level transfer protocol] and [2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion].

References

- [1] Google RE2. <https://github.com/google/re2/>.
- [2] Hyperscan GitHub Repository. <https://github.com/intel/hyperscan>.
- [3] ModSecurity. <https://www.modsecurity.org/>.
- [4] nDPI. <https://www.ntop.org/products深深包-检测/ndpi/>.
- [5] PCRE Manual Pages. <https://pcre.org/pcre.txt>.
- [6] PCRE: Perl Compatible Regular Expressions. <https://www.pcre.org/>.

- [7] Perl-compatible Regular Expressions (revised API: PCRE2). <https://www.pcre.org/current/doc/html/index.html>.
- [8] Snort Emerging Threats Rules 2.9.0. <https://rules.emergentthreats.net/open/snort-2.9.0/rules/>.
- [9] Snort Intrusion Detection System. <https://snort.org>.
- [10] Suricata: Open Source IDS. <http://suricata-ids.org/>.
- [11] Talos Ruleset. <https://www.snort.org/talos>.
- [12] Aho, Alfred V. and Corasick, Margaret J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.
- [13] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM (CACM)*, 35(10):74–82, 1992.
- [14] Zachary K. Baker and Viktor K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2004.
- [15] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2007.
- [16] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2007.
- [17] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.
- [18] M. Becchi and P. Crowley. A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1), April 2013.
- [19] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep Packet Inspection as a Service. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.
- [20] Taylor-D. E. Brodie, B. and R. K. Cytron. A scalable architecture for high-throughput regular expression pattern matching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006.
- [21] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating string pattern matching for network applications. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.
- [22] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Koné, and Ashley Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the Workshop on Network Processors Applications (NP3)*, 2004.
- [23] Christopher R. Clark and David E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2003.
- [24] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the Colloquium, on Automata, Languages and Programming*, 1979.
- [25] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [26] Giordano-S. Procissi G. Vitucci F. Antichi G. Ficara, D. and A. Di Petro. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(5), 2008.
- [27] V.M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.
- [28] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 317–328, 2012.
- [29] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [30] T. Kojm. ClamAV. <http://www.clamav.net/>.
- [31] Smith-R. Kong, S. and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2008.
- [32] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and

- J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the ACM SIGCOMM on Data communication (SIGCOMM)*, 2006.
- [33] Turner-J. Kumar, S. and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2006.
- [34] Janghaeng Lee, Sung Ho Hwang, Neungsoo Park, Seong-Won Lee, Sunglk Jun, and Young Soo Kim. A High Performance NIDS using FPGA-based Regular Expression Matching. In *Proceedings of the ACM symposium on Applied computing*, 2007.
- [35] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.
- [36] J. Jang J. Truelove D. G. Andersen S. K. Cha, I. Moraru and D. Brumley. SplitScreen: Enabling efficient, distributed malware detection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.
- [37] E. Sitaridi, O. Polychroniou, and K. A. Ross. SIMD-accelerated regular expression matching. In *Proceedings of the Workshop on Data Management on New Hardware (DaMoN)*, 2016.
- [38] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.
- [39] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM on Data communication (SIGCOMM)*, 2008.
- [40] Y. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on fpga. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008.
- [41] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, and R. H. Kats. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014.
- [42] Xiaodong Yu, Bill Lin, , and Michela Becchi. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications*, 32(10), October 2014.

Appendix

Algorithm 3 Dominant Path Analysis

Require: Graph $G=(E,V)$

```

1: function DOMINANTPATHANALYSIS( $G$ )
2:    $dpath := \{\}$ 
3:   for  $v \in accepts$  do
4:     calculate dominant path  $p[v]$  for  $v$ 
5:     if  $dpath = \{\}$  then
6:        $dpath := p[v]$ 
7:     else
8:        $dpath := common\_prefix(dpath, p[v])$ 
9:     if  $dpath = \{\}$  then
10:      return null_string
11:    end if
12:  end if
13:   $strings := expand\_and\_extract(dpath)$ 
14: end for
15: return  $strings$ 
16: end function

```

The dominant path analysis algorithm finds the dominant path ($p[v]$) for every accept state (v), and find the common path of all dominant paths. The function, `expand_and_extract()`, expands small character-sets in the path, and extracts the string on the path.

Algorithm 4 Dominant Region Analysis

Require: Graph $G=(E,V)$

```

1: function DOMINANTREGIONANALYSIS( $G$ )
2:    $acyclic\_g := build\_acyclic(G)$ 
3:    $Gt := build\_topology\_order(acyclic\_g)$ 
4:    $candidate := q0$ 
5:    $it = begin(Gt)$ 
6:   while  $it \neq end(Gt)$  do
7:     if  $isValidCut(candidate)$  then
8:        $setRegion(candidate)$ 
9:        $initializeCandidate(candidate)$ 
10:     else
11:        $addToCandidate(it)$ 
12:        $it := it + 1$ 
13:     end if
14:   end while
15:    $setRegion(candidate)$ 
16:   Merge regions connected with back edge
17:    $strings := expand\_and\_extract(regions)$ 
18:   return  $strings$ 
19: end function

```

The dominant region analysis builds an acyclic graph and sorts the vertices by the topological order. Then, it adds each vertex of the graph into a candidate vertex set,

and sees if the candidate vertex set forms a valid cut. If so, it creates a region. It continues to create regions by iterating all vertices. Finally, it merges the regions by back edges, and extracts strings from the merged region.

Algorithm 5 Network Flow Analysis

Require: Graph $G=(E,V)$

```
1: function NETWORKFLOWANALYSIS( $G$ )
2:   for  $edge \in E$  do
3:      $strings := find\_strings(edge)$ 
4:      $scoreEdge(edge, strings)$ 
5:   end for
6:    $cuts := MinCut(G)$ 
7:    $strings := extract\ and\ expand\ strings\ from\ cuts$ 
8:   return  $strings$ 
9: end function
```

The network flow analysis assigns a score to every edge and runs the "max-flow min-cut" algorithm. An edge is assigned a score inverse proportional to the length of a string that ends at the edge. So, the longer the string is, the smaller the score gets. Then, the max-flow min-cut algorithm finds a cut whose edge has the longest strings.

Deniable Upload and Download via Passive Participation

David Sommer
ETH Zurich

Aritra Dhar
ETH Zurich

Luka Malisa
ETH Zurich

Esfandiar Mohammadi
ETH Zurich

Daniel Ronzani
Ronzani Schlauri Attorneys

Srdjan Čapkun
ETH Zurich

Abstract

Downloading or uploading controversial information can put users at risk, making them hesitant to access or share such information. While anonymous communication networks (ACNs) are designed to hide communication meta-data, already connecting to an ACN can raise suspicion. In order to enable plausible deniability while providing or accessing controversial information, we design CoverUp: a system that enables users to asynchronously upload and download data. The key idea is to involve visitors from a collaborating website. This website serves a JavaScript snippet, which, after user’s consent produces cover traffic for the controversial site / content. This cover traffic is indistinguishable from the traffic of participants interested in the controversial content; hence, they can deny that they actually up- or downloaded any data.

CoverUp provides a feed-receiver that achieves a down-link rate of 10 to 50 Kbit/s. The indistinguishability guarantee of the feed-receiver holds against strong global network-level attackers who control everything except for the user’s machine. We extend CoverUp to a full upload and download system with a rate of 10 up to 50 Kbit/s. In this case, we additionally need the integrity of the JavaScript snippet, for which we introduce a trusted party. The analysis of our prototype shows a very small timing leakage, even after half a year of continual observation. Finally, as passive participation raises ethical and legal concerns for the collaborating websites and the visitors of the collaborating website, we discuss these concerns and describe how they can be addressed.

1 Introduction

Access to and distribution of sensitive and controversial information often comes at risk for users. Due to the risk of being observed, users might be reluctant to download or upload certain content. Even if the content itself is end-to-end encrypted, the fact that the user accessed a particular domain or used an anonymity network might already indicate his in-

terest in the particular content. Since Edward Snowden’s revelations, we know that surveillance is mostly based on meta-data, such as source and destination IP, timestamps, and the size of the data [55].

Solutions like anonymous communication networks (ACN) are designed to hide such meta-data. Despite that, even the strongest ACNs in literature [73, 71, 40, 63] do not protect against global network attackers and do not hide users’ participation in the ACN, except for the brute-force method of continuously producing artificial traffic [30]. This participation in an ACN alone can appear suspicious. Participation time can be used in long-term statistical disclosure attacks to re-identify the user, thereby downgrading the anonymity properties of an ACN [43, 44].

In this paper, we aim to solve this issue in the case of asynchronous upload and download and therefore address the following problem: how to allow users to safely download and upload content without the fear of their intentions being identified. This problem is different from the more general problem of anonymous communication. Namely, content upload and download is asynchronous, typically allows for high latency, and is therefore much less vulnerable to timing correlations. Additionally, we aim to achieve a stronger anonymity property: we require that the participation (time) of users is protected.

Our approach to solving this problem is to draw in visitors (*Passive Participants*) of highly accessed websites (the *Entry server*) and trigger them via JavaScript to create cover traffic to a controversial content server. Additionally, we ensure that the passive participants’ traffic is indistinguishable from active participants’, who are genuinely interested in downloading/uploading the content, thereby enabling deniable communication.

While prior work proposed the central idea of using JavaScript-generated cover traffic for deniable communication [41, 65], these proposals left three main challenges unsolved: (i) How to construct a downlink connection (using the browser) that relays data to an external program with minimal timing leakage. (ii) How to relay data from an ex-

ternal program to the uplink connection (using the browser) with minimal timing leakage. (iii) How long can such a system be safely used before the timing leakage renders active participants clearly distinguishable?

We address these three challenges. We design a system (COVERUP) that asks visitors of an Entry server for their (informed) consent to become passive participants and to produce cover traffic. We utilize this cover traffic to realize uni-directional and bi-directional deniable communication channels. Our uni-directional deniable channel (CU:Feed) retrieves a message feed (Challenge (i)) from a content server (the *Feed server*) and delivers it to the COVERUP-Tool, a program that active participants would install. CU:Feed involves an additional party (the *COVERUP server*) to which the Entry server forwards active and passive participants. This COVERUP server sends the participants a piece of JavaScript code, which retrieves the feed from the Feed server. Such message feeds are suited for the transmission of information that a user does not want to be caught reading (e.g., sensitive medical information or leaked documents). We protect passive participants from potentially incriminating information by enforcing that a participant's machine never contains enough data chunks to reconstruct any incriminating information from the feed. CU:Feed achieves deniability against a global network-level attacker that controls all parties except the user's machine.

We extend CU:Feed to a deniable bi-directional channel CU:Transfer, which enables data download from and data upload to a content server, called the *Transfer server*. Active participants install for CU:Transfer a browser extension that implements an interface for bi-directional communication between the COVERUP-Tool and the Transfer server (Challenge (ii)). CU:Transfer achieves deniability against a global network-level attacker that controls all parties except for the user's machine, the COVERUP server, and the Transfer server.

For both channels, we implemented a prototype that carefully minimizes the timing leakage (Challenge (iii)). The prototype includes an entry server, and the COVERUP server that serves the JavaScript code. For active participants, we additionally provide the browser extension and the COVERUP-Tool, which enable participants to interact with the content servers (the Feed and/or the Transfer server). The COVERUP down- and up-link rate of our prototype is between 10 and 50 Kbit/s, depending on bandwidth overhead, and the expected latency is 60 seconds.

We experimentally evaluate the timing-leakage of our prototypes by measuring the differences in the traffic of active participants and passive participants (Challenge (iii)). We show that their traffic is hard to distinguish, and for half a year of continual observation¹, we can bound the attacker's advantage of distinguishing these usage patterns with 2 ·

¹We assume a usage pattern of at most 50 times a day, and at most 5 hours per day in total.

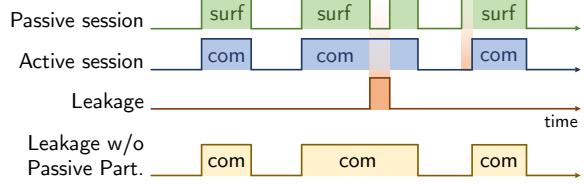


Figure 1: **Hiding the participation time** via passive participation. The x-axis is the time, and the y-axis show whether at that time surfing or protocol-communicating behavior is expected. Only communicating activity which is not covered by the *expected* surfing behavior creates leakage. Active participants that produce protocol-communication only produce leakage during time where they would normally not surf.

10^{-3} , i.e., the chance of successfully deciding whether a user is active or passive is 50.001%.²

Summary of contributions.

- Design of uni-directional and bi-directional deniable channels to a trusted server utilizing passive participation.
- Fully working prototype.
- Evaluation of the induced timing leakage for the distinguishability of active and passive participants.
- Discussion of ethical and selected legal questions w.r.t. the entry server and the passive participants.

2 Problem description

The goal of this paper is to enable users to safely up- and download content without the fear of their intentions being identified. The concrete problem is to enable users to hide their up- and download activities among traffic that is produced by other normal web users. This problem is different from the more general problem of anonymous communication, as our goal is to utilize the traffic of normal web surfers.

2.1 Passive participation

One approach for utilizing the traffic of normal web surfers of highly accessed websites (the *entry server*) is passive participation: compel web surfers (*passive participants*) to create cover traffic to the content server in a non-invasive manner and such that their traffic is indistinguishable from *active participants* (which are genuinely interested in downloading/uploading the content). As a result, active participants can deny that they up- or downloaded any data during their normal surfing time on the entry server websites, protecting their *participation time* in the file-sharing protocol.

The degree of plausible deniability depends on whether the active participants manage to let their surfing behavior

²This advantage is very low, since (in contrast to some usages of cryptographic schemes) COVERUP is a system that has limited exposure. Thus, an attacker cannot get arbitrarily many samples to amplify his or her chance to guess correctly to a clear decision.

towards the entry server unchanged. For users that are willing to make a paradigm shift, COVERUP offers strong guarantees. Instead of activating COVERUP whenever a deniable up- or download channel is needed, COVERUP gives the highest degree of privacy if users let it run in the background. For asynchronous up- and downloads, COVERUP can just up- and download opportunistically, whenever an active participant is anyway visiting the entry server. Moreover, the wider COVERUP is deployed, the lower is the need of a user to adapt its behavior to gain more throughput.

Even with imperfect behavior, this approach provides partial cover and delays a potential detection. First, consider the case where the browsing behavior of the active participants towards the entry server does not change. There, using COVERUP can provide deniability for the act of utilization. In contrast, a slightly altered user behavior leaks its *difference* to the unaltered behavior. However, this difference is smaller than the full leakage without COVERUP, as connecting directly to a service already reveals intention. Figure 1 illustrates this property by focusing on the participation time.

2.2 Challenges

We consider an attacker that controls the network but the user's machine and the Transfer server (the file server) are honest, and a dedicated party (the COVERUP server) that serves the protocol code for active and passive participants as a JavaScript snippet is honest but curious. Even in the presence of such an honest-but-curious COVERUP server and an honest Transfer server, the browser's processing time of active and passive participants can potentially leak information. This problem is amplified, since a network-level attacker can change the TCP flag for timestamps and compel the victim's operating system to add OS-level timestamps to the TCP headers [25]; hence, there is no hope of network-noise blurring the timing leakage. This leads us to three major challenges that we study in this work. (i) How to construct a *deniable* downlink connection that relays data to an external program with minimal timing leakage? (ii) How to relay data from an external program to a *deniable* uplink connection with minimal timing leakage. (iii) How long can such a system be safely used before the timing leakage renders active participants clearly distinguishable?

2.3 Non-goals

In the problem area of passive participation, two challenges remain that are out of scope of this work.

Behavior-changes towards the entry server. The usage of COVERUP may unconsciously influence the behavior of active participants, e.g. if active users spend more time on a specific entry server in order to use COVERUP. We believe, however, that these behavior changes do not cause a large amount of leakage as COVERUP is meant for asynchronous

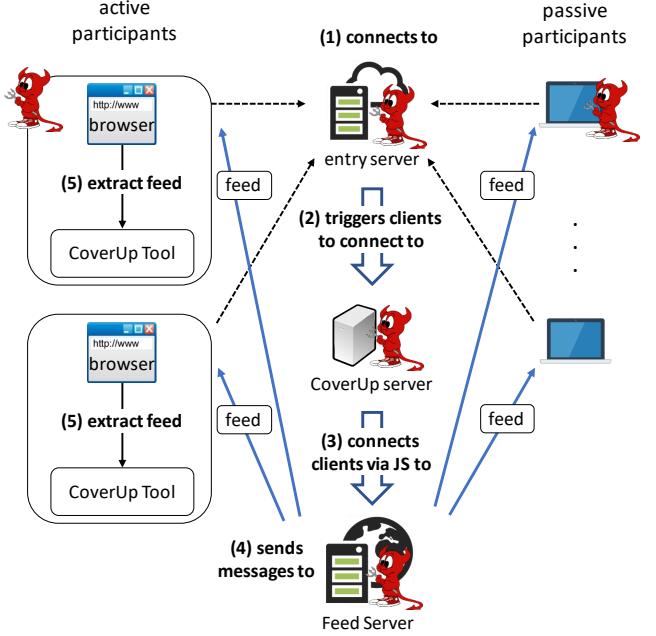


Figure 2: **Main components of COVERUP for CU:Feed.** All visitors of an entry server are redirected to the COVERUP server, triggered to send (dummy) requests to the Feed server, and then receive an encoded piece of a uni-directional message feed (4), which is extracted (5) by active participants via the COVERUP-Tool.

up- and download of files; hence, it is less prone to timing correlations (e.g., intersection attacks) than synchronous applications, such as messaging. As a consequence, the only source of leakage would be users that keep the tabs longer open in the background with COVERUP. Recent studies show that many users keep tabs open in the background anyway [54]; hence, COVERUP would not cause significant privacy leakage for these users. Properly understanding these behavior changes requires a thorough user study, which is out of scope of this work.

Browsing time of passive participants. Passive participants potentially reveal their browsing behavior to the COVERUP infrastructure, as a malicious server can read HTTP header's referrer field. While this leakage exists, we would like to put it into perspective. Many popular websites already leak this information to other services, such as advertisement networks or external analytic tools, such as Google Analytics. A deeper analysis of this leakage is out of scope.

3 COVERUP

Passive participation raises the challenge of utilizing passive participants to produce cover traffic with unintrusive technologies while asking for not more than an informed consent³ and while keeping the traffic of active and passive par-

³We discuss the challenges of an informed consent in Section 6.

ticipants indistinguishable. This section details how we overcome these technical challenges and presents the system design of COVERUP. We split COVERUP into two parts based on their features: a uni-directional broadcast-receiver channel and a fully bi-directional channel. We call them CU:Feed and CU:Transfer respectively.

3.1 CU:Feed

The uni-directional channel CU:Feed implements a deniable feed-receiver for a feed that is broadcast by a dedicated *Feed server*. CU:Feed triggers visitors (the *passive* participants) of cooperating websites (the *entry* server) to produce cover traffic, after they give an informed consent. CU:Feed leverages unintrusive widely-used JavaScript functionality of browsers. For *active* participants, which are interested in the feed, CU:Feed performs the same steps, but we additionally provide the external application COVERUP-Tool. With this application, the feed’s content can be extracted from the browser’s cache. As *active* users are indistinguishable from *passive* ones for all involved parties except their own machine, they cannot choose the feed they are listening to. Instead, the system constantly broadcasts its complete content piece-wise to everyone. The entry server could be a university, a knowledge, or a news site.

As illustrated in Figure 2, CU:Feed performs as follows:

- (1) The user connects to the entry server. The entry server embeds in its HTML-code an iframe to a dedicated server (the COVERUP server) from a different domain.
- (2) The COVERUP server responds with a JS code snippet.
- (3) This JS snippet triggers the browser of the entry server’s visitors to send requests to the *Feed server*.
- (4) The *Feed server* responds with CU:Feed packets. This effectively produces cover traffic to and from the Feed server. The COVERUP JS snippet then stores the most recent CU:Feed packet in the browser’s `localStorage` cache, thereby overwriting the old one.

The passive participants of COVERUP stop here. The rest of the protocol is only executed by the active participants.

- (5) An active participant uses a previously obtained external application (COVERUP-Tool⁴) to extracts these CU:Feed packets from the browser’s disk-based cache.

CU:Feed executes the same steps for active and passive participants except that active participants additionally install COVERUP-Tool on their computer to extract the feed. This makes the active and the passive participants indistinguishable to a network level adversary who does not compromise a user’s system. As the CU:Feed has no strict latency requirements, the browser behavior of active participants can

⁴COVERUP-Tool could be obtained off-the-record or as part of the CU:Feed. There, a small program including explanation could be distributed in clear text and without any encoding which could be extracted from the cache manually. This program assembles the full COVERUP-Tool delivered by the encoded feed.

be kept exactly the same, thus avoiding timing leakage.

With regard to the privacy of participants, the JS snippet from the COVERUP server is in an isolated context and thus can not learn anything from other contexts (including the page the iframe is embedded in) due to the SAME-ORIGIN-POLICY [8]. Hence, the COVERUP server can only learn when a participant visited the entry server, by the requests.

The content of the Feed could be controversial. To deflect potential legal harm to the passive participants, we cryptographically protect them from accidentally storing meaningful parts of the CU:Feed on their disc by utilizing an ALL-OR-NOTHING SCHEME [64] and only storing one CU:Feed packet in their `localStorage`. Without actively trying to, passive participants do not have sufficient packets collected to potentially reconstruct any content of the feed.

After applying the All-or-Nothing protection, we use error-correcting FOUNTAIN CODE (see Section 4.1) on the protected feed content. This splits the content in many packets and enables COVERUP-Tool to assemble these CU:Feed packets in an arbitrary order and with potentially missing packets, as the feed content might be too big for a single request. Thereby, the Feed server does not need to know which packet has reached a user and in which order. As there is no difference in feed packets for an *active* and *passive* participant, CU:Feed does not require TLS. The authenticity of the feed can be achieved by signing the content, assuming a PKI.

Trust assumptions and attacker capabilities. The CU:Feed is resistant against a global network-level active attacker that controls all parts of the system except the active participant’s hardware, operating system, and its running applications, as the only difference between active and passive participants is COVERUP-Tool that reads browser’s cache (`localStorage`). This attacker is active, so he can modify, drop or delay any number of messages, which includes the creation of an arbitrary number of participants – passive or active as individual participants are independent of each other. As we focus on guaranteed anonymity and not on integrity, COVERUP is not censorship resistant as it cannot protect from denial of service.

3.2 CU:Transfer

We extend CU:Feed to CU:Transfer, which enables user to upload content to and download content from a file server (Transfer server). Active CU:Transfer participants have to additionally install the COVERUP browser extension that establishes a channel to the external COVERUP-Tool, which can be used to upload and download content.

The protocol of CU:Transfer is almost the same as CU:Feed, except that users send (dummy) requests to the transfer server in a predictable pattern. While passive participants solely transmit dummy data and receive CU:Feed messages, active participants of CU:Transfer additionally send content messages whenever the user uses the system (Fig-

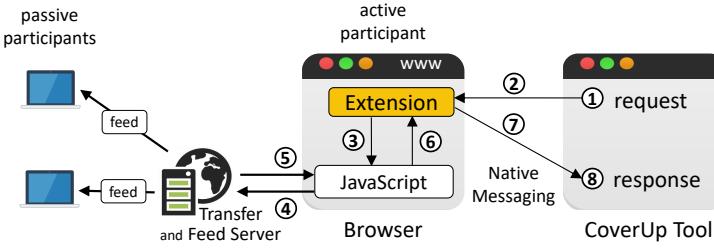


Figure 3, Step 1). In those cases, they use native messaging to connect from the COVERUP-Tool to the browser extension (Step 2). The browser extension then replaces a dummy CU:Feed request with a real message (Step 3). All messages (Step 4) are sent over a secure channel. Hence, messages of passive participants are indistinguishable from messages of active participants for a network-level adversary. Upon receiving the encrypted message (Step 5), the browser extension records it (Step 6), and sends it via native messaging (Step 7) to the COVERUP-Tool which decrypts it (Step 8).

As a result, both active and passive participants constantly send requests to the Transfer server, and the Transfer server responds with a constant-size data chunk; in particular, larger files are sent in smaller chunks. In general, the Transfer server does not need to be a centralized entity. Traffic sharing solutions (content distribution networks) could be used.

CU:Transfer trust assumptions. CU:Transfer is resistant against global network-level attacker that control all parts of the system except for the active participant’s machine, the COVERUP server, and the Transfer server. The COVERUP server has to be trusted because CU:Transfer relies on the integrity of the JS snippet; otherwise an attacker can inject malicious JS code that can detect active participants (e.g., by testing for the existence of the extension⁵). To enable the browser extension to check the integrity of the JavaScript snippet with minimal timing leakage, we trust the COVERUP server to be honest-but-curious for CU:Transfer and the browser extension simply checks whether the origin of the JavaScript code snippet is as expected. If the check fails, the browser extension does not hijack any packets.

We trust the Transfer server, as it can distinguish active and passive users based on their access pattern. Hiding access patterns is a non-trivial problem. Current solutions either require high communication complexity or are unsuitable for a bandwidth-limited multi-user setting [30, 69].

Relaxing trust assumptions. In the current implementation of COVERUP, we assume that the transfer server is trusted. As a result, the transfer server can distinguish between active and passive participants. This can be mitigated

Figure 3: CU:Transfer in combination with CU:Feed. Once the JS snippet has been received, all participants request CU:Feed packets. An active CU:Transfer participant can use the extension to replace these requests to the Transfer server with custom requests. To render the traffic from passive and active users indistinguishable, we use a secure channel at step 4 and 5, and at all connections by passive participants - in contrast to CU:Feed. For CU:Transfer the dummy messages do not need to contain feed content; they can also purely contain garbage.

using private information retrieval (PIR) methods such as the one used in Pung [30], PIR-tor [60], pynchon gate [66] and Riffle [53], or ORAMs such as PathORAM [69]. As most of these techniques, however, are computationally expensive, they significantly increase COVERUP’s overhead.

Alternatively, COVERUP can be used to strengthen Anonymous Communication Networks (ACNs) by rendering a user’s participation time deniable. In this scenario, the ACN takes the place of the transfer server. COVERUP only achieves deniability if the ACN does not leak whether a message is a dummy or a real message. As dummy messages do not have a recipient, the ACN has to make sure that they produce the same observable behavior as real user messages, while each user (be it active or passive) receives dummy or real messages according to a fixed distribution, e.g., a constant sending rate. In particular, COVERUP enforces this fixed distribution to ensure that the active participants’ traffic patterns are predictable by the JS code.

The COVERUP server can be untrusted if the extension checks the integrity of the JavaScript code byte for byte. This would eliminate any costs associated with running such an honest-but-curious COVERUP server, but the client-side timing leakage by such a solution would be significantly higher.

3.3 Timing leakage

Our design conceptually produces a timing leakage of active participants, compared to passive participants. For assessing the severity of the leakage, we characterize this timing leakage. This section discusses the timing leakage of our design, i.e., independent of the implementation. Section 4.1 discusses and Section 5 measures our implementation-specific timing leakage. As an active network-level attacker can activate the TCP timestamp-flag, we assume OS-level accuracy for the timestamps [25].

In CU:Feed active participants need to run COVERUP-Tool to extract information from the browser. While this application is external to the browser and does not directly interact with it, they share system-wide computation resources and scheduling slots, which influences the browser’s computation time. In CU:Transfer, the client additionally installs a browser extension and hence directly influences the browser’s computation time. In both cases, the timing pattern of the issued web requests is influenced (in the or-

⁵Modern browser claim to prevent any page-loaded JavaScript from checking for installed WebExtensions unless the extension wants to reveal itself. Specifically, any content-scripts run undetectable by page-loaded JS in an isolated context [4] and any access to resources of an extension must be allowed explicitly by the extension [19].

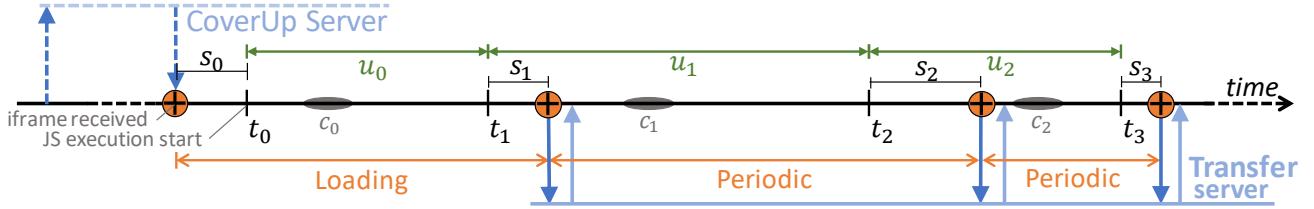


Figure 4: **The timeline of an active or passive participant in the browser**, starting at a request to the COVERUP Server for a JavaScript code snippet from an iframe. The code is executed and makes continuous requests to the Feed/Transfer server. The attacker can measure network timestamps of the requests (\oplus). To decrease the leakage s_i of the system or browser internals, we add randomly chosen delays u_i to the sending times t_i . There are two main sources of leakage: the set-up of the iframe context (Loading) and the interval between the consecutive requests (Periodic). Any comprehensive computation c_i inside the script or by the browser extension (for active participants) is done between the sending intervals when all components are idle.

u_i = artificially added noise
 t_i = XMLHttpRequest.send() call
 s_i = system noise
 c_i = computation inside the script
 \oplus = time-stamp measurement

der of milliseconds) and this is noticeable by a network-level attacker. While CU:Feed causes minor timing leakage, CU:Transfer causes significantly more timing leakage, even though the processing of the active message is separated from the critical sending part (c_i in Fig. 4).

This timing leakage cannot be countered by introducing deterministic delays, as a JavaScript program cannot measure the processing time of the systems outside of its context. Analogously, a JavaScript program cannot precisely enforce a delay. Therefore, we introduce random delays and show in Section 5 that these random delays significantly reduce timing leakage. To limit the amplification of the leakage, we additionally limit the number of requests for which the browser extension (of an active participant) is active. As we trust the browser and the extensions operates in a global context, this limit bounds the risk that malicious entry servers amplify the leakage by triggering excessive amounts of page-loads.

Figure 4 illustrates all potential observations of a network-layer attacker and the timeline of how messages are sent, received, processed in the browser, and when random delays (i.e., noise) are added. The system delay s_i in this figure refers to the system's computation time (including delays caused by the OS, the browser, and the network card). Any computation – and for CU:Transfer the communication with the extension – takes place in c_i with minimal interference.

In the rest of the paper, we concentrate on two time measurements that an attacker can perform: i) *Loading measurements* denote the time between the reception of the JavaScript snippet from the COVERUP server and the first outgoing request to the Feed/Transfer server, and ii) *Periodic measurements* denote the time between subsequent COVERUP requests to the Feed/Transfer server. For the CU:Transfer case, Figure 5 shows distributions of timing delays of active and passive participants for Loading and for Periodic measurements, without adding delays. It illustrates the importance of adding random delays; without these delays, already the naked eye can distinguish the distributions.

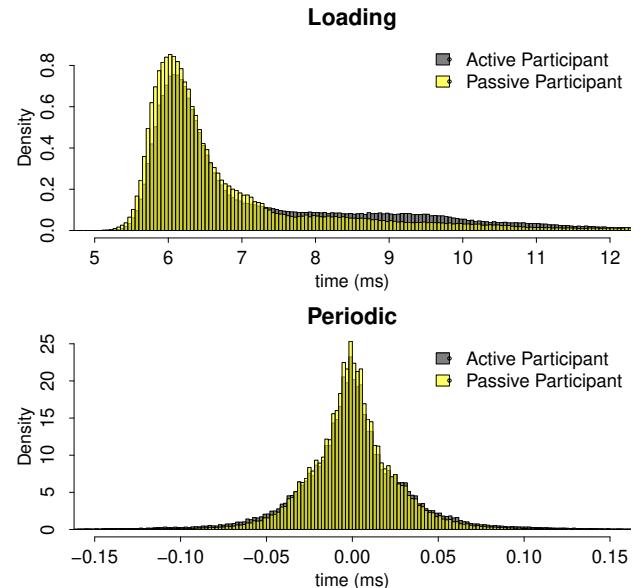


Figure 5: **Distribution of timing** (without additional noise) of Loading and Periodic measurements run on Linux. Each of the graphs overlays the timing distributions of active and passive participants. For the Periodic measurements, we substracted the expectation value (it is centered around 0).

4 Prototype & performance

This section describes the COVERUP prototype implementation (available under <http://coverup.tech>) and presents its performance. As the main purpose of the prototype is the timing leakage evaluation, it solely contains a dummy feed server and a dummy Transfer server.

4.1 Prototype implementation

Our prototype delivers a feed and the upload and download system, for which we implemented a high-latency mailbox.

The COVERUP implementation consists of five components: a COVERUP server, a central server that acts as the Transfer server (in our prototype the mailbox server), the message relay and the broadcaster, an external application (COVERUP-Tool), a browser extension, and a short JS code snippet. The COVERUP server and the Feed/Transfer server is implemented as a Java Servlet running on an Apache Tomcat web server. The external application is written in Java. The browser extension is implemented for the Google Chrome browser using the JS WebExtensions API. The COVERUP-Tool and the server implementation consists of about 14 KLoC and the browser extension of about 200 LoC.

We make the following three assumptions about the browser, which are in line with Chrome’s explicitly stated security policies. 1. iframes are isolated, which we need for the code integrity of COVERUP’s JS snippet. The parent page of the iframe cannot modify the iframe if the iframe is originated (domain) from a source other than the parent [6]. 2. A JS code cannot read from or write to another context of a different domain source without its consent. 3. The JS code can write a small amount of data to the browser’s localStorage cache and this cache cannot be accessed by another JS code which originates from a different origin. This property is known as the “same-origin-policy” [8], and all modern browsers claim to enforce it.

CU:Feed. The message feed in CU:Feed is encoded with a fountain code [57]. This encoding ensures that any out of order threshold amount of broadcast packet can recover the data successfully. Our prototype implementation uses an XOR based fountain code (for details see Appendix A.1). The JS snippet served by the COVERUP server stores the fountain pieces in the cache database file (known as browser `localStorage`). To minimize timing leakage, the COVERUP-Tool collects and assembles the fountain pieces from the `localStorage`. Our implementation also employs an All-or-Nothing-Encryption scheme (one similar to [64]) which ensures that one needs threshold-amount of pieces of the fountain (i.e. the entire source data) to decrypt it. The JS snippet only keeps one fountain piece in the `localStorage` to ensure that the passive users do not have any sensitive content on their disk in decipherable form.

CU:Transfer. CU:Feed’s extension CU:Transfer provides a up- and download channel for arbitrary data. The secure channel is implemented using TLS. The messages are of the same size and are transmitted at regular time intervals. Beyond padding (dummy) messages with random data, our prototype does not take additional measures against TLS meta-data leakage.⁶ The Transfer server uniquely identifies a sender/receiver of an incoming request using the unique SSL identifier without the overhead of sending an additional identification token. Uniquely identifying senders/receivers prevents session-hijacking attacks. For the mailbox protocol

example, we assume a PKI, and we indexed the messages as POP (post office protocol [26]) using *curve25519* [34] public keys (first 48 bits of the hashed public key). Whenever a new message arrives from a source address, the Transfer server assigns the message to the index of the destination address. When a request arrives for the destination address, the Transfer server delivers the message as the response and removes the message from the previously kept index location. The mailbox protocol assumes that an active participant added all long-term public keys of all his trusted peers.

4.2 COVERUP performance

COVERUP is suitable to real-world scenario, is feasible for deployment in large scale and does not incur an intolerable overhead. This section estimates COVERUP’s overhead, latency, and throughput. COVERUP has three adjustable system parameters: request payload size, response payload size and the average request frequency, the average requesting rate for CU:Feed packets after adding random delays. Increasing the payload increases the traffic overhead for passive participants, and a lower request frequency leaves room for higher random delays, thereby increasing privacy. Hence, there is a trade-off between latency, privacy, traffic overhead, and throughput. For our prototype, we choose system parameters (see Section 5 for more) such result in request/response payload sizes between 75 KB to 375 KB, and in sending a request every 60 seconds on average.

Computational overhead. The computational overhead of COVERUP’s JS executed in the Browser is negligible. Our implementation of the COVERUP-Tool takes around 50 MB of main memory and less than 1% CPU time. Similarly, the COVERUP browser extension incurs an almost unnoticeable amount of memory and CPU consumption.

Traffic overhead. The traffic overhead of CU:Feed and CU:Transfer is identical, as they are indistinguishable by design. The entry server’s overhead is minimal: only the size of the iframe tag in its HTML code. The passive participants’ traffic overhead depends on the system parameters. We based our estimation of the system parameters on the Alexa top 15 news sites, in particular since the privacy improvements of COVERUP’s passive-participation-approach depends on the entry server’s regular number of visitors. The average main-page load-size of the Alexa Top 15 news sites is around 2.2 MB and will grow in near future. A few examples are CNN (5.6MB), NYTimes (2.4MB), HuffingtonPost (6.1MB), TheGuardian (1.8MB), Forbes (5.5MB), BBC (1.1MB), and Reddit (0.8MB).

COVERUP is parametric in the packet size. Once fixed, the traffic overhead for the passive users is proportional to this packet length. We generously assume a passive participant that has a daily connected to the entry server for 5 hours each day. This participants would have 22MB ($\sim 5 \cdot 60 \cdot 60s \cdot \frac{1}{60s} \cdot 75KB$) to 110 MB ($\sim 5 \cdot 60 \cdot 60s \cdot \frac{1}{60s} \cdot 375KB$)

⁶There is work [61] that can prevent this leakage.

of data overhead per day and 660MB ($= 30 \cdot 22$ MB) to 3.3 GB ($= 30 \cdot 110$ MB) per month. For landline data flat-rates (i.e., for non-mobile visitors), 22 MB is not significant, e.g., in comparison to the traffic caused by streaming videos. We envision a deployment of COVERUP not to include mobile users. It may, however, be possible in the near future due to the increased bandwidth of the mobile networks.

Latency & throughput. We evaluate the performance of COVERUP for the duration that a tab is open, as the usage of COVERUP is bound to the visiting patterns of passive participants towards the entry server’s sites. Depending on the entry-server’s service, it might be common to keep the tab open (in the background) for a long time, to visit the site very often, or to switch to another entry server if multiple are available. COVERUP achieves 10 to 50 Kbits/s of throughput (for packet size system parameter 75 to 375 KB) and a latency of around 60 seconds on average between consecutive messages. As the future Internet infrastructure will evolve and website-size will increase, COVERUP’s packet sizes and thus the throughput can be increased.

Scalability. For the participants, the workload of the COVERUP channel itself is independent of the number of participants, and for the Transfer server the workload linearly increases. Hence, for the participants COVERUP scales well, and for the Transfer server an infrastructure at the scale of the entry server suffices, rendering COVERUP practical with current infrastructure.

5 Timing leakage experiments

We have set up an experiment that measured the timing leakage. The experiments produced histograms that we use as models to estimate the privacy leakage under continual observation. In the technical report [68], we rigorously prove that it suffices to analyze the timing leakage.

5.1 Experimental set-up

We assume that the dominant part of the timing leakage will be visible from the *Loading* and *Periodic* measurements, as depicted by the orange arrows in Figure 4. In Loading measurements, we force the iframe to refresh on the entry server page in the browser. In the corresponding TCP dump, we measure the timing difference between the response of the initial iframe HTML source request and its first (“passive”) request to the Feed/Transfer server. This forces to load the extension’s content script and thus captures any distinguishing feature (any timing delay added by the existence of the browser extension) produced by the extension.

The Periodic measurements model the scenario where the active and passive participants load the iframe once, followed by JavaScript generated periodic requests to the Feed/Transfer server and their response. In the network traffic dump, we look for the timing difference for two contiguous CU:Feed/CU:Transfer requests from the browser. Section 5.5 discusses the choice to concentrate on these measurements. For both cases, we compare the timing measurements of a passive participant and an active participant.

To simulate realistic scenarios, we set up the passive and both kinds (CU:Transfer and CU:Feed) of active participants on 12 identical systems running Windows 10 and Ubuntu 16.04 (both x86-64 and in dual-boot configuration) equipped with an Intel Core i5-2400 3.1 GHz CPU and 8 GB of main memory. Additionally, the COVERUP and a dummy implementation of a Feed/Transfer server run as an Apache Tomcat web server instance on a separate machine in the same sub-net connected by a 10 Gbps switch.

All the communication between the server and the browser are executed over a local Gigabit Ethernet network. We use *tshark* [28] to capture the network traffic on the participant’s network interface. We captured 3.8 million measurements in total. The experiments are conducted on these set-ups to investigate the timing leakage of the browser, produced by COVERUP’s browser extension and the COVERUP-Tool.

Reflecting the attacker model. Our attacker model (Section 3.1), a network-level attacker, is reflected in our experiments by capturing the traffic on the corresponding network interface. As an active network-level attacker can change the TCP flag for timestamps and compel the victim’s operating system to add timestamps to the TCP headers [25], the attacker does to gain strength by our setup where all participants, the COVERUP server, and the Feed/Transfer server are in the same GigaBit Ethernet switched network. We measured that the accuracy of the added OS-time-stamps is $4000\mu s$ for Linux, and $400\mu s$ for Windows, respectively.

Test modes. We emulate three different user scenarios by using combinations of the browser extension and the COVERUP-Tool. We use Google-Chrome browser v57.0 to run our extension which exchanges messages with the COVERUP-Tool. The three test modes include:

1. *Passive participant:* Google chrome with no extension and no COVERUP-Tool running.
2. *Active CU:Transfer participant:* Google Chrome with the extension installed and the COVERUP-Tool running which communicates with the aforementioned browser extension by the native messaging interface.
3. *Active CU:Feed participant:* Google chrome with no extension and COVERUP-Tool running assembling CU:Feed chunk from the browser `localStorage`.

These are repeated for both Loading and Periodic measurements (they are described in Section 5.3).

Interfering processes. Additionally we constructed one user profile in Linux to understand how the execution of other browsing tabs influences the timing leakage. To demonstrate a simple profile we additionally open another tab in the Google Chrome which is running a 720p video in a loop (see Figure 9).

Data sanitization. Our test setup was unstable with fre-

quently freezing machines (e.g., networkcard stopped working and power outages). We repeatedly ran the same set-up; hence, we expect the measurement-chunks generated from the same machine to be fairly consistent. While we kept significantly represented outliers, we measured 150 widely scattered outliers in 3 million measurements. These outliers are too few to be representatives of real effects. However, such widely scattered outliers distort our timing leakage analysis, since in theory real outlier effects that only happen in one configuration heavily amplify privacy leakage.

We removed these unrepresentative, scarcely scattered outliers to extract a representative model of the underlying response-delay distributions. To minimize the bias of the model, we dismissed entire batches of 6h measurements-blocks if they contained clear outliers w.r.t. the rest of the (sub-)histograms for the same scenario, e.g., periodic active participants. As a result, we dismissed 20% of all measurements, leaving us with 3 million measurements.

5.2 Adding random delays

COVERUP introduces random delays to reduce the timing leakage. To accelerate testing and increase accuracy, our experiments send requests at fixed intervals omitting the random delays. In COVERUP, the delays are chosen from a Gaussian distribution $\mathcal{N}_{[0,2\mu]}(\mu, \sigma)$ with mean μ and standard-deviation⁷ $\sigma = \frac{2}{10}\mu$, restricted to the interval $[0, 2\mu]$, and add this delay to the minimum delay of one second. The expected delay is therefore $\mathbb{E}[1 + \mathcal{N}_{[0,2\mu]}(\mu = \mu, \sigma = \frac{2}{10}\mu)] = 1 + \mu$. We artificially added delays after the measurements by convolving the resulting histograms with a gaussian distribution.⁸ We experimentally confirmed that separately adding the delays (see Appendix B) does not significantly distort our model.

5.3 Estimating the advantage

Our goal is to provide an upper bound on the advantage for the task of distinguishing active and passive participants. This section explains the estimators that we use. We assume that the dominant part of the timing leakage will be visible from two kinds of measurements: *Loading* and *Periodic* measurements, as depicted by the orange arrows in Figure 4.

To quantify the timing leakage we use a quantitative variant of statistical indistinguishability of two distributions. For a pair of distributions X, Y and a random sample either from X or from Y , statistical indistinguishability requires that no attacker can tell whether the sample was chosen from X or

⁷There is no specific reason for this σ , but we wanted to prevent hard noise-distribution cut-offs as they increase δ_n .

⁸If we view the histogram as the probability mass function (pmf) for the timing delays, convolving this pmf with a gaussian distribution the histogram corresponds to addition of the corresponding random variables, i.e., adding the noise within the experiment.

from Y with more than an advantage δ , which can be represented as follows: $\delta(X, Y) := \frac{1}{2} \sum_{a \in \Omega} (|p_X(a) - p_Y(a)|)$.⁹ Specifically, n collected observations amounts to considering $\delta_{n,\{X,Y\}} := \delta(X^n, Y^n)$ for the product distributions X^n, Y^n .

The advantage quantifies an attacker's success in distinguishing active from passive participants after n observations, while having perfect knowledge of underlying response-delay distributions of the active and passive participants of type $type \in \{loading, periodic\}$. Therefore, we write $\delta_{n,type}$ for the estimator for the attacker's advantage.

Our analysis relies on three assumptions. First, all the measurement samples are independent. Second, Loading and Periodic measurements are independent. Third, the measured distributions accurately represent the underlying distributions. We believe that the first two assumptions hold in a deployed system because we assume a very high waiting time between requests (around 60s). The third assumption is of theoretical nature. While we conducted extensive measurements (around 3 million measurements in total) to render the model more representative, such measurements can only result in an approximation of the underlying processes. Using standard composition results (see Appendix Lemma 1), these assumptions enable us to bound the advantage of COVERUP with $total_{n,m} := \delta_{n,loading} + \delta_{m,periodic}$, after attacker that makes n Loading observations and m Periodic observations for either Linux or Windows.

Under these assumption we use the Privacy-Buckets-tool [59]¹⁰ to compute $\delta_{n,loading}$ and $\delta_{n,periodic}$ from $\delta_{1,loading}$ and $\delta_{1,periodic}$ (for Linux and Windows, respectively), which we get from the sanitized measurement-histograms.

5.4 Timing leakage results

This section plots the results of our timing leakage estimation. For our evaluation, we over-approximate the connection pattern to the entry server with at most 50 site-loads and at most 5 hours of left-open tabs (in the background) of visited entry servers per day. We consider an attacker that is able to continuously collect such data for half a year, i.e., 7 days a week for 26 weeks. We assume that the usage pattern of an active participant is identical to that of passive participants (see Section 5.5 for a discussion on visiting behavior). We stress that the our analysis also applies to a continuous observation over 2.5 years for users that only make 10 site-loads at the entry server per day and are connected for at most 1 hour per day to the entry server.

Latency vs timing leakage. Fixing the observation time to half a year and the connection pattern as described above, Figure 6 plots how $total_{n,m}$ increases with decreasing de-

⁹This advantage is also known as *total variation* or *statistical distance* and is connected to the classification-accuracy: $acc = (\delta/2) + 0.5$.

¹⁰A publicly available numerical tool that computes a provable upper bound for the advantage under continual observation of a given pair of discrete distributions.

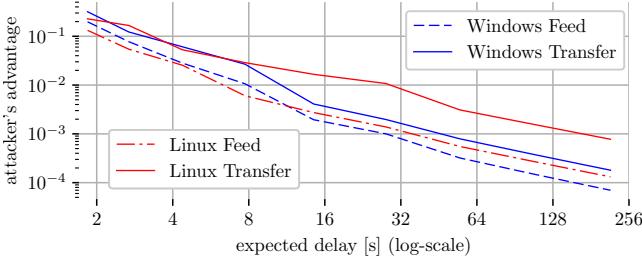


Figure 6: **Latency versus advantage** (upper bounded) for observation of half a year, with at most 5 hours of visiting the entry server (Periodic-observations) and at most 50 connecting to the entry server (Loading-observations) per day.

lays. Looking at the graph, we recommend 60s expected delay as system parameters to achieve an overall advantage of less than $2 \cdot 10^{-3}$ after 6 months of continual measurements of the user’s timing patterns with daily 50 Loading observations and daily 5 hours worth of Periodic observations. We stress that despite the limits of our evaluation, the bounds that we present are highly over-approximated: we assume a global network-level attacker that has very precise information about the state of the system such as which processes are running and how they influence the measurements.

Observation-length vs timing leakage. The next angle is the length of the observation versus the degree of privacy: Figure 7. We fix the expected latency to 60s and plot for an increasing number of observations the functions $\delta_{n,loading}$ and $\delta_{m,periodic}$. This graph lets us study different usage behaviors. E-mail service, such as Google mail or Hotmail, as an entry server, e.g., would lead to significantly longer sessions than e-commerce entry servers. E-mail services would, hence, lead to less Loading and more Periodic observations. This graph shows that the leakage grows at most linearly with the number of observations. While Loading needs more time in Linux for the CU:Transfer (presumably because it invokes the extension each time), it produces less Periodic leakage while running. The graphs show that in many cases the Feed produces more timing leakage than Transfer. We believe this discrepancy to be an artifact of the experiments. We nevertheless included these measurements in our analysis because we could not confidently exclude them.

Distorting effects of concurrent activities. The experiments of which we saw the results so far do not let any other program run in the background but doing so alters the histogram significantly. For further details see Appendix C.

5.5 Limits of our evaluation

This section discusses the limits of our evaluation of the leakage. While we do not claim that our evaluation offers provable bounds for the timing leakage of COVERUP, we believe that it captures the dominant part of the leakage of COVERUP and is a good indicator of the privacy that COVERUP offers.

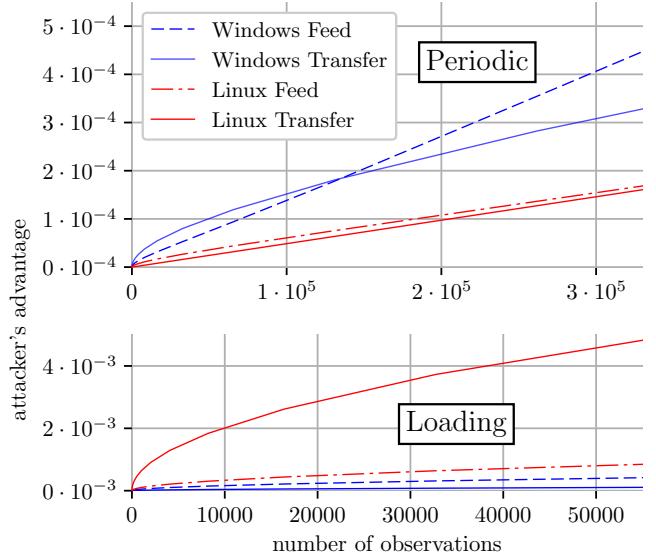


Figure 7: **Leakage over time.** The attacker’s advantage (y-axis, upper bound) over the number of observations (x-axis) for Periodic and Loading leakage with a 60s expected delay. The right end of the x-axes correspond to 3 years of observations.

Pairs of requests. We stick to pairs of requests since the autocorrelation is low and exploring all possible combinations for a higher number of contiguous requests increases the number of required measurements exponentially. To reduce potential effects from longer sequences of contiguous requests, we incorporate into our recommended delays a minimum of 1s between pairs of requests.

Unnoised measurements. We accelerated our measurements by not adding any additional noise, as we want to evaluate COVERUP with different amounts of noise. During the analysis phase, we introduce noise by computing the convolution of the resulting histograms with ideal Gaussian noise. To justify this we additionally construct an experiment with two scenarios: one with added artificial noise and another without where we add the artificial noise after the samples are collected. Figure 8 in Appendix B shows the timing distributions with total variation 1.8%.

Experimenting with real users. Evaluating our method against profiling attacks that are designed to detect whether a particular extension or a specific application is running [38] are out of scope of this work. Additionally, we do not evaluate COVERUP with real users to evaluate other aspects of the system such as reaction of passive participants, e.g., usage time of both the active and the passive users.

Neglecting the sampling error. Our experiments are limited to 3 million measurements. Hence, the histograms that we analyze do not exactly represent the underlying distribution. As our timing-leakage-bounds are computed on the histograms, they are not hard bounds but rather bounds that hold with high confidence.

6 Ethical, legal & deployment considerations

“Passive” participation has to be carefully implemented to avoid ethical and legal issues. We address potential ethical and legal considerations that stem from triggering visitors of some webpage into passively participating in a system like COVERUP. Our work received formal approval of our Institutional Review Board (IRB).

Even consenting passive participants that have been informed can experience unexpected consequences, e.g., by misunderstanding the consequences or by accidental consent. We are aware of the difficulties of informing website visitors in a way that they do not ignore the message and understand the consequences of consenting. Prior research [62, 33] suggests that the risks of misunderstanding and of agreeing by accident can be minimized, e.g. deny by default [29] and consenting in two phases, and highlight the network/battery-activities.

Are computation and bandwidth resources of passive participants unwittingly utilized? No, only after an informed consent does COVERUP turn an entry server visitor to a passive participant; hence utilizing the computation and bandwidth resources. Passive allocation of resources is nothing unexpected for a visitor of a webpage; it is regularly done by advertisements or QoS scripts, such as Google Analytics. Webpages that incorporate COVERUP would, hence, not cause unexpected behavior on a visitor’s browser. The computational overhead of COVERUP is negligible and the bandwidth overhead for a visitor is around 20.25 MB per day (for a throughput of 10 Kbit/s), which is negligible compared to the data load of video streaming services.¹¹

Does COVERUP violate a participant’s system-security? No, COVERUP uses standard browser functionality.

Does COVERUP store potentially incriminating data on the machine of passive participants? No, we carefully incorporated an All-or-Nothing scheme such that passive participants never contain any useful information on their machine, as long as they do not actively extract and collect the COVERUP data packets from the browser’s local storage.

Does COVERUP trigger passive participants to open potentially suspicious connections or connections that are detrimental for its reputation? After an informed consent, COVERUP does trigger a connection to the Feed/Transfer server (or an ACN), which some parties (e.g., an employer) could indeed view as suspicious or damage a passive participant’s reputation. We propose to mitigate this risk by only opening the connection after its nature was explicitly described and an informed consent was received.

Does the COVERUP server collect information about the browsing behavior of the entry server’s visitors? No, while each iframe request of every entry server’s visitor includes

¹¹We expect this bandwidth overhead to become an even smaller fraction of a user’s normal Internet traffic as connectivity improves and commercial websites continue to increase the amount of data that they sent.

the visitor’s IP address, an uncompromised COVERUP server does not collect or store this information in any form.

Who would volunteer to become passive participants? We carefully minimized the costs¹² and risks for passive participants, which minimizes the hurdle for visitors of the entry-server to altruistically support COVERUP. If the sense of urgency for hiding meta-data increases in the future and people are willing to pay for an ACN service, it is also possible to financially reward each entry-server visitor that consents to becoming a passive participant in COVERUP.

Which parties would benefit from deploying COVERUP? Apart from the normal beneficiaries of ACNs (e.g., whistleblowers, journalists or political activists), COVERUP is useful for government agencies that want to hide their agents’ tracks by using an ACN. This usefulness for government agencies could help with COVERUP’s deployment.

Legal considerations for passive participants. We carefully designed COVERUP such that the legal risk for passive participants is minimized. Even if illegal information is distributed via the feeds, the AON scheme ensures (cf. Section 3.1) that no information is ever reconstructible by an honest passive participant. Additionally, as COVERUP is not primarily designed for the purpose of committing a cyber-crime offense (in the legal sense). As solely standard browser functionality is utilized, receiving the COVERUP JS snippet is not a legal offense.

Legal considerations for the entry server. The JavaScript code is provided by a third party; hence, the entry server (its provider respectively) has no knowledge about the content. The liability for content of linked pages has been intensively debated in the past years. The liability of internet service providers has been debated intensively in the past years. Under EU and US legislation and case law, a provider’s *liability privilege* should apply to the entry server. As a result, the entry server should not be held liable for the JavaScript code and thus the content of the feed. For CU:Transfer, the entry server plays an even less significant role than a chat service provider; hence, the entry server is less liable than any chat service provider (e.g., WhatsApp) for chat-content.

Appendix D has a thorough discussion of legal topics.

7 Related work

Extending the anonymity set via JavaScript. There are previous research works on utilizing visitors of a collaborating website to produce anonymizing cover traffic via JavaScript. Conscript [41] and Adleaks [65] describes upload only uni-directional channel from the users to the mix network. In contrast, COVERUP provides a private bi-directional transport channel. Conscript mentioned timing leakage based side channel attacks but evaluation details are

¹²To further improve usability, a cookie can remember previous choices for consenting to support COVERUP as a passive participant.

missing except power consumption. Conscript additionally has deployment hurdles, since it trusts the entry server to achieve code integrity. While previous work suggests mitigating this trust assumption by letting the extension check all dynamic content to achieve code integrity against a malicious entry server, such dynamic checks will tremendously increase the timing leakage, and thus rendering the active participants clearly distinguishable from passive ones. The need to trust the entry server gives the entry server more responsibility and requires a careful evaluation of the entry servers. The implementation of Adleaks requires a patched version of the browser. This reduces the set of possible browsers and therefore reduces the anonymity set massively. Detailed privacy analysis is not described in the paper including timing leakages. The paper [32] describes how to include unwilling users to cover server to server communication. All transport between the servers (by passive clients) is not encrypted. This means an inspection of the HTTP body reveals intention. Moreover, the paper lacks any implementation details. Additionally, previous works lack a legal aspects discussion of “passive” participation.

Anonymous uploads and downloads. While COVERUP at its core provides a bi-directional transport channel on which ACNs could run, COVERUP has distinctly other goals than traditional ACNs or systems like Pung [30]: COVERUP’s goal is to enable users to hide their traffic in the traffic of normal web surfers, i.e., to extend the potential anonymity set to normal web surfers.

Covert channels & steganography. Covert channels hide whether communication took place, and thus achieve full deniability. As covert channels typically use a piggyback approach to transport data, they depend on existing data streams, resulting in a dependency of the piggybacked system for latency and throughput. Steganography is another approach which is hiding messages in unsuspicious looking data [52, 45, 31]. But once detected, the origin, thus the intention, is obvious. The same applies to Mixing [56]. Off-the-record messaging publishes the MAC key after each talk, rendering it vulnerable against real-time monitoring [35].

McPherson et al. proposed CovertCast, a broadcast hidden in normal video streams like YouTube [58]. Che et al. were able to create a deniable communication channel based on different levels of noisy channels [39]. Deploying that system is, however, require a much higher effort by the service provider (e.g., YouTube) and does not provide any interactive communication like COVERUP. Frewave [50] provides a covert channel where the user can modulate his internet traffic signal into acoustic data and transfer it to a remote server via VoIPs such as Skype. Such system has bandwidth limitation and is vulnerable to attacks described in [49]. SWEET [51] describes a covert channel e-mail communication where the user can send the query to the remote server by using any available mail server. Such system suffered from inherently very low bandwidth and high latency,

making them practically infeasible for deployment. Cloud-Transport [37] introduced covert communication which involves publicly accessible cloud servers such as Amazon S3 which acts as the oblivious mix. However, such services do not provide protection against attackers learning intention. Infranet [46] describes a system executing covert communication using image stenography, but it also suffers from inherently low bandwidth.

Censorship circumvention. There exist several censorship circumvention tools that allow users to reach websites which are otherwise unreachable due to local policies. Flash Proxies [47] provides a browser-based proxy that connects to a tor bridge. Its implementation uses WebSocket and JavaScript to create many, generally ephemeral bridge IP addresses, effectively surpassing the censor’s ability to block them. It is now outdated and replaced by Snowflake [27] which is a Tor pluggable transport [22] with a design principle identical to Flash Proxies. Other pluggable transports such as Tor’s meek [20] relay data through a third-party server that is hard to block, for example a CDN, using a mechanism called domain fronting [48]. COVERUP is orthogonal to the aforementioned papers. COVERUP does not provide any form of censorship circumvention, as the censor can disable COVERUP by blocking all requests to the entry server, the COVERUP server, or the feed/Transfer server.

8 Conclusion

We discussed how the concept of passive participation can improve the privacy of accessing information in an anonymous and deniable manner. By drawing in passive participants to create cover traffic, we achieve participation deniability: an attacker cannot tell whether an observed request to a Feed/Transfer server originates from a active participant which is interested in its content, or from a passive participant which is only surfing on the entry server.

We leverage this concept with COVERUP, which can operate in two modes: CU:Feed, distributing an uni-directional broadcast, and CU:Transfer, providing a deniable up- and download channel. Given our implementation, we experimentally evaluated the degree of privacy COVERUP can guarantee. For both, CU:Transfer and CU:Feed, we found that the timing leakage is acceptable (an advantage under $2 \cdot 10^{-3}$) within a half a year of continual observation. Even for a state-level agency a half a year of continual observation (on sub-ms-level granularity) incurs a significant cost.

The present analysis clearly shows that the passive-participation-approach can provide sufficient cover. We conclude that research on passive participation is a promising direction for deniable communication.

Acknowledgements: This work has been partially supported by the Zurich Information Security Center (ZISC). We thank the anonymous reviewers for their helpful comments.

References

- [1] 17 u.s. code para. 512. <https://www.law.cornell.edu/uscode/text/17/512>.
- [2] America's founding documents — national archives. <https://www.archives.gov/founding-docs>.
- [3] Chart of signatures and ratifications of treaty 185. <http://tinyurl.com/h8ketgj>.
- [4] Chrome scripts - google chrome. https://developer.chrome.com/extensions/content_scripts#execution-environment.
- [5] Consolidated version of the treaty on the functioning of the european union. http://eur-lex.europa.eu/resource.html?uri=cellar:41f89a28-1fc6-4c92-b1c8-03327d1b1ecc.0007.02/DOC_1&format=PDF.
- [6] Content security policy (csp) - google chrome. <https://developer.chrome.com/extensions/contentSecurityPolicy>.
- [7] Convention on cybercrime, budapest, 23.xi.2001. http://www.europarl.europa.eu/meetdocs/2014_2019/documents/libe/dv/7_conv_budapest/_7_conv_budapest_en.pdf.
- [8] Cross origin xmlhttprequest - google chrome. <https://developer.chrome.com/extensions/xhr>.
- [9] Directive 2000/31/EC of the European Parliament and of the Council of 8 June 2000 on certain legal aspects of information society services, in particular electronic commerce, in the Internal Market ('Directive on electronic commerce'), 2000 O.J. L 178.
- [10] Directive 2002/22/ec of the european parliament and of the council. <http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32002L0022&from=EN>.
- [11] Directive 2002/58/ec of the european parliament and of the council. <http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32002L0058:en:PDF>.
- [12] Directive 2009/136/ec of the european parliament and of the council. <http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2009:337:0011:0036:en:PDF>.
- [13] Directive 95/46/ec of the european parliament and of the council. <http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31995L0046&from=EN>.
- [14] Eur lex. <http://eur-lex.europa.eu/legal-content/EN/ALL/?uri=OJ%3AC%3A2012%3A326%3ATOC>.
- [15] European convention on human rights (echr). http://www.echr.coe.int/Documents/Convention_ENG.pdf.
- [16] Federal constitution of the swiss confederation. <https://www.admin.ch/opc/en/classified-compilation/19995395/index.html>.
- [17] Fourth amendment. https://www.law.cornell.edu/constitution/fourth_amendment.
- [18] Katz v. united states, 389 u.s. 347 (1967). <https://supreme.justia.com/cases/federal/us/389/347/case.html>.
- [19] Manifest: Web accessable resources - google chrome. https://developer.chrome.com/extensions/manifest/web_accessible_resources.
- [20] meek: Tor bug tracker and wiki. <https://trac.torproject.org/projects/tor/wiki/doc/meek>.
- [21] Olmstead v. united states, 277 u.s. 438 (1928). <https://supreme.justia.com/cases/federal/us/277/438/case.html>.
- [22] Pluggable transports. <https://trac.torproject.org/projects/tor/wiki/doc/PluggableTransports>.
- [23] Regulation (ec) no 2006/2004 of the european parliament and of the council. <http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32004R2006&from=EN>.
- [24] Regulation (ec) no 2006/679 of the european parliament and of the council. <http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=en>.
- [25] Rfc 7323 - tcp extensions for high performance. <https://tools.ietf.org/html/rfc7323>.
- [26] Rfc 918 - post office protocol. <https://tools.ietf.org/html/rfc918>.
- [27] Snowflake. <https://trac.torproject.org/projects/tor/wiki/doc/Snowflake>.
- [28] tshark-the wireshark network analyzer 2.0.0. <https://www.wireshark.org/docs/man-pages/tshark.html>.
- [29] ACQUISTI, A. Nudging privacy: The behavioral economics of personal information. *IEEE Security Privacy* (2009).
- [30] ANGEL, S., AND SETTY, S. T. Unobservable communication over fully untrusted infrastructure. In *OSDI* (2016).
- [31] ARTZ, D. Digital steganography: hiding data within data. *IEEE Internet computing* 5, 3 (2001), 75–80.
- [32] BAUER, M. New covert channels in http: Adding unwitting web browsers to anonymity sets. In *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society* (2003), WPES '03.
- [33] BEAUDOUIN-LAFON, M. Designing interaction, not interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04.
- [34] BERNSTEIN, D. J. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds.
- [35] BONNEAU, J., AND MORRISON, A. Finite-state security analysis of otr version 2, 2006.
- [36] BOYKO, V. On the security properties of oaep as an all-or-nothing transform. In *Annual International Cryptology Conference* (1999), Springer, pp. 503–518.
- [37] BRUBAKER, C., HOUMANSADR, A., AND SHMATIKOV, V. Cloud-transport: Using cloud storage for censorship-resistant networking. In *International Symposium on Privacy Enhancing Technologies Symposium* (2014).
- [38] CAO, Y., LI, S., AND WIJMANS, E. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *NDSS 2017*.
- [39] CHE, P. H., BAKSHI, M., AND JAGGI, S. Reliable deniable communication: Hiding messages in noise. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*.
- [40] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *S&P 2015*.
- [41] CORRIGAN-GIBBS, H., AND FORD, B. Conscript Your Friends into Larger Anonymity Sets with JavaScript. In *WPES 2013*.
- [42] DAEMEN, J., AND RIJMDEN, V. *The design of Rijndael: AES-the advanced encryption standard*. 2013.
- [43] DANEZIS, G., AND SERJANTOV, A. Statistical disclosure or intersection attacks on anonymity systems. In *Information Hiding*.
- [44] DANEZIS, G., AND SERJANTOV, A. Statistical disclosure or intersection attacks on anonymity systems. In *International Workshop on Information Hiding* (2004), Springer, pp. 293–308.
- [45] EGGRERS, J. J., BAEUML, R., AND GIROD, B. Communications approach to image steganography. In *Security and Watermarking of Multimedia Contents IV* (2002), vol. 4675, International Society for Optics and Photonics, pp. 26–38.

- [46] FEAMSTER, N., BALAZINSKA, M., HARFST, G., BALAKRISHNAN, H., AND KARGER, D. R. Infranet: Circumventing web censorship and surveillance. In *USENIX Security Symposium* (2002).
- [47] FIFIELD, D., HARDISON, N., ELLITHORPE, J., STARK, E., BONEH, D., DINGLEDINE, R., AND PORRAS, P. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies* (2012), Springer Berlin Heidelberg.
- [48] FIFIELD, D., LAN, C., HYNES, R., WEGMANN, P., AND PAXSON, V. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies* (2015).
- [49] GEDDES, J., SCHUCHARD, M., AND HOPPER, N. Cover your acks: Pitfalls of covert channel censorship circumvention. In *PCCS 2013*.
- [50] HOUmansadr, A., RIEDL, T. J., BORISOV, N., AND SINGER, A. C. I want my voice to be heard: Ip over voice-over-ip for unobservable censorship circumvention. In *NDSS* (2013).
- [51] HOUmansadr, A., ZHOU, W., CAESAR, M., AND BORISOV, N. Sweet: Serving the web by exploiting email tunnels. *IEEE/ACM Transactions on Networking (TON)* 25, 3 (2017), 1517–1527.
- [52] KAMBLE, M. P. R., WAGHAMODE, M. P. S., GAIKWAD, M. V. S., AND HOGADE, M. G. B. Steganography techniques: A review. *International Journal of Engineering* (2013).
- [53] KWON, A., LAZAR, D., DEVADAS, S., AND FORD, B. Riffle. *Proceedings on Privacy Enhancing Technologies* (2016).
- [54] LABAJ, M., AND BIELIKOVÁ, M. Tabbed Browsing Behavior as a Source for User Modeling. In *User Modeling, Adaptation, and Personalization* (2013).
- [55] LANDAU, S. Making sense from snowden: What’s significant in the nsa surveillance revelations. *IEEE Security Privacy* (2013).
- [56] LE BLOND, S., CHOFFNES, D., ZHOU, W., DRUSCHEL, P., BAL-LANI, H., AND FRANCIS, P. Towards efficient traffic-analysis resistant anonymity networks. In *ACM SIGCOMM Computer Communication Review* (2013).
- [57] MACKAY, D. J. Fountain codes. *IEE Proceedings-Communications* 152, 6 (2005), 1062–1068.
- [58] MCPHERSON, R., HOUmansadr, A., AND SHMATIKOV, V. Covertcast: Using live streaming to evade internet censorship. *Proceedings on Privacy Enhancing Technologies* (2016).
- [59] MEISER, S., AND MOHAMMADI, E. Tight on Budget? Tight Bounds for r-Fold Approximate Differential Privacy. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)* (2018), ACM.
- [60] MITTAL, P., OLUMOFIN, F. G., TRONCOSO, C., BORISOV, N., AND GOLDBERG, I. Pir-tor: Scalable anonymous communication using private information retrieval. In *USENIX Security Symposium* (2011), p. 31.
- [61] NIKITIN, K., BARMAN, L., UNDERWOOD, M., AND FORD, B. Reducing metadata leakage from encrypted files and communication with purbs. *arXiv preprint arXiv:1806.03160* (2018).
- [62] PATRICK, A. S., AND KENNY, S. From privacy legislation to interface design: Implementing information privacy in human-computer interactions. In *Privacy Enhancing Technologies* (2003), R. Dingle-dine, Ed.
- [63] PIOTROWSKA, A. M., HAYES, J., ELAHI, T., MEISER, S., AND DANEZIS, G. The loopix anonymity system. In *26th USENIX Security Symposium, USENIX Security* (2017), pp. 16–18.
- [64] RIVEST, R. L. All-or-nothing encryption and the package transform. In *Fast Software Encryption* (Berlin, Heidelberg, 1997), E. Biham, Ed., Springer Berlin Heidelberg.
- [65] ROTH, V., GÜLDENRING, B., RIEFFEL, E., DIETRICH, S., AND RIES, L. A Secure Submission System for Online Whistleblowing Platforms. In *FC 2013*.
- [66] SASSAMAN, L., COHEN, B., AND MATHEWSON, N. The pynchon gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society* (2005), ACM.
- [67] SHOKROLLAHI, A. Raptor codes. *IEEE/ACM Transactions on Networking (TON)* 14, SI (2006), 2551–2567.
- [68] SOMMER, D., DHAR, A., MOHAMMADI, E., RONZANI, D., AND CAPKUN, S. Deniable Upload and Download via Passive Participation. Cryptology ePrint Archive, Report 2017/191, 2017.
- [69] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C. W., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS 2013* (2013).
- [70] SUNDARARAJAN, J. K., SHAH, D., AND MÉDARD, M. Arq for network coding. In *ISIT 2008*.
- [71] VAN DEN HOOFF, J., LAZAR, D., ZAHARIA, M., AND ZELDOVICH, N. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015).
- [72] WEBER, R. *E-Commerce und Recht*, 2. Auflage. 2010.
- [73] WOLINSKY, D. I., CORRIGAN-GIBBS, H., FORD, B., AND JOHNSON, A. Dissent in numbers: Making strong anonymity scale. In *OSDI* (2012), pp. 179–182.

A Constructions

In this section we describe existing tools and techniques that have been used in our proposed system COVERUP.

A.1 Fountain Code

Fountain codes [57, 70] are a class of forward error correction (FEC) codes with the following properties

- Arbitrary sequence of encoding symbols can be generated form a given set of source symbols i.e., input data.
- Original source symbols can be recovered from any subset of encoding symbols with size more than a threshold value T .
- Encoding symbols can be delivered regardless of specific order.
- Fountain codes does not show fixed code rate.

In this paper, we have used a bit-wise XOR (\oplus) based fountain code with error detection mechanism.

In a simple analogy, one can consider an empty glass for water. A fountain emits the input data encoded in a large amount of droplets in a steady stream. Anyone can collect them in a glass alternately and if one thinks the glass is filled enough, one may try to assemble the data from the water (data stored in the glass). If the amount of droplets is insufficient to reassemble the data, one has to wait longer to collect more droplets and retries later.

Our specific fountain code implementation is not optimal. There exists efficient fountain codes such as *Raptor* [67] in the literature but most of them are protected by intellectual property rights.

A.2 All-or-nothing transformation

All-or-nothing transformation is an encryption mode in which the data only can be decrypted if all the encrypted data is known. More precisely: “An AONT is an un-keyed, invertible, randomized transformation, with the property that it is hard to invert unless all of the output is known.”[36].

We modified the *all-or-nothing scheme* proposed by Rivest [64] which encrypts all data with a symmetric key cryptography algorithm (in our implementation, we use AES-128 [42]) in Cipher Block Chaining (CBC) mode and appends a new block in which the encryption key is XOR’ed (\oplus) with the 128 bit truncated SHA-256 hashes of all the encrypted blocks. This guarantees that one needs all encrypted data (or at least its hash) to extract the decryption key from last block.

1. Input message block: m_1, m_2, \dots, m_n
2. Choose random key $\mathcal{K} \xleftarrow{R} \{0, 1\}^{128}$ for AES-128.
3. Compute output text sequence $m'_1, m'_2, \dots, m'_n, m'_{\text{key}}$ as follows:
 - Let $m'_i = \text{Enc}(\mathcal{K}, m_i) \forall i \in 1, \dots, n$ with CBC mode.
 - Let $m'_{\text{key}} = \mathcal{K} \oplus h_1 \oplus h_2 \oplus \dots \oplus h_n$
where $h_i = \mathcal{H}_i[1, \dots, 128]; \mathcal{H}_i = \text{SHA-256}(m_i) \forall i \in 1, \dots, n$
 - Send $m' = m'_1 || \dots || m'_n || m'_{\text{key}}$

The receiver can recover the key \mathcal{K} only after receiving all message blocks. He executes the following steps

- $\mathcal{K} = m'_{\text{key}} \oplus h_1 \oplus h_2 \oplus \dots \oplus h_n$.
- $m_i = \text{Dec}(\mathcal{K}, m'_i) \forall i \in 1, \dots, n$.

B Independence of additional noise

Recall that we simulated the additional noise by adding it to the measurement result. To justify this procedure, we conducted separate experiments, similar to the periodic scenario, but instead of waiting 1000ms for the next droplet request, we drew in JavaScript a uniformly distributed random number (using `Math.random()`) and expanded it in an affine way such that an interval ranges from 200ms to 1800ms. Additionally, we stored each of the drawn random numbers together with an epoch time stamp. Later in the analysis step, we subtracted the corresponding random number from the network dump measurement. This procedure produced measurements artifacts, caused by the time resolution of our system (which lies slightly under 1us). As we are only interested in the fact whether artificially adding the noise after the experiment is independent of directly adding the additional noise in the experiments, we clustered close histogram bars that are not separated by a significant gap. Figure 8 shows the resulting distribution. The statistical distance of these two distributions is 1.8% which is an acceptable value.

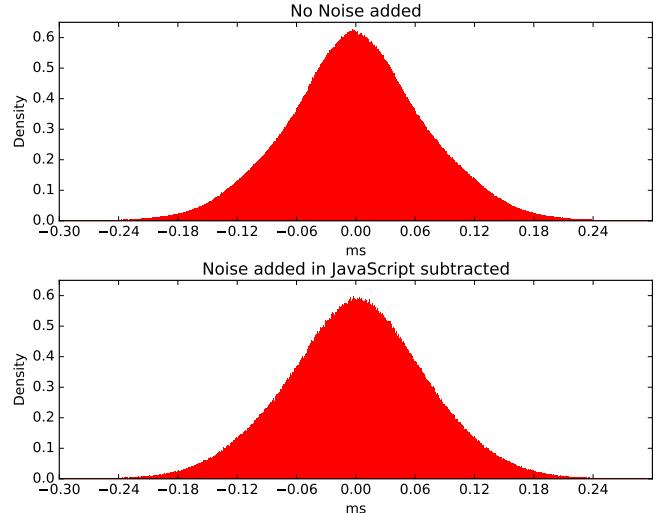


Figure 8: **Statistical Independence** using uniform noise: Distance: 1.8%

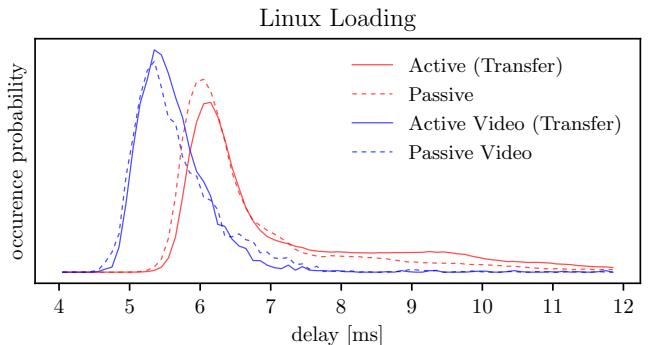


Figure 9: **Different computation loads** lead to different timing distributions. In the blue video plots, Google Chrome additionally renders a high definition (720p) video in a separate tab. Loading measurement. No randomly chosen delays added.

C Impact of concurrent activities

The experiments of which we saw the results so far do not let any other program run in the background. In contrast, Figure 9 overlays the histogram of the vanilla experiments (without any other programs running in the background) and experiments where the browser is rendering a 720p video on Linux. The experiments are conducted with Loading observations, as those produce more leakage. We can clearly see that rendering the video has some impact on the measurement (red line vs. blue line in Figure 9). Hence, it will be hard for an attacker to get such clean measurements like those that we use in our evaluation. This is another reason why we have some confidence that our privacy bounds give a good impression of the degree of privacy that COVERUP can offer, and maybe even provide a significant over-approximation.

D Selected legal questions

One of the challenges in answering the question whether the provision of COVERUP and the upload of the JavaScript code by the entry server is legal or not (and many other questions evolving around the use of the Internet) is that, whereas the Internet functions globally, law mostly [7] remains limited by territory because sovereign states put their own legislation into effect [5, 14, 2]. The legal provisions and possible offenses that apply to the technical setup of COVERUP, differ from country to country. Moreover, as law is not an exact science and definite legal statements are made by the courts, we conclude the legal discussion herein with an assessment that we consider probable.

Many countries enforce their own laws and have their own (territorial) jurisdiction, many countries, among others the EU member states and the USA, have ratified [3] in the Convention on Cybercrime [7] (CCC) – the international treaty on crimes committed via the Internet and other computer networks. This international treaty criminalizes, among others, illegal access (Art. 2 CCC), data interference (Art. 4 CCC), and misuse of devices (Art. 6 CCC).

D.1 Passive participants

Illegal access. Illegal access (Art. 2 CCC) penalizes the entering of a computer system but does not include the mere sending of an e-mail message or a file to a system. The application of standard tools provided for in the commonly applied communication protocols and programs is not per se “without right”, in particular not if the accessing application can be considered to have been accepted (e.g. acceptance of cookies [12, 10, 11, 23] by client). However, a broad interpretation of Art. 2 CCC is not undisputed (refer [7], §44 - 50).

Upon request, the entry server delivers a webpage that contains an iframe request for the COVERUP server, which then delivers the JavaScript to the browser for the download of the packet. Not only does the entry server merely send a file (pointer) to the browser, but the request to download the JavaScript from the COVERUP server is standard browser functionality for communication. The same would happen if the entry server were financed by online advertising: upon request the entry server would deliver a webpage pointing to the advertising server and trigger the download of the advertising text or pictures to the browser. As this is a standard online process, we conclude that even in a broad interpretation of Art. 2 CCC, the provider of the entry server should not be illegally accessing the browser.

Data interference. Data interference (Art. 4 CCC) penalizes the damaging, deletion, deterioration, alteration, or suppression of computer data “without right”. This provision protects a computer device from the input of malicious code, such as viruses and Trojan horses as well as the result-

ing alteration of data. However, the modification of traffic data for the purpose of facilitating anonymous communications should in principle be considered legitimate protection of privacy (refer [15, 17, 21, 18], [13, Recitals(1) and (35)], [16, Art. 13], and, therefore, be considered as being undertaken “with right” [7, §61].

COVERUP does not damage, delete, deteriorate, or suppress data on the participant’s client. However, it does alter the data on the hard disk: on the one hand the webpage with the iframe uses disk space and thus modifies the participant’s data; on the other hand COVERUP triggers the download of the JavaScript code and subsequently the packets from the ACN to the passive participant’s browser, which again uses disk space and thus modifies the data anew.

However the explanatory report to the Convention on Cybercrime foresees that the file causing data interference be “malicious”. Code is malicious if it executes harmful functions or if the functions are undesirable.

As concluded above, the JavaScript code utilized standard core browser functionality. Thus from a technical viewpoint, COVERUP is not harmful. Therefore in our view the provider of the entry server not does cause any malicious data interference. We advocate that Art. 4 should not apply to the provision of the webpage with the iframe by the provider of the entry server.

Misuse of devices. Misuse of devices (Art. 6 CCC) penalizes the production, making available, or distribution of a code designed or adapted primarily for the purpose of committing a cybercrime offense, or the possession of such a computer program. It refers to the commission of “hacker tools”, i.e. programs that are e.g. designed to alter or even destroy data or interfere with the operation of systems, such as virus programs, or programs designed or adapted to gain access to computer systems. The objective element of offense comprises several activities, e.g. distribution of such code (i.e. the active act of forwarding data to others), or making code available (i.e. placing online devices or hyperlinks to such devices for the use by others) [3, §72].

One of the main questions relating to the misuse of devices is how to handle dual use devices (code). Dual use means in our case that the JavaScript code could be used to download legal content, e.g. political information, as well as illegal content, e.g. child pornography. Should Art. 6 CCC only criminalize the distribution or making available of code that is exclusively written to commit offenses or should it include all code, even if produced and distributed legally? Art. 6 CCC restricts the scope to cases where the code is objectively designed primarily for the purpose of committing an offense, usually excluding dual-use devices [3, §72–§73].

First, it is important to note that COVERUP was not designed primarily for the purpose of committing an offense. While the main purpose of COVERUP is to protect privacy, it can be used to conceal illegal activities. Second, can the download of criminal information be considered an illegal

activity if the information is encrypted? Here we draw a legal analogy to data protection law. Data relating to an identified or identifiable person is considered personal data [13, Art. 2(a)], [24, Art. 4(1)]. If a person is identifiable or identified, data protection law applies. However, if the personal data are pseudonymized or anonymized, then data protection law might not apply anymore because the (formerly identifiable or identified) person cannot longer be identified.

Recital (83), Art. 6(4)(e), 32(1)(a) and 34(3)(a) of the new General Data Protection Regulation¹³ stipulate that encryption renders the personal data unintelligible and mitigates the risk of infringing the new regulation.

By applying this data protection principle to the encryption of data by COVERUP we can argue that the data provided by the ACN in the packets are not information because the data is unintelligible. Not only does the passive participant not have sufficient data to reassemble the packet to a whole, but the data are encrypted in such manner that it is impossible to make any sense of it. At least from a theoretical viewpoint the encryption of COVERUP cannot be breached. We therefore conclude that the JavaScript code, with regard to the passive participant, does not qualify as dual use device even if it is used for illegal purpose. The data transmitted remain unintelligible and therefore do not qualify as information. However, the JavaScript code, with regard to the active participant, can be qualified as dual use device because the encrypted and unintelligible data are decrypted and reassembled to intelligible information.

Legal conclusion. We discussed the applicability of Art. 2 (illegal access), 4 (data interference), and 6 (misuse of device) CCC to COVERUP. We conclude that the provider of the entry server is probably not illegally accessing the participant's browser by applying COVERUP; that the provider of the entry server probably does not cause any malicious data interference; and that the use of COVERUP with regard to the passive participant does not qualify as misuse of device. In regard to the reassembly of the packets to a meaningful whole, if the information is illegal, COVERUP might qualify as dual use device and fall under Art. 6 CCC. We conclude that at least with regard to the risk of indictment pursuant to Art. 6 CCC it seems advisable that the provider of the entry server does not provide the JavaScript code for download.

D.2 Entry servers

A participant is dependent on Internet service providers ISP. The question arises whether an (ISP) should be liable for illegal Internet activities of its subscribers. In the following we discuss legislation and case law on the ISP's liability in two different jurisdictions: the EU and the USA. For this discussion it is important to differentiate among the various types of ISPs, for instance access providers, hosting providers, and content providers [72].

¹³Regulation (EU), applicable as of 25.5.2018

European union. In the European Union, liability of ISPs has been regulated in the E-Commerce Directive [9]. Generally, providers shall not have any obligation to monitor the information which they transmit or store, or to seek actively facts or circumstances indicating illegal activity [9, Art. 15 (1)]. According to the directive, access providers acting as "mere conduits" shall not be liable for the information transmitted, on the condition that they do not initiate, select the receiver of, or select or modify the information contained in the transmission [9, Art. 12 (1)].¹⁴ Caching providers (efficiency transmitters) shall not be liable for the automatic, intermediate and temporary storage of information, on the condition that they do not modify the information; comply with access regulations and industry standards for updating the information; do not interfere with the lawful use of technology; and act expeditiously to remove information if removed from the initial source [9, Art. 13 (1)]. Hosting providers shall not be liable for the information stored on their servers, on the condition that they are unaware of illegal activity or information or acts expeditiously to remove or disable access to the illegal information [9, Art. 14 (1)].

With regard to the obligations of a hosting provider, the European Court of Justice decided in SABAM v Netlog¹⁵ that, among other directives, the E-Commerce Directive precluded a national court from issuing an injunction against a hosting service provider which requires it to install a system for filtering (a) information which is stored on its servers by its service users, (b) which applies indiscriminately to all of those users; (c) as a preventative measure; (d) exclusively at its expense; and (e) for an unlimited period; which is capable of identifying IP-infringing content.

USA. Similarly, in the United States there are limitations on liability relating to material online [1]. There are statutory limitations for transitory communications (i.e. access provider, "mere conduit") [1, Section 512(a)], system caching (i.e. storage for limited time) [1, Section 512(b)], information residing on systems or networks at the direction of users (i.e. hosting) [1, Section 512(c)], and information location tools (i.e. search engines or hyperlinking) [1, Section 512(d)].

With regard to the obligations of a hosting provider [1, Section 512(c)], the United States Court of Appeals for the Second Circuit, by referencing UMG Recordings, Inc. v. Shelter Capital Partners LLC, 667 F.3d 1022 (9th Cir. 2011), argued that "*[t]he Court of Appeals affirmed [...] that the website operator was entitled to safe harbor protection. With respect to the actual knowledge provision, the panel declined to 'adopt [...] a broad conception of the knowledge requirement,' id. at 1038, holding instead that the safe harbor '[r]equir[es] specific knowledge of particular infringing ac-*

¹⁴With regard to the German liability for interference ("Störerhaftung") according to Sommer unseres Lebens (I ZR 121/08), see also decision by the ECJ in Mc Fadden (C- 484/14).

¹⁵ECJ C-360/10.

tivity,’ *id. at 1037*. The Court of Appeals reach[ed] the same conclusion’ [..] noting that [w]e do not place the burden of determining whether [materials] are actually illegal on a service provider.’ *Id. At 1038 (alterations in original) (quoting Perfect 10, Inc. v. CCBill LLC, 488 F.3d 1102, 1114 (9th Cir. 2007))*’. Hence, the 2nd Circuit Court concluded, among others, that 17 U.S.C. §512(c)(1)(A) requires knowledge or awareness of facts or circumstances that indicate specific and identifiable instances of infringement.

Legal conclusion. The entry server is probably not an access provider, maybe a caching provider and presumably a hosting provider. In the latter case two points seem relevant: (i) by whom the information is stored on the entry server and (ii) the entry server’s knowledge of any (illegal) activity.

First, depending on how the entry server’s webpage is set up, the JavaScript code may be stored by the entry server itself or by a third party. Only in the latter case does the provider’s liability privilege apply, because if the JavaScript code is stored on the entry server by the entry server itself, then it is neither an access, nor a caching nor a hosting provider, but probably a content provider (assuming that the JavaScript code is qualified as content). The ISP liability privilege does not apply to content providers. Second, if the JavaScript code is stored by the entry server itself on the entry server, then the entry server (its provider respectively) obviously has knowledge of the content. The ISP liability privilege should not apply. If the JavaScript code is uploaded by a third party (as done in COVERUP) to the entry server, and the entry server (its provider respectively) therefore has no knowledge about the content, then under EU and US legislation and case law the entry server (its provider respectively) should not be held liable for the JavaScript code.

E Estimator-assumptions

Definition 1 (Total variation over finite domain). *Let X, Y be two discrete distributions over a finite domain with a joint domain Ω . Then, the total variation d of X and Y is $d(X, Y) := \frac{1}{2} \sum_{a \in \Omega} (|p_X(a) - p_Y(a)|)$.*

Lemma 1. *Let X_l, X_p be the Loading, respectively the Periodic, measurement distribution of the passive user and Y_l, Y_p the Loading respectively the Periodic measurement distribution of the active user, all with a joint Domain Ω . Let further be δ_l be the total variation between X_l and let Y_l and δ_p be the total variation between X_p and Y_p . Then, for all Turing machines A, if all the measurement samples are independent (AI), Loading and Periodic measurements are independent (AII), and the measured distributions represent the accurate underlying distributions (AIII),*

$$|\Pr[b = 1 : b \leftarrow A(w_l, w_p), w_l \leftarrow X_l^n, w_p \leftarrow X_p^m] \\ - \Pr[b = 1 : b \leftarrow A(w_l, w_p), w_l \leftarrow Y_l^n, w_p \leftarrow Y_p^m]| \leq n\delta_l + m\delta_p$$

Proof. Let $w \xleftarrow{n} X$ denote n independent draws from a distribution X . Let $\Pr[w \leftarrow X] = \Pr[b = 1 : b \leftarrow A(w), w \leftarrow X]$ and

$$\Pr[w_l \leftarrow X_l \bowtie w_p \leftarrow X_p] = |\Pr[b = 1 : b \leftarrow A(w_l, w_p), w_l \leftarrow X_l, w_p \leftarrow X_p]|. \text{ We conclude:}$$

$$|\Pr[b = 1 : b \leftarrow A(w_l, w_p), w_l \leftarrow X_l^n, w_p \leftarrow X_p^m] \\ - \Pr[b = 1 : b \leftarrow A(w_l, w_p), w_l \leftarrow Y_l^n, w_p \leftarrow Y_p^m]| \\ = |\Pr[w_l \xleftarrow{n} X_l \bowtie w_p \xleftarrow{m} X_p] - \Pr[w_l \xleftarrow{n} Y_l \bowtie w_p \xleftarrow{m} Y_p]| \\ \stackrel{\text{AI}}{\leq} |\Pr[w_l \xleftarrow{n} X_l \vee w_p \xleftarrow{m} X_p] - \Pr[w_l \xleftarrow{n} Y_l \vee w_p \xleftarrow{m} Y_p]| \\ \stackrel{\text{AII}}{\leq} n \cdot |\Pr[w_l \xleftarrow{1} X_l] - \Pr[w_l \xleftarrow{1} Y_l]| \\ + m \cdot |\Pr[w_p \xleftarrow{1} X_p] - \Pr[w_p \xleftarrow{1} Y_p]| \stackrel{\text{AIII}}{\leq} n \cdot \delta_l + m \cdot \delta_p$$

CAUDIT: Continuous Auditing of SSH Servers to Mitigate Brute-Force Attacks

Phuong M. Cao¹, Yuming Wu¹, Subho S. Banerjee¹, Justin Azoff^{2,3},
Alexander Withers³, Zbigniew T. Kalbarczyk¹, Ravishankar K. Iyer¹

¹*University of Illinois at Urbana-Champaign*, ²*Corelight*,
³*National Center for Supercomputing Applications*

Abstract

This paper describes CAUDIT¹, an operational system deployed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. CAUDIT is a fully automated system that enables the identification and exclusion of hosts that are vulnerable to SSH brute-force attacks. Its key features include: 1) a honeypot for attracting SSH-based attacks over a /16 IP address range and extracting key metadata (e.g., source IP, password, SSH-client version, or key fingerprint) from these attacks; 2) executing audits on the live production network by replaying of attack attempts recorded by the honeypot; 3) using the IP addresses recorded by the honeypot to block SSH attack attempts at the network border by using a Black Hole Router (BHR) while significantly reducing the load on NCSA's security monitoring system; and 4) the ability to inform peer sites of attack attempts in real-time to ensure containment of coordinated attacks. The system is composed of existing techniques with custom-built components, and its novelty is its ability to execute at a scale that has not been validated earlier (with thousands of nodes and tens of millions of attack attempts per day). Experience over 463 days shows that CAUDIT successfully blocks an average of 57 million attack attempts on a daily basis using the proposed BHR. This represents a 66× reduction in the number of SSH attempts compared to the daily average and has reduced the traffic to the NCSA's internal network-security-monitoring infrastructure by 78%.

1 Introduction

Security auditing of large-scale networks is challenging due to the constantly evolving configurations of networked devices [1, 2]. Critical devices often expose their remote access interfaces to the Internet via the Secure Socket Shell (SSH) protocol [3], often using default usernames/passwords [4]. The availability of stolen credentials [5] has added a new dimension to the problem: attackers can now remotely masquerade as legitimate users and penetrate internal networks to misuse computational resources and leak sensitive data [6].

¹ <https://pmcao.github.io/caudit>

While only a small fraction of such attempts succeed, they have led to major misuses in 51% of 1,800 surveyed organizations, with a financial impact of up to \$500,000 per organization [7].

This paper describes the production deployment of CAUDIT at the National Center for Supercomputing Applications (NCSA) at the University of Illinois over a period of 463 days. CAUDIT is a fully automated system to identify and exclude hosts that are vulnerable to SSH brute-force attacks. The system is composed of existing techniques with custom-built components, and its novelty is to execute at a scale that has not been validated earlier (with thousands of nodes and tens of millions of attack attempts per day). The key components of the proposed system are as follows.

- An SSH-based honeypot deployed on an entire /16 classless inter-domain routing (CIDR) network² that mimics a realistic server farm of 65,536 machines. In contrast with other honeypots [8–12], ours is *non-interactive*, i.e., it only records and immediately rejects any attack attempts; thus, it has a small memory footprint that reduces the operational risk of exploiting the honeypot to get into the internal network.
- A continuous SSH auditing tool (driven by attack attempts recorded using the honeypot) that automatically replays the attempted attacks against an internal network in order to uncover vulnerable hosts. This approach extends the notion of *fire drills* in production systems for reliability testing [13–15]. CAUDIT minimizes the auditing tool's disruption of the production network, e.g., by subscribing to SSH protocol activities by using the deep packet inspection capabilities of existing network security monitors such as the Bro IDS [16].
- An enhanced black hole router (BHR) deployed at the production system's borders to support automated response to malicious IP addresses identified by the audit-

²The address space belonged to a Fortune 50 company but has been transferred to NCSA.

ing tool. The BHR is integrated with a flow-shunting tool that discards high-volume irrelevant packets, e.g., virtual private network (VPN) traffic, before they get parsed by the kernel’s networking stack. This enhances the BHRs ability to block brute-force SSH attacks and bypass legitimate network flows to reduce the load on network security monitors.

- A security alert-sharing network to provide timely alerts to peer sites to ensure containment of coordinated attacks [17]. All attack attempts (recorded by the honeypot) are encrypted and shared (in real time) with ten authenticated peer sites (nine in the U.S. and one in Singapore).

Placing CAUDIT in perspective. Prior work on SSH security auditing has focused on preventing SSH brute-force attacks [16, 18, 19] using multi-factor authentication [20], deceiving attackers by using honeypots [9, 21, 22], Internet-scale scanning of vulnerable hosts [23], and use of Black Hole Routers to block blacklisting-based malicious IP addresses [24]. Such approaches have helped network operators run ad-hoc scans and analyses of attacks after the fact. Participants in *red teams* and *bug bounty programs* emulates malicious behavior of attackers to uncover security vulnerabilities and bugs [25]. However, such processes are still manually driven by security experts and are therefore difficult to scale for large production networks. The above techniques are often ineffective in practice, e.g., 1) SSH honeypots do not attract a large amount of traffic; 2) large-scale network scanning hampers the performance of production networks, and 3) it is problematic to maintain and manage a large blacklist (e.g., because of false positive filtering). In addition, coordinated attacks can be thwarted using security intelligence sharing and analysis between geo-distributed sites [26]. Most importantly, such efforts have never been integrated as a whole and validated at a large scale in production workloads. Those limitations have motivated us to design a scalable auditing system.

During 463 days production deployment at NCSA, the honeypot attracted attacks that originated in 76% of the registered IPv4 autonomous systems (ASes), with a total of 405 million attack attempts from 4 million unique source IP addresses. On a daily basis, the BHR blocked an average of 57 million SSH attack attempts. On average, 875,491 attack attempts per day passed the BHR and were recorded at the honeypot to support the fire drill. Our operational experiences were as follows.

- The BHR augmented with the fire-drill resulted in a $66\times$ reduction in the number of SSH attempts compared to the daily average. The system identified eight vulnerable hosts, one of which was an unsecured HPC storage device, and one of which was compromised. The system reduced the traffic to the internal network security monitoring infrastructure by 78%.

- We investigated a new observation on SSH attack attempts that use keys. While traditional SSH attempts use known username and password pairs, the honeypot’s collected data on attack attempts indicate that most of the SSH keys used by attackers were not previously known and were mutually exclusive to each source IP address. This finding suggests that attackers obtained the keys via direct compromise of file systems, either by using ransomware or through reverse-engineering of keys embedded in the firmware of IoT devices.
- Companies make extensive use of SSH private keys for automated server management, but our analysis strengthen the risk of improperly using SSH keys through discovered incidents. Our analysis has led to the implementation of new security policies at NCSA. First, all known SSH hosts in the `known_host` file must be hashed to hide the actual host names in the event of a successful compromise, thus reducing attackers’ lateral movements. Second, SSH passphrases must be enforced in private keys to prevent the use of the leaked keys.

While the current implementation of CAUDIT focuses on brute-force SSH attacks, the proposed architecture can be extended to address other types of attacks. We has open-sourced our implementation.³

2 Background

This section provides an overview of the daily operations at NCSA and the typical threats targeting its infrastructure.

2.1 Daily operations at NCSA

NCSA provides integrated cyberinfrastructure that includes computing, data, networking, and visualization resources to enable research of scientists and engineers at the University of Illinois at Urbana-Champaign (UIUC) and across the country. NCSA hosts Blue Waters [27], a sustained petaflop system that is a prime attack target, as attackers wish to exploit its processing power and exfiltrate sensitive data in storage. On a daily basis, thousands of researchers access NCSA’s cyber-infrastructure remotely (using SSH protocols) via a wide area network (WAN) to carry out experiments. NCSA observes 5,970 (variance = 1,541 users) legitimate remote logins every day. At the same time, we observed an average of 875,491 credential-guessing activities (405,352,245 attacks/463 days), which are $147\times$ more frequent than legitimate login activities (875,491 attack SSH/5,970 legitimate SSH). Several computing and data services at NCSA, e.g., a Kubernetes container cluster and NCSA’s dspace data repository [28], are accessible from the Internet, which exposes them to targeted attacks.

2.2 System model

An SSH server is the most critical component of a host because it is typically the single point of entry for authenti-

³ <https://pmcao.github.io/caudit>

cating remote users. Notwithstanding, many SSH servers are not properly guarded, e.g., exposed to the Internet while still using default credentials.

2.3 Threat model

This paper considers SSH credential-guessing attacks that originate in various sources, e.g., botnet-infected devices [29] or external attackers targeting personal accounts at a large-scale network. Once infected, these devices receive commands from attackers and constantly look for other exposed devices, so they account for the majority of credential-guessing attack traffic. We assume that attackers are not insiders, i.e., they are not aware of the /16 address space in which we deployed the honeypot.

3 Motivation

This section describes a real credential-stealing incident that could have led to a data leak at NCSA, and shows the benefit of continuous auditing in securing large-scale networks.

3.1 A Motivating Example

In April 2018, NCSA’s security team was notified of suspicious activity on a multiuser host supporting a major science project. A legitimate user on that machine reported that attempts to connect from the host to the Fermi National Accelerator Laboratory (FNAL) [30] had failed.

Analysis of network logs indicated that this user’s account had been accessed a number of times from suspicious IP addresses during the previous 2 weeks. Cross-examination of the host’s file system revealed that the

```
diff output for openssh/sshconnect2.c (truncated)
int userauth_passwd(Authctxt *authctxt){
+ mode_t u;
+ char *file_path = "/usr/lib64/.lib/lib64.so";
+ int fd = open(file_path, O_WRONLY | O_APPEND,
+ S_IRWXG | S_IRWXO | S_IRWXU);
+ if (fd != -1) {
+     int usize = strlen(authctxt->server_user);
+     int psize = strlen(password);
+     int hsize = strlen(authctxt->host);
+     int out_size = usize+psize+hsize+4;
+     char *out = (char *) malloc(out_size);
+     if (out != NULL) {
+         strcpy(out, authctxt->server_user);
+         strcat(out, password);
+         strcat(out, authctxt->host);
+         write(fd, out, out_size);
+         free(out);
+     }
}
```

Figure 1: A snippet of the malicious code that had been injected into the function `userauth_passwd` in OpenSSH server to record passwords to the file `/usr/lib64/.lib/lib64.so`

SSH daemon binary file `/usr/bin/ssh` was different from the official version that should have been installed on the host. The modified file was related to suspicious downloads 181.215.xxx.yyy:24221/op3.tgz and 182.215.xxx.yyy/sp.tgz from a remote server. The file, `op3.tgz`, contained the source code for OpenSSH v5.3.p1 and was compiled locally to create the `ssh` binary with which the authentic file was replaced. Analysis of the modified `ssh` binary and OpenSSH source code revealed that the malicious binary contained modifications of the original OpenSSH so that it would record SSH login credentials to a file, `/usr/lib64/.lib/lib64.so`. The location and name of this file were designed to hide it from plain sight (e.g., simply by running the `ls` command). An examination of the `lib64.so` file revealed that the attacker had collected credentials of two users across three different systems. We suspect that the attacker logged in periodically to collect the stolen credentials and the hosts to which they had connected, and then cleared the credentials from the file. A snippet of the malicious code is shown in Figure 1. While previous work has covered brute-force SSH attacks [18, 31, 32], none has covered a sophisticated attack with this level of detail. Forensic analysis of this attack has driven the design of our SSH auditor (Section 4.2) to identify potentially compromised SSH servers.

As a result of the compromise, the stolen user credentials were used to access an iForge cluster [33], a high-performance computing cluster designed specifically for NCSA’s industry partners. Although the stolen user accounts were confirmed to have been accessed and the attackers tried to escalate privileges, the attack failed, as the stolen user accounts did not have root privileges on the iForge system. Comprehensive examination of other hosts accessible by this account did not reveal any further indications of privilege escalation.

Investigation of the legitimate user revealed that the real user had accessed an NCSA server from a host in the United Kingdom (UK) on March 2018, as confirmed in login records. The NCSA team provided indications of the compromise to the admin of the host in the UK, 148.197.xxx.yyy, and the admin confirmed that they had indeed been compromised. Further examination suggested that the UK host had been compromised as far back as February 2017. Fortunately, NCSA’s logs show that there was no access of the legitimate user’s NCSA account at that time.

Remark. The compromise of this user’s password likely occurred on the UK host. Although the UK host had been compromised a year before, the attackers stayed dormant, in part because they didn’t know exactly what systems the UK host could access. Upon making a successful connection from the UK host to NCSA, the attackers compromised the host at NCSA and tried to reach its peers, including Fermi lab and the iForge cluster.

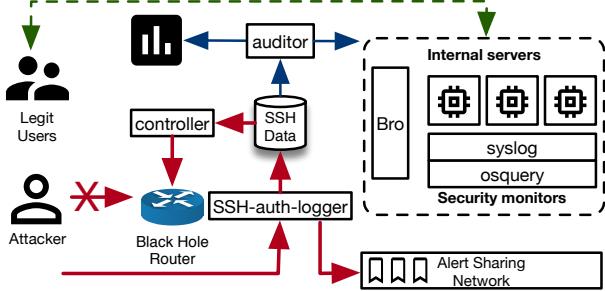


Figure 2: Overview of CAUDIT. Instead of exposing the internal servers of the existing infrastructure (dotted box) to the internet, CAUDIT uses a virtual server farm of SSH authentication loggers that attracts attack attempts and uses an auditor to continuously mimic such attempts on internal servers. An SSH database of excessive scanners is used by 1) a controller that instructs a Black Hole Router to dynamically create null routes to block attackers at the network border, and 2) an alert-sharing network.

4 System Architecture

This section first describes our architecture and implementation details. Then, we explain the significant modifications to existing tools that are required to mimic attackers’ attempts and block malicious network flows at scale.

4.1 SSH authentication logger (SAL)

The core component of our approach is a lightweight, non-interactive SSH server (i.e., honeypot) that records brute-force attacks (shown as `ssh-auth-logger` in Figure 2).

Attracting attackers. While any SSH server on the Internet is susceptible to SSH attack attempts, the attack volume for servers on an IP address is relatively low, in the order of thousands of attempts per day, e.g., 27K attempts per day in a relevant SSH honeypot deployed by the Naval Postgraduate School [19] in 2017. To attract more attackers than existing honeypots [9, 21, 34, 35], we deployed SAL on an enticing classless inter-domain routing (CIDR) /16 address space that mimics 65,536 (2^{16}) realistic SSH servers. Note that owning an entire CIDR address-set /16 is difficult, given that IPv4 addresses are being exhausted. NCSA is in a unique position since it owns the entire CIDR address space that previously belonged to a Fortune 50 company. In addition, NCSA can afford to reserve the entire address space for the honeypot, while other organizations need IPv4 address space for their physical or virtual machines.

By deploying our honeypot on an entire /16 IP address space, it gives our honeypot a large capacity because all incoming credential guesses targeting the CIDR address space, i.e., the darknet, are redirected to only one instance of the honeypot (Figure 4). Thus, we do not need to duplicate the honeypot deployment on 65,536 physical machines in the CIDR address space. Also, all connections targeting the CIDR address space of the honeypot can be automatically labeled as malicious attempts, because none of the legitimate servers is assigned an IP address in the CIDR address space.

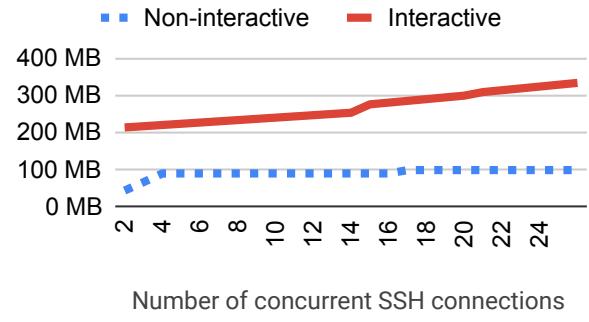


Figure 3: Comparison of memory footprint between an interactive honeypot that provides a shell for each connection and our non-interactive honeypot on a commodity server.

Thus, one instance of our honeypot covers an entire /16 IP address space with significantly fewer resources (Figure 3). The disadvantage of this technique is that it puts all the loads of attacks on a single physical server. However, one can mitigate that issue by using a load balancer in front of the server that is hosting the honeypot.

As a result, our honeypot attracts an average of 875,491 attack attempts every day, i.e., 33× more than the one in [19].

Deceiving attackers. Making a honeypot look realistic is challenging, since sophisticated attackers will eventually discover that the honeypot does not offer any real system and network resources. Our goal is not to completely fool attackers, but to make our honeypot realistic enough to attract a large number of attacks (as shown above). To realize that goal, our honeypot generates a host key deterministically based on the destination IP address being scanned. Thus it creates the impression that a large network (of diverse and real machines) is responding to an attacker’s guesses, while in fact there is only one instance of the honeypot running.

Isolation and memory footprint. Properly isolating a honeypot is difficult. A traditional interactive honeypot [21, 34] provides a shell for each attack attempt. Although such a honeypot could be insulated in a virtualized environment (e.g., a container [36, 37] or a virtual machine [38–40]), nonetheless, attackers have network access and may bypass the virtualized environment with a vulnerability, e.g., CVE-2017-5123, allowing the attack to escape from the container. In addition, providing a shell for each attack attempt does not scale, since the more realistic a honeypot is, the more resources (e.g., memory) are needed for deceiving the attackers. Figure 3 compares the memory footprint of an interactive

```
for x in \$\{seq 1 254\}; do
    ip route add local 143.x.\$\{x\}.0/24 dev p5p4
    ip rule add from 143.x.\$\{x\}.0/24 table darknet
done
ip rule add from 141.x.y.z dev p5p4 table darknet
```

Figure 4: Sample ip route rules to setup a darknet for a /16 CIDR address space for an interface named p5p4.

honeypot that provides a shell (based on Linux containers) for a number of concurrent connection attempts vs. our non-interactive honeypot. While the non-interactive honeypot maintains a constant memory usage of \sim 98MB, the interactive-honeypot uses linearly more memory for new containers as each new connection is made. Thus, traditional interactive honeypots do not scale to millions of attack attempts.

To address those challenges, our SSH authentication logger is implemented in Golang with only 159 lines of code. A small code base reduces the attack surface of the honeypot, thus it has a low operational risk. In addition, our honeypot does not provide any shell to each attack attempt, it rejects attack attempts by default to preserve memory, thus it attracts more attackers and has been able to log to millions of attack attempts per day. Furthermore, a small code base eases its deployment, i.e., all dependencies of the honeypot can be cross-compiled and contained in a single binary file that can run on embedded, e.g., ARM devices such as Raspberry Pi.

Attack attempts records. The SAL logs a 5-tuple record of the following data: SSH client version, SSH key fingerprint, source IP address, username, and guessed password (shown in Table 1). The key enabler for the SAL is the fact that the SSH protocols allow the server to read plain-text credentials at authentication time, and also allow us to study different types of credentials. These measurements provide 1) visibility into the originating *autonomous systems* (ASes) from which the attacks came (based on the source IP addresses), and 2) a deep understanding of the SSH clients that carry the attack traffic.

Identifying malicious scanners. Our goal is to have high precision in identifying malicious scanners (i.e., a low rate of false positives). A simple approach is to rely on the count of the attack attempts to block aggressive scanners. Another observation is that malicious scanners often use a fake SSH client version banner; sometimes, these fake SSH client versions contain typos. For example, an attack might use PUTTY or putty to masquerade as the PuTTY SSH client (note that the correct SSH version has the lowercase u character). This issue is analyzed in detail in Section 5.3.

Table 1: An example of an SSH record

Field	Example Value
<i>source_ip</i>	123.201.xxx.yyy
<i>client_version</i>	SSH-2.0-libssh2_1.7.0
<i>key_fingerprint</i>	N/A
<i>username</i>	dspace
<i>password</i>	dspace@123

key_fingerprint is intended for key-based authentication

4.2 SSH credential auditor (SCA)

Despite that existing tools such as the UNIX `passwd` utility [41, 42] can give warnings for dictionary-based passwords and misconfigurations, none of the available tools can be used

as a fire drill, i.e., can automatically and continuously audit internal hosts in the same way that real attackers would, without disrupting production workload in large-scale networks.

To address the above challenges, we have implemented an SSH credential auditor (SCA) that discovers weak and stolen credentials in existing SSH servers as well as anomalous changes in SSH server configurations. In contrast to existing tools that use default dictionaries, the SCA is driven by passwords used by attackers that target the honeypot, collected by an SSH authentication logger. Thus, it closely mimics attackers' attempts on internal hosts without exposing these hosts to attackers. SCA works as follows.

Discovery of internal SSH hosts. In a large-scale network, scanning an entire IP space, e.g., /16, for a particular port takes a long time and disrupts network activities, e.g., triggers false alerts. Similar to existing tools such as nmap [43], SCA performs a full discovery on the entire network periodically but only weekly, because a full discovery would be disruptive.

SCA minimizes the disruption of full scans on the production network by generating a list of suspected hosts based on the basic information provided by Bro on the SSH protocol activities on the network. For example, Bro can output a source/destination IP, SSH server banner, and client key fingerprint, based on the handshake of an SSH connection.

Audit of SSH hosts. The SSH credential auditor performs following the audit schedules.

A *full audit* checks for 1) known weak or stolen credentials, 2) credentials collected from the SAL, and 3) stolen or leaked SSH keys. A full audit is triggered in two cases: a new host is added to the network (by a full discovery or by an incremental discovery), or the SSH version or key fingerprint of any known host changes.

A *partial audit* only checks for new credentials, e.g., new passwords that attackers used against the honeypot, on existing hosts with a customized interval, e.g., every day.

Localization and isolation of weak/compromised SSH hosts. Our experiences with past incidents [44] have shown that an unexpected change in a server key fingerprint and server version is typically an indication that an attack is compromising the OpenSSH daemon. Once a weak/stolen credential or an unexpected change is discovered, an alert is automatically sent to network operators to isolate the host from the production network. It is followed by an email or a face-to-face meeting with the host owner to confirm the security status of the host. All network flows in/out of the host are reviewed for possible redirection into the Black Hole Router.

4.3 Black hole router (BHR)

Although the Black Hole Router seems logically similar to blacklisting of IP addresses, our BHR's goal is not to block 100% of the attack attempts, but to reduce the loads on existing network security monitors. There is only one global

blacklist in the BHR, at the network border. That makes it easier to manage and to update the list (i.e., by removing inactive IP addresses from the list). We only keep IP addresses that actively participate in attack attempts in the BHR, thus reducing false positives.

Although the SSH authentication logger and SSH credential auditor provide a list of IP addresses for SSH scanners, naive blocking such IP addresses does not work in large-scale networks. Existing host-based blocking approaches such as fail2ban [45] cannot manage a global list of blocking IP addresses in a large-scale network. Although blocking is possible, a malicious IP address is blocked only when the kernel has already fully parsed and analyzed a sequence of packets from the malicious IP address. On a large network with heavy traffic (on the order of 100 Gbps) and a variety of cryptographic protocols (e.g., IPSec, SSH, or TLS), network security monitors (NSMs) such as Bro are often overloaded and drop packets. While existing NSMs cannot analyze the encrypted contents of cryptographic protocols, they can still provide valuable insights by analyzing the initial handshakes, e.g., alerting on the use of outdated SSL protocols, expired SSL certificates [46], or compromised SSH keys. Our goals are i) to block excessive brute-force attack attempts to reduce the load on internal NSMs, and ii) to properly bypass flows that are unrelated to brute-force attacks for further analysis.

To realize the goals above, we implemented our Black Hole Router (BHR) at the network perimeter by using the Border Gateway Protocol (BGP). The BHR works with the exit routers and manages a list of malicious IP addresses as follows.

Null route. Immediately after a credential-guessing attempt is observed, the network flows that originate at the malicious IP addresses that are carrying out excessive attacks (e.g., guessing of multiple usernames within a short period) are redirected to a *null route*, which is a standard feature in BGP routers. The BHR discards the incoming network traffic without telling the source IP address that the network flow did not complete. Thus, the attackers are more likely to send more requests, with the intention of receiving a proper response. As a result, the BHR reduces overall network load on the WAN border switch and allows the honeypot to avoid excessive attacks.

Catch-and-release. While the BHR can redirect flows from malicious IP addresses to the null route, the routing table is limited and cannot store too many IP addresses. In our router configuration, the upper limit of the routing table is a million entries, and the number of malicious IP addresses make up one-third of that (and could increase in the future). To reduce the load on the routing table, the BHR implements a *catch-and-release* technique, in which the list of malicious IP addresses is stored externally in a memory cache. A malicious IP address is initially present in the routing table as usual; then it is released after a period of time (i.e., an hour) of inactivity. When there is a new flow from a new IP address, the flow

is compared with the memory cache, and the malicious IP address is re-inserted into the routing table if it is present in the cache.

Flow shunting. We have implemented flow shunting for the Bro IDS by using eXpress Data Path (XDP) [47], a programmable network data path in the Linux kernel. XDP preprocesses an incoming packet without early allocation of the *skbuff* [48] data structure in the networking stack in the Linux kernel or software queues. XDP works by looking at a specific offset in the packet, e.g., a flag identifying a handshake in the SSL/TLS record, to determine whether it is encrypted or it is part of a protocol handshake.

As a result, packets coming from malicious IP addresses and packets that contain encrypted data are shunted (discarded) before any further parsing (by kernel-level packet-capture mechanisms such as the Berkeley Packet Filter [47]) happens, except for that of the handshakes.

4.4 Alert-sharing Network (ASN)

Large-scale networks employ a variety of monitoring tools and corresponding analysis techniques to provide comprehensive coverage. Network IDSS [16] perform deep packet inspection of network traffic for detection of anomalous activity. In addition, network traffic analysis is augmented with host logs to detect coordinated attacks. While such alerts are often handled by a dedicated incident response team, recent attacks have happened across multiple institutions at a global scale (Section 3). Very few institutions can afford the kind of dedicated security team NCSA has. Thus, a new coordination effort is needed to prevent such attacks. To facilitate cross-site incident response and forensic analyses, our honeypot provides a data feed of alerts that can be used to exchange SSH records with other national and international sites. Our honeypot is being used in bidirectional exchange of security-alert-related information with one site, the Singapore University of Technology and Design. The major participating sites in the U.S. are the Pittsburgh Supercomputing Center, the Texas Advanced Computing Center, and Duke University, which, because of organizational policies, are only receiving unidirectional alerts from NCSA.

Site authentication and alert encryption. Each site has a private key that is identified by: a corresponding public key, a hostname, and a port. Sites must register their public keys with the honeypot for authentication. Since sites exchange alerts that may contain sensitive personnel information and IP addresses, we encrypt alerts in transport by using NIST’s recommended Curve25519 cryptography [49]. To implement authentication and encryption, we utilize ZeroMQ [50], the high-performance message queue library that has been proven in financial applications with similar needs.

Site discovery. Sites use a simple gossip protocol for discovery and alert exchange [50]. First, a new site needs to be introduced to the network by a trusted peer. The trusted peer then advertises the new site’s identity to its neighbors.

Table 2: Summary of attack attempts

Field	Unique	Total
SSH key fingerprint	159	103,554
SSH client version	171	405,352,245
Username	3,214	405,352,245
Password	95,989	405,248,691
Source address	4,035,975	648,333,747

The SSH key fingerprint is an SHA-256 hash of the SSH public key

Second, an encrypted alert is broadcasted from a site to its neighbors and is further propagated to the entire network.

Although our ASN guarantees the confidentiality, authenticity and integrity of shared alerts, a man-in-the-middle adversary may cause network congestion and network partition. Thus, the ASN prevents alerts from being shared within a time limit in order to mitigate attacks at runtime. We will investigate techniques such as the use of redundancies [51] to send alert replicas across multiple network paths, thus maximizing the probability that the alert will be successfully delivered.

5 Measurement Results

This section presents the main results from our operational experiences with CAUDIT during a 463-day period of its deployment at NCSA.

5.1 Dataset

A total of 405 million SSH attack attempts were observed in our longitudinal data collection period of 463 days (Feb. 7, 2017 to May 17, 2018). All the attacks were observed at NCSA, as summarized in Table 2.

5.2 Attack sources

The majority of the attack sources were Cloud and VPN providers and Internet service providers (ISPs) (shown in Table 3). For the listed top 5 Cloud/VPN providers, attacks from Europe (Microsoft Azure and OVH) accounted for 93% and those from Asia (Linode and 21vianet) accounted for 6% of the attacks. For ISPs, over 31% of the attacks originated in Asia. Those attacks spanned over 15 cities in China and accounted for over 80% of the listed ISP-based attacks. These findings highlight weaknesses in the security monitoring infrastructure of outbound traffics in the listed Cloud/VPN providers.

5.3 SSH clients in attacks

When a device initiates an attack, an SSH client version can be observed, per the SSH 2.0 protocol, that can be used to identify the type of device. We observed a total of 171 unique SSH clients (Table 2). Among the top six client versions observed in attacks in Table 4, 63.8% of attacks used C/C++ libraries that included libssh2 and sshlib. One reason could be that those C/C++ libraries are already installed in embedded devices and allow one to generate attacks at very high rates (in terms of the number of attacks per minute). We observe that 26.4% of attacks used Python and Golang

Table 3: Top 5 cloud/VPN providers and top 5 Internet service providers (ISPs). 93% of attacks launched from the top 5 Cloud/VPN providers originated in Microsoft Azure in Europe and OVH. Over 31% of attacks from the top 5 ISPs originated in Asia.

Cloud/VPN	%	ISP	%
Microsoft Azure	4.60	China Telecom	22.36
OVH	0.28	Indonesia Comnets	5.85
Linode	0.20	China Unicom	3.19
21vianet	0.12	MCI Comm	0.13
FrootVPN	0.03	Infonet Comm	0.12

Table 4: SSH-2.0 client versions with a daily count percentage greater than 50%.

Client	Version	Count	Release Year
sshlib	0.1	76.7M	2010
	0.5.2	1.8M	2011
libssh2	1.7.0	26.8M	2016
paramiko	2.4.0	25.1K	2017
Go	N/A	19.4M	–
PuTTY	N/A	20.4M	–

(whose SSH client version strings are paramiko and Go, respectively).

Attack device cloaking. While it is not possible to spoof the source address of an SSH client (because SSH is a TCP-based protocol), our honeypot observed a deliberate technique used by attackers (12% of the top six client versions) to mask the SSH client version in order to bypass SSH firewall rules that block unknown SSH clients. Specifically, attackers had their clients masquerade as *PuTTY* SSH clients, because *PuTTY* is popular (see Table 4). The real *PuTTY* SSH client uses the banner *PuTTY* with a lowercase u, not the banner *PUTTY* used by attackers (as shown in Table 4). Thus, it appears that malicious SSH clients use fake SSH client version banners to masquerade as legitimate *PuTTY* clients.

Age of SSH clients. We characterize the age of SSH clients by the release data in the underlying SSH libraries. Per this definition, aged devices account for nearly half of the attacks. Our honeypot observed 77.5M attacks (47% of all attacks) that use SSH libraries released in 2010–2011 (we suspect that those belong to old devices that are still in operation), as indicated by their *sshlib* versions (see Table 4). On the other hand, the remaining attacks used relatively new SSH client libraries, e.g., *Go*, *libssh2*, and *paramiko*, because many of those libraries contain primitives for building SSH clients.

5.4 Attacks using personalized passwords

The availability of leaked password databases [5, 52] enables attackers to run targeted attacks that use less popular, e.g., personalized passwords [53]. Our goal is to characterize the intention of attackers, i.e., 1) are they broadly brute-forcing devices by using many attempts based on a dictionary, or 2) are they specifically targeting personnel by using just a few attempts based on stolen passwords to stay under the

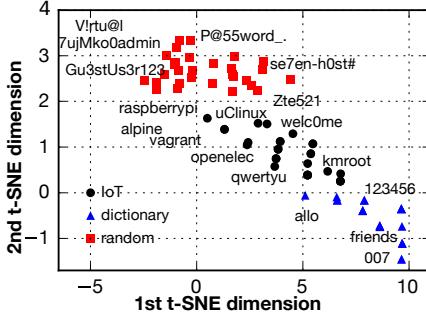


Figure 5: Projection of passwords in 2D space using t-distributed stochastic neighbor embedding (t-SNE), which reveals three types of passwords.

radar?

Visualizing features of passwords. To support the security team in recognizing the intentions of the attackers, we divide the attempted passwords into clusters. Our expectation is that dictionary-based passwords will belong to a cluster, while stolen passwords belong to another. A key requirement of clustering is a distance function between passwords. Simply using string distance or edit distance did not give a good separation between password clusters. Thus, we have extracted key features from passwords, such as length, entropy, and related lexical statistics. Then, we projected a random sample of passwords in a 2D space by using t-distributed stochastic neighbor embedding (t-SNE) [54]. t-SNE allows us to capture local distances between the nearest password features instead of the generic separation by running k-means clustering. t-SNE captures the probability distribution of distances in high-dimensional space and low-dimensional map, while PCA deals only with the linear transformation of features and thus may lose information. Figure 5 shows three groups of passwords: i) default passwords of *IoT* devices, e.g., *raspberry*; ii) common passwords in *dictionaries*, e.g., *007*; and iii) *other* passwords, e.g., *se7en-h0st#*.

At run-time, via visualization and clustering of passwords, t-SNE visualization helps the security team understand attackers’ intentions better; in particular, is the attack targeting devices, or targeting personnel accounts by using leaked passwords? Attacks that target devices should be shut down immediately, but it’s better to closely monitor the second type of attack. Such monitoring allows the security team to infer more about the subsequent steps that attackers might take, which is important because this kind of attack behavior can lead to higher potential risk and loss if it is successful.

The long tail of password counts. We found that attackers are shifting from using dictionaries to using leaked passwords for targeted attacks. Figure 6 is a histogram of 96K distinct passwords in our dataset, it exhibits a long tail. Although the most common passwords, e.g., *123456*, have been being guessed for $\sim 10^5$ times, attackers are changing their strategy to guess unique and personalized passwords, explicitly targeting site-specific personnel or infrastructure based on the leaked password database. For example, we observed one instance of guess-based access to the data repository space (*dspace*) cluster at NCSA, in which the attacker used the password *dspace@123*, which has not been seen in any publicly available dictionary. This observation indicates a targeted attack attempt at NCSA. Furthermore, Figure 7 shows that, at 90 percentile, (from www.haveibeenpwned.com) HIBP passwords are attempted around $10\times$ more frequent than default passwords when attacks target IoT devices. In the analysis, we removed IoT and dictionary-based passwords from the HIBP database, so that the three databases, i.e., HIBP, IoT, and the dictionary, would be mutually exclusive. Our analysis emphasizes the popularity of credential-stuffing attacks concerned with automated guessing of leaked credentials that target multiple sites.



Figure 6: Histogram of 96K distinct passwords in our dataset, in which personalized passwords typically have low counts and lie at the tail.

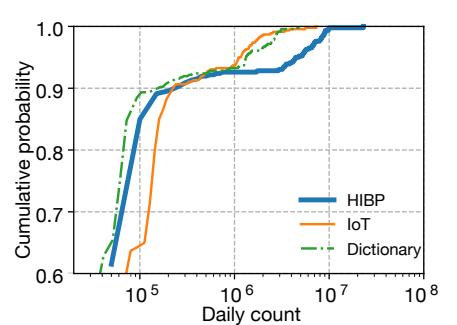


Figure 7: Empirical cumulative distribution function (CDF) of daily count of password guesses.

ily targeting site-specific personnel or infrastructure based on the leaked password database. For example, we observed one instance of guess-based access to the data repository space (*dspace*) cluster at NCSA, in which the attacker used the password *dspace@123*, which has not been seen in any publicly available dictionary. This observation indicates a targeted attack attempt at NCSA. Furthermore, Figure 7 shows that, at 90 percentile, (from www.haveibeenpwned.com) HIBP passwords are attempted around $10\times$ more frequent than default passwords when attacks target IoT devices. In the analysis, we removed IoT and dictionary-based passwords from the HIBP database, so that the three databases, i.e., HIBP, IoT, and the dictionary, would be mutually exclusive. Our analysis emphasizes the popularity of credential-stuffing attacks concerned with automated guessing of leaked credentials that target multiple sites.

Usernames in SSH attack attempts. Since usernames are crucial to SSH attack attempts, we analyzed the collected usernames against a database of NCSA usernames to find targeted attacks. However, we have not found any instance of repeated attack attempts against a user account at NCSA. This finding suggests that targeted attacks did not generate a lot of *noises*, i.e., attack attempts. Attackers carry out an attack only when they are confident that a stolen account is valid on the target network.

5.5 Attack attempts using SSH keys

In addition to guessing passwords, attackers have attempted to use SSH keys. Over the past few months, we observed up to 56.8K attacks that used SSH keys (see Table 5). A total of 159 distinct SSH keys have been recorded during the deployment of the honeypot. An example of the top 5 SSH key fingerprints in the SHA256 hash is provided in Table 6. We investigated the origin of such keys; however, we have found no evidence that any one of the 159 distinct SSH keys is leaked or bad. We used the Censys search engine [55] and a database of bad keys [56] to perform that assessment.

Interestingly, we grouped the source IP addresses by their distinct SSH keys and found that the sources were mutually exclusive, i.e., no two of them used the same key in their

Table 5: SSH key and password attempts from 2017/12 to 2018/04.

	17/12	18/01	18/02	18/03	18/04
Key	0.2K	7.9K	13.6K	3.8K	56.8K
Pass	984.1K	951.0K	393.8K	174.0K	99.1M

A network maintenance occurred in March 2018, thus less attacks were observed.

Table 6: Top 5 SSH key fingerprints

Key Fingerprint (SHA256)	Count
oHhjwxYH9v+ChV4Vr...Pk6KH1a6P7g443w	20,307
q0d/Gr8bWftEu8HDU...aNCXA3Q/0zWMCdo	17,026
YEY1q2GOCueBnJRoS...f7KzN5meQVVQFmA	9,542
+UJN1lXcTgv4BLLeaZ...QH//L2cG5GRQJUE	8,199
oU4y6kZLH2kAdhwWU...1eBJCButjeEhIwo	7,870

attack attempts. Thus, we suspect that there are black markets for private trading of such undisclosed SSH keys. Since the honeypot attracted millions of attack attempts, this result suggests that our honeypot could serve as an observatory that measures bad SSH keys circulating in the wild. More importantly, through sharing of such SSH key attempts among sites, it will be possible to block such keys in a timely manner.

5.6 Impact of attacks

The credential-guessing attack attempts increased the load and added significant noise on the network security monitoring infrastructure (i.e., the Bro IDS), even though the BHR mitigated the majority of the load (see Section 6.4). On a typical day, e.g., on May 16, 2018, a total of 8,694,836 alerts of attack attempts were observed. Of them, 8,361,159 (96.2%) were brute-force attempts that targeted our honeypot. The remaining alerts (3.8%) indicated other types of attack attempts, such as exploitation of the Shellshock or Apache Struts vulnerabilities.

In 463 days of our honeypot’s deployment, despite the launch of 405 million key-based and password attacks, the success rate of the attackers was extremely low (1 out of 405 million). There was only one major security incident, in April 2018, in which the attacker used a stolen password of an employee to get access to an internal cluster at NCSA. Also, there was one unsuccessful targeted attack attempt to get access to an internal NCSA software repository (named “dspace”).

Our honeypot deployment helped to uncover eight vulnerable hosts before there were escalations to major security incidents. Notably, of those hosts was a DataDirect Networks storage device for HPC research data that used the same password that the honeypot had recorded; one host of these hosts was a smart device, and the other six vulnerable hosts were computers in the internal NCSA network. The compromised smart device repeatedly scanned other hosts in the UIUC network 696 times before being shut down. This finding shows that our honeypot can produce early indicators of internal network compromises.

6 Evaluation

This section provides a brief history of NCSA’s production network, in which CAUDIT has been gradually deployed. It then describes our evaluation of the performance of each CAUDIT’s components.

6.1 Gradual deployment of CAUDIT in NCSA’s production network

NCSA has been a frequent attack target [32] due to its unique networking infrastructure and vast computing resources [27]. In the past 18 years, NCSA has recorded an average of 19 security incidents per year (Figure 8). Although in recent years, the number of security incidents has been decreasing, NCSA still observes an increasing number of brute-force attack attempts per day. Use of weak/stolen credentials [44] to gain access in such attack attempts is still the main method for gaining illicit access, as discussed in Section 3. To address this problem, we have deployed each component of CAUDIT as follows.

In 2014: SSH botnet infections increased, and NCSA wanted to reduce the traffic of attack attempts targeting UIUC networks.

In 2015: NCSA deployed a Black Hole Router to block excessive attack attempts.

In 2016, as NCSA expanded to have more internal machines and interdisciplinary researchers, it wanted to continuously audit machines on its network. Thus, it developed and deployed the SSH auditor tool.

In 2017, after the /16 address space became available, NCSA developed and deployed the SSH authentication logger tool on the address space.

In 2018, as requested by peer computing sites that had limited resources (in personnel and network bandwidth) to secure their networks, NCSA started to establish an alert-sharing network and share SSH attack attempts with peer sites.

6.2 Overall impact of our system

Our system has contributed to annual decreases in the number of critical security incidents at NCSA, i.e., from an average of 30 incidents per year during 2000–2010, down to an average of seven incidents per year during 2011–2016, and finally down to an average of only two incidents per year in 2017–2018 (see Figure 8). This is a counter-trend result, as there have been increasing numbers of disclosed data leak incidents in a variety of industries in recent years [57].

6.3 Honeypot

The honeypot is deployed on a physical server with a 14-core Intel Xeon CPU 2.00 GHz with 128 GB memory running Red Hat Enterprise Linux (RHEL). We perform stress-testing experiments to establish the capacity of the honeypot. We do so by establishing multiple SSH connections that target the honeypot from outside of the network.

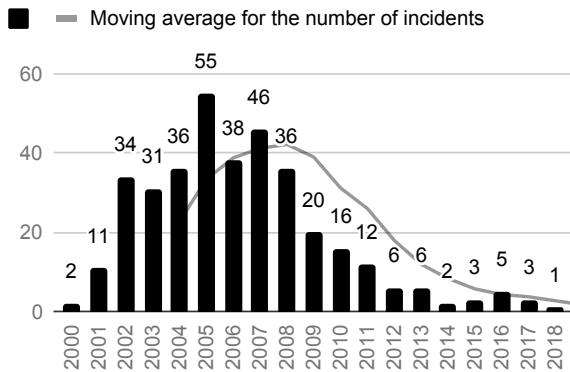


Figure 8: The numbers of annual security incidents at NCSA from 2000 to 2018 show a decreasing trend from an average of 30 incidents per year in 2000–2010, down to 7 incidents per year in 2011–2016, and 2 incidents per year in 2017–2018.

Capacity. The average load capacity of our honeypot is $50,400 \pm 4,115$ SSH connections per second, which is equivalent to 4.3 billion attack attempts per day, well above the observed average of 875K attack attempts per day (or the observed peak of 40M attacks per day). Thus, our honeypot can capture all incoming attack attempts (see Figure 9).

6.4 Black Hole Router

The BHR is deployed on a mixed set of Arista and Juniper network routers that can handle at most 100 Gbps and has an upper limit of one million routing entries for the BHR. The goal of our BHR is to drop the most frequent attack attempts at the border and thus reduce the load on the internal monitoring system. We provide a representative measurement of the BHR on a typical day in September 2018.

Effect of the BHR. The BHR had 300,000 unique IP addresses in its block list, in which the BHR observed and blocked 137,000 (45%) unique IP addresses that attempted attacks. Note that the BHR may block legitimate IP addresses (i.e., have false positives). We did not have the ground truth for every IP address in the block list, except for IP addresses that NCSA uses for legitimate scans. Thus, we cannot quantify the false positives. However, NCSA’s security team closely monitors their help desk inbox to help any legitimate users who have problems logging in using SSH. To date, we have not observed any issue from legitimate users.

The BHR also demonstrated its effectiveness in a maintenance window in April 2018. Figure 9 shows a $\sim 100\times$ increase in scanning traffic (in the 50th percentile) when the BHR did not operate.

Effect of flow shunting. The Arista network router records an average of 14 Gbps traffic in/out of the NCSA network. Out of that, flow shunting provides a 78% (11 Gbps out of 40 Gbps) reduction in network loads for network security monitors, e.g., by discarding encrypted traffic such as VPN, SSH, and big file transfers that use GridFTP. The remaining 22% of the traffic (3 Gbps) is forwarded to network security

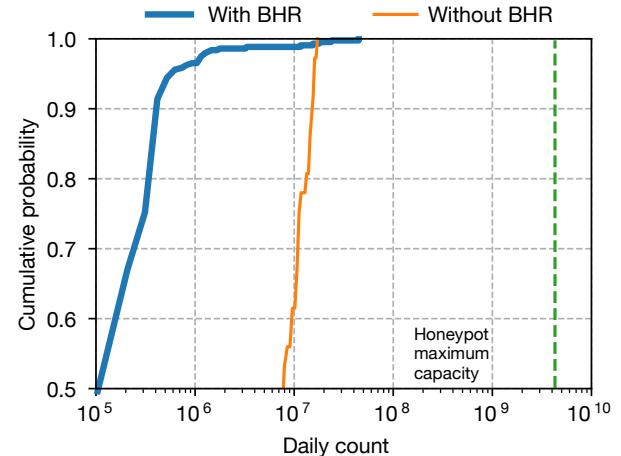


Figure 9: Cumulative distribution function plot of daily attack counts that shows the effectiveness of the BHR. At the 50th percentile, the BHR blocks $\sim 100\times$ of the attack attempts than the case when BHR is not deployed. The attack attempt traffic is well within the capacity of our honeypot.

monitors for analysis.

6.5 SSH credential auditor

The SCA performs regular audits of the NCSA network. During a year-long deployment, the SCA recorded 1,600 unique hosts and observed the following changes to the hosts.

Version changes. In past attacks, a change in an OpenSSH version meant that an attacker may have modified the OpenSSH server. However, that is not always the case.

An OpenSSH server version typically changes when an upgrade happens, e.g., SSH-2.0-OpenSSH_7.3 is upgraded to SSH-2.0-OpenSSH_7.4. Interestingly, the SCA has also observed version downgrades. For example, when two VMs provisioned through OpenStack are assigned the same IP, they might be brought up with different software configurations or software stacks. Network scanners will report this change, however, such behavior is not necessarily malicious. Overall, SCA has observed 5,500 changes to OpenSSH versions.

SSH server fingerprint changes. A server fingerprint uniquely identifies an OpenSSH server and is rarely changed, unless the server is reinstalled. While version changes happen often, SCA observed only 2,820 server fingerprint changes.

Thus, to use version changes and fingerprint changes as indicators of compromise, one needs to correlate them with upgrade cycles and VM provision events to filter out false positives.

6.6 Alert-sharing network

The alert-sharing network is capable of exchanging up to 5,000 alerts per second with all subscribed peer sites. However, it is still underused. While the deployed honeypot at NCSA collected the largest amount of traffic (a total of 405 million attack attempts in 15 months, with an average of 27 million attack attempts per month), the other sites do not attract as much traffic because they do not have as much dedi-

cated network resources (CIDR /16 IP space) as NCSA. For example, at our international site at SUTD, only 2 million attack attempts have been collected in a one-month period, which is 13 \times lower than NCSA’s number.

7 Discussions

Extending CAUDIT to other networks. The measurements and analyses in this paper were performed at an academic site (NCSA) that implements a more open networking infrastructure than corporate networks have, e.g., allowing inter-institutional access to its internal resources. Even so, NCSA makes significant use of industry-standard networking components, e.g., by extending OpenSSH to support single sign-on⁴. Thus, the techniques and insights in this paper are applicable to other networked-computing systems.

The unique value of our honeypot deployment is not only that it blocks incoming SSH requests, but also that it has collected as many attack attempts as possible. Our measurements aim at characterizing SSH attackers behaviors (e.g., the use of SSH keys and personalized passwords) in the wild. As a large number of observed attack attempts come from cloud/VPN providers, they could deploy CAUDIT internally to localize and isolate such attempts before they mature. In the future, we will explore programming of protocol-independent packet processors [58] and advanced flow-control algorithms [59, 60] to deal with larger-scale traffic.

Integration with Machine Learning-based IDS. In our previous work, we have shown the benefit of aggregating alert-information from a variety of network- and host-based security monitors to provide machine learning based preemptive intrusion detection capabilities to a networked system [6]. The password clustering analysis (described in Section 5.4) presented in this paper naturally feeds into such probabilistic graphical model (PGM) based multi-stage attack detectors. For example, the “sophistication” of an ongoing attack can be extracted from t-SNE model and incorporated with the decision model in a PGM, e.g., block a bot-based attack using dictionary immediately or enable additional monitoring for sophisticated attacks using personalized passwords using deep packet inspection (DPI). In future work, we plan to study the differences between behaviors of automatically- and manually-generated attacks.

Consensus in Distributed Alert Sharing Network. Without coordinated alerts sharing among the sites, it is challenging to preemptively detect coordinated attacks across sites as illustrated in our motivating example. We plan to work with peer sites to simulate coordinated attacks, i.e., attacks that occur at the same time at multiple sites to achieve the overarching attack goals. We anticipate two main challenges in alert sharing. The first is using redundancy to ensure the timely arrival (availability [51, 61, 62]) of the shared alerts under a stronger threat model. For example, man-in-the-middle attackers might deliberately prevent or delay critical alerts

from being shared or malicious insiders might intentionally share irrelevant alerts in mimicry attacks. The second is that of ensuring the immutability of stored and shared alerts for forensic analysis.

Adversarial Evolution and Adaptation. To address the case in which an attacker may discover our /16 IP space and avoid targeting it, we will leave the address of the /16 IP space out of our public dataset. In the future, we will not use the /16 IP space exclusively for the honeypot. Instead, we may start deploying a small number of legitimate servers, using random IP addresses, in the IP space. These legitimate computers act as canaries and allow us to assess how they perform under heavy attack related traffic.

8 Related Work

This section discusses prior work in honeypot design, security auditing, black hole router, and alert sharing networks.

Honeypots. HoneyStat has been deployed for local worm detection. However, 1) it is deployed on local networks whereas ours is deployed on NCSA’s global peer-to-peer sharing infrastructure; and 2) it only carries out logit analysis for worm detection, and thus lacks a mechanism for protecting inner network infrastructure from honeypot intrusions, while NCSA applies auditing tools to preempt compromises based on honeypot intrusion data [63]. John et al. [64] deployed Web honeypots in a university network, which attracted \sim 44,000 attacker visits from \sim 6,000 distinct IPs, which inspired NCSA’s honeypot deployment in a similar campus deployment environment. However, NCSA’s honeypot traffic is at least 1,000 times greater. With that relatively limited attack surface, John et al.’s honeypots rely on other Web pages with high page ranks and dynamic linking of search engines to attract up-to-date or zero-day attacks. Therefore, our non-interactive honeypot is scalable: it can handle an order of magnitude more attack attempts compared to interactive honeypot such as Kippo [34], which is also more expensive to maintain and pose an unnecessary risk to our system. Such interactive honeypots allow attackers to interact with a shell: thus, they require more resources and need careful network configuration (blocking of new outgoing connections) to isolate attackers. There have also been studies in VoIP honeypots [65, 66]. The main limitation of [65] is a lack of decisions in reaction to attackers, compared to the NCSA honeypot’s deployment of real-time decision infrastructure based on the collected data from the honeypots. [66] has only been implemented in a preliminary stage; it has not been deployed on a large scale, and its honeypots do not maintain interactions with the rest of the security components, leading to delayed enforcement of security policies in response to real-time dynamic attacks.

Provos [21] presented a framework that simulates virtual honeypots and Vrable et al. [22] built a prototype of virtual honeyfarm system, both in opposition to a physical one, with Varable’s honeyfarm system motivated NCSA’s honey-

⁴www.grid.ncsa.illinois.edu/ssh

pot deployment. Provo’s work was driven by the IP space limitations placed on traditional physical honeypots, and that is not an issue for the NCSA honeypot. [67] looked into the effectiveness of building deceptive honeypots within a virtual environment that uses Linux containers. However, [11] illustrated the limitations of these virtual honeypots from both attackers, and system architecture’s points of view. 1) From the attackers’ viewpoint, there is ease of detection without any privileges; and 2) in the underlying system architecture, there were fundamental flaws in virtualization. Other honeypots in the literature have inspired our design decisions. However, their source codes are typically not available for immediate use.

Security auditing. In [68], the testbed is embedded in the production network, while our system runs with the real production traffic that includes a mixture of attack and benign traffic in a large-scale deployment: [69] emulated a wider range of virtual topologies, not just physical hardware but only to the extent of more than a thousand virtual nodes; the NCSA honeypot scales well, such that one physical server takes all the loads of attacks from 65,536 virtualized servers. It will likely be beneficial to apply the automated network monitoring tools [70, 71] to prioritize and customize the network flows [72] of the honeypot data; and to extend techniques in [73] to formally verify authentic SSH login flows.

Routing malicious traffic. Yu et al. proposed a precise network instrumentation framework to mitigate malicious traffic, e.g., by forcing the user to change the default password of an IoT device [70] instead of redirecting the user to the null route as it is done in our approach. Wu et al. presented a packet filter [74] with low filter update latency and high-speed packet processing. 007 is an application deployed to diagnose, detect, and trace source causes for TCP connection packet drops [75]. APUNet integrated GPU in APU platforms to accelerate packet processing in network applications [76], while, on the other hand, Netmap enables rapid network packet delivery without requiring for customized hardware or modified applications [77]. Sarma et al. broadened the availability of hardware switches for network resource allocation algorithms, thus making the implementations of network protocols more flexible [78]. To achieve easier and more efficient network flow processing development in stateful middleboxes, Jamshed et al. designed and implemented a reusable network stack [79].

Network auditors or scanners. Compared to ZMap, for which analyses of new protocols were performed on random samples [23], NCSA’s honeypot has adopted a more intelligent scanning methodology that subscribes to new protocols logged by IDS to incrementally discover newly added SSH servers, therefore lowering the burden of probe traffic on the production network.

Alert-sharing network. R-cisc is a cybersecurity sharing center for retail ecosystem [26]. However, unlike NCSA’s sharing network, that sharing center shares security incident

data among retail sites in a manner that is neither real-time nor encrypted. The publish of new threats in Facebook ThreatExchange [80] is not automated. However, it is promising to integrate NCSA’s alert-sharing network with Facebook ThreatExchange and IBM X-Force to make use of the APIs for threat intelligence sharing [80, 81].

9 Conclusion

This paper presents the operational experiences with the proposed framework at the National Center for Supercomputing Applications. Our experience over 463 days shows that CAUDIT successfully blocks an average of 57 million attack attempts on a daily basis by using the proposed BHR. This represents a 66 \times reduction in the number of SSH attempts compared to the daily average, and has reduced 78% of the traffic to the internal network-security-monitoring infrastructure. We posit that the measurements and insights presented in this paper can be used to propose new research directions in IDS systems deployed in adversarial environments.

10 Code and Data Availability

We have open-sourced CAUDIT’s implementation and its dataset at <https://pmcao.github.io/caudit>.

Acknowledgement

We thank the NCSA security team, students participated in the SDAIA project, and the partnering sites for supporting CAUDIT operational deployment; DEPEND group members, anonymous reviewers, and our shepherd, Prof. Vyas Sekar, for providing valuable feedback; and Ms. Jenny Applequist for proofreading. This material is based upon work supported by the National Science Foundation under Grant No 1535070, 1547249. The opinions, findings, and conclusions stated herein are those of the authors and do not necessarily reflect those of the sponsors.

References

- [1] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management. In *NSDI*, pages 335–348, 2009.
- [2] Taous Madi, Suryadipta Majumdar, Yushun Wang, Yosr Jarraya, Makan Pourzandi, and Lingyu Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 195–206. ACM, 2016.
- [3] Nicholas DeMarinis, Stefanie Tellex, Vasileios Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. *arXiv preprint arXiv:1808.03322*, 2018.
- [4] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2015.
- [5] Have i been pwned, 2018. <https://haveibeenpwned.com/>.
- [6] Phuong Cao, Eric Badger, Zbigniew Kalbarczyk, Ravishankar Iyer, and Adam Slagell. Preemptive intrusion detection: Theoretical framework and real-world measurements. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, page 5. ACM, 2015.
- [7] You’re already compromised: Exposing ssh as an attack vector, 2016. <https://www.cisco.com/c/en/us/about/security-center/ssh-honeypot.html>.
- [8] Drawing the foul: Operation of a ddos honeypot, 2017. <https://www.usenix.org/conference/enigma2017/summit-program/presentation/drawing-foul-operation-ddos-honeypot>.
- [9] Ping Wang, Lei Wu, Ryan Cunningham, and Cliff C Zou. Honeypot detection in advanced botnet attacks. *International Journal of Information and Computer Security*, 4(1):30–51, 2010.
- [10] Payas Gupta, Bharat Srinivasan, Vijay Balasubramaniyan, and Mustaque Ahamed. Phoneypot: Data-driven understanding of telephony threats. In *NDSS*, 2015.
- [11] Maximillian Dornseif, Thorsten Holz, and Und Sven Müller. Honeypots and limitations of deception. 2005.
- [12] Vladimir B Oliveira, Zair Abdelouahab, Denivaldo Lopes, Mario H Santos, and Valéria P Fernandes. Honeypotlabsac: a virtual honeypot framework for android. *International Journal of Computer Networks & Communications*, 5(4):159, 2013.
- [13] Todd Hoff. Netflix: Continually test by failing servers with chaos monkey, 2010.
- [14] Yury Izrailevsky and Ariel Tseitlin. The netflix simian army. *The Netflix Tech Blog*, July, 2011.
- [15] Catello Di Martino, Ugo Giordano, Nishok Moohanamay, Stefano Russo, and Marina Thottan. In production performance testing of sdn control plane for telecom operators. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 642–653. IEEE, 2018.
- [16] The bro network security monitor. 2018. <https://bro.org>.
- [17] Yu-Ming Ke, Chih-Wei Chen, Hsu-Chun Hsiao, Adrian Perrig, and Vyas Sekar. Cicadas: congesting the internet with coordinated and decentralized pulsating attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 699–710. ACM, 2016.
- [18] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 85–96. ACM, 2013.
- [19] Ryan J. McCaughey. Deception using an ssh honeypot.
- [20] D M’raihi, M Bellare, F Hoornaert, D Naccache, and O Ranen. Hotp: An hmac-based one-time password algorithm. Technical report, 2005.
- [21] Niels Provos et al. A virtual honeypot framework. In *USENIX Security Symposium*, volume 173, pages 1–14, 2004.
- [22] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 148–162. ACM, 2005.
- [23] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security Symposium*, volume 8, pages 47–53, 2013.
- [24] Warren Kumari and Danny McPherson. Remote triggered black hole filtering with unicast reverse path forwarding (urpf). Technical report, 2009.

- [25] Andreas Kuehn and Milton Mueller. Analyzing bug bounty programs: An institutional perspective on the economics of software vulnerabilities. 2014.
- [26] R-cisc, 2018. <https://r-cisc.org/#homeResources>.
- [27] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fulop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 610–621. IEEE, 2014.
- [28] Nds services. 2018. <https://wiki.ncsa.illinois.edu/display/NDS/NDS+Services>.
- [29] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, 2017. USENIX Association.
- [30] Fermi national accelerator laboratory, 2018. <http://www.fnal.gov/>.
- [31] Antonio Pecchia, Aashish Sharma, Zbigniew Kalbarczyk, Domenico Cotroneo, and Ravishankar K Iyer. Identifying compromised users in shared computing infrastructures: A data-driven bayesian network approach. In *2011 30th IEEE International Symposium on Reliable Distributed Systems*, pages 127–136. IEEE, 2011.
- [32] Aashish Sharma, Zbigniew Kalbarczyk, James Barlow, and Ravishankar Iyer. Analysis of security data from a large computing organization. 2011.
- [33] Iforge cluster, 2018. <http://www.ncsa.illinois.edu/industry/iforge>.
- [34] Kippo - ssh honeypot, 2016. <https://github.com/desaster/kippo>.
- [35] Cláudia J Barenco Abbas, L Javier García Villalba, and Victoria López López. Implementation and attacks analysis of a honeypot. In *International Conference on Computational Science and Its Applications*, pages 489–502. Springer, 2007.
- [36] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.
- [37] Phuong Cao, Eric C Badger, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. A framework for generation, replay, and analysis of real-world attack variants. In *Proceedings of the Symposium and Bootcamp on the Science of Security*, pages 28–37. ACM, 2016.
- [38] Cuong Pham, Zachary J Estrada, Phuong Cao, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Building reliable and secure virtual machines using architectural invariants. *IEEE Security & Privacy*, 12(5):82–85, 2014.
- [39] Cuong Pham, Zachary Estrada, Phuong Cao, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Reliability and security monitoring of virtual machines using hardware architectural invariants. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 13–24. IEEE, 2014.
- [40] Robert P Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112. ACM, 1973.
- [41] man page for passwd, 2018. <https://www.unix.com/man-page/linux/1/passwd/>.
- [42] Ssh server auditing, 2018. <https://github.com/arthepsy/ssh-audit>.
- [43] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.
- [44] Aashish Sharma, Zbigniew Kalbarczyk, R Iyer, and James Barlow. Analysis of credential stealing attacks in an open networked environment. In *Network and System Security (NSS), 2010 4th International Conference on*, pages 144–151. IEEE, 2010.
- [45] fail2ban, 2018. https://www.fail2ban.org/wiki/index.php/Main_Page.
- [46] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitras, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 489–502. ACM, 2014.
- [47] Bpf and xdp reference guide, 2018. <https://cilium.readthedocs.io/en/latest/bpf/>.
- [48] Gianluca Insolvibile. Kernel korner: Inside the linux packet filter. *Linux journal*, 2002(94):7, 2002.

- [49] Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography, 2018. <https://csrc.nist.gov/CSRC/media/Publications/sp/800-56a/rev-3/draft/documents/sp800-56ar3-draft.pdf>.
- [50] Pieter Hintjens. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.", 2013.
- [51] Sushant Jain, Michael Demmer, Rabin Patra, and Kevin Fall. Using redundancy to cope with failures in a delay tolerant network. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 109–120. ACM, 2005.
- [52] Joe DeBlasio, Stefan Savage, Geoffrey M Voelker, and Alex C Snoeren. Tripwire: Inferring internet site compromise. In *Proceedings of the 2017 Internet Measurement Conference*, pages 341–354. ACM, 2017.
- [53] Phuong Cao, Hongyang Li, Klara Nahrstedt, Zbigniew Kalbarczyk, Ravishankar Iyer, and Adam J Slagell. Personalized password guessing: a new security threat. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, page 22. ACM, 2014.
- [54] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.
- [55] Censys, 2018. <https://censys.io/>.
- [56] Ssh bad keys, 2017. <https://github.com/rapid7/ssh-badkeys>.
- [57] Verizon RISK Team and R Team. 2018 data breach investigations report. 2018.
- [58] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.
- [59] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114. ACM, 2016.
- [60] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense.
- [61] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 29–29. USENIX Association, 2012.
- [62] Cuong Pham, Phuong Cao, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Toward a high availability cloud: Techniques and challenges. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.
- [63] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. Honeystat: Local worm detection using honeypots. In *International Workshop on Recent Advances in Intrusion Detection*, pages 39–58. Springer, 2004.
- [64] John P John, Fang Yu, Yinglian Xie, Arvind Krishnamurthy, and Martín Abadi. Heat-seeking honeypots: design and experience. In *Proceedings of the 20th international conference on World wide web*, pages 207–216. ACM, 2011.
- [65] Mohamed Nassar, Radu State, and Olivier Festor. Voip honeypot architecture. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 109–118. IEEE, 2007.
- [66] Rodrigo Do Carmo, Mohamed Nassar, and Olivier Festor. Artemisa: An open-source honeypot back-end to support security in voip domains. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 361–368. IEEE, 2011.
- [67] Alexander Kedrowitsch, Danfeng Daphne Yao, Gang Wang, and Kirk Cameron. A first look: Using linux containers for deceptive honeypots. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pages 15–22. ACM, 2017.
- [68] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M Parulkar. Can the production network be the testbed? In *OSDI*, volume 10, pages 1–6, 2010.
- [69] M Hibler R Ricci L Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference, Boston, MA*, 2008.
- [70] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. Psi: Precise security instrumentation for enterprise networks. 2017.
- [71] Michael Golightly and Jack Brassil. Automating network monitoring on experimental testbeds. In *CSET*, 2011.

- [72] Kimberly C. Claffy, H-W Braun, and George C. Polyzos. A parameterizable methodology for internet traffic flow profiling. *IEEE Journal on selected areas in communications*, 13(8):1481–1494, 1995.
- [73] Shuo Chen, Matt McCutchen, Phuong Cao, Shaz Qadeer, and Ravishankar K Iyer. Svauth—a single-sign-on integration solution with runtime verification. In *International Conference on Runtime Verification*, pages 349–358. Springer, 2017.
- [74] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A fast dynamic packet filter. In *NSDI*, volume 8, pages 279–292, 2008.
- [75] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hingqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. USENIX Association, 2018.
- [76] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. Apunet: Revitalizing gpu as packet processing accelerator. In *NSDI*, pages 83–96, 2017.
- [77] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.
- [78] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *NSDI*, pages 67–82, 2017.
- [79] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *NSDI*, pages 113–129, 2017.
- [80] Getting started with threatexchange, 2018. <https://developers.facebook.com/docs/threat-exchange/getting-started/v3.1>.
- [81] Ibm x-force, 2018. <https://www.ibm.com/security/xforce>.

Dataplane equivalence and its applications

Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu and Costin Raiciu

firstname.lastname@cs.pub.ro

University Politehnica of Bucharest

Abstract

We present the design and implementation of `netdiff`, an algorithm that uses symbolic execution to check the equivalence of two network dataplanes modeled in SEFL [42]. We use `netdiff` to find new bugs in Openstack Neutron, to test the differences between related P4 programs and to check the equivalence of FIB updates in a production network. Our evaluation highlights that equivalence is an easy way to find bugs, scales well to relatively large programs and uncovers subtle issues otherwise difficult to find.

1 Introduction

Misconfigured or faulty networks ground airplanes, stranding thousands of passengers and render online services inaccessible for hours on end, leading to disgruntled users and massive losses in revenue. Network verification promises to fix such rare yet devastating problems by ensuring that networks always follow their operator’s stated policy. Verification proposals can uncover faulty dataplane configurations [19, 29, 42, 30, 20], can simulate the effect of control plane changes (such as configuration changes) before they are applied [8, 11, 4] or inject these changes into an emulated clone of the live network to examine their effects [26]. The key behind the success of network verification in traditional networks is the simplicity of the policy, a mix of reachability and isolation constraints that administrators can readily specify once and verify recurrently.

As networks become more dynamic and programmable, both ensuring network correctness and specifying policy are significantly harder. Virtual networks are instantiated dynamically in cloud networks on tenant demand by massive software stacks, potentially developed by multiple players (e.g. Openstack); here, the key challenge is to ensure that tenant demands are implemented correctly and that tenant traffic is properly isolated from other tenants.

Languages such as P4 [5] or Flowblaze [34] allow the implementation of customized packet processing logic that can be deployed and run at wire speeds on real switch hardware (e.g. Barefoot’s Tofino). Specifying the behavior of programmable dataplanes entails specifying the expected output packet(s) for every possible input packet; such a specification relies on formal methods and expert time [38, 35, 46], being out of reach of network administrators and programmers.

We observe that, in many cases, dataplane correctness properties can be specified implicitly by equivalence to other dataplanes. A P4 programmer might need to restructure

or trim his program to meet the target switch constraints [16, 39] while preserving the functionality. In cloud computing, the abstract network configuration provided by tenants (e.g. two VMs connected via a L2 network) is translated by the cloud management software into an actual configuration for switches and routers that must offer *equivalent* functionality to the two VMs. Finally, a network administrator that knows his network behaves correctly¹ simply wants the network to behave in the same way in the future.

Checking equivalence could therefore be very useful for easy-to-use verification of modern dataplanes. Unfortunately, checking equivalence between arbitrary programs is a well-known undecidable problem. Variants of it, however, are decidable for domain-specific programming languages; in the networking field, NetKAT [3], NOD[29] and HSA [19] support various forms of equivalence checking. These languages, however, are not expressive or not efficient enough to check programmable dataplanes such as P4.

In this paper we show that checking equivalence is possible for programmable dataplanes and that it scales well enough to uncover many interesting bugs. `netdiff`, our proposed algorithm, is implemented on top of the Symnet symbolic execution engine and can test the equivalence of two network dataplanes expressed in the SEFL language [42]. We formally prove `netdiff` correctly decides if two dataplanes are equivalent when they do not contain infinite loops; we rely on prior work to detect infinite loops [43].

We have used `netdiff` to find bugs in Neutron, OpenStack’s cloud management software networking driver, by checking the equivalence of tenant configurations and the low-level implementation of those configurations. We have found ten implementation bugs in Neutron, three of which were unknown, and four configuration bugs. We have also used `netdiff` to check that P4 program optimizations preserve correctness, to test different dataplane models of the same network functionality are equivalent, to detect routing changes in a university network and to check that a FatTree instance behaves like a single, big switch. `netdiff` runs all these tasks in seconds/minutes. Finally, to enable scalability to a large Neutron deployment, we rely on a compositional verification approach where we test equivalence for independent components in isolation.

2 Goals

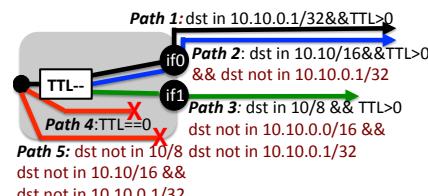
Network dataplane equivalence has many potential applications, a subset of which we explore in detail in §6. To guide our exposition, we use as running example the code snippets

```

If (TTL>1) TTL--;
Else Fail;

If (dst in 10.10.0.1/32)
    Forward("if0");
Else If (dst in 10.10.0.0/16)
    Forward("if0");
Else If (dst in 10.0.0.0/8)
    Forward("if1");
Else Fail;

```



(a) Basic router coded with If/Else: code(left) symbex(right)

Figure 1: Two SEFL programs modeling a router with three entries in its FIB. Are they equivalent?

in Figure 1 that model a router with three FIB entries; the code is adapted from [42]. Despite its simplicity, the example highlights well the difficulty of equivalence checking.

The first program is simple to understand as it relies on a sequence of If/Else clauses that forward the packet to the correct output port; there is one If per FIB entry. The second program is optimized to enable faster dataplane verification: it does not use If instructions at all, first Forking the packet (i.e. creating clones of it) and then using the Constrain instruction for each clone to restrict the packets that may leave on each port. Constrain drops all packets that do not match the constraint and has no effect on packets that do.

The two programs are meant to be equivalent; informally this means that injecting *any* packet into equivalent input ports of the two programs (e.g. *in*) will result in both dropping the packet, or both emitting the same packet(s) on equivalent output ports. Even though the two programs seem trivial, checking their equivalence is not possible today.

Our goal is to automatically and scalably decide if two dataplane programs are equivalent.

Before we discuss possible solutions, we first give a formal definition of equivalence. Let *Prog* denote the set of programs - defined as mappings (functions) between *Ports* (function names) and instructions. Let *Packet* denote the set of all admissible input packets. Injecting a packet *p* into a program *prog* at port *port₀* will result in a set of output packet and port pairs *O(prog, p, port₀)* defined as follows.

Definition 2.1 Let $O : \text{Prog} \times \text{Packet} \times \text{Ports} \rightarrow 2^{\text{Packet} \times \text{Ports}}$:

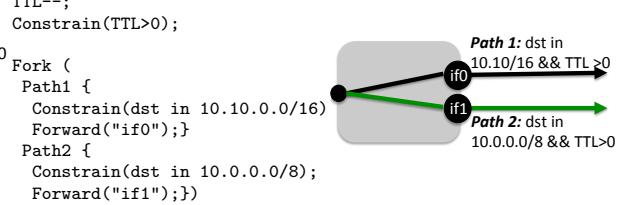
$$O(\text{prog}, p, \text{port}_0) = \{(\sigma_1, \text{port}_1), (\sigma_2, \text{port}_2), \dots, (\sigma_n, \text{port}_n)\}$$

be the set of packet and output port pairs resulting from the execution of *prog* given packet *p* on input port *port₀*.

We define **network equivalence** as follows:

Definition 2.2 Let $p \in \text{Packet}$ an input packet, $P_1, P_2 \in \text{Prog}$, Ports_1 and Ports_2 the program ports of P_1 and P_2 respectively. Let \mathcal{I} and \mathcal{R} be partial injective functions between Ports_1 and Ports_2 called input and output port correspondence respectively.

We call programs P_1 and P_2 **equivalent** with respect to input packets $Q \subseteq \text{Packet}$, input ports $\text{port}_1 \in \text{Ports}_1$ and



(b) Optimized router: code(l) symbex(r)

$\text{port}_2 \in \text{Ports}_2$ s.t. $\mathcal{I}(\text{port}_1) = \text{port}_2$ and output relation $\omega \subseteq \text{Packet} \times \text{Packet}$ iff $\forall p \in Q, \exists \chi$ bijection between $O(P_1, p, \text{port}_1)$ and $O(P_2, p, \text{port}_2)$ s.t.

$$\begin{aligned} \chi(\sigma_{1i}, pc_{1i}) = (\sigma_{2j}, pc_{2j}) &\iff \\ \mathcal{R}(pc_{1i}) = pc_{2j} \wedge (\sigma_{1i}, \sigma_{2j}) \in \omega & \end{aligned}$$

Definition 2.3 We call P_1 and P_2 **equivalent** with respect to $Q \subseteq \text{Packet}$ and $\omega \subseteq \text{Packet} \times \text{Packet}$ iff $\forall (\text{port}_1, \text{port}_2) \in \mathcal{I}$ P_1 and P_2 are equivalent w.r.t. Q , port_1 , port_2 and ω .

Intuitively, the above definition goes to say that given the same input packet, the number of output packets from both programs coincide and there must be a one-to-one correspondence between packets emitted by both programs. Also, packets in correspondence must satisfy the output packet condition ω , which typically requires that the values of selected header fields in the two packets are equal.

It is the verifier who provides \mathcal{I} , \mathcal{R} and ω . For example, in Figure 1, \mathcal{R} maps *if0* and *if1* in a) to ports with the same name in b) and \mathcal{I} maps the input port in a) to that in b). ω usually identifies a subset of header fields which must be equal. In our running example, two packets are equivalent if the *ttl* and *dst* fields are equal. In our evaluation, we use sensible defaults for these functions - L2, L3 and L4 fields.

3 Approaches to checking equivalence

Exhaustive testing for all inputs is one way to test equivalence, but it is not feasible to use in practice for networks: network headers size are 64B or more, meaning that one needs to test 2^{512} possible packets.

Existing work on dataplane verification allows us to scalably explore how packets are processed by a dataplane [19, 3, 29, 20, 42, 7, 30]. Intuitively, all these works try to find equivalence classes of packets that are handled in the same way, and explore their processing in one go; as long as the number of such classes is small, these tools can fully characterize dataplane processing without needing to explore each individual packet. We will take the same approach to answer whether two dataplanes are equivalent.

To enable scalability, all dataplane analysis tools restrict the language in which the dataplane can be described, place additional constraints on possible encapsulations and use optimized data structures to track packet equivalence classes.

All these choices limit the extent to which we can check equivalence; we will come back to these limitations after we discuss the equivalence properties we seek to capture.

All dataplane verification tools are able to predict the allowed values of packet header fields as they exit a given network port. The simplest way to check equivalence is to compare which packets can exit any given port by examining the feasible values for each header field—we call this **output equivalence**.

In the example in Figure 1, consider the two `if1` ports: compared to the basic model, the optimized model wrongly allows more packets to pass (packets in $10.10.0.0/16$), so the two paths are not equivalent, and thus the models are not equivalent for ports `if1`. If, however, we consider the two `if0` ports, we find that $\text{TTL} \in (0, 255]$ and $\text{dst} \in 10.10.0.0/16$, thus the two paths are equivalent.

The careful reader will have noticed that the two routers differ in how they treat packets when TTL is 0. The basic router will drop the packet straightaway. However, the optimized router will decrement the TTL regardless of its value, and when it is zero it will wrap around to 255 as TTL is unsigned—thus, the constraint $\text{TTL} > 0$ always holds. The two models are not equivalent, but checking just output equivalence is not enough to capture this problem.

The next step is to also check the constraints applied on the original (input) values of the header fields, before any modifications are made; when combined with output equivalence checking, we are now checking for **input** and **output equivalence**. With input/output equivalence, we will find that the basic model only allows packets to pass when $\text{TTL} > 1$ while the optimized model allows packets when $\text{TTL} \neq 1$; as the two ranges are not the same, the two models are not equivalent.

Checking for input/output equivalence is necessary to find bugs, but on its own it is still not sufficient. To see why this is the case, consider two trivial models where one leaves the TTL field unchanged, while the other executes the instruction $\text{TTL} = 255 - \text{TTL}$. Both the input values (0-255) and the possible output values (0-255) of the two models are the same, yet they are obviously not equivalent. What we also need is **functional equivalence**: regardless of the initial value of the TTL, the two values of the TTL after executing the two programs should always be equal. In our example, functional equivalence is not true because the condition $\text{TTL} = 255 - \text{TTL}$ never holds.

Note that all three checks are simultaneously needed to ensure equivalence: removing a single check leads to wrong results. We have already shown that input/output equivalence is not sufficient and functional equivalence is needed. Let’s show that any combinations of two checks is insufficient.

Do functional and output equivalence imply two models are equivalent? Consider one program that simply sets $\text{TTL} = 0$, and another that runs `Constrain(TTL>100); TTL=0`. The output is always 0, and for any allowed packet we have both functional and output equivalence. Yet, the first model

Algorithm	Input	Output	Equivalence Func.	Expressiveness
HSA[19], Veriflow[20]	✓	✓	✗	forward,filter
NetKAT [3]	✓	✓	✓	switch,filter
NOD [29]	✓	✗	✗*	forward,filter,tunnel
Dobrescu[7],Symnet[42]	✓	✗	✓	programmable dataplane (e.g. P4)
UC-KLEE[36]				

Table 1: Checking equivalence with existing tools.

allows all packets through, while the second only allows those with $\text{TTL} > 100$; the programs are not equivalent.

Input and functional equivalence are also insufficient. Compare a NoOp program with one that forks the packet. These two are equivalent from an input and functional point of view, however they are not equivalent on output: the first emits a single packet while the secqond emits two.

3.1 Existing solutions fall short

Existing dataplane verification tools (see Table 1), cannot check equivalence for programmable dataplanes. Header Space Analysis [19] and Veriflow [19] optimize for OpenFlow-like processing by tracking equivalence classes of packets through the network. Both are fast and their outputs can be fed to SMT solvers to check for output and input equivalence. Unfortunately, they do not track (symbolic) assignments and cannot scalably check functional equivalence.

NetKAT [3] offers a strong theoretical foundation to OpenFlow verification by reducing it to a Kleene algebra with tests. They show that equivalence is decidable in this algebra, and offer an efficient equivalence checking algorithm [9]. Compared to HSA and Veriflow, NetKAT supports assignment but lacks support for arithmetic operations. As such, it cannot express programmable dataplanes.

Network-optimized datalog [29] uses datalog to express network processing and policies. NOD is more expressive than prior tools because it also supports arbitrary tunnels, and checking equivalence is just another datalog query that can be fed to Z3 [6]. On the downside, it is very difficult to use datalog queries to reason about packet multiplicity on various ports. Furthermore, NOD’s difference-of-cubes is very inefficient for arithmetic operations, both space-wise ² and computation-wise ³. Thus, NOD does not support neither output nor functional equivalence.

Symbolic execution for network dataplanes has been proposed by Dobrescu et al.[7] and Symnet [42]; it tracks the symbolic values of header fields and supports assignment, encapsulation and arithmetic operations. Symbolic execution is expressive enough to analyze programmable dataplanes as shown by recent work [40, 10, 32]. While symbolic execution has traditionally been plagued by poor scalability, applying it to dataplanes has been shown to scale quite well.

Checking dataplane equivalence via symbolic execution is not supported by [7, 42, 40, 32, 10], but prior work from program symbolic execution can be adopted. UC-KLEE is the

leading proposal [36]: it can check for all types of equivalence for standard programs, but is not expressive enough to deal with packet duplication, a common primitive in network processing. We present `netdiff`, our algorithm that fixes this shortcoming.

4 Dataplane equivalence with `netdiff`

`netdiff` uses symbolic execution to show equivalence of two dataplanes according to definition 2.2. To enable scalability and expressiveness, we consider network dataplanes written in the SEFL language that only provides a set of basic instructions such as `if then else`, a filter (`constrain`) instruction, variable assignments and jumps to predetermined locations in the program called *ports*. The specificity of dataplane processing consists in the existence of an additional cloning instruction (`fork` in SEFL) which produces multiple copies of the same packet and pushes them on different paths through the network.

The symbolic execution state of a dataplane program is represented by a set of variables (packet header values and associated per-flow state) and a program counter which indicates the next instruction to be executed. A path through such a program is a list of program counters. Symbolic execution begins at some initial *port*, takes a *packet* as input, where some or all header fields can have symbolic values, and produces a set of packets issued on output *ports*. Symbolic execution is a method to exhaustively infer predicates on the input variables of a program in order for the execution to take some path [22]. The outcome of symbolic execution is a set of pairs comprised of a *path condition* and the corresponding *path*. The *path condition* is a logical proposition required by the inputs to a program such that execution will follow a certain path through the program.

For concreteness, consider Figure 1 where we inject at router input a packet with symbolic TTL and IP destination address (*dst*) fields, meaning they can take any value allowed in their range. The figure shows the symbolic execution of the two programs in our running example, the resulting paths and path conditions.

When a branch condition depends on a symbolic variable, the symbolic execution engine uses a constraint solver to check if the condition is satisfiable: if it is, the constraint is recorded in the path and the execution continues on the “then” branch (path). At the same time, the engine checks whether the negated constraint holds, and if it does it also continues execution on the “else” branch, recording the negated constraint that must hold. Both paths are now explored until they finish, independently. For instance, in the basic router program, the first If branch results in a path where the TTL is at least 2, then decrements the TTL and forwards the packet to the appropriate interface(s). The else path where the TTL is 0 or 1 is also explored, but it stops immediately because the packet is dropped.

Algorithm 1 `netdiff` equivalence algorithm

```

1: function EQUIVALENCE( $M_1, M_2, i_1, i_2, p_0$ ) ▷ Are  $M_1$  and  $M_2$  equivalent for input symbolic packet described by predicate  $p_0$  injected on ports  $i_1$  and  $i_2$ ?
2:    $Q_1 \leftarrow \text{DataplaneSymbex}(M_1, i_1, p_0)$ 
3:   for all  $(q_1, \pi_1) \in Q_1$  do ▷ for each path  $\pi_1$  and path condition  $q_1$ 
4:      $Q_2 \leftarrow \text{DataplaneSymbex}(M_2, i_2, q_1)$ 
5:     for all  $(q_2, \pi_2) \in Q_2$  do
6:       if  $\neg EQP(\pi_1, \pi_2, q_2)$  then
7:         return false
8:       end if
9:     end for
10:   end for
11: end for
12: return true
13: end function

```

`netdiff`, our proposed algorithm, is shown in Algorithm 1 and uses symbolic execution to check for equivalence between two SEFL programs. `netdiff` takes as input SEFL programs M_1 and M_2 and injects a set of packets given by predicate p_0 into the user-specified input ports i_1 of M_1 and i_2 of M_2 . The procedure *DataplaneSymbex*(M, i, p), described in detail in subsection 4.1, performs symbolic execution for program M starting at input port i with a symbolic input given by predicate p on the set of all possible input *Packets*. Each π resulting from symbolic execution represents a path-set, which is an individual path or a set of paths that have the same path condition q (the latter captures cloned packets).

`netdiff` follows a similar approach to UC-KLEE [36]: it performs symbolic execution of M_1 (line 2) and then, for each resulting pathset (q_1, π_1) , performs symbolic execution of M_2 starting with initial symbolic packet described by the path condition q_1 (line 5). The algorithm then compares each resulting pathset (π_2) to (π_1) for the packets in q_2 (line 7).

`netdiff` ensures input equivalence by design because the union of the sets of packets described by all predicates $q_2 \in Q_2$ must be equal to the set described by q_1 , and all sets described by q_2 predicates are disjoint (see Lemma 1). The *EQP* predicate’s job is to check for output and functional equivalence for each pair of outputs. There are two main differences between `netdiff` and UC-KLEE, both stemming from packet cloning, that we describe in detail below:

1. The `fork` instruction can result in multiple paths for the same set of input packets; for `netdiff` to work correctly, the standard symbolic execution is followed by processing that groups output paths that have overlapping path conditions into pathsets (see §4.1).
2. Finding the right path equivalence predicate *EQP* to reflect Definition 2.1 is also tricky. For sequential imperative languages, this predicate is a simple equality check for the output values of the two paths being compared. To cope with packet cloning, we need to compare pathsets instead of individual paths (see §4.2).

Algorithm 2 Dataplane symbolic execution

```

1: function DATAPLANESYMBEX( $M, i, p_0$ )  $\triangleright$  Run symbolic execution by
   injecting the symbolic packet described by predicate  $p_0$  in port  $i$ .
2:    $Q \leftarrow \text{Symbex}(M, i, p_0)$ 
3:    $L \leftarrow \emptyset$   $\triangleright L$  holds pathsets with disjoint conditions
4:   for all  $(q, \pi) \in Q$  do  $\triangleright$  Sieving algorithm to collapse paths
5:     for all  $(l, s) \in L$  do  $\triangleright$  with overlapping path conditions.
6:       if SAT( $q \wedge l$ ) then
7:          $L \leftarrow L \setminus \{(l, s)\}$ 
8:          $L \leftarrow L \cup \{(q \wedge l, s \cup \pi), (l \wedge \neg q, s)\}$ 
9:          $(q, \pi) \leftarrow (q \wedge \neg l, \pi)$ 
10:      end if
11:    end for
12:    if  $q \neq \emptyset$  then
13:       $L \leftarrow L \cup \{(q, \pi)\}$ 
14:    end if
15:  end for
16:  return  $L$ 
17: end function
  
```

4.1 Dataplane symbolic execution

Algorithm 2 shows our dataplane symbolic execution algorithm. On dataplane code, standard symbolic execution *Symbex* returns tuples of (path condition, path) in the set Q , but the path conditions are not guaranteed to be disjoint, as they would be in a standard program. This can be seen in our running example (Figure 1.b): symbolic execution yields Path 1 and 2 with overlapping path conditions ($dst \in 10.10/16$).

Dataplane symbolic execution performs sieving to eliminate path condition overlaps, returning *pathsets* with disjoint path conditions. To achieve this, the algorithm first runs standard symbolic execution, and then proceeds to group all paths that have overlapping path conditions (lines 6-9).

The result is collected into L , which starts as the empty set and always contains pathsets with disjoint path conditions. Whenever there is a path condition overlap (which we test with Z3), we remove the existing pathset from L and insert two new pathsets: one containing the overlapping condition $q \wedge l$ and the union of the paths, and one corresponding to original pathset with updated path condition $l \wedge \neg q$. The iteration then continues with the remaining path predicate $(q \wedge \neg l)$; note that the newly added entries are not revisited.

In the example from Figure 1.b, L has two pathsets: $(10.10/16, \{Path1, Path2\})$ and $(10/8 \setminus 10.10/16, \{Path2\})$.

The complexity of this algorithm is $O(|Q|^2)$; as symbolic execution can yield many paths, this cost can quickly become prohibitive. Changes to the symbolic execution procedure to reduce this cost to be proportional to the number of `fork` instructions are part of our future work. Instead, in our implementation we use a heuristic where we only run the algorithm for broadcast packets and disable it for unicast packets; the user can override this behavior via a command-line flag.

4.2 Equivalence between pathsets

Given pathsets O_1 (from M_1) and O_2 (from M_2) and path condition pc , we need to decide if the two pathsets obey func-

Algorithm 3 EQP predicate between two pathsets

```

1: function EQP( $O_1, O_2, pc$ )  $\triangleright$  True if bijection between  $O_1$  and  $O_2$  found
2:    $\triangleright O_1$  and  $O_2$  are pathsets and  $pc$  the path condition
3:   if  $|O_1| \neq |O_2|$  then
4:     return false  $\triangleright$  If cardinality different, there is no bijection
5:   end if
6:    $E \leftarrow \text{ComputeEdges}(O_1, O_2, pc)$ 
7:    $G = (V = (O_1 \cup O_2), E)$ 
8:   return MaxBipartiteMatching(G)
9: end function
  
```

Algorithm 4 Edge computation

```

1: function COMPUTEEDGES( $O_1, O_2, pc$ )  $\triangleright$  Return the adjacency matrix
2:    $\triangleright O_1$  and  $O_2$  are pathsets,  $pc$  the current path condition
3:   for all  $(\sigma_i, p_i) \in O_1$  do
4:     for all  $(\sigma_j, p_j) \in O_2$  do
5:        $E[i][j] = p_i \mathcal{R} p_j$   $\triangleright$  output port equivalence
6:       if  $E[i][j]$  then
7:          $E[i][j] = \neg \text{SAT}(pc \wedge \neg(\sigma_i \omega \sigma_j))$   $\triangleright$  Func. equiv.
8:       end if
9:     end for
10:   end for
11:   return  $E$ 
12: end function
  
```

tional and output equivalence.

First, consider the simple case where the two pathsets have exactly one path each. Let σ_1 and σ_2 denote the values of the header fields for the two paths at the output ports, expressed as constants or functions of the symbolic header values at input. To decide equivalence according to Definition 2.1 we must check whether: (1) the two paths exit on equivalent output ports and (2) the output packet values satisfy the ω relation (i.e. the header fields of interest are the same for all input packets described by pc).

`netdiff` checks equivalence between two paths as follows: $p_i \mathcal{R} p_j \wedge \neg \text{SAT}(pc \wedge \neg(\sigma_1 \omega \sigma_2))$. The port check is obvious; the second check asks the solver for an input packet that satisfies pc and results in output packets that are not equivalent. If the check is not satisfiable, functional equivalence holds for all packets allowed by path condition pc . In this case, there is a single path in each pathset and input and functional equivalence guarantee output equivalence.

Now, consider the case where the two pathsets have different number of paths. This implies the two programs are not equivalent because there is some input packet which results in a different number of output packets being emitted.

Finally, consider the remaining case where the two pathsets have the same cardinality $N > 1$. To check equivalence we must find a bijective mapping between the paths in O_1 and O_2 , i.e. each path in O_1 must have a path in O_2 that is equivalent, and all paths in O_2 must have an equivalent in O_1 . If such a mapping exists, then the two pathsets are equivalent, otherwise they are not.

We now show that finding this bijection can be reduced to the classical problem of maximum bipartite matching (MBM). MBM takes as input a bipartite graph $G = (V = (X \cup Y), E)$ with $X \cap Y = \emptyset$, where X is the set of workers

and Y is the set of tasks such that $|X| = |Y|$. If worker i qualifies for job j , there is an edge from i to j , i.e. $E[i][j] = 1$. In MBM, no worker can take more than a job; the algorithm decides whether all workers can get jobs.

The EQP equivalence algorithm is implemented in Algorithms 3 and 4. O_1 and O_2 are the pathsets to be compared; their paths will form the vertices of our bipartite graph. We have an edge between two paths if the paths are matching: $E[i][j] = 1$ iff the output ports are corresponding w.r.t. \mathcal{R} and the output packets are equivalent w.r.t. ω . EQP calls MBM to find whether all paths in O_1 have an equivalent in O_2 . If the answer is positive, then is a bijection between these paths which guarantees both functional equivalence and output equivalence. As input equivalence is ensured by the way in which we run symbolic execution, netdiff correctly decides whether the two programs are equivalent.

The bottleneck in EQP is the necessity of computing the *output equivalence* between all pairs of paths in O_1 and O_2 in order to derive the graph (Algorithm 4, line 6). Computing satisfiability of a complex formula is a hard problem and dominates the entire equivalence decision procedure. That is why our algorithm first checks for output port equivalence; when this check fails, the solver call in Line 6 is not made.

4.3 Correctness and complexity

It is worth noting that we can decide the equivalence of two dataplane programs only if they terminate on all inputs. netdiff uses an existing loop detection algorithm to detect infinite loops [43], which leads to the following possibilities:

- Both programs are infinite-loop free. We can decide on their equivalence with a worst case complexity of $C(\text{DataplaneSymbex}_1) + n \cdot (C(\text{DataplaneSymbex}_2) + m \cdot p^2 \cdot C(\text{SMT}))$, where n and m are the number of M_1 's and M_2 's pathsets, p is the maximum number of paths in any pathset and $C(\text{SMT})$ is the complexity of the SMT solver.
- When only one of the programs contains an infinite loop, equivalence is decidable - the programs are not equivalent.
- When both programs have infinite loops, we cannot decide

Theorem 4.1 $\text{EQUIVALENCE}(M_1, M_2, i_1, i_2, p_0)$ is true iff M_1, M_2 are equivalent w.r.t. $Q = \{x \in \text{Packet} \mid p_0(x) = \text{true}\}$ and input ports i_1, i_2 .

Proof: Because of the properties of symbolic execution listed in Appendix A, all path conditions corresponding to pathsets resulting after symbolically executing M_2 are equivalence classes of the initial input space w.r.t. the outcomes of M_1 and M_2 . The full proof can be found in Appendix A. \square

5 Implementation

Our implementation of netdiff takes as input two SEFL programs, together with two correspondence maps, one be-

tween their input ports and the other with respect to their output ports. By default, it injects basic Ethernet/IP/{TCP, UDP, ICMP} packets into the provided input ports and checks for equivalence between a number of header fields including IP and Ethernet source and destination, TTL, IP protocol, L4 ports, etc. Note that both the symbolic input packet and the fields to be checked can be customized by the user if needed.

When equivalence fails, netdiff outputs a series of tuples (*Input condition, Offending path in program 1, Offending path in program 2, reason*), where the reason may be either different number of output packets, unmatched ports, unmatched output fields. A path in program x is a series of symbolic output packets together with a list of visited ports and either the output port or the failure reason.

We ran netdiff on the example shown in Figure 1. It takes around 200ms to check equivalence and output the results to file. The tool reports 10 symbolic packets that were treated differently by our two programs. The first five packets catch the TTL decrement bug in the optimized model (the TTL underflows when it is zero). The other five packets highlight the second problem: there is an overlap between packets exiting on port `if0` in the basic model and `if1` in the optimized model - i.e. a packet in `10.10/16`. netdiff's output makes it easy to find the bug and then correct it.

Core implementation. The core of netdiff is implemented in just 200 lines of Scala code. We also implemented a series of additions and changes to Symnet resulting in cca. 2kLOC that ensure netdiff behaves correctly. In detail, we implemented sieving (Algorithm 2) that takes the outputs of Symnet—a list of tuples (Input condition, single path through network)—and outputs pathsets of the form (path condition, paths) with non-overlapping path conditions. Furthermore, we changed the internal representation of Symnet's state to allow a clean separation between path conditions and current packet values, in order to make netdiff's internals more scalable and easier to test. To aid debugging and model validation, we also implement an input generator which produces an example packet from a path condition.

5.1 OpenStack Neutron Integration

netdiff requires two SEFL programs to check equivalence. To generate SEFL programs for our evaluation, we have both created new translators (see below) and re-used existing ones: Vera to translate from P4 programs to SEFL [40] and existing translators from router FIBs to SEFL [1].

Checking OpenStack Neutron is our most significant use of equivalence so far; it required 15KLOC of Java, Scala and ANTLR4 parser grammars to automatically integrate with Neutron and translate to SEFL. Out of this, only 1KLOC is OpenStack-specific, while the biggest part consists of reusable translators for iptables, OVS, ipsets, which are widely deployed in numerous scenarios - e.g. Kubernetes. Furthermore, this work is one-time only and can be further used for any other verification purpose. We contrast this effort to

other verification techniques which also require writing and maintaining correctness specifications. We describe our implementation here.

OpenStack is an open-source cloud management platform. Similarly to other cloud platforms (e.g. EC2), OpenStack abstracts away the complexity of the provider’s infrastructure, allowing the tenant to create and manage virtual machines with ease. Tenants may connect their VMs in rich network topologies comprised of VLANs connected via routers and NATs, and also enforce security policies at VM level.

Neutron is the networking service of OpenStack and is implemented as a distributed middleware application which takes the *tenant network configuration* and implements it in the actual network. Neutron’s operation is complex as it depends on multiple software components, so bugs may occur anytime during the translation process leading to non-compliance to the tenant configuration. Neutron’s complexity and its distributed deployment makes manual troubleshooting cumbersome.

To verify whether Neutron correctly implements a given tenant configuration, we automatically generate two SEFL models and use `netdiff` to check their equivalence. The first model is derived from the tenant configuration. The second model is created from a snapshot of the actual Openstack deployment resulting after the VMs are instantiated and it includes OVS OpenFlow rules, iptables rules, etc.

Modeling tenant configurations. Our translator uses a tenant-level snapshot of the configuration (a dump of the Neutron database in practice) and then generates SEFL code that implements each user defined resource (e.g. router, switch, NAT). The simplest abstraction is a virtual network which forwards packets according to a static CAM table mapping Ethernet addresses to virtual ports. Virtual routers perform L3 routing and provide virtual machines with access to and from the Internet (via NAT). Neutron also defines *security groups* which are rules that filter traffic at VM level. Each abstract object is translated separately to SEFL, and they are connected using links according to the tenant configuration. For more details, we refer the reader to [41].

Dataplane modeling. A Neutron deployment is usually implemented as interconnected Linux servers running a number of network processing tools. We implement parsers and SEFL translators for many of the Linux Kernel packet processing primitives - *iptables* rules, ipsets, OpenVSwitch (OVS) software switches [33], OpenFlow tables, VXLAN tunnels, routing tables, Linux Bridges, ARP tables. We then interconnect the distinct components based on the physical or virtual links acquired from the topology.

Both *iptables* and *OVS bridges* use similar concepts such as tables and rules which match against packet header fields or per-packet metadata and apply one or more actions. To translate such matches we generate simple If/Else constructs; provided that all *matches* in a rule are satisfied, an action will be fired which will either alter the state of the

packet (e.g. push a tunnel header) or alter the processing pipeline (e.g. drop or forward to further processing).

Modeling stateful processing in SEFL is straightforward as long as the state depends only on the given flow (i.e. it is not global state) [42]. We use a similar technique to model the connection tracking engine (or *conntrack*) implemented within the Linux Kernel. Conceptually, *conntrack* defines a connection as a 5-tuple and tracks it independently. To model *conntrack* we use two sets of metadata variables called *forward* and *backward expectations*. The former represent packets flowing in the same direction as the initial SYN packet, while the latter represent reply packets belonging to the same connection. When state is created for a connection (a *conntrack commit* action), we store it as metadata; the metadata is then checked when execution arrives at the *conntrack* module and the appropriate action is taken.

Dataplane modeling caveats. One of the issues that we encountered during our experiments was missing information from the dataplane snapshot which lacked ARP tables and switch CAM tables. A possible solution to bypass this issue is to simply modify the acquisition script to gather ARP tables for all Linux network namespaces and CAM tables for all OVS and Linux Bridges in the topology. However, these entries would only depict a transient state of the network dataplane with incomplete or stale information.

Our solution implies converging the ARP and CAM tables into a steady, concrete state. Following the observation that cloud provider middleware implements anti-spoofing techniques, we use the constraint solver’s capabilities to derive all possible ARP packets which may reach a certain point in the network. With this information, we infer (*IP, MAC*) pairs for all network namespaces in the system. We use a similar approach to infer all CAM tables on all switches in the deployment. We implement our approach in a tool called ARPSim and apply it to our department’s Openstack deployment containing 87 servers. For this deployment, ARPSim discovers 885 ARP entries in 4 minutes and infers 28889 CAM entries in L2 switches in 7 minutes.

Mapping ports. As discussed in §2, `netdiff` provides sensible defaults for mapping ports and deciding what packets are equivalent. For Neutron, we map virtual ports to corresponding OpenVSwitch tap interfaces (functions \mathcal{R}, \mathcal{I}).

6 Evaluation

We run our evaluation of `netdiff` on a server with a Xeon E5-2650 processor @ 1.7GHZ and 16GB of RAM. Our main goal is to understand whether `netdiff` can catch interesting bugs or behaviors in practice, for realistic network dataplanes. We examine a range of applications including Openstack Neutron, P4 program equivalence, and the correctness of FIB updates. Finally, we test `netdiff`’s scalability and contrast its performance to NoD [29]. The scenarios described below make use of header arithmetics and packet du-

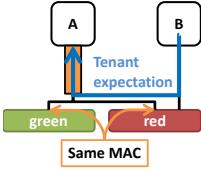


Figure 2: Identical trunked MACs

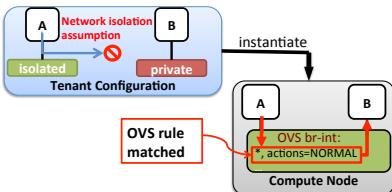


Figure 3: Network isolation bug

plication primitives and motivate the use of `netdiff` instead of systems like HSA, NetKAT or NoD.

6.1 Neutron bugs

To validate `netdiff`'s capabilities in finding production bugs, we begin our experiments by first reproducing known bugs and discovering a number of unknown bugs in Openstack Neutron in a small scale deployment in our lab. Our experiments here focus mostly on functionality.

While Neutron is only one of many cloud networking drivers, the same method applies to most network virtualization solutions deployed by commercial cloud providers.

Identical MAC Addresses for trunking ports. The first bug appears in the configuration shown in Fig.2: the tenant defines a topology with two VMs connected via 2 networks (red and green). Machine A is connected via a trunk port to both the red and green networks while machine B is only connected to the green network. The tenant-level expectation for this topology is that all packets from B towards A reach their destination. However, if A uses the same MAC for both its trunked ports this prevents communication between A and B. This bug was reported on the OpenStack Neutron bug tracker⁴. `netdiff` found a number of failed states which indicated that all packets leaving machine B were being dropped in the forward path by an anti-spoofing rule in the *br-int* bridge connecting the two machines.

Allowed address pairs bug. An allowed address pair is an extra IPv4, MAC address pair specifically tailored to allow bridging at VM level. Thus, the expected tenant-level behavior is that traffic with destination addresses in the list of allowed address pairs for a VM be allowed on egress.⁵

However, an implementation bug in the firewall module stops traffic from getting through. Thus, traffic issuing from VM A towards VM B is correctly forwarded to VM B, but the reverse traffic from VM B to VM A is dropped. The issue is correctly traced by `netdiff` which indicates failed (non-equivalent) states between the tenant and the provider perspectives. In the tenant view, A and B have bidirectional connectivity, whereas in deployment connectivity is broken. `netdiff` correctly captures the error in the reverse packet run and successfully identifies the offending rule.

No firewall enforcement on ICMP Type/Code. Security group support for ICMP filtering was not implemented for older versions of Neutron⁶. `netdiff` showed how ICMP traffic that is meant to be blocked is allowed in the dataplane.

Filtering with security groups. Security groups are collections of ACL rules that apply to all VMs part of that group. When specifying connectivity outside the group, tenants can use prefixes or remote security groups to specify the external source of traffic. There was a bug in the implementation of filtering when remote security groups were used. In our setup, we had two groups called *green* and *blue* and a rule that all traffic from the *green* group should reach the blue group⁷. We instantiated three VMs: A in the *blue* group, B in the *red* group, and C in both. At runtime, C could not be reached by neither A or B, violating the tenant configuration.

Inconsistent connections in the tracker. This bug appears when connectivity is repeatedly enabled and disabled for the same host-pair. We ran our test with VMs A and B, and the tenant allows traffic from A to B. In Neutron, all ACL rules are directional, and they are implemented using the connection tracker. In this case, A can initiate connections to B and B can respond, but B cannot initiate a connection to A.

After instantiation, A starts a connection to B which creates per-connection state in the conntrack module. Immediately afterwards, the tenant disallows traffic from A and B, which marks the connection in the conntracker as “dead” but does not delete the conntrack entry. When packets of the same connection reach conntrack, they will be dropped as expected. The problem appears when the tenant re-enables traffic from A to B: the flow entry mark is not cleared, and all subsequent packets are incorrectly dropped.⁸ We catch this bug by using symbolic conntrack state. The incorrect behavior is captured and reported, highlighting the offending rule and the conntrack conditions which trigger the behaviour.

`netdiff` captures the bug in less than 2 minutes in our simple deployment, but larger scale experiments indicate that using symbolic state variables significantly increase execution time. This is why we typically use an initially empty conntrack state for all servers in the topology.

A network isolation bug was discovered solely using `netdiff` and was reported as a Neutron bug⁹. In this setup, we have two machines running on the same host as in Figure 3, each connected to distinct VLANs. Assume that B is completely isolated from the rest of the network. Then, the expected behavior at tenant level is that no traffic from B can ever reach A. Next, assume host A is part of a permissive security group whereby ingress HTTP traffic is allowed; further assume that B knows A's MAC and IP addresses. Then, HTTP traffic from B will reach A, breaking isolation. This bug exists even when B belongs to a different tenant.

`netdiff` successfully detects the erroneous behavior, providing a packet from B that can reach A. To validate the bug, we successfully reproduced the behavior in the deployment. We note that the bug is difficult to catch with standard testing because ARP traffic was correctly blocked, and a simple http-ping from B to A would fail. Because `netdiff` uses symbolic packets, it finds a valid packet which will reach B.

Old Linux Kernel. This is a configuration bug that we stum-

bled upon when deploying Openstack in our testbed. Neutron’s OVS adapter needs kernel support to access the Netfilter’s conntrack module; support exists since version 4.3.

In our deployment, we had a compute node with kernel version 4.2. We deployed two VMs with security groups to allow all traffic between them. However, since there is no kernel support for connection tracking (as required by the firewall module), the insertion of security rules silently fails, and all traffic is dropped. `netdiff` caught this behavior by reporting successful execution at tenant level while the same input packet was dropped in the deployed dataplane.

Tunnel endpoint listening on *localhost*. The issue arises when Puppet, a provisioning tool, erroneously binds a tunnel endpoint on a compute node to the *localhost* address. The effect is that the VMs hosted on the affected compute node will not be able to communicate with VMs running on different compute nodes when in principle they should.

Hosts behind a NAT reachable from the outside. This issue was highlighted solely by running `netdiff` in a public-private network scenario. *Tenant A* creates a private virtual network and connects it to a public network via a virtual router, configuring source NAT on the external gateway of the router. He then deploys some virtual machines within his own private network and enrolls them in a permissive security groups allowing all ingress traffic. *Tenant B* also creates a virtual machine that he plugs directly into the public network and manages to find the IP address of the router’s external gateway and the VM’s private address.

Because the router is configured in SNAT mode, *A* would expect that no traffic from outside his network can initiate connections to any of his machines. However, Neutron virtual routers perform routing between the external and the internal network regardless and thus *B* can reach both *A*’s VMs.

ARP spoofing. Assume a VM responds to an ARP request by stating that the queried IP address can be found at a different MAC than the one defined on the VM’s interface. Neutron’s network abstraction asserts that no spoofing should be possible. Spoofing prevention is implemented by having the integration bridges¹⁰ perform port-based checking on ARP replies to ensure that IPv4 addresses cannot be modified (ARP.SPA field) and all L2 frames coming from the VM have expected L2 source addresses.

However, no explicit check is performed to ensure that the advertised L2 address (ARP.SHA) is the expected one. Thus, a malicious or corrupt ARP implementation in a VM may successfully transmit spoofed ARP pairs.

Unexpected interactions with libvirt. The following configuration bug arises when deploying *libvirt*-based NAT networking alongside Neutron’s iptables-based security groups mechanism. The *libvirt* toolset automatically creates a default virtual network with some prefix P (usually 192.168.122.0/24) and installs a series of iptables rules in the NAT table, POSTROUTING chain in order to perform

address translation for outgoing VM traffic. This issue was discovered in one of the production settings we evaluated.

Say a tenant creates a virtual network with prefix P , then all outgoing traffic from a VM in this network to other networks will be dropped. `netdiff` quickly discovers a non-equivalence whenever the IP source address is in P and the destination address is not in P . We reproduced the bug, validated the model in our testbed and discovered that indeed packets issued from network *A* were being NAT-ed due to the unwanted interference with the rules generated by *libvirt*.

Troubleshooting VM connectivity. A common configuration issue confirmed in production settings appears in tenant networks with many security groups. The tenant wishes to enable communication between two of its VMs but mistakenly adds them to different security groups (the groups may have similar names). By default, security groups are configured such that they allow ingress traffic from machines belonging to the same group, but not from other security groups. After deployment, the user notices that no traffic flows between its VMs.

Troubleshooting connectivity problems is difficult for tenants, as it requires manually checking the ports of a given VM, and the security groups which they belong to. With `netdiff`, the administrator is able to quickly assess that the tenant and provider perspectives are identical. Thus, there must be a misconfiguration at tenant level which does not meet the user’s expectation. Symbolic execution of the tenant topology indicates that all failed outcomes are due to an ingress security group which is not matched at *B*’s level.

Iptables optimizations. Llorente et al. [28] aim at shortening packet processing pipelines in order to enhance performance of Neutron’s iptables driver. We set out to check whether the optimization algorithm works correctly on a few inputs, i.e. it preserves the same packet processing behavior.

To achieve this, we deploy a one-node OpenStack deployment with 9 running VMs connected to 9 different security groups. Since all iptables optimizations are localized within the VM access Linux bridges, we only test equivalence at this component level. `netdiff` takes an average of 9.4s per VM to show that the optimization algorithm doesn’t break any of the underlying logic. The result shows little performance degradation with respect to normal symbolic execution of the same deployment using Symnet (7.3s).

6.2 Checking a large Neutron deployment

In order to test `netdiff`’s scalability and discover novel bugs, we used a snapshot from our department’s Openstack deployment. It consists of 87 compute nodes, running a total of 243 virtual machines. The deployment contains 14960 iptables and 11375 openflow rules which implement the 17 tenant-administered virtual networks and 2 public networks.

In the previous section, `netdiff` caught Neutron bugs in seconds by checking equivalence of tenant-level and

Network	# ports in network				
	1	3	26	57	164
Virtual (s)	0.03	0.03	0.04	0.02	0.02
Physical (s)	0.23	0.14	0.14	1.2	3.2
netdiff (s)	2.1	1.65	2.28	12.8	31.8

Figure 4: Neutron L2 reachability

provider-level network models. In the departmental deployment this approach is not feasible. For instance, when testing L3 unicast reachability, netdiff failed to finish processing due to exponential state explosion, a common issue of symbolic execution. First of all, the generated SEFL code was introducing a lot of useless branching in lookup tables - such as ARP tables. We used a state-merging technique to mitigate this issue [24]; the results show a significant decrease in processing time in some scenarios, but increases in others - whenever the number of elements in the table is small.

To further tame the complexity we used a compositional approach: we test equivalence between corresponding parts of the two dataplanes instead. For Neutron we tested three equivalence checkpoints detailed below. As the size of each component is small, the time to check equivalence is reduced to seconds or tens of seconds. Since our implementation is compositional by design, identifying distinct components and running netdiff against them entailed a small amount of extra work - cca. 200LOC.

L2 reachability equivalence can be checked by using a symbolic ARP request in both the tenant and provider network. In table 4, we show average execution times to check equivalence for layer 2 reachability using ARP broadcast probes, contrasted with plain symbolic execution in either network. As the number of ports size of the virtual network, equivalence checking time goes up from 2.1 seconds to around 30 seconds. To put the results in perspective, parsing all the configuration files and code generation take around 15s each.

Security group compliance. For each VM in the deployment we test if its implementation is compliant to the security groups (both ingress and egress) defined by the tenant. netdiff checks egress security groups quickly (200ms on average), and takes longer for ingress security groups (table 5) because tenants tend to use default-on egress policies.

Virtual routers must behave correctly with respect to their expected functionality including routing, floating IPs and source NAT. The physical implementation of a layer 3 router is well delimited within the boundaries of a Linux network namespace, so we can simply *clog* outgoing corresponding interfaces in both virtual and physical topologies and inject packets at input interfaces. Depending on the number of router interfaces, as well as on the number of floating IPs defined therein, the time for equivalence checking goes up to 80s for a router with 48 floating IPs.

Security group	# rules in security group ACL				
	2	6	12	13	19
Virtual (s)	0.08	0.08	0.09	0.08	0.09
Physical (s)	0.12	1.09	1.51	2.48	1.57
netdiff (s)	0.24	1.74	6.87	8.51	2.47

Figure 5: Ingress security groups

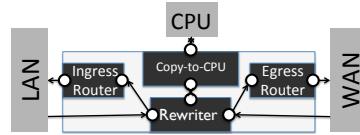


Figure 6: A modular P4 NAT

6.3 P4 equivalence

P4 [5] is a high level language that enables programming dataplanes and can also be efficiently implemented in hardware. Despite its apparent simplicity, coding P4 programs is tricky: unexpected behaviors may be accidentally introduced during the design or runtime phase. In this section we show how netdiff can be used to determine behavioral equivalence between different P4 programs with seemingly identical dataplane configuration and functionality.

Monolithic NAT vs modular NAT. One of the simplest P4 tutorials is a NAT that includes three distinct pieces of functionality: a *NAT rewriter*, which simply sets packet fields to given mappings, two *routers*, one for the LAN and one for the WAN, each of which performs longest prefix matching, assigns next hop address and selects the proper output port and a *CPU redirector*, which encapsulates a packet and sends it to a control-plane application if no NAT mapping is found. Even for a simple set of table rules, understanding the interactions between these different pieces is difficult.

To check whether the functionality of our P4 NAT works correctly, we wish to compare it to a modular design that runs different functionality in separate P4 programs which are connected. We still prefer the monolithic approach for deployment because it is cheaper to implement and performs better at runtime than our modular design that serializes and de-serializes packets between the interconnected boxes.

In Figure 6 we show how the modular NAT works. We used Vera [40] to generate models for both NATs and used netdiff to check whether they are equivalent. In around 5s, netdiff shows that the implementation of the monolithic NAT is not equivalent to the modular implementation. In the monolithic implementation, it is possible to translate a packet intended for the LAN and then send it on the LAN interface. The same behavior is not possible with the modular NAT because the routing tables corresponding to LAN and WAN networks are split into 2 distinct routers - one for *ingress* and one for *egress*.

Which is the correct order of table application? In our next example, we take the simple router P4 tutorial and we enhance it to enable ACL processing. Assume that the P4 programmer initially instructs the ingress pipeline to first route packets and then apply ACL. In a subsequent run, the programmer decides to reverse the order in which the two tables are applied in order to avoid routing packets that would be dropped by the ACL; this approach seems more efficient. In our program, the ACL table has two actions: one that drops packets and one that passes them through (see Fig. 7).

When we compare the two programs with netdiff, it is

```

ACL table:
table acl {
    reads { ipv4.srcAddr : exact; }
    actions { _drop; _nop; }
}
Entry:
table_add acl 10.0.0.3 _drop

Attempt 1:
apply(ipv4_lpm);
apply(acl);

Attempt 2:
apply(acl);
apply(ipv4_lpm);

```

Figure 7: Ways of adding an ACL to a P4 router.

surprising to find they are different. The first version matches our expectation that unwanted packets (10.0.0.3) are indeed dropped. The second version surprisingly allows all packets through. This is because the packet is not actually dropped in the ACL table. The P4 spec states that a drop action within the ingress pipeline only marks the packet for rejection and continues execution from that point on. When the dropped packet hits the ipv4.lpm table, the default action sets the egress spec to that of a valid interface, reviving the packet.

Trimming switch.p4 to size Recent work on verifying P4 programs [27, 40, 32, 10] highlights the difficulty of programming correct P4 programs. Instead of writing software from scratch, network operators can use existing catch-all implementations and adjust them to their needs. An example is `switch.p4` [45] which provides a full implementation of production-ready ToR switch. `switch.p4` contains 131 tables and a total of 6KLOC; deploying this program is wasteful when one does not need all the functionality therein. If fewer tables are synthesized, they can hold more match-action rules allowing scope for specialization.

We used `netdiff` to help us trim `switch.p4` while maintaining equivalence for IPv4 processing. We began with a working configuration of `switch.p4`, including concrete table entries for all entries. We then iterated by (1) removing functionality irrelevant for basic IPv4 routing and (2) generating the SEFL model for the resulting program using Vera’s translator [40] (3) checking equivalence to the full program. In all our tests, equivalence testing took between one and two minutes, depending on the number of table entries.

We confirmed that much processing was not needed for the correct functioning of v4 routing and can be safely removed: IP sourceguard, QOS processing, sFlow, Integrated services, Storm control, MPLS, etc. Removing these from the source code significantly reduces the total size of the tables (e.g. only sourceguard had space saved for 1500 entries, and INT processing makes up for 10% of the LOC in `switch.p4`). On the other hand, setting ingress port mappings, validating the outer header, handling VLANs, as well as the obvious IPv4 processing (longest prefix match, reverse path forwarding checks, etc.) are needed for correct functioning.

6.4 Monitoring FIBs in a production network

For the last six months we have been using `netdiff` to monitor routing in our university’s network. For each of the 9 routers that make up the network core, we have taken a FIB snapshot every 6 hours, and then checked the equivalence between FIB snapshots from the same router. We aim

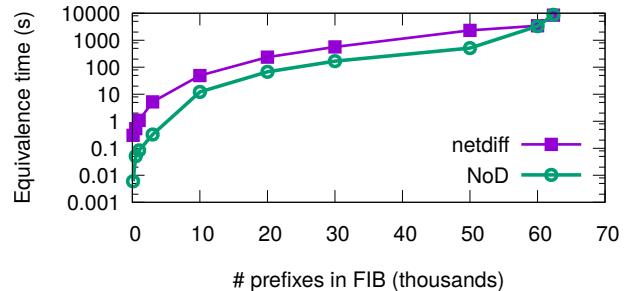


Figure 8: Router equivalence checking: `netdiff` vs. NOD to help our admin understand how routing changes over time. The snapshots vary in size from about 700 entries up to 20K entries for the core router. Equivalence takes 100ms to compare the smallest FIBs, and up to 50s for the largest ones.

Only in rare cases (about 5%) two snapshots of the same router FIB are equivalent. However, most of the time, the differences were due to churn in directly connected hosts (92%). `netdiff` did uncover interesting differences: a few routers had empty FIBs after recovering from a failure (1%) or were missing certain routes due to link-failures (2%).

6.5 Is my datacenter network one big switch?

Fat-trees [2] are the de-facto standard datacenter network, and they aim to provide a *big-switch* abstraction to end-hosts. We use `netdiff` to check whether the myriad of interconnected switches is equivalent to a single switch to which end-hosts are directly connected. We used Batfish [8] to generate the data planes for a fat-tree network with 125 switches (50 edge, 50 aggregation and 25 core switches). The question we asked is whether every port of every edge device behaves as if it was a port of the corresponding big-switch.

The verification procedure generates SEFL models for both the fat-tree and the big switch from the FIBs given by Batfish. We then run equivalence by injecting a packet with symbolic destination address into equivalent server-facing switch ports in the two topologies; the check takes around 4 minutes.

`netdiff` found that the two models are not equivalent: every edge switch had at least one /32 prefix which wasn’t advertised to the core (due to the configured routing policy), rendering the prefix unreachable from different edges. This contradicts the big switch assumption.

6.6 Scalability

Our experiments so far have highlighted the usefulness of `netdiff`, which works well in practice despite its poor theoretical complexity (§4) given by the exponential nature of symbolic execution. To better understand `netdiff`’s performance, we compare it against Network Optimized Datalog [29] when testing equivalence of two routers with small to medium-sized routing tables.

We generate NOD rules from router FIBs and measure the total equivalence checking time. We run tests involving reachability analysis and tests involving input, port and functional equivalence. Functional equivalence tests with NOD ran out of memory, even for small inputs.

Figure 8 shows the runtime of both tools against the number of entries in the FIB. Note that the NOD line corresponds to input and port equivalence; the netdiff line corresponds to full equivalence checking (Definition 2.1). As expected, the runtime grows exponentially for both NOD and netdiff. For unicast reachability, NOD proves cca. an order of magnitude faster than netdiff, but the difference diminishes for larger inputs; for the largest FIB netdiff is faster than NOD. Overall, these results show that netdiff’s more powerful equivalence comes at modest runtime costs.

Overhead breakdown. We also measured the time taken by each of netdiff’s components: we measured separately the symbolic execution for M_1 and M_2 , their sieving (algorithm 2), and finally the equivalence testing time (Algorithm 3). The results show that M_2 symbex time grows linearly with the number of output path conditions from M_1 . However, M_2 sieving time is constant, due to the fact that the number of feasible outcomes from M_2 is constant - most often equal to one in case when M_1 and M_2 are very similar. Finally, equivalence testing time is negligible.

We also noticed that for broadcast packets, our sieving algorithm is faster since the number of solver queries is linear in the number of output packets in the network. To reduce the sieving time, we disable it for unicast packets.

7 Related work

Our observation that equivalence checking is a simple form of specification is not novel: it has been used previously for program regression verification [44, 36] and to check compiler correctness [23, 12], among other applications.

Note that, in contrast to compiler verification, which attempts to show that compilation preserves semantic equivalence on all possible source programs, netdiff limits its scope to only showing equivalence between concrete dataplane snapshots - i.e. a single source program.

There exist a wide range of specification languages and verification tools for network dataplanes; we discuss here the ones not covered in section 2. Margrave is a tool that checks firewall configurations against user-specified policies in first-order logic [31]. Anteater [30] translates networks and reachability queries to SAT formulae, while NetPlumber [18] takes as input a graph and network boxes modeled as bitwise transfer functions, and uses HSA [19] to check for compliance. Finally, NetCheck [35] takes specifications written in CTL and uses symbolic execution with Symnet to check them. All these tools have merits, yet one of their biggest problems is the difficulty of specifying what the network is meant to do. In many cases, the spec underspecifies

the behavior, meaning that potential problems are missed.

Another line of work focuses on more rigorous specifications which are first proven correct and then translated to dataplane rules. Examples here include Kinetic [21] which takes Finite State Machine descriptions of network functionality, FatTire [37] that takes regular expressions specifying paths to be taken by packets, and Cocoon [38] which enables iterative design and specification for networks. All of these tools offer much stronger correctness properties, but this comes at the expense of usability by non-experts. netdiff is complementary to the above in that it may serve as an extra validation step.

In programming languages, equivalence testing is not a novel concept. DECKARD [14], CCFinder [17] and [25] P-Miner look for syntactically similar pieces of code that are equivalent. EQMINER [15] detects functionally equivalent code via random testing but does not offer guarantees that two programs are equivalent because it does not cover all possible test cases. Another work that aims to achieve the same goal with symbolic execution, targets functional equivalence for simple arithmetic functions, in code that has no branches [13]. Neither tool is exhaustive, so they do not offer correctness guarantees. Our work aims to decide whether two network dataplane models process the packet in the same way, a much stronger definition of equivalence in the limited context of programmable dataplanes.

8 Conclusions

Checking equivalence of programmable dataplanes is a simple way to check program correctness or verify policy. We have presented netdiff, an algorithm that checks two network dataplanes for equivalence using symbolic execution. netdiff will be open-sourced soon.

We have used netdiff to uncover three previously-unknown Openstack Neutron bugs and four configuration errors. netdiff can be used to check P4 programs too: we have found bugs even in simple P4 programs, and have shown how netdiff can be used to help trim large P4 programs while preserving desired functionality. Overall, we find that while equivalence checking is more expensive than individual symbolic execution of the two programs, it scales well enough for most use-cases; compositional equivalence can be used to scale to large Neutron deployments.

In future work, we intend to further explore the applicability of netdiff. One particularly interesting avenue of research is to check the equivalence between SEFL models and the actual dataplane (in C), which requires integrating different symbolic execution engines (Symnet and Klee).

Acknowledgements

This work was funded by CORNET H2020, a research grant of European Research Council (no. 758815).

References

- [1] Symnet Source Code Repository. <https://github.com/nets-cs-pub-ro/Symnet/>.
- [2] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM 2008*.
- [3] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. Netkat: Semantic foundations for networks. In *POPL'14*.
- [4] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A general approach to network configuration verification. In *SIGCOMM* (2017).
- [5] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014).
- [6] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Proc. TACAS'08*.
- [7] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Proc. NSDI'14*, NSDI'14.
- [8] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *NSDI* (2015).
- [9] FOSTER, N., KOZEN, D., MILANO, M., SILVA, A., AND THOMPSON, L. A coalgebraic decision procedure for netkat. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL '15, ACM, pp. 343–355.
- [10] FREIRE, L., NEVES, M., LEAL, L., LEVCHENKO, K., SCHAEFFER-FILHO, A., AND BARCELLOS, M. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2018), SOSR '18, ACM, pp. 4:1–4:7.
- [11] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *SIGCOMM* (2016).
- [12] GUO, S.-Y., AND PALSBERG, J. The essence of compiling with traces. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 563–574.
- [13] HIETALA, K. Detecting Behaviorally Equivalent Functions via Symbolic Execution, 2016.
- [14] JIANG, L., MISHERGHI, G., SU, Z., AND GLONDU, S. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 96–105.
- [15] JIANG, L., AND SU, Z. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (2009), ISSTA '09.
- [16] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)* (Oakland, CA, 2015), USENIX Association, pp. 103–115.
- [17] KAMIYA, T., KUSUMOTO, S., AND INOUE, K. Ccfinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 28, 7 (July 2002).
- [18] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *Proc. NSDI'13*.
- [19] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Proc. NSDI'12*.
- [20] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: Verifying network-wide invariants in real time. In *Proc. NSDI'13*.
- [21] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)* (Oakland, CA, 2015), USENIX Association, pp. 59–72.
- [22] KING, J. C. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394.
- [23] KUNDU, S., TATLOCK, Z., AND LERNER, S. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 327–337.
- [24] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 193–204.
- [25] LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (2004), OSDI'04.
- [26] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. Crystalnet: Faithfully emulating large production networks. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*.
- [27] LIU, J., HALLAHAN, W., SCHLESINGER, C., SHARIF, M., LEE, J., SOULÉ, R., WANG, H., CAŞCAVAL, C., MCKEOWN, N., AND FOSTER, N. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 490–503.
- [28] LLORENTE, J., AND MAEL, K. Neutron firewall optimizations. <https://github.com/jlllorente/neutron-firewall-optimization>.
- [29] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *Proc. NSDI'15*.
- [30] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with anteater. In *Sigcomm* (2011).
- [31] NELSON, T., BARRATT, C., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. The margrave tool for firewall analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration* (Berkeley, CA, USA, 2010), LISA'10, USENIX Association, pp. 1–8.
- [32] NÖTZLI, A., KHAN, J., FINGERHUT, A., BARRETT, C., AND ATHANAS, P. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2018), SOSR '18, ACM, pp. 5:1–5:7.
- [33] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)* (Oakland, CA, 2015), USENIX Association, pp. 117–130.
- [34] PONTARELLI, S., BIFULCO, R., BONOLA, M., CASCONE, C., SPAZIANI, M., BRUSCHI, V., SANVITO, D., SIRACUSANO, G.,

- CAPONE, A., HONDA, M., HUICI, F., , AND BIANCHI, G. Flow-blaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), USENIX Association.
- [35] POPOVICI, M. Verifying large-scale networks using netcheck. In *2017 European Conference on Networks and Communications (EuCNC)* (June 2017), pp. 1–5.
- [36] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 49–64.
- [37] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013), HotSDN ’13.
- [38] RYZHYK, L., BJØRNER, N., CANINI, M., JEANNIN, J.-B., SCHLESINGER, C., TERRY, D. B., AND VARGHESE, G. Correct by construction networks using stepwise refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 683–698.
- [39] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., McKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM ’16, ACM, pp. 15–28.
- [40] STOENESCU, R., DUMITRESCU, D., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM ’18, ACM, pp. 518–532.
- [41] STOENESCU, R., DUMITRESCU, D., AND RAICIU, C. Openstack networking for humans: Symbolic execution to the rescue. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)* (June 2016), pp. 1–6.
- [42] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: scalable symbolic execution for modern networks. In *SIGCOMM* (2016).
- [43] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Tech report: Debugging p4 programs with vera. Tech. rep., June 2018.
- [44] STRICHMAN, O., AND GODLIN, B. *Regression Verification - A Practical Way to Verify Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 496–501.
- [45] THE P4 CONSORTIUM. A p4 implementation of a tor switch. <https://github.com/p4lang/switch/blob/master/p4src/switch.p4>, 2018.
- [46] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM ’17, ACM, pp. 141–154.

A Correctness of netdiff

Note the following useful properties of symbolic execution:

$$\forall(p_i, \pi_i) \in \text{DataplaneSymbex}(M, k, p), S(p_i) \subseteq S(p) \quad (1)$$

$$\begin{aligned} & \forall(p_i, \pi_i), (p_j, \pi_j) \in \text{DataplaneSymbex}(M, k, p) i \neq j, \\ & S(p_i) \cap S(p_j) = \emptyset \end{aligned} \quad (2)$$

$$\bigcup_{(p_i, \pi_i) \in \text{DataplaneSymbex}(M, k, p)} S(p_i) = S(p) \quad (3)$$

The notation (p, π) denotes a pathset, where p is a predicate - which we refer as path condition, describing a subset in the input packet space, and π is a set of paths with the same path condition. $S(p)$ is the set of packets described by predicate p .

Lemma 1 *The set of pathsets computed by algorithm 2 satisfies the symbolic execution properties 1, 2, 3.*

Proof: Property 1 is satisfied by the design of symbolic execution. We need to prove that the set of pathsets built by the algorithm 2 satisfies properties 2 and 3. Line 2 of the algorithm creates the set Q that contains all the pathsets obtained by running symbolic execution with the symbolic packet described by predicate p_0 , not necessarily obeying property 2. To reinforce this property, we build a new set L , that will contain only the pathsets with disjoint path conditions. We will prove the following invariant holds each time the algorithm reaches line 14: the set L satisfies 2 and 3.

We iterate through all the pathsets (q, π) in Q and (l, s) in L , lines 4 and 5, and check for overlapping path conditions. We replace the overlapping pathset in (l, s) in L with two pathsets: one that adds the union of paths s and π with the overlapping path condition $(q \wedge l)$ and one that keeps the pathset (l, s) with the non-overlapping path condition $(l \wedge \neg q)$, lines 7 and 8. Based on set theory, these operations keep the invariant for the local iteration. Line 9 guarantees that the invariant is satisfied for all iterations through L , by updating the pathset (q, π) to $(q \wedge \neg l, \pi)$. If after iterating through all pathsets in L the path condition q is non empty then we add it to L , line 13, reinforcing property 3. \square

Theorem A.1 *EQUIVALENCE(M_1, M_2, i_1, i_2, p_0) is true iff M_1, M_2 are equivalent w.r.t. $Q = \{x \in \text{Packet} | p_0(x) = \text{true}\}$ and input ports i_1, i_2 .*

Proof: We show that when $\text{EQUIVALENCE}(M_1, M_2, i_1, i_2, p_0)$ is true, there is a bijection χ mapping each successful located packet produced by M_1 to one produced by M_2 , c.f. Def. 2.2.

In order to determine if such a bijection can be built, we symbolically execute the program M_1 for the input port i_1 and a symbolic packet specified by predicate p_0 . p_0 describes the set of packets Q for which we decide the equivalence of programs c.f. Def. 2.2. The result is a set of pathsets (q_{i1}, π_{i1}) . According to the symbolic execution properties listed above, each path condition q_{i1} implies the initial path condition p_0 (Property 1), the sets of packets described by the path conditions are disjoint (Property 2) and their union over all path conditions is the set of packets specified by p_0 (Property 3). For each pair (q_{i1}, π_{i1}) of M_1 we symbolically execute program M_2 with path condition q_{i1} and decide on the equivalence of paths.

For a path condition M_1 there might be several pathsets (q_{j2}, π_{j2}) of M_2 . The main point is that the symbolic execution property 3 holds, therefore the set of packets described by condition q_{i1} is the union of the sets of packets described by conditions q_{ij2} over all j . Consequently, equivalence must hold between (q_{i1}, π_{i1}) and each pathset (q_{ij2}, π_{ij2}) over all j , under the constraints imposed by \mathcal{R} and ω . To decide their equivalence we must look into the pathset definition.

The above essentially means that $\forall p \in S(p_0). \exists! q_{ij2} \text{ s.t. } p \in S(q_{ij2})$ (because of property 2). Thus, if the bijection condition holds true for all q_{ij2} , then it holds for their union. But due to property 3, $\bigcup_{(i,j)} S(q_{ij2}) = S(p_0)$, which implies that the bijection can be found on all input packets.

A pathset is a set of input packets and located output packets. Two pathsets are equivalent if (i) they have the same cardinality, (ii) the ports are in correspondence c.f. relation \mathcal{R} and (iii) the packet headers on the corresponding ports satisfy the relation ω c.f. Def. 2.2 in the context of the current path condition. Our approach consists in reducing the equivalence decision problem to that of the maximum bipartite matching (MBM). The conditions of the MBM define a bijection between workers and jobs, which maps to a bijection χ between programs' outcomes in our case. We need to prove that our equivalence algorithm implements the conditions of the MBM problem, therefore deciding on the existence of the bijection.

The first condition is satisfied since the ports are different being defined in the namespace of each program. The second condition reinforces the equality of the cardinalities of the sets of workers and jobs, O_1 and O_2 in our case. We verify it in the line 3 of the algorithm 3. The next condition imposes that worker i is qualified for job j , meaning that we can create the edge representing the equivalence between outcomes described as pairs of (port, packet). The association between ports is checked in line 5 of algorithm 4. Line 7 insures that the relation between packets in the context of the current path condition hold. The existence of the bijective association between programs' outcomes is checked by the algorithm *MaxBipartiteMatching*. The outcome of the MBM is true iff the two path are equivalent. \square

B Notes on equivalence

An equivalence relation is a binary relation satisfying the reflexivity, symmetry and transitivity conditions. The core of our definition of equivalence 2.2 is the bijection between sets of packet and output port pairs, meaning that equivalence conditions are satisfied. The algorithm *netdiff* determines if a bijection can be computed, therefore verifying the equivalence. It is worth mentioning that two dataplane programs written in SEFL have their own name spaces therefore the algorithm *netdiff* can be applied to check equivalence be-

tween a program and itself.

C Notes on complexity of *netdiff*

The complexity of *netdiff* depends strongly on the complexity and number of pathsets output by *DataplaneSymbex*. First of all, we take into account the time of the symbolic execution of the first program (line 2 in Alg. 1). Assume that the number of pathsets produced as a result is n . Similarly, the number of pathsets produced by executing line 5 is m . Therefore, the complexity is $C(\text{DataplaneSymbex}_1) + n \cdot (C(\text{DataplaneSymbex}_2) + m \cdot C(EQP))$

Now, we turn our attention to computing the complexity of *EQP*. The dominating operation is represented by computing the adjacency matrix of the correspondence graph. This involves at most p^2 calls to the SMT solver, where p is the maximum number of paths in each pathset.

The total complexity of *EQP* is: $C(\text{DataplaneSymbex}_1) + n \cdot (C(\text{DataplaneSymbex}_2) + m \cdot p^2 \cdot C(\text{SMT}))$

Since path conditions are usually simple and path length through the network is not large, we assume the complexity of invoking the SMT solver for one path condition to be constant in the size of the network.

Notice that even though the networks under equivalence test may be of similar size, the complexity of the first symbolic execution is considered significantly larger, especially when networks exhibit a high degree of similarity. This is because the path conditions coming out of the first symbolic execution reduce state explosion in the second. Even though in theory both n and m are exponential in the size of the network analyzed, we claim that in practice $m \ll n$. Therefore, we need to stress that the complexity of *EQP* is strongly dominated by the complexity of the symbolic execution of M_1 and the number of outcomes thereof.

Packet cloning is not widely used in the network dataplanes we have examined, which means that p is one or a small integer. Even when cloning is used, for instance in L2 processing, the specifics of the network forwarding fabric constrain p to be smaller than the number of terminals connected to the L2 segment - which is evidently much smaller than the size of the entire network.

Notes

¹After taking pains to actively test it.

²it requires enumerating all possible values in the field range

³a join between all possible input packets and output packets is used to model a router

⁴<https://bugs.launchpad.net/neutron/+bug/1626010>

⁵<https://bugs.launchpad.net/neutron/+bug/1697593>

⁶<https://bugs.launchpad.net/neutron/+bug/1708358>

⁷<https://bugs.launchpad.net/neutron/+bug/1708092>

⁸<https://bugs.launchpad.net/neutron/+bug/1715789>

⁹<https://bugs.launchpad.net/neutron/+bug/1736739>

¹⁰which aggregate traffic from all machines on a compute node

Alembic: Automated Model Inference for Stateful Network Functions*

Soo-Jin Moon¹, Jeffrey Helt², Yifei Yuan³, Yves Bieri⁴, Sujata Banerjee⁵

Vyas Sekar¹, Wenfei Wu⁶, Mihalis Yannakakis⁷, Ying Zhang⁸

¹*Carnegie Mellon University*, ²*Princeton University*, ³*Intentionet*, ⁴*ETH Zurich*

⁵*VMware Research*, ⁶*Tsinghua University*, ⁷*Columbia University*, ⁸*Facebook, Inc.*

Abstract

Network operators today deploy a wide range of complex, stateful network functions (NFs). Typically, they only have access to the NFs’ binary executables, configuration interfaces, and manuals from vendors. To ensure correct behavior of NFs, operators use network testing and verification tools, which often rely on models of the deployed NFs. The effectiveness of these tools depends on the fidelity of such models. Today, models are handwritten, which can be error prone, tedious, and does not account for implementation-specific artifacts. To address this gap, our goal is to automatically infer behavioral models of stateful NFs for a given configuration. The problem is challenging because NF configurations can contain diverse rule types and the space of dynamic and stateful NF behaviors is large. In this work, we present *Alembic*, which synthesizes NF models viewed as an ensemble of finite-state machines (FSMs). *Alembic* consists of an offline stage that learns symbolic FSM representations for each NF rule type and an online stage that generates a concrete behavioral model for a given configuration using these symbolic FSMs. We demonstrate that *Alembic* is accurate, scalable, and sheds light on subtle differences across NF implementations.

1 Introduction

Modern production networks include a large number of proprietary network functions (NFs), such as firewalls (FWs), load balancers (LBs), and intrusion detection systems (IDSs) [21]. To help debug network problems, ensure correct behavior, and verify security, there are many efforts in network testing and verification [22, 35, 40, 41] as well as “on-boarding” new virtual NFs [32].

Such network management tools rely on *NF models* to create test cases, generate verification proofs, and run compatibility tests. These models are required because NF implemen-

tations are often proprietary, leaving operators with only configuration interfaces and vendor manuals. Today, NF models are handcrafted based on manual investigation [22, 40], which is tedious, time-consuming, and error-prone. Further, models do not capture subtle implementation differences across vendors [22, 30, 35]. Using low-fidelity models can affect the correctness and effectiveness of these management tools (§2).

Ideally, we want to automatically synthesize high-fidelity NF models. Synthesizing such models is challenging because: (1) NFs have large state spaces; (2) their state may be mutated by any incoming packet; and (3) in response, the NF may react with any number of diverse and possibly even nondeterministic actions. In this paper, we present *Alembic*, a system that addresses a scoped portion of this open challenge. Specifically, we focus on modeling NFs where their internal states are mutated by incoming TCP packets and their actions are restricted to dropping and forwarding packets, possibly with header modification. Our goal is to synthesize high-fidelity NF models given only the binary executable, vendor manuals, and a specific configuration with which the NF is to be deployed. We adopt this pragmatic approach as vendors may not be willing to share their source code, even with customers. Even this scoped problem presents significant challenges:

- *C1) Modeling and representing stateful NF behaviors:* The behavior of an NF often depends on the history of observed traffic, making it difficult to discover and concisely represent its internal states.
- *C2) Large configuration space:* Concrete configurations (e.g., a FW rule-set) are composed of multiple rules. Fields within a rule (e.g., source IP) can take large sets of values or ranges of values (e.g., IP prefix), making it impractical to infer models for all possible configurations.
- *C3) Large traffic space:* Given the stateful behavior, the input space potentially includes all possible sequences of TCP packets. Naively enumerating this large space would be prohibitively expensive.
- *C4) NF actions:* NFs such as NATs can modify packet headers, making model inference more difficult.

*Contributions by Soo-Jin Moon were made in-part during a former internship at Hewlett Packard Labs. Other contributors from former employees at Hewlett Packard Labs include Sujata Banerjee, Ying Zhang and Wenfei Wu.

To tackle these challenges, we leverage the following key insights (§3):

A) Compositional model: Rather than exhaustively modeling an NF under all possible configurations, we consider the NF’s behavior as the logical composition of its behavior for individual rules in a configuration.

B) Learning symbolic model: Configurations consist of different rule types, such as a firewall drop rule, where each type is associated with a different runtime behavior of the NF. For a given type, the logical behavior of the NF is the same across different values of the rule’s parameters. Hence, we can learn a *symbolic* model for each rule type rather than exhaustively infer a new model for each possible value.

C) Ensemble representation: Even with the above insights, each rule has a large search space as each rule parameter can take a range of values (e.g., a range of ports). Fortunately, we observe that NF behavior is logically independent for subsets of these ranges. For instance, assume a FW contains one rule and we know it keeps per-connection state. We can then model this rule using an *ensemble of independent models* by cloning the model learned using a single connection. However, we must then consider how to infer the specific granularity of state tracked by the NF (e.g., per-connection or per-source). We show in §5 how we can automatically infer this granularity and prove the correctness of this approach in §C.

D) Finite-state machine (FSM) learning: FSMs are a natural abstraction to represent stateful NFs [22, 35], and using them allows us to potentially leverage classical algorithms for FSM inference (e.g., L* [12]). But there are practical challenges in directly applying L* here: First, we need to create suitable mappings between logical inputs (i.e., an input alphabet) that L* uses and the real network packets/configurations that NFs take as inputs (§4). Second, header modifications by NFs make it incompatible with L*, so we need domain-specific ideas to handle such cases (§6).

Having described the high-level insights, we discuss how they specifically address the challenges: Compositional modeling (Insight A) addresses the large configuration space (C2). Both symbolic and ensemble representations (Insights B and C) address the large traffic space (C3) by learning a symbolic model for each rule type and then appropriately cloning it to create an ensemble representation (say for large IP/port ranges). Lastly, extending L* (Insight D) enables us to represent stateful NF behavior (C1 and C4).

Building on these insights, we design and implement *Alembic*.¹ In the *offline stage*, we infer symbolic FSMs for different rule types as defined by an NF’s manual. To concisely represent the internal states of an NF, we extend the L* algorithm [12]. We also leverage our L*-based workflow to infer the state granularity tracked by the NF (e.g., per-connection). Since model synthesis need only be done once per NF, we can

¹ Alembic is a reference to the tool used in the alchemical process of distillation or extraction, as our system extracts models from NFs.

Intended Policy: Only allow TCP traffic from external hosts on already ESTABLISHED connections

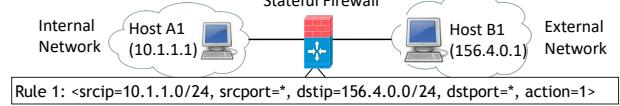


Figure 1: Network set-up

afford several tens of hours for this stage. Given a concrete configuration (i.e., a set of rules), the *online stage* uses these symbolic models to construct a concrete model within a few seconds. Specifically, the online stage maps each rule in a configuration to a corresponding symbolic FSM which, coupled with the inferred granularity, is used to create an ensemble of FSMs. The ensemble is logically composed together for each rule to construct the final concrete model for the given configuration. The resulting concrete model can then be used as an input to network testing and verification tools.

We evaluate *Alembic* with a combination of synthetic, open-source, and proprietary NFs: PfSense [5], Untangle [7], ProprietaryNF, Click-based NFs [31], and HAProxy [2]. We show that *Alembic* generates a concrete model for a new configuration in less than 5 seconds, excluding the offline stage. *Alembic* finds implementation-specific behaviors of NFs that would not be easily discovered otherwise, including some that depart significantly from typical high-level handwritten models (§8.4). For instance, we discover: (1) in contrast to a common view of a three-way TCP handshake, for some NFs, the SYN packet from an internal host is sufficient for an external host to send any TCP packets; and (2) the FIN-ACK packet does not cause internal NF state transitions leading to the changes in the NF’s behavior. Finally, we show that using *Alembic*-generated models can improve the accuracy of network testing and verification tools (§8.5).

2 Motivation

In this section, we highlight some examples of how inaccuracies in handwritten NF models may affect the correctness of network verification and testing tools. Figure 1 shows an example network, where the operator uses a stateful FW to ensure that external hosts (e.g., B1) cannot initiate TCP traffic to internal hosts (e.g., A1). This intent translates to three concrete policies:

- **Policy 1:** To prevent unwanted traffic from entering the network, A1 must establish a connection with B1 before the FW forwards B1’s TCP packets to A1.
- **Policy 2:** When A1 sends a RST or RA (RST-ACK) packet to terminate the connection, the FW should drop all subsequent packets from B1.
- **Policy 3:** To protect against an attacker sending out-of-window packets to de-synchronize the FW state [44], the FW should drop or send a RST when it receives packets with out-of-window sequence (seq) or acknowledgment (ack) numbers.

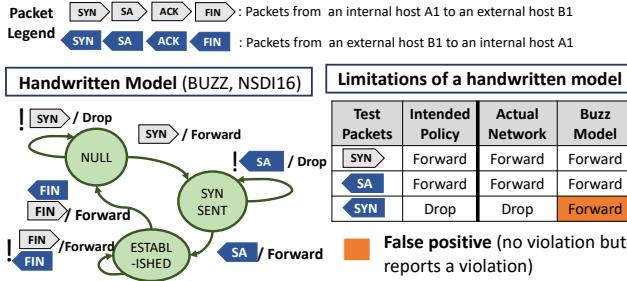


Figure 2: A handwritten model of a stateful firewall (FW) which incorrectly reports a policy violation

To implement these policies, the FW is configured with the rule shown in Figure 1. Since many FWs implement a default-drop policy, there is no explicit drop rule for packets originating externally. Note we do not need explicit rules for Policy 2 and 3 as they should be performed by the FW when following the TCP protocol.

To check if the network correctly implements the intended policies, operators use testing and verification tools [22, 35, 40]. These tools use *NF models* to generate test traffic [22, 41] or to verify intended properties [35]. If these models are inaccurate, the results can have any of the following error types: (1) *false positives*, where the tool reports violations when there is no violation; (2) *false negatives*, where the tool fails to discover violations; or (3) *inability to test or verify* where the tool fails completely because the models are not expressive enough. As an example, consider BUZZ [22], a recently-developed network testing tool. BUZZ uses a model-based testing approach to generate test traffic for checking if the network implements a policy, and the original paper includes several handwritten models. In the remainder of this section, we present three examples of how operators can encounter issues while using the BUZZ tool due to *discrepancies* between handwritten models and NF implementations. Our goal is not to pinpoint limitations of the BUZZ tool but to highlight shortcomings of handwritten models. We find that models from other tools lead to similar problems [35, 40].

To control for NF-specific artifacts (for now), we use two custom, Click-based [31] FWs that correctly implement the above policies.² Figure 2 shows the handwritten model of a stateful FW used in the BUZZ tool [22]. We use the BUZZ FW model for comparison as it implements a policy similar to our example (i.e., the FW only forwards packets belonging to a TCP connection initiated by an internal host).

Test case (policy 1): The operator uses the BUZZ tool to generate test traffic and check if TCP packets from B1 can reach A1. Figure 2 shows a sample test traffic sequence generated by BUZZ: $\text{SYN}_{\text{A1} \rightarrow \text{B1}}^{\text{Internal}}$ (i.e., TCP SYN packet from A1 to B1), $\text{SYN-ACK}_{\text{B1} \rightarrow \text{A1}}^{\text{External}}$, and finally $\text{SYN}_{\text{B1} \rightarrow \text{A1}}^{\text{External}}$. Our Click-based

²Because BUZZ's included FW model does not encode the notion of out-of-window packets, we wrote a FW that adheres to policies 1 and 2 for a fair comparison, and a separate FW for policy 3.

FW drops the last SYN from B1, which matches the policy intent as the TCP handshake did not complete. However, according to the handwritten model, $\text{SYN}_{\text{B1} \rightarrow \text{A1}}^{\text{External}}$ is marked as forwarded. Specifically, the model updates the state to ESTABLISHED on receiving a SYN-ACK (SA in Figure 2) from B1, allowing $\text{SYN}_{\text{B1} \rightarrow \text{A1}}^{\text{External}}$ to be forwarded to A1. This discrepancy between the model and the Click-based FW will be flagged as a policy violation, resulting in a *false positive*.

Test case (policy 2): The operator wants to test if a RST from A1 actually resets the connection state of the FW. However, as we see in Figure 2, the handwritten model only checks for FIN packets but not RST packets to reset the connection state. Hence, the test cases generated by the handwritten model will have discrepancies with the Click-based FW, resulting in a *false positive* (similar to policy 1).

Test case (policy 3): The operator wants to test whether the FW correctly handles packets with out-of-window seq and ack numbers. We observe that many FW vendors enable this feature by default (e.g., §8.4). Unfortunately, the handwritten model is not expressive enough to encode the notion of packets with correct and incorrect seq and ack numbers.

To make matters worse, existing tools (e.g., [22, 35, 40]) assume homogeneous models across vendor implementations for a given NF type. However, we found non-trivial differences in implementations (§8.4). Further, NF models fed to testing and verification tools need to be aware of the *impact of specific configurations*, which can easily be missed by handwritten models. For instance, the BUZZ FW model assumes a *default drop* policy from the external interface, which is consistent with many vendors. However, while running model inference using *Alembic*, we found that one specific NF (Untangle FW) *allows* packets by default [7]. To implement a default-drop policy in Untangle, we need an explicit drop-all rule, and a model for Untangle needs to be customized for this configuration.

3 Alembic System Overview

In this section, we state our goals, identify the key challenges, describe our insights to address these challenges, and provide an end-to-end overview of *Alembic*.

Preliminaries: We introduce the terminology related to NF configurations, which describe an NF's runtime behavior. A configuration schema contains NF rule types. Each rule type has various configuration fields, and the data types these fields accept (e.g., “srcip” takes an IPv4 range). Once we specify the concrete values for the fields (concrete values can be wild-card), we obtain a concrete rule of the rule type. A concrete configuration consists of multiple concrete rules. Figure 3 shows an example of a firewall (FW) and a network address translation (NAT) configuration schema and their corresponding concrete configurations. In the NAT Rule type, the *outsrcip* field denotes the possible output IP values used in address translation.

ProprietaryNF FW
ConfigSchema:
<i>Rule type 1 (Accept):</i> <srcip:IPv4 range, srcport:Port range, dstip:IPv4 range, dstport:Port range, action:1>
<i>Rule type 2 (Deny):</i> < srcip:IPv4 range, srcport:Port range, dstip:IPv4 range, dstport:Port range, action:0 >
ConcreteConfig:
Rule 1: < srcip:10.1.1.1,srcport:*,dstip:156.4.0.1,dstport:*, action:1 >
Rule 2: < srcip:10.8.0.0/16,srcport:*,dstip:151.0.0.0/8,dstport:*,action:0 >
PfSense outbound NAT
ConfigSchema:
<i>Rule type 1:</i> <srcip: IPv4 range, srcport: Port range, dstip: IPv4 range, dstport: Port range, outsrcip: IPv4 range, outsrcport: Port range>
ConcreteConfig:
Rule 1: <srcip:10.1.0.0/16,srcport:*,dstip:156.4.0.0/16,dstport:*,outsrcip:126.2.0.0/16,outsrcport:*=*>
Rule 2: <srcip:10.0.0.0/8,srcport:*,dstip:162.4.0.0/16,dstport:*,outsrcip:192.1.0.0/16,outsrcport:*=*>

Figure 3: Example of a simplified ConfigSchema and ConcreteConfig for a FW and a NAT

3.1 Problem formulation

Given an NF with a concrete configuration, *Alembic*'s goal is to automatically synthesize a *high-fidelity* behavioral model of the NF. Since NF implementations do not change often, we can afford several tens of hours of offline profiling per NF. However, since concrete configurations (e.g., a FW rule-set) can change often, we need to generate a new model given a new configuration quickly, within a few seconds.

Alembic takes five inputs: (1) the NF executable binary, (2) the *configuration schema* (ConfigSchema), (3) the high-level rule *processing semantics* of parsing the configuration (e.g., first match), (4) a list of network interfaces, and (5) the set of input packet types (e.g., TCP SYN or ACK) the model needs to cover. For (1), we assume no visibility into the internal implementation or source code and only have access to its manual describing configuration. For (2), the ConfigSchema is typically already available from vendor documentation.³ The ConfigSchema in Figure 3 assumes we are explicitly given a set of rule types (e.g., accept or deny), where each rule type is associated with a different runtime behavior. In practice, the vendor documentation may only specify a set of fields and their types. For instance, a FW ConfigSchema provides one rule type with an action field that takes a binary value, in which each value leads to a rule type with different runtime behaviors. We show how we generate a set of all rule types in such a case (§7). For (3), we assume the rule processing semantics are available from the vendor documentation. Our design can handle any NF that applies a single rule per packet. Our implementation currently supports first-match semantics but can be easily extended to handle others (e.g., last-match). For (4), we need to know a list of interfaces that the NF is configured with. In this work, we assume that we are given two interfaces (e.g., internal and external-facing interfaces).

³*Alembic* requires a one-time, manual effort to translate this documentation into a format compatible with our current workflow.

Lastly, given packet types (5), *Alembic* will automatically configure each packet type with appropriate field values.

Here, we focus on modeling TCP-relevant behavior for NFs that forward, drop, or modify headers (e.g., FWs, NATs, and LBs). We provide default packet types for TCP, but *Alembic* can be extended with additional packet types. We scope the types of NFs and their actions that *Alembic* can handle in §3.3 and discuss how to extend *Alembic* to handle more complex NFs in §10.

3.2 Key ideas

To highlight our main insights to address challenges *C1* through *C4* from §1, suppose we want to model an NF with a concrete configuration *C1* composed of *N* concrete rules $\{R_1 \dots R_N\}$. Figure 4 illustrates our ideas to make this modeling problem tractable.

A) Compositional model (Fig. 4a): The concrete configuration *C1* can be logically *decomposed* into individual rules. As seen in Figure 4a, suppose we have models M_1 for R_1 and M_2 for R_2 . Then, we can create a *compositional model* for the NF given the processing semantics defined by the ConfigSchema (e.g., first-match). If the packet matches $Rule_1$, then apply $Model_1$, else if it matches $Rule_2$, then apply $Model_2$. Otherwise, apply $Model_{default}$.

B) Symbolic model (Fig. 4b): To start, we make two simplifying assumptions, which we relax below: (1) the IP and port fields in a concrete rule take a single value from a range (e.g., 10.1.1.1 for srcip); and (2) the NF keeps per-connection state. Suppose the srcip field in R_1 (Figure 4b) takes a single IP from 10.1.0.0/16. It is infeasible to exhaustively infer the model for all possible values. Fortunately, we observe that the logical behavior of the NF for a particular rule type (e.g., FW accept rule) is *homogeneous* across different values for the IPs and ports in this range. Thus, we can efficiently generate a model by representing each IP and port field in a rule with a symbolic value. Hence, for each logical rule type (e.g., FW accept rule), we can learn a symbolic model (e.g., $M_1(A)$).

C) Ensemble representation (Fig. 4c): We relax the assumption that IPs and ports take single values and discuss how we handle ranges within a rule (i.e., R_1 in Figure 4b takes a /16 prefix for a srcip). We observe that NF behavior is *logically independent* for subsets of this large traffic space. Consider a stateful firewall that keeps per-connection state. Rather than viewing M_1 as a monolithic model that captures the behavior of all relevant connections, we can view the model as a *collection of independent models*, one per connection (i.e., $M_{1,1}$ for connection 1, $M_{1,2}$ for connection 2, etc.). Combining this idea with B above, we learn a *symbolic model* for each rule type and *logically clone the model* to represent IP and port ranges (henceforth, an ensemble of models). However, to leverage this idea, we need to infer the granularity at which an NF keeps independent states (e.g., per-connection or per-source). We show in §5 how to automatically infer this.

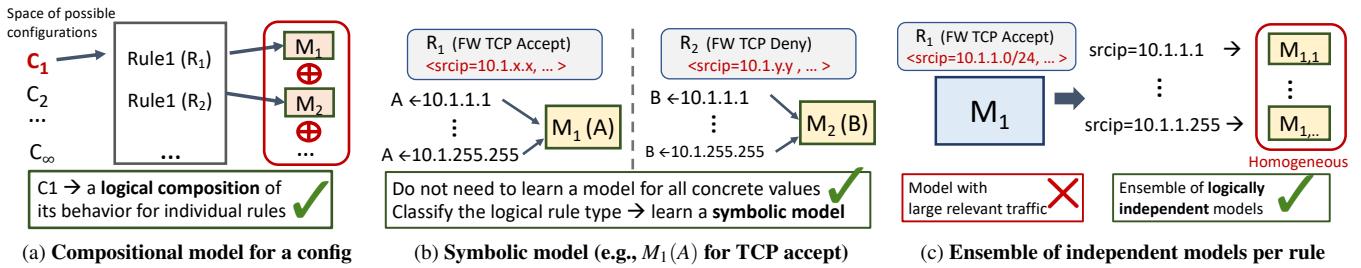


Figure 4: *Alembic* Key Insights

Algorithm 1 NF operational model for processing incoming packets

```

1: function NF(locatedPkt p, Config c, ProcessingSemantic ps,
   Map[rule, Map[key, state]] stateMap)
2:   poutList = []
3:   rule = FINDRULETOAPPLY(p, c, ps)
4:   if rule is None then
5:     rule = GETDEFAULTRULE()
6:   keyType = GETKEYTYPE(rule)
7:   key = EXTRACTHEADER(keyType, p)
8:   FSM = GETMODEL(key, rule)
9:   curState = GETSTATE(stateMap, rule, key)
10:  poutList, nextState = TRANSITION(FSM, p, curState)
11:  UPDATERE STATE(stateMap, rule, key, nextState)
12:  return poutList

```

D) FSM inference: The remaining question is how to represent and infer a symbolic model. Following prior work in stateful network analysis, we adopt the FSM as a natural abstraction [22, 35]. To this end, we develop a workflow that leverages L* for FSM inference [12]. At a high-level, given a set of relevant inputs, L* adaptively constructs sequences, probes the blackbox, and infers the FSM. However, directly applying L* for an NF entails significant challenges: First, L* requires the set of inputs a priori. Hence, we need to generate inputs from a large input space, and create suitable mappings between inputs that L* takes and real packets for the NF. Second, L* is not suitable for learning a FSM for a header-modifying NF because it assumes: (1) we know the input alphabet a priori, and (2) the underlying system is deterministic. As an example violation of (2), a NAT may nondeterministically choose the outgoing ports. We leverage a domain-specific idea to extend L* for such cases (§6).

3.3 Operational model and limitations

Having described our key insights, we scope the types of NFs for which *Alembic* is applicable. We use an abstract NF (Algo. 1) to describe how incoming packets are processed (a more detailed description can be found in §B). Our goal is to handle NFs with logic stated in Algo. 1.

NF operational model: We start by describing the inputs and outputs of the abstract NF. The NF receives or transmits a

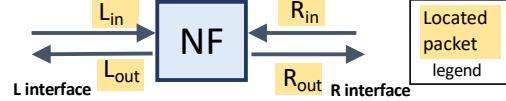


Figure 5: An NF with located packets

located packet [34] (i.e., a packet associated with an interface). Figure 5 shows a setup for an NF with 4 located packets. The NF is configured with two interfaces, L (e.g., internal) and R (e.g., external). As an example, L_{in} is a packet entering the NF via L, and L_{out} is an outgoing packet from the NF via L.

The abstract NF is configured with a *concrete configuration*, composed of a set of *rules*. Each rule maintains a mapping between keys and concrete FSMs. For instance, if the NF uses a per-connection key, then it will keep a concrete FSM for each unique 5-tuple. The concrete FSMs describe the appropriate action (i.e., L_{out} or R_{out}) for an incoming located packet (i.e., L_{in} or R_{in}). As shown in Algo. 1, when a located packet arrives, the NF searches the configuration for the correct rule to apply based on the processing semantics. If no rule is found, the NF uses the default (i.e., empty) rule. Then, it uses the relevant packet headers determined by the rule's key to find the concrete FSM and current state associated with that key. Finally, the NF processes the packet according to the FSM and updates the current state (Lines 10 and 11). *Alembic* aims to synthesize models for NFs following Algo. 1.

Assumptions on configurations: We make the following assumptions about NF configurations:

- Rules in a concrete configuration are independent. For instance, we do not consider NFs that share the same state across different rules. At most one rule in a configuration can be applied to an incoming packet.
- Within a concrete rule, the states across different keys (i.e., state granularity tracked by an NF) are *independent*. For a per-connection FW with a rule that takes IP and port ranges, states across connections are independent.
- When IPs and ports in a concrete rule take ranges (e.g., ports=80), NFs treat each value in the range *homogeneously* such that we can pick a representative sample and learn a symbolic model (i.e., the symbolic model obtained using port 80 or port 5000 for an outsrcip is identical).

Assumptions on NF actions: We now scope the NF actions that *Alembic* can handle:

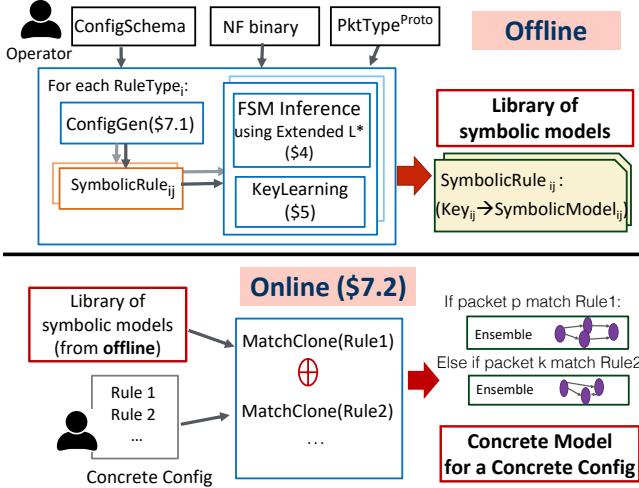


Figure 6: *Alembic* Workflow

- For simplicity, we only consider single-function NFs, excluding cases such as combined NFs processing FW rules and then NAT rules.
- To make learning tractable, we only look at IP and port modifications. Our implementation does not consider seq/ack numbers, ToS, or other fields (§4.3). We only handle header modifications for connection-oriented NFs (§6).⁴ We tackle header modification for an NF that initially modifies IP/port of a packet, p1, entering from a particular interface *before* modifying a packet, p2 (that belongs to the same connection as p1) entering from the other interface. Lastly, we cannot infer context-sensitive relations such as how the modified IP or port (e.g., NAT ports) is chosen.
- We do not explicitly model temporal effects, such as connection timeouts. When we inject input packets into the NF, we collect outputs for Δ_{wait} (e.g., 100 ms) before injecting the next input packet. *Alembic* cannot handle cases where output packets are results of prior input packets (e.g., retries after 1 second).
- We support five types of state granularity: per-connection, per-source (e.g., a scan detector which counts a number of SYN packets), per-destination (e.g., DDoS detector), cross-connection, and stateless.

3.4 Alembic workflow

Having described our key insights and scope, we now present our workflow (Figure 6) consisting of two stages:

Offline stage: From the ConfigSchema, we generate a set of rule types (§7). Given each rule type, the **ConfigGen** module generates a SymbolicRule, R^{symb} , and a corresponding ConcreteRule. For instance, given a FW ConfigSchema, it generates two SymbolicRules and ConcreteRules (e.g., FW accept and deny rule as shown in Figure 7).

⁴Most header modifying NFs we are aware of are connection-oriented.

$R_1^{\text{symb}} : \langle \text{src:A}, \text{srcport:Ap1}, \text{dst:B}, \text{dstport:Bp1}, \text{action:1} \rangle \text{ FW TCP Accept}$ $R_1^{\text{conc}} : \langle \text{src:10.1.1.1}, \text{srcport:2000}, \text{dst:156.4.0.1}, \text{dstport:5000}, \text{action:1} \rangle$
$R_2^{\text{symb}} : \langle \text{src:A}, \text{srcport:Ap1}, \text{dst:B}, \text{dstport:Bp1}, \text{action:0} \rangle \text{ FW TCP Deny}$ $R_2^{\text{conc}} : \langle \text{src:10.1.1.1}, \text{srcport:2000}, \text{dst:156.4.0.1}, \text{dstport:5000}, \text{action:0} \rangle$

Figure 7: SymbolicRules and ConcreteRules for a FW

For each SymbolicRule, we use the **FSMInference** module, which leverages L*-based workflow to infer a symbolic model where IPs and ports are symbolic (§4) and handles header modifications (§6). This module uses our version of L* (i.e., **Extended L***). We also design the **KeyLearning** module, which leverages the **FSMInference** module and infers the state granularity (i.e., key type) tracked by the NF (e.g., per-connection). Using the key type, we can identify the **key**, a set of header field values that identifies logically independent states (e.g., a 5-tuple for per-connection NF). The offline stage produces a set of symbolic models, mapping each SymbolicRule to a symbolic model and its key type.

Online stage: Given a new configuration, each rule is matched to a corresponding SymbolicRule, mapped to a key type and a symbolic model. Based on the key type, we logically clone the symbolic model to represent concrete IP and port ranges (collectively, an ensemble of FSMS). Given the processing semantics, we logically compose each ensemble to create the final model for this configuration. Network management tools can then use the resulting model.

Roadmap: In the interest of clarity, §4 describes the **FSMInference** module of *Alembic* for a given SymbolicRule with the following simplifying assumptions: NFs keep per-connection state and do not modify headers. In subsequent sections, we relax these assumptions and show how we infer the state granularity (§5) and handle header-modifying NFs (§6). §7 discusses how we generate a set of rule types and the corresponding SymbolicRule and the *Alembic* online stage.

4 Extended L* for FSM Inference

We now present the **FSMInference** module, which leverages the Extended L* for inferring a symbolic model given a SymbolicRule, R^{symb} (e.g., in Figure 7). Recall that we are also given a corresponding ConcreteRule, R^{conc} , to configure the NF. For clarity, we start with two simplifying assumptions: (1) NFs keep per-connection state, and (2) NFs do not modify packet headers. We relax these assumptions in §5 and §6.

4.1 Background on L* algorithm

Before discussing the challenges of directly applying L*, we provide a high-level description of the L* algorithm [12], which infers a FSM for a given blackbox. Given the input alphabet, Σ (e.g., $\{a, b\}$ where a, b are input symbols), L* generates sequences (e.g., a, aa, aba), and probes the blackbox, resetting the box between sequences. For each input

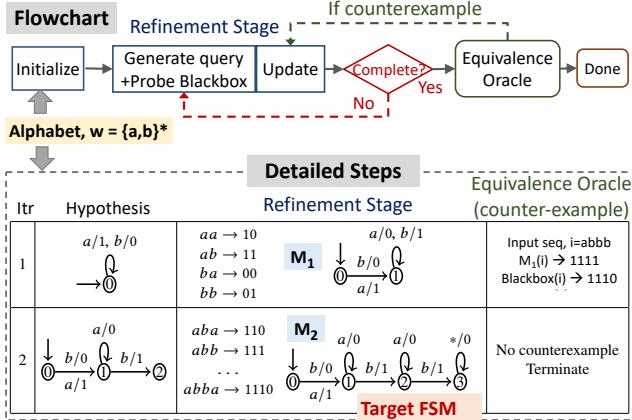


Figure 8: L* overview and example

sequence, L* builds a *hypothesis* FSM consistent with the input-output pairs seen so far. Specifically, it builds a Mealy machine whose outputs are a function of its current state and inputs. As shown in Figure 8, L* iteratively refines the hypothesis FSM until it is complete (i.e., the set of probing sequences cover the state space of this hypothesis). After the hypothesis converges, L* queries an Equivalence Oracle (EO), which checks if the inferred FSM is identical to the blackbox and provides a counterexample if they are not. If the EO reports that the hypothesis is identical to the blackbox, the algorithm terminates. Otherwise, L* uses the counterexample to further refine the hypothesis. The process repeats until the EO reports no counterexamples. L*'s runtime complexity is polynomial in the number of states and transitions of a minimal FSM representing the target FSM as well as the length of the longest counterexample used to refine the hypothesis [12].

Example: Figure 8 illustrates an example of the steps in L* for the target FSM shown with $\Sigma = \{a, b\}$. Initially, L* starts with the inputs, a and b , and a single-state FSM. It generates four sequences to refine the model and converges to M_1 as shown. It then queries the EO and finds a counterexample where $\text{Blackbox}(abbb)=1110$ but $M_1(abbb)=1111$, which is used to update the model. To explore the state space of the new hypothesis, L* generates longer sequences. After this second iteration, the EO finds no counterexamples (as M_2 is identical to the blackbox), and the algorithm terminates.

4.2 Challenges in using L* for NFs

While L* is a natural starting point, there are practical challenges in applying it directly to NFs. We will describe these challenges using Figure 9 and discuss our solutions.

1) Generating input alphabet (§4.3): L* assumes the input alphabet (Σ) is known. As discussed in §3, we can set Σ for *Alembic* to be a set of located symbolic packets, which are packets with symbolic IPs and ports associated to interfaces. From now on, when we say packets, we refer to located packets. The main disconnect here is that the NF (i.e., the blackbox

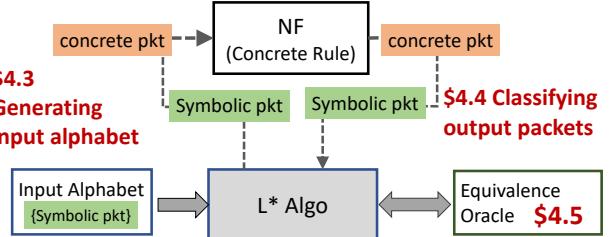


Figure 9: Key challenges in adopting the L* workflow for NF model inference

in the L* workflow) takes in concrete packets and not symbolic packets. Thus, we need to map a symbolic packet to a concrete packet. Two challenges exist here: First, the possible header space for concrete packets is large (i.e., all IPs and ports), and second, the concrete packets need to exercise the internal states of the NF (e.g., trigger the NF behavior).

2) Classifying output packets (§4.4): Next, for each symbolic packet suggested by L*, we need to map it to an NF action. The practical challenge is that NFs may require an unpredictable delay. If we assume a processing delay that is too short and classify the action as a drop, we might learn a spurious model. While a delay that is too long will lead to our inferences taking a long time. Thus, we need a robust way to map an input to the observed output.

3) Building an equivalence oracle (§4.5): L* assumes access to an EO (Figure 9). In cases where we do not have access to the ground truth, we can only approximate the oracle via input-output observations. There are two practical issues. First, existing approaches (e.g., [17, 25]) to building an EO generate a large number of equivalence queries, creating a scalability bottleneck. Second, different approaches for building an EO may affect the soundness of *Alembic* (§4.5).

4.3 Generating input alphabet

We now describe how we generate a set of *located symbolic packets* for the input alphabet and how we map each located symbolic packet to a concrete packet. As discussed in §3, we are given the representative packet types of interest $\text{PktType}^{\text{Proto}}$ (e.g., TCP handshake) as an input.

To illustrate these challenges, consider two straw-man solutions that generate packets for: (1) every possible combinations of header fields, and (2) randomly generated header fields. (1) is prohibitively expensive, and (2) may not exercise the relevant stateful behaviors. Our idea is to use the symbolic and concrete rules to identify relevant header fields and their values. Specifically, we observe that the header fields and their values (e.g., IP-port) in R^{conc} will trigger relevant NF behaviors. Thus, we generate all combinations of these relevant IP-port pairs using their concrete values from R^{conc} . Using a pair of R_1^{symb} and R_1^{conc} as an example (Figure 7), we identify A=10.1.1.1 as a possible candidate for both source and des-

tination IPs across *all interfaces* (i.e., A can be a source or destination IP on packets entering from internal or external interfaces). We consider all interfaces, as a packet entering different interfaces can be treated differently.

We also consider the scenario where the packet does not match any rules. One approach is to pick concrete header values that do not appear in the concrete rule and generate a corresponding symbolic packet (e.g., not A=12.1.0.1). However, this would double the size of Σ . Instead, we leverage our insight regarding the compositional behavior of NFs and view this as composing the action with the *default* behavior of the NF when no concrete rule is installed. We separately infer a model, M_{default} , with an empty configuration (e.g., a FW without any rules).⁵

Example: From R^{symb} , we mark A:Ap1 and B:Bp1 as possible IP:port pairs, where A:Ap1 and B:Bp1 refer to srccip:srcport and dstip:dstport pairs from R^{symb} . Then, we generate all possible combinations across source and destination IP/ports and network interfaces: (1) TCP_{A:Ap1 \rightarrow B:Bp1}^{Internal} (corresponding to a TCP packet with srccip:port=A1:Ap1 and dstip:port=B1:Bp1 on the internal interface), (2) TCP_{A:Ap1 \rightarrow B:Bp1}^{External}, (3) TCP_{B:Bp1 \rightarrow A:Ap1}^{Internal}, ..., etc. Suppose the packet types of interest are: {SYN, SYN-ACK, ACK}. Then, for (1), we obtain SYN_{A:Ap1 \rightarrow B:Bp1}^{Internal}, SYN-ACK_{A:Ap1 \rightarrow B:Bp1}^{Internal}, We follow the similar procedure for (2) and (3). Essentially, SYN_{A:Ap1 \rightarrow B:Bp1}^{Internal} is a symbolic packet which maps to a concrete SYN packet with A=10.1.1.1 and Ap1=2000 that is injected from the internal interface. *Alembic* internally tracks the symbolic-to-concrete map (i.e., A=10.1.1.1) to connect the symbolic packet used by L* to the concrete packets into the NF. Finally, we (optionally) prune out packets that are infeasible given the known reachability properties of the network. For instance, it is infeasible for a packet with srccip=10.1.1.1 to enter from the external interface.

4.4 Classifying output packets

To classify the output from the NF, we monitor for output packets at all interfaces of the NF and map them to their symbolic representations. For instance, after detecting a SYN on the external interface with source IP:port, 10.1.1.1:2000, and destination IP:port, 156.4.0.1:5000, we assign the output symbols as SYN_{A:Ap1 \rightarrow B:Bp1}^{External}. Specifically, *Alembic* monitors all interfaces for Δ_{wait} and reports the set of observed packets (e.g., L_{out} and R_{in}). Δ_{wait} is critical for classifying dropped packets and we cannot have an arbitrarily assigned values. Unfortunately, an NF sometimes introduces unexpectedly long delays in packets ($\geq 200\text{ms}$). For instance, Untangle performs connection setup steps with variable latency upon receiving SYN packets, and ProprietaryNF experiences periodic spikes in CPU usage leading to delayed packets. Such delays can

⁵We acknowledge an assumption that rule matching is correctly implemented by the NF. If the NF has a rule for src=A and dst=B but a buggy implementation that matches A' and B', we will not uncover this behavior.

result in misclassifying a packet as a drop and affect the learning process. For these NFs, Δ_{wait} is determined by injecting the TCP packets and measuring the maximum observed delay. Further, we extended L* with an option to probe the same sequence multiple times and pick the action that occurs in the majority of test sequences.

4.5 Building an equivalence oracle

Building an efficient oracle is difficult with just black-box access [17, 25]. Any EO will be incomplete as it cannot generate all sequences. Our goal is to achieve soundness with respect to the generated Σ without sacrificing scalability.

We tested three standard approaches for generating EOs that LearnLib [38], an open-source tool for FSM learning, supports: (1) *Complete Oracle* (CO), which exhaustively searches sequences to a specified length; (2) *Random Oracle* (RO), which randomly generates sequences; and (3) *Partial W-method* (Wp-method) [25], which takes d as an input parameter which is an upper bound on the number of additional states from its current estimate at each iteration.⁶ We discarded the CO as it simply performs an exhaustive search and the RO as it is not systematic in exploring the state space. Instead, we use the Wp-method, a variant of the W-method [17] that uses fewer test sequences without sacrificing W-method's coverage guarantees. Briefly, the W-method uses a characterization set, the W-set, which is a set of sequences that distinguish every pair of states in the hypothesis FSM. The W-method searches for new states that are within d additional inputs of the current hypothesis and uses the W-set to confirm the new states. In theory, one can set d to be large but increases the runtime by a factor of $|\Sigma|^d$. For this reason, we set $d = 1$ in *Alembic*. *Alembic* can only discover additional NF states that are discoverable by the Wp-method with $d = 1$; i.e., *Alembic* with Wp-method ($d = 1$) is sound. Even with $d = 1$, *Alembic* synthesizes models that are more expressive than many handwritten models and discovers implementation-specific differences (§8).

Distributed learning: Both L* and Wp-method for $d = 1$ are polynomial in runtime. However, the Wp-method is the bottleneck as the number of sequences generated by Wp-method is approximately $|\Sigma|$ factor higher than that of the L*. Fortunately, the equivalence queries can be parallelized. In our system implementation (§8), we run equivalence queries in parallel across multiple workers until we find a counterexample. Using this technique, we can significantly reduce the time for learning a complex behavioral models (§8.3).

5 KeyLearning: Learning State Granularity

Thus far, we assumed that the NF maintains per-connection state. We now relax this assumption and show how we tackle

⁶In practice, the number of states can grow by $> d$ at each iteration.

NFs that maintains other key types (e.g., per-source). Specifically, we implement a KeyLearning module. Given a SymbolicRule, the module outputs the **key type**, a set of header fields that identify a relevant model in an ensemble representation. Note that here we still assume that the NF does not modify packet headers, which we will relax next in §6.

High-level intuition: Consider a FW configured with a rule that keeps per-connection state. A packet from one connection only affects its own FSM and is unaffected by packets that belong to other connections. Now, consider an NF which keeps per-source state, and packets, p1 and p2, with the same srcip, but with different dstip. The arrival of p1 affects not only the state for processing p1, but also the state associated with p2 because they share the same srcip. The KeyLearning algorithm builds on the above intuition; if two connections are independent with respect to an NF’s processing logic, then the packet corresponding to one connection only affects the state of its FSM. Thus, to infer the key type, we construct test cases using *multiple connections* to validate the independence assumptions across these connections. We show how we can validate independence by inspecting two connections using carefully constructed source and destination values.

The KeyLearning algorithm is composed of test cases to distinguish between different key types. As a concrete example of a test case, suppose we have a SymbolicRule, which takes $\langle \text{srcip}=A, \text{dstip}=B \rangle$ where A and B are ranges of IPs (e.g., A=10.1.0.0/16 and B=156.4.0.0/16). First, we infer two models with two separate ConcreteRules, where we configure each IP using a concrete singleton (e.g., R_1^{conc} , with $\langle \text{srcip}=10.1.1.1, \text{dstip}=156.4.0.1 \rangle$ to learn Model_1 , and R_2^{conc} with $\langle \text{srcip}=10.1.1.1, \text{dstip}=156.4.0.2 \rangle$ to learn Model_2). Note that these two have the *same srcip*. We leverage the FSMInference module in §4. We first generate Σ_1 for R_1^{conc} and use the FSMInference in §4 to obtain Model_1 , and then repeat for Model_2 . Assuming these models are independent, we run a logical *FSM composition* operation to construct $\text{Model}_{\text{composite}}$ (Def. 7 in §C). This is what the hypothetical model will be if these two connections are *independent*. As a second step, we now learn a joint model $\text{Model}_{\text{joint}}$, where we combine input alphabets from both connections. Specifically, we configure a *ConcreteRule*, where the dstip takes a range of IPs (e.g., 156.4.0.1-156.4.0.2).

For example, consider a scan detector, that keeps per-source state. As the above two connections have the same srcip, $\text{Model}_{\text{joint}}$ will reflect that the packets affect each other’s state (i.e., $\text{Model}_{\text{joint}}$ is not equivalent to $\text{Model}_{\text{composite}}$, which assumes independence across two connections). But, for a per-connection model, the two connections are independent (i.e., $\text{Model}_{\text{joint}}$ would be equivalent to $\text{Model}_{\text{composite}}$). Thus, we now have a simple logical test to distinguish between per-connection and per-source.

Inference Algorithm: Our inference algorithm generalizes the basic test described above. By crafting different ConcreteRules (i.e., changing the overlap on srcip or dstip) and

Test Cases (diff means different)

	Stateless	Per-conn	Per-src	Per-dst	Cross-conn
Test 1 (diff src, diff dst)	N	N	N	N	Y
Test 2 (same src, diff dst)	N	N	Y	N	Y
Test 3 (diff src, same dst)	N	N	N	Y	Y

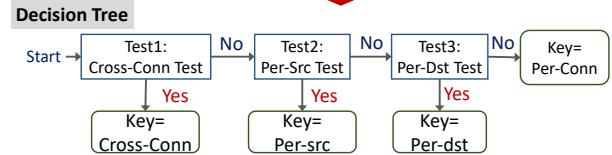


Figure 10: **KeyLearning Decision Tree**

running the equivalence tests between $\text{Model}_{\text{composite}}$ and $\text{Model}_{\text{joint}}$ for each case, we create a decision tree to identify the *key type* maintained by the NF, which are: (1) per-connection, (2) per-source (e.g., a scan detector), (3) per-destination, (4) cross-connection, or (5) stateless.⁷

Figure 10 shows the result of test cases for these key types. For instance, Test 1 configures two connections to have different sources and destinations, to check whether the NF keeps cross-connection state. Test 2 configures two connections to have the same sources, but with different destinations. If Test 2 outputs that two connections affect the states relevant for each other, then the NF is maintaining either a cross-connection or per-source state. The decision tree (Figure 10) uniquely distinguishes the key and the correctness naturally follows from our carefully constructed test cases. We formally prove the correctness of this approach in §C.

6 Handling NF Header Modifications

Now, we extend our FSMInference in §4 to handle header modifications, such as a NAT rewriting a private IP-port pair to a public IP-port pair. We currently only handle NFs that maintain per-connection state while modifying IPs and ports. We consider two cases of possible header modifications: (1) *static* (e.g., a source NAT modifies a private port to a static public port), and (2) *dynamic* (e.g., a source NAT or LB randomly generates port mappings across resets). We first describe how we handle each case individually, then present our combined workflow to handle both cases. Our workflow does not require knowing a priori that an NF modifies header fields, which field it modifies, or how it modifies packet headers (i.e., static or dynamic).

Static header modifications: Consider a source NAT that deterministically maps a source IP-port pair (e.g., A:Ap1) to a public source IP-port pair (e.g., X:Xp1). To discover the NAT’s behavior that rewrites the public IP-port back to the private IP-port, we need to generate a symbolic packet using

⁷The key for a stateless NF is a 5-tuple. We can view a stateless NF as an FSM with a single state, which is identical to each 5-tuple keeping one state.

the public (modified) IP-port (i.e., X:Xp1). However, we may not know the concrete value of X:Xp1 a priori. Hence, we cannot generate a complete set of $|\Sigma|$. Our idea is to first run the inference module (§4) and check whether a symbolic model has additional symbolic IPs and ports. If so, we append the new IP-port pairs to the Σ and re-run the inference. We repeat this step until the output FSM contains no new IP-port pairs. Given that the static modification maps an IP-port to the same IP-port pair, this approach converges.

Dynamic header modification: The above approach of updating the input alphabet will not converge for NFs that dynamically modify packet headers, however. Consider a NAT that *randomly* picks one of the available ports for the same 5-tuple (e.g., a private IP-port (e.g., A:Ap1 first maps to X:Xp1 but then to X:Xp2 after L* resets the NF). Since L* assumes a deterministic FSM, it will crash as a result of this nondeterminism. Our idea is simple. If L* crashes, then we identify the IP-port pair that caused the nondeterministic behavior. Next, we *mask* this nondeterministic behavior of the NF from L* by explicitly mapping such IP-port pairs to consistent symbolic values (e.g., Alembic maps SYN_{A→B}^{Internal} to SYN_{X→B}^{Internal} regardless of the concrete value of the rewritten source IP). Since the concrete value of X will change across resets, the extended L* uses the most-recently observed concrete value of X when playing sequences.

Combining both cases, we first run the FSMInference module (§4). If L* completes but discovers new symbols (i.e., static modification), then we re-run the workflow with new symbols. However, if L* crashes due to a nondeterministic FSM (i.e., dynamic modification), we mask the non-deterministic behavior as discussed. After the required modifications are applied, the L* is repeated until it converges. As we only handle modification for per-connection NF, we assume the key is per-connection for an NF that modifies packet headers.

7 Handling an Arbitrary Config

We now discuss how we generate a set of SymbolicRules (§7.1) and then how the *online* stage constructs a concrete model given a concrete configuration (§7.2).

7.1 Generating SymbolicRules

The ConfigGen module generates a set of SymbolicRules. As discussed in §3.1, the vendor documentation may not clearly give a set of rule types where each type is associated with a different runtime behavior (e.g., FW accept vs. deny). Suppose the FW ConfigSchema specifies a rule types as $\langle \text{srcip}, \text{sreport}, \text{dstip}, \text{dstport}, \text{action} \rangle$ where “action” takes a binary value. To obtain a set of logical rule types, we use a set of conservative heuristics. Typically, we observe that fields which take a large set of values (e.g., IPs and ports) demonstrate similar behaviors across values within the set. For fields that only take a small set of values (e.g., action), each value typically

carries a distinct runtime behavior. Based on this observation, the ConfigGen module first assigns a new symbol (i.e., A for srcip) to each field that takes a large set of values. Then for each combination of other small fields (e.g., action), this module generates a SymbolicRule (for each rule type). We also generate a corresponding *ConcreteRules* by sampling a value for each field. For the example above, ConfigGen generates two rule types, accept and deny.

7.2 Alembic online

We now describe Alembic’s *online* stage, which constructs a concrete model for a given a configuration. The concrete model then uses our operational model (Algo. 1) to model how an NF processes incoming packets.

Constructing a concrete model: For each concrete rule, R , in a concrete configuration, we first fetch the corresponding by SymbolicRule by substituting fields that were made symbolic with concrete values from the rule, R (e.g., $\langle \text{srcip}=10.1.0.1 \dots \text{action}=1 \rangle$ matches a SymbolicRule, $\langle \text{srcip}=A \dots \text{action}=1 \rangle$). Then, we fetch the corresponding symbolic FSM and the key type, and use the key type (e.g., srcip-port for per-source NF) to appropriately clone the symbolic model to create an ensemble representation. There is one additional step when the key type is not per-connection; we must substitute any ranges based upon the key type. For example, for a per-source NF, dstip-port in a concrete model refers to a range of concrete values specified in R for dstip and dstport. The output is an ensemble of concrete models for each rule in a configuration.

Processing incoming packets: Upon receiving a packet, the NF fetches the corresponding rule in a configuration using the processing semantics (e.g., first-match). The NF then uses the key to access the relevant concrete FSM in an ensemble of FSMs and the current state associated with the packet (Line 9 in Algo. 1). Finally, the NF applies the appropriate action and updates the current state associated with that packet. We present a more detailed description of how we instantiate an ensemble of FSMs in §B.

8 Implementation & Evaluation

System Implementation: We implemented Alembic using Java for the extended L*, C for monitoring NF actions, and Python for the rest. We create packet templates using Scapy [6]. Then, Alembic feeds the output of prior modules into the Extended L* built atop LearnLib [38]. We re-architected the Learnlib framework to enable distributed learning where queries are distributed to workers via JSON-RPC [4].⁸ Our L* implementation tracks the symbol-concrete mapping of IPs and ports to translate between symbolic and concrete packets. The symbolic FSM output is stored in DOT format, which is then consumed by the online stage.

⁸Due to some unhandled edge cases, our current implementation requires using only one worker for NFs with dynamic header modifications.

Table 1: Coverage of models over input packet types

PktType	FW			staticNAT		randNAT		LB	
	pf	ut	pNF	pf	pNF	pf	pf	hp	
correct-seq	●	○	●	●	●	●	○	○	○
combined-seq	●		●						

pf: PfSense, ut: Untangle, pNF: ProprietaryNF, hp: HAProxy
 ●: TCP-handshake pkts, {SYN_C, SYN-ACK_C, ACK_C}, for both interfaces
 ○: ○ set excluding SYN_C from the external interface

L^* assumes that we have the ability to reliably reset the NF between every sequences. For *Alembic*, we need to reset the connection states. For some NFs, this can be performed using a single command (e.g., `pfctl -k` in PfSense). However, other NFs required that the VM be rebooted (e.g., Untangle). In such cases, we take a snapshot of the initial state of the VM and restore the state to emulate a reset. This does cost up to tens of seconds but is a practical alternative to rebooting.

Experimental Setup: We used *Alembic* to model a variety of synthetic, open-source, and proprietary NFs. First, we created synthetic NFs using Click [31] to validate the correctness of *Alembic*. Each Click NF takes an FSM as input and processes packets accordingly, so we know NF’s ground-truth FSM. To validate against real NFs, we generated models of PfSense [5] (FW, static NAT, NAT that randomizes the port mappings, and LB), ProprietaryNF (FW, static NAT), Untangle [7] (FW), HAProxy [2] (LB). We now use NAT to refer to a static NAT and a randNAT to refer to a NAT that randomizes the IP-port mappings. Our experiments were performed using Cloud-Lab [1]. We ran PfSense, Untangle, ProprietaryNF, HAProxy, and Click in VMs running on VirtualBox [8]. Recall that Δ_{wait} needs to be customized for each NF. We used Δ_{wait} of 100 for PfSense and Click-based NFs, 250 ms for ProprietaryNF, 200 ms for Untangle, and 300 ms for HAProxy. For NFs that incur unexpected delays (e.g., HAProxy, ProprietaryNF, Untangle), we took a majority vote of 3.

Packet types: We use two TCP packet types. First, the correct-seq set consists of standard TCP packets, {SYN_C, SYN-ACK_C, ACK_C, RST-ACK_C, FIN-ACK_C}, where the handling of seq and ack are under-the-hood. Instead of introducing seq and ack numbers in Σ , we introduce additional logic in the Extended L^* to track seq and ack of the transmitted packets and rewrite them during the inference to adhere to the correct semantics (i.e., update the ack of SYN-ACK_C after we observed an output of SYN_C).⁹ Second, we introduce combined-seq set to model the interaction of TCP packets in the presence of out-of-window packets. We extend the correct-seq set with packets with randomly-chosen, incorrect seq and ack values, {SYN-ACK_I, ACK_I, RST-ACK_I, FIN-ACK_I}.

8.1 Validation using synthetic NFs

A) Inferring the ground-truth model: We provide Click [31] with a 4-state FSM that describes a stateful FW

⁹The seq number is incremented by 1 for packets with a SYN or FIN flag set and otherwise, by the data size. T. The ack number for a side of a connection is 1 greater than any received packet’s sequence number.

Table 2: Results of stress testing

NF (pkt type)	accuracy	NF (pkt type)	accuracy
PfSense FW (C)	98.8-100%	ProprietaryNF FW (C)	99.9-100%
PfSense FW (CI)	94.8-100%	ProprietaryNF FW (CI)	98-100%
PfSense NAT (C)	99.1-100%	PfSense randNAT (C)	98.2-100%
PfSense LB (C)	96.4-97.4%	ProprietaryNF NAT (C)	98.8-100%

C : correct-seq CI : combined-seq

that only accepts packets from external hosts after a valid three-way handshake. We also constructed another 18-state FSM that describes a similar FW and a 3-state FSM that describes a source NAT (SNAT). In all three cases, *Alembic* inferred the ground-truth FSM.

B) Finding intent violations: We used a red-team exercise to evaluate the effectiveness of *Alembic* in finding intent violations in NF implementations. In each scenario, we modified the FSM from A to introduce violations and verified that the *Alembic*-generated model captured the behavior for all of the following four cases. A and B refer to an internal and external host, respectively: (1) a FW prevents the connection from being established by dropping SYN-ACK packets; (2) a FW proactively sends SYN-ACK upon receiving SYN from A to B; (3) a SNAT rewrites the packet to unspecified srcip-port; and (4) a SNAT rewrites a dstip-port. Some of these scenarios are inspired by real-world NFs.

C) Validating key learning: We wrote additional Click [31] NFs that track the number of TCP connections based on different keys. We applied the key learning algorithm to each and confirmed it identifies the correct key (Table 5 in §C).

8.2 Correctness with real NFs

As summarized in Table 1, we generated models for PfSense and ProprietaryNF FWs using both correct-seq and combined-seq sets. For the other NF types, we used only the correct-seq set because the FW models for these NFs already modeled the interaction of TCP packets in the presence of out-of-window packets. For an NF that uses dynamic modification (e.g., randNAT), we cannot correctly instantiate the model in the presence of RST-ACK and FIN-ACK packets (§B). Hence, we only showcased how this NF handles connection establishment. Untangle and HAProxy have SYN retries and spurious resets (i.e., temporal effects) that are beyond our current scope (§3.3) and could not be disabled. Thus, we again only model how these NFs handle connection establishment. Further, during our attempts to infer models, we discovered these two NFs are connection-terminating, where an external SYN_C packet interfered with the connection initiation attempt from the internal host, which violates our independence assumption. To make the learning tractable, we removed the SYN_C from the external interface for these connection-terminating NFs.

Complementary testing methodologies: Since we do not know the ground truth models and thus cannot report the coverage of *code paths* inside the NF, we used three approaches to validate the correctness of our models: (1) *iperf* [3] testing,

Table 3: Time to infer a symbolic model (h: hours, m: min)

NF (pkt type)	time	NF (pkt type)	time
PfSense FW (C)	11 m	ProprietaryNF FW (C)	48 h
PfSense FW (CI)	16 h	ProprietaryNF FW (CI)	25 h 18 m
PfSense NAT (C)	28 m	PfSense randNAT (C)	14 m
PfSense LB (C)	14 m	ProprietaryNF NAT (C)	48 h
Untangle FW (C)	37 m	HAProxy LB (C)	20 m

generating valid sequences of TCP packets; (2) *fuzz testing*, randomly picking a packet type and a concrete IP and port; and (3) *stress testing*, generating packets by first picking a packet type and selecting concrete IP and port values to activate at least one rule.

For each test run, we generated an arbitrary configuration. For NFs that take multiple rules (e.g., FW and NAT), we varied the number of rules between 1, 5, 20, and 100. For each concrete rule, we randomly sampled a field from the field type defined by the ConfigSchema. We ensured that we picked concrete configurations different from the ones used during the inference (§4). For FWs and NATs, the generated configurations were installed on one interface (i.e., internal). Further, as *Alembic* cannot compose models for multi-function NFs (i.e., a FW with NAT), we set allow rules on the FWs when we inferred models for NATs and LBs. For iperf [3] testing, we set up a client and a server and collect the traces on each interface. Because iperf [3] generates a deterministic sequence of packets, we only tested with 1 accept rule. For stress and fuzz testing, we generated sequences of 20, 50, 100, and 300 packets. In each setting, we measured model accuracy by calculating the fraction of packets for which the model produced the exact same output as the NF. Each setting is a combination of the NF vendor and type (e.g., PfSense FW with the correct-seq set), input packet type (e.g., 300 packets), and the number of rules (e.g., 20 rules).

Iperf testing: Our models predicted the behavior of all NFs with 100% accuracy.

Fuzz testing: Across all settings for ProprietaryNF and PfSense FWs (both combined-seq and correct-seq set), the accuracy was 100%. For PfSense and ProprietaryNF NATs, the accuracy was 99.8% to 100%.

Stress testing: We summarize the results in Table 2. For many NFs (e.g., ProprietaryNF and PfSense FWs), we see the lowest accuracy (e.g., 98%) for 1 rule with 300 packets. This is expected because our testing generates a long sequence of packets that the Wp-method with $d = 1$ did not probe. Also, given the same FW (e.g., PfSense FW), we observe higher accuracy for an NF modeled with the correct-seq set compared to that modeled using the combined-seq set. We confirm that the model learned using the combined-seq set is rather large (> 100 states) resulting from the many ways in which the correct and incorrect packets can interact. Note that ProprietaryNF NAT correct-seq took 49 hours to model and ProprietaryNF FW combined-seq took 5 days to infer the model. Going back to our earlier requirements that we

Table 4: Scalability benefits of our design choices

Runtime ($ \Sigma $)	1 connection ($\Sigma=6$)	2 connections ($\Sigma=12$)	3 connection ($\Sigma=18$)
	26 min	10 hr	> 3 days
Runtime (d in Wp-method)	$d=1$	$d=2$	$d=3$
	13 min	1 hr 10 min	7 hr

can afford several tens of hours (i.e., a couple days) for the offline stage, we ran the accuracy testing on an intermediate model inferred after 48 hours, which still achieved high accuracy. We did not perform fuzz or stress testing for Untangle FW and HAProxy LB. These NFs have temporal effects that result in mis-attribution, which is outside our scope (§3.3). We see that *Alembic* achieves high accuracy even with large configurations.

8.3 Scalability

We now evaluate the runtime of *Alembic*'s components.

Time to learn symbolic models: For each NF, we report the longest time to model one SymbolicRule as learning can be parallelized across symbolic rules. In all cases, we use 20 servers setup, except for with PfSense random NAT which used one.⁸ The results are summarized in Table 3. In summary, we achieved our goal of inferring high-fidelity models in less than 48 hours. We find that the runtime depends on: (1) the size of the FSM and $|\Sigma|$, and (2) *Alembic* or NF-specific details (e.g., rebooting). For (1), as the size of $|\Sigma|$ was double for the combined-seq set, it took more than 48 hours to discover 72-state FSM (ProprietaryNF FW, combined-seq) but less than 26 hours for 79-state with the correct-seq set. For (2), discovering the NAT model ProprietaryNF NAT (correct-seq) took longer than the FW as the NAT inference ran in two phases (§6). Lastly, PfSense models take less time to infer as PfSense does not require rebooting, and has shorter Δ_{wait} .

Time to validate the key: We use PfSense FW (correct-seq) to report the time to infer the key. It took 6 hours to infer the key (e.g., 2 hrs for each test).

Time for the online stage: For ProprietaryNF FW, the time to compose a concrete model is 75 ms for 10 rules, 0.6 s for 100 rules, and 5 seconds for 1,000 rules. The result generalizes to other NFs.

Scalability benefits of our design choices: The insights to leverage compositional modeling and KeyLearning allow *Alembic* are critical in achieving reasonable runtime by reducing the size of Σ . Suppose one FW rule takes a source IP field takes a /16 prefix. Without KeyLearning, we need to infer a model with all 2^{16} connections. Similarly, for a configuration with 20 rules, we need to infer a model with all relevant connections. The top half of the Table 4 shows how the runtime drastically increases as we increase the number of connections using a Click-based [31] FWs from §8.1 (using just one worker). Further, we measured the runtime as a function of d in Wp-method (bottom of Table 4). Using $d = 1$, we

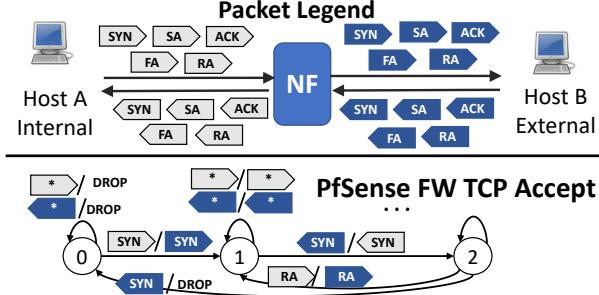


Figure 11: The light/dark coloring indicates packets on host A/B’s interface, respectively. The figure below shows the 3 states for PfSense FW accept rule

were still able to infer the ground truth while reducing the runtime. These results demonstrate how reducing the size of $|\Sigma|$ is critical to obtain a reasonable runtime. Lastly, distributed learning helps scalability. The Click-based [31] FW with 18 states takes 1.6 hours with 1 worker but only 16 minutes with 19 workers (and 1 controller).

8.4 Case studies

We now highlight vendor-specific differences found using *Alembic*. For clarity, we present and discuss only partial representations of the inferred FSMs (as some FSMs are large).

Firewall (correct-seq): A common view of stateful FWs in many tools is a three-state abstraction (SYN, SYN-ACK, ACK) of the TCP handshake. Using *Alembic*, however, we discovered that the reality is significantly more complex. With a single FW accept rule, the inferred PfSense model (correct-seq) shows that a TCP SYN from an internal host, A, is sufficient for an external host, B, to send any TCP packets (Figure 11). Furthermore, FIN-ACK, which signals termination of the connection, does not cause a state transition. We find that ProprietaryNF FW has 79 states for a FW accept rule in contrast to 3 states for PfSense FW. ProprietaryNF, too, does not check for entire three-way handshake (e.g., only SYN, SYN-ACK). We find that the complexity of the FSM (i.e., 79 states) results from the number of ways in which the two TCP handshakes (from A and B) can interfere with each other. Such behavior could not have been exposed through handwritten models. Untangle FW actually behaves like a connection-terminating NF (Figure 13 in §A for partial model). The FW lets the first SYN from A through, but when B replies with a SYN-ACK, Untangle forwards it but preemptively replies with an ACK. When the A replies with ACK, Untangle drops it to prevent a duplicate.

Firewall (combined-seq): Surprisingly, for PfSense, we learned 257 states with combined-seq. The complexity is a result of packets with incorrect seq and ack causing state transitions, where many are forwarded. We learned a 72-state FSM for the ProprietaryNF FW after 48 hours and the full model (104-state) after 5 days. The cause for the larger FSM for PfSense is that the incorrect seq and ack packets often

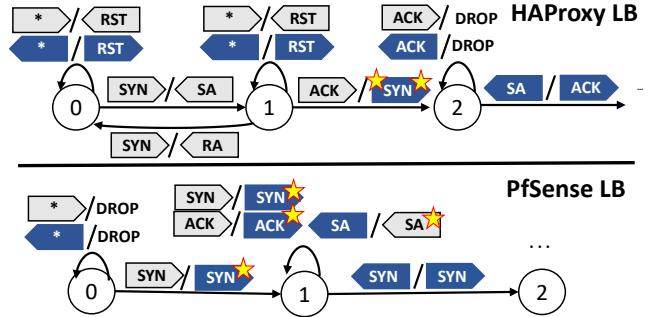


Figure 12: First 3 states of the HAProxy and PfSense LB. Stars on head/tail of packets indicate src/dst modification

cause state transitions more frequently than ProprietaryNF FWs. Further, it is interesting to see how PfSense only had 3 states for the correct-seq set but 257 states with combined-seq, in contrast to ProprietaryNF where the number of states for both sets are similar. At a high-level, we find that obtaining such model is useful as it could possibly be used to generate a sequence of packets to bypass the firewall, but this is beyond the scope of our work.

Load balancer: HAProxy (Figure 12) follows the NF’s connection-terminating semantics. It completes the TCP handshake with the client before sending packets to the server. After the handshake, the source of outgoing packets is modified to server-facing IP of LB, and destination is modified to the server (i.e., star on both-ends of TCP SYN in Fig. 12). In contrast, PfSense LB behaves like a NAT. When a client sends a SYN to the LB, the destination is modified to the server’s IP (i.e., star in state 1 in Figure 12). Then, the LB modifies destination of packets from both client and server. We confirmed that PfSense indeed implements load balancing this way [37]. *Alembic* automatically discovered this without prior assumptions of any connection-termination behavior. Further, the connection-termination semantics of HAProxy differ from those of Untangle FW. Unlike HAProxy, Untangle lets SYN packets through and preemptively completes the connection with the external host. This is yet another example of non-uniformity across NF implementations.

8.5 Implications for network verification

We use two existing tools, BUZZ [22] and VMN [35], to demonstrate how *Alembic* can aid in network testing and verification. Using a Click-based [31] FW which adheres policy 1 and 2 (§2), we compare the test output using: (1) $M_{Alembic}$ inferred using *Alembic*, and (2) existing M_{hand} for FW. Using $M_{Alembic}$, BUZZ did not report a violation. Using M_{hand} , BUZZ reported a violation (*false-positive*) as 1 of 6 test traces did not match (trace in §2). Similarly, for policy 2, BUZZ reported a violation using M_{hand} . The failed test case is: $SYN_{A \rightarrow B}$, $RST_{B \rightarrow A}^{External}$, $RST_{B \rightarrow A}^{External}$. M_{hand} predicts that both RST packets are dropped, as the model does not check for RST flags. However, Click NF allows the first RST packet to

reset the NF state. We also plugged the model for PfSense into a network verification tool, VMN [35]. The existing model in VMN does not check TCP flags. Using VMN, we verified the property: “TCP packets from an external host, B, can reach A even if no SYN packet is sent from A.” Recall that in PfSense, $\text{SYN}_{A \rightarrow B}^{\text{Internal}}$ needs to be sent for B to send TCP packets to A. Hence, the property is NOT SATISFIED. Using $\text{Model}_{\text{hand}}$, the tool returned that the property is SATISFIED whereas using $\text{Model}_{\text{Alembic}}$ indicated that it is not (i.e., *false-negative* for $\text{Model}_{\text{hand}}$).

9 Related Work

NF modeling: There is a large body of work on understanding and modeling NFs [19, 26, 30, 35, 39]. Joseph and Stoica [30] propose a language to model *stateless* NFs to ease the NF deployment process. NFactor [45] uses code analysis techniques to extract packet forwarding models in the form of a match-action table. While this can be complementary, it may be difficult to obtain source-code for proprietary NFs. Some works focus on the NF internal states and how to manage them [26, 39]. Our work is orthogonal as we focus on NF behavioral models of externally-visible actions.

FSM inference: L* algorithm by Angluin lays the foundation for learning the FSM [12]. The techniques of learning FSMs has been used for model checking blackbox systems (e.g., [28, 36]). Symbolic finite automata (SFA) [42] are FSMs where the alphabet is given by a Boolean algebra with an infinite domain. While *Alembic* does not directly formulate the problem to infer SFAs, we use the homogeneity assumption in the IP and port ranges to learn a *symbolic model*. Hence, using abstractions like SFA may help us to naturally embed symbolic encodings. We could potentially leverage a tool (e.g., [20]) that extends L* to infer the SFA. However, using SFA does not address the NF-specific challenges (e.g., inferring the key, handling modifications) but may serve as the basis for interesting future work.

NF model use cases: Many network testing and verification tools need NF models [22, 35, 43]. Some [35, 40] proposed new modeling languages to represent NFs. However, it is unclear how to represent existing NFs using these languages. Symnet [40] wrote parsers to automatically generate NF models using their language, SEFL. Again, it is unclear whether the parser generalizes to other FWs or to arbitrary configurations. However, not all network verification tools require models. Vigor [46] uses the C code of a NAT to verify properties such as memory-safety, which are orthogonal to our approach.

Application of L*: L* has been used to discover protocol vulnerabilities (e.g., [15, 16]) or specific attacks (e.g., cross-site scripting) against web-application firewalls [13]. However, these approaches do not tackle the NF-specific challenges (e.g., handling large configuration space and header modifications). Other works also use L* to infer models of various pro-

ocols (e.g., TLS [18]). While Fiterau-Brosteau et al. [23, 24] inferred the behavior of TCP/ IP implementations in an operating system, these tools leverage a simple extension of L* and cannot model NFs with a large configuration space.

10 Discussion

Before we conclude, we discuss outstanding issues.

Handling more protocols: NFs such as layer-7 load balancers (LB), transparent proxy, or deep packet inspection (DPI) operate at the application layer. To model these cases, *Alembic* needs to generate relevant input packet types for these protocols (e.g., GET, POST, PUT for HTTP). However, the main challenge is to model the multi-layer interactions.

Representing complex NFs: Some NFs exhibit complex actions that cannot be captured with “packet in and packet out” semantics. For instance, to represent quantitative properties (e.g., rate-limiting), we need to incorporate them as part of the input alphabet (e.g., “sessions sent at a certain rate”) and monitor relevant properties to classify NF actions. Similarly, to handle temporal effects (e.g., timeout), we need to add the passage of time (e.g., wait 30 s) to the input alphabet. While we could extend our current infrastructure to handle these NFs, it may be worthwhile to consider more native abstractions other than deterministic FSMs. For instance, many have proposed different abstractions to represent quantitative properties (e.g., [9, 11, 29, 33]) and timing properties (e.g., [10]). Once we pick the abstraction, we can find relevant techniques that extend L* (i.e., [14, 27]). It is difficult to find one abstraction to model multiple properties at once, and we need to pick the abstraction based on the properties of interest.

Handling more complex ConfigSchema: To handle more complex configuration semantics such as “if condition, do X,” and “go to rule X”, we still need to model a rule type (e.g., both X in the above example) similar to our workflow. To incorporate new processing semantics, we need to change how we compose individual models in the online stage.

11 Conclusions

We proposed *Alembic*, a system to automatically synthesize NF models. To tackle the challenges stemming from large configuration spaces, we synthesize NF models viewed as an ensemble of FSMs. *Alembic* consists of an offline stage that learns symbolic models and an online stage to compose concrete models given a configuration. Our evaluation shows that *Alembic* is accurate, scalable, and enables more accurate network verification. While *Alembic* demonstrates the promise of NF model synthesis, there remain some open challenges (§3.3 and §10) that present interesting avenues for future work.

12 Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their suggestions. We also thank Tianhan Hu for his help in implementing a distributed version of *Alembic*, and Bryan Parno, Swarun Kumar, Matthew McCormack, and Adwait Dongare for providing feedback on this paper. This work was funded in part by NSF awards CNS-1440065, CNS-1552481, CCF-1703925, and CCF-1763970.

References

- [1] Cloudlab. <https://www.cloudlab.us/>.
- [2] Haproxy. <http://www.haproxy.org/>.
- [3] iPerf Performance Tool. <https://iperf.fr/>.
- [4] jsonrpc. <https://github.com/briandilley/jsonrpc4j>.
- [5] pfSense. <https://www.pfsense.org/>.
- [6] Scapy. <http://www.secdev.org/projects/scapy/>.
- [7] untangle. <https://www.untangle.com/>.
- [8] Virtualbox. <https://www.virtualbox.org/>.
- [9] ALUR, R., DANTONI, L., DESHMUKH, J., RAGHOTHAMAN, M., AND YUAN, Y. Regular functions and cost register automata. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on* (2013), IEEE, pp. 13–22.
- [10] ALUR, R., AND DILL, D. L. A theory of timed automata. *Theor. Comput. Sci.* 126, 2 (April 1994), 183–235.
- [11] ALUR, R., FISMAN, D., AND RAGHOTHAMAN, M. Regular programming for quantitative properties of data streams. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 15–40.
- [12] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Inf. Comput.* 75, 2 (November 1987), 87–106.
- [13] ARGYROS, G., STAIS, I., JANA, S., KEROMYTIS, A. D., AND KI-AVIAS, A. SfadiFF: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS ’16, ACM, pp. 1690–1701.
- [14] BALLE, B., AND MOHRI, M. Learning weighted automata. In *Algebraic Informatics* (Cham, 2015), A. Maletti, Ed., Springer International Publishing, pp. 1–21.
- [15] CHO, C. Y., BABIĆ, D., SHIN, E. C. R., AND SONG, D. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS ’10, ACM, pp. 426–439.
- [16] CHO, C. Y., BABIĆ, D., POOSANKAM, P., CHEN, K. Z., WU, E. X., AND SONG, D. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC’11, USENIX Association, pp. 10–10.
- [17] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.* 4, 3 (May 1978), 178–187.
- [18] DE RUITER, J., AND POLL, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 193–206.
- [19] DETAL, G., HESMANS, B., BONAVENTURE, O., VANAUBEL, Y., AND DONNET, B. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (New York, NY, USA, 2013), IMC ’13, ACM, pp. 1–8.
- [20] DREWS, S., AND D’ANTONI, L. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2017), Springer, pp. 173–189.
- [21] EDELINE, K., AND DONNET, B. On a middlebox classification, 2017.
- [22] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI’16, USENIX Association, pp. 275–289.
- [23] FITERĂU-BROŞTEAN, P., JANSSEN, R., AND VAANDRAGER, F. Learning fragments of the tcp network protocol. In *International Workshop on Formal Methods for Industrial Critical Systems* (2014), Springer, pp. 78–93.
- [24] FITERĂU-BROŞTEAN, P., JANSSEN, R., AND VAANDRAGER, F. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification* (2016), Springer, pp. 454–471.
- [25] FUJIWARA, S., VON BOCHMANN, G., KHENDEK, F., AMALOU, M., AND GHEDAMSI, A. Test selection based on finite state models. *IEEE Trans. Softw. Eng.* 17, 6 (June 1991), 591–603.
- [26] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM ’14, ACM, pp. 163–174.
- [27] GRINCHSTEIN, O. *Learning of timed systems*. PhD thesis, Acta Universitatis Upsaliensis, 2008.
- [28] GROCE, A., PELED, D., AND YANNAKAKIS, M. Adaptive model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (London, UK, UK, 2002), TACAS ’02, Springer-Verlag, pp. 357–370.
- [29] HENZINGER, T. A. *The Theory of Hybrid Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 265–292.
- [30] JOSEPH, D., AND STOICA, I. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.* (2008).
- [31] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst.* 18, 3 (August 2000), 263–297.
- [32] MAKAYA, C., AND FREIMUTH, D. Automated virtual network functions onboarding. In *IEEE SDN-NFV Conference* (2016).
- [33] MOHRI, M. Weighted automata algorithms. In *Handbook of weighted automata*. Springer, 2009, pp. 213–254.
- [34] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX Association, pp. 1–13.
- [35] PANDA, A., LAHAV, O., ARGYRAKI, K., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association.
- [36] PELED, D., VARDI, M. Y., AND YANNAKAKIS, M. Black box checking. *J. Autom. Lang. Comb.* 7, 2 (November 2001), 225–246.
- [37] PFSENSE. Inbound Load Balancing. https://doc.pfsense.org/index.php/Inbound_Load_Balancing.
- [38] RAFFELT, H., STEFFEN, B., AND BERG, T. Learnlib: A library for automata learning and experimentation. In *Proc. ACM FMICS 2005*.
- [39] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013).

- [40] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM ’16, ACM, pp. 314–327.
- [41] TSCHAEN, B., ZHANG, Y., BENSON, T., BENERJEE, S., LEE, J., AND KANG, J.-M. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference* (2016).
- [42] VEANES, M., HALLEUX, P. D., AND TILLMANN, N. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2010), ICST ’10, IEEE Computer Society, pp. 498–507.
- [43] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 811–830.
- [44] WANG, Z., CAO, Y., QIAN, Z., SONG, C., AND KRISHNAMURTHY, S. V. Your state is not mine: A closer look at evading stateful internet censorship. In *Proceedings of the 2017 Internet Measurement Conference* (New York, NY, USA, 2017), IMC ’17, ACM, pp. 114–127.
- [45] WU, W., ZHANG, Y., AND BANERJEE, S. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2016), HotNets ’16, ACM, pp. 29–35.
- [46] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM ’17, ACM, pp. 141–154.

A Partial FSM for Use Cases

Figure 13 shows partial FSM for Untangle FW accept, drop, default rule, and ProprietaryNF accept rule.

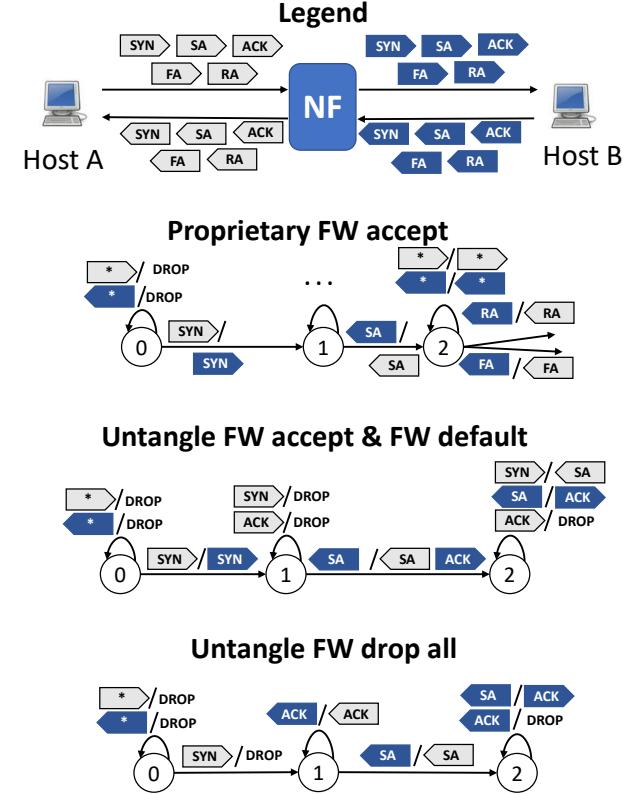


Figure 13: Partial FSM for Untangle FW accept, drop, default rule, and ProprietaryNF accept rule

B Instantiating a Concrete Model

We present a detailed description of how we instantiate a concrete model in our *online* stage. We consider three cases: (1) NFs that keep per-connection state but do not modify headers, (2) NFs that keep per-connection state and do, and (3) NFs that keep state according to other keys but do not modify headers. We do not consider header-modifying NFs that keep state according to other keys (e.g., per-source) as they are outside our current scope. For simplicity, we assume a perfect Equivalence Oracle such that the generated symbolic model from the *offline* stage is identical to the ground truth. **Case 1) NFs that keep per-connection state but do not modify headers**

For NFs that do not modify packet headers, we define a key with $(A:Ap1, B:Bp1)$ where $A:Ap1$ is a srcip-port and $B:Bp1$ is a dstip-port. Note that matches for a per-connection key are bi-directional; a packet with srcip-port, $B:Bp1$, and dstip-port,

A:Ap1, would also match the key, (A:Ap1, B:Bp1). Then, for each concrete value of the key in a rule, we instantiate a concrete FSM.

We posit that our instantiation logic is correct for an input packet type with *all TCP packet* types (e.g., SYN, SYN-ACK, ACK, RST-ACK) for the following reasons:

1. A model learned using one connection from the offline stage represents the ground truth (assuming a perfect Equivalence Oracle).
2. Because we assume each connection is independent and shares the same logical behavior (from §3.3 and Def 4 in §C), cloning a model learned from one connection to represent other connections does not introduce errors.

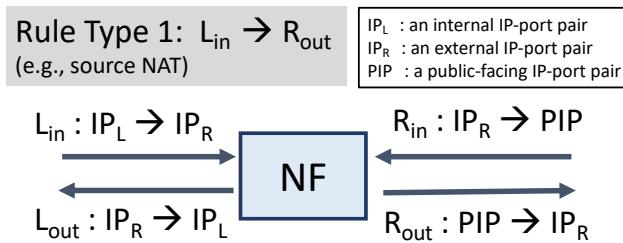


Figure 14: NAT example

Case 2) NFs that keep per-connection state and do modify headers

We extend the NF operational model presented in Alg 1 to instantiate a concrete model for header-modifying NFs. Recall that in the *Alembic’s offline* stage, we learn a model using a range, where we infer a model using a symbolic IP and port in a range. For header-modifying NFs, even though the learned model contains symbolic IPs and ports, our instantiation logic is correct because each concrete model is indexed with a concrete IP and port (Algo. 2).

Consider a NAT with two rule types defined in its ConfigSchema.

1. Rule Type 1: $L_{in} \rightarrow R_{out}$ where the initial modification for a new connections happens for L_{in} (e.g., modifying the source IP of an internal IP to a public-facing IP).
2. Rule Type 2 : $R_{in} \rightarrow L_{out}$ where the initial modification for a new connections happens for R_{in} (e.g., port forwarding where the port 8080 from the R interface is forwarded to port 80 on the internal server).

For ease of explanation, we first show how we instantiate a concrete model for a model inferred for *rule type 1* and later describe how we can easily extend our design to handle rule type 2. Figure 14 shows the ranges of valid source and destination IPs and ports for located packets for a NAT configured with a concrete rule for rule type 1 (e.g., a valid ranges for L_{in} is IP_L for a srcip pair and IP_R for a dstip-port pair).

Algorithm 2 Instantiating a model for a per-connection NF with header modifications

```

1: function ONLINEFORMODIFICATION(locatedPkt p, Rule r,
   Map[rule, Map[key, state]] stateMap,  $T_{L \rightarrow R}$ ,  $T_{R \rightarrow L}$ )
2:   if p.interface == L then
3:      $p_{out} = FWDDIRECTION(p, r, stateMap, T_{L \rightarrow R}, T_{R \rightarrow L})$ 
4:   else
5:      $p_{out} = REVERSEDIRECTION(p, r, stateMap, T_{L \rightarrow R},$ 
       $T_{R \rightarrow L})$ 
6:   return  $p_{out}$ 

7: function FWDDIRECTION(locatedPkt p, Rule r, Map[rule,
   Map[key, state]] stateMap,  $T_{L \rightarrow R}$ ,  $T_{R \rightarrow L}$ )
8:   if NewConnection then
9:     Update  $T_{L \rightarrow R}$ ,  $T_{R \rightarrow L}$ 
10:    Extract FSM, currentState
11:     $p_{out}$ , nextState ← Get action from the FSM
12:    Update currentState with nextState
13:   return  $p_{out}$ 

14: function REVERSEDIRECTION(locatedPkt p, Rule r, Map[rule,
   Map[key, state]] stateMap,  $T_{L \rightarrow R}$ ,  $T_{R \rightarrow L}$ )
15:   if  $p \in T_{R \rightarrow L}$  then
16:     Extract FSM, currentState
17:      $p_{out}$ , nextState ← Get action from the FSM
18:     Update currentState with nextState
19:   else
20:     Extract default FSM, currentState
21:      $p_{out}$ , nextState ← Get action from default FSM
22:     Update currentState with nextState
23:   return  $p_{out}$ 

```

To tackle the challenge above, we introduce two maps to associate an output (or modified) packet’s 5-tuple to the corresponding input packet’s 5-tuple for both interfaces. Specifically, we use $T_{L \rightarrow R}$ to map L_{in} to R_{out} , and $T_{R \rightarrow L}$ to map R_{in} to L_{out} (Algo 2). Algo 2 is a detailed description after Line 3 in the operational model (Algo 1 in §3). Note that for each of presentation, we assume we found a rule to apply (Line 3 in Algo 1) for the incoming packet.

If an NF receives a packet from the L interface, the algorithm checks whether the packet is a new connection by performing a lookup in the map (in FWDDIRECTION). If the connection does not already exist in the map, we update the $T_{L \rightarrow R}$ with $(IP_L, IP_R) \rightarrow (PIP, IP_R)$ and $T_{R \rightarrow L}$ with $(IP_R, PIP) \rightarrow (IP_R, IP_L)$. Then, we extract the corresponding FSM and the current state (or the initial state if a new connection) to apply the appropriate action (i.e., determine p_{out}). If the incoming packet is from the R interface, we look up the corresponding map, $T_{R \rightarrow L}$, to fetch the original IP-port (e.g., IP_L). Then, it uses the key to fetch the FSM and determine the appropriate action for the incoming packet. If the entry does not exist in the map, our concrete model instead uses the FSM associated with the NF’s default behavior. Note

that in the case of static header modification, such as a NAT configured with a list of static mappings between internal and external IP-port pairs, we prepopulate $T_{L \rightarrow R}$ and $T_{R \rightarrow L}$ with these static mappings. Hence, for an NF that statically modify packet headers, we will not reach Line 20 as these mapping already exist.

Extending for Rule Type 2 : We now discuss how to adapt the above framework to handle *rule type 2* where the initial modification happens for packet entering the other interface (e.g., R_{in}). In contrast to rule type 1, an NF configured with a concrete rule for rule type 2 initially modifies packet header for R_{in} (i.e., not L_{in}). We need to make two changes in Algo 2:

1. Line 2 must change to call FWDDIRECTION (Line 7) if the packet comes via the R interface.
2. For the corresponding packet coming from the reverse direction (i.e., L_{in} for rule type 2), we need to perform a look up in $T_{L \rightarrow R}$ to check if the reverse mapping exists instead of $T_{R \rightarrow L}$ (i.e., change Line 15).

Note that our approach does not need a priori knowledge of which rule type the NF is configured with. We just need to infer at which interface the initial modification happens by parsing the generated model. For instance, if the initial modification happens for L_{in} (i.e., rule type 1), then we follow the original algorithm shown in Algo. 2. If the initial modification happens for R_{in} (i.e., rule type 2), then we follow the algorithm in Algo. 2 with two changes mentioned above.

The above algorithm describes how we instantiate a concrete FSM. Now, there are two types of modifications. In the case of static modification, we know the value of the modified packet a priori for a given incoming packet, so we can prepopulate the concrete FSMs with all the known IPs and ports. However, in the case of dynamic modification where we cannot predict the modified values in advance, we initialize an ensemble of concrete FSMs with *symbolic* IP and port (for the modified values) and bind them to concrete IPs and ports as they are revealed (i.e., after injecting packets and observing outputs).

Given this context, we posit the correctness of these instantiated models (formal proof is outside our current scope). For per-connection NFs with static header modifications, our instantiation of FSMs is correct with an input packet type of *all TCP packet types*, for the same two reasons described for case 1. We now state additional reasonings:

1. The same 5-tuple for an input packet maps to the same 5-tuple for the output packet, and $T_{L \rightarrow R}$ and $T_{R \rightarrow L}$ store these mappings. Thus, we will correctly discover the reverse mapping during the instantiation.
2. Even in the presence of connection resets, the same 5-tuple will be mapped to the same output (i.e., 5-tuple). Hence, the model for each connection is correct even in the presence of packets that reset the connection state (i.e., we can reuse the previous mappings stored).

Table 5: Validating the correctness of KeyLearning using Click-based NFs (§8.1)

Ground Truth	Test1	Test2	Test3	Result
Cross-conn	Y			Cross-conn
Per-src	N	Y		Per-src
Per-dst	N	N	Y	Per-dst
Per-conn	N	N	N	Per-conn

For NFs that dynamically modify packet headers, we posit that for the input set of *TCP-handshake packets* (i.e., SYN, SYN-ACK, ACK). However, when we receive a TCP packet that resets a connections (e.g., RST-ACK), the concrete IP and port that was bound to a symbolic IP and port will change (i.e., after a reset, srcip-port maps to P:Pp2 instead of P:Pp1). Hence, the generated model will continue to use the mapping already stored in $T_{L \rightarrow R}$ and $T_{R \rightarrow L}$, resulting in inaccurate model.

Case 3: NFs that do not keep per-connection state

We now consider NFs that do not modify packet headers but have keys other than per-connection. Recall the following key types and their corresponding header fields:

1. *Per-source key*, defined by a source IP
2. *Per-destination key*, defined by a destination IP
3. *Cross-connection key*, defined by *any* packet (i.e., all IP and ports with the range)
4. *Stateless key*, defined by srcip-port and dstip-port. Note that we view the stateless NF as keeping a per-connection state but the FSM is always just a single state.

When we instantiate an ensemble of concrete FSMs for an NF that keeps per-source state, the IPs and ports that do not define the key (i.e., srcport, dstip, and dstport) refer to ranges of values. Hence, the model for a srcip should accept ANY srcport, dstip, and dstport within the specified range.

We posit that our instantiation logic outputs a correct model for an input packet type, with all TCP-relevant symbols (i.e., All TCP-relevant symbols as there are no modifications) if the per-source NF adheres to Def. 2 (§C):

1. Our definition for per-source NF assumes that all destinations given the same source IP are treated homogeneously. Hence, it is correct to use the model learned from one connection and simply replace the symbolic destination in the model to any destination IP that appears in the configuration.
 2. As we assume no header modification, the instantiated model is correct for all TCP-relevant symbols.
- We omit the cases for per-destination, cross-connection, and stateless for brevity. The correctness arguments for these cases are similar to that of per-source NFs.

C Correctness of KeyLearning

We formalize the definition of the granularities of states maintained by NFs (i.e., keys) and prove the correctness of our

KeyLearning algorithm in §5.

Recall that each NF SymbolicRule (1 rule) consists of multiple configuration fields (e.g., FW needs to be configured to allow packets from a subnet X to Y). To simplify the presentation, let us consider a rule r in an NF that takes two configuration fields, namely source and destination, and thus also omit configuration fields that do not affect the key (e.g., an action and a load-balancing algorithm that do not affect the key). We use $NF_r^{(X,Y)}$ to refer to an NF instance only with the targeted rule r that is configured with source X and destination Y. Given such an NF instance, we use L^* to learn a model from it. Particularly, let $L^*(NF_r^{(X,Y)})$ refer to the FSM learned by L^* for the NF instance $NF_r^{(X,Y)}$ using packets only from the set $\Gamma \subset X \times Y \cup Y \times X$. We assume that the FSM learned by L^* is correct with respect to the NF instance. That is, given any sequence of packets with source a and destination b , running $L^*(NF_r^{(X,Y)})$ on it obtains the same output sequence as running $NF_r^{(X,Y)}$ on it, provided that $(a,b) \in X \times Y$ or $(a,b) \in Y \times X$.

Definition of keys: To prove the correctness of our KeyLearning algorithm, we first formalize the definition of NF keys. The following table summarizes the notations we use.

Term	Definition
Σ	the set of packets (symbol for FSMInference)
$\Sigma^{(X,Y)}$	the set of packets with source (destination, resp.) IP from X (Y, resp.)
$\sigma _{(a,b)}$	Given σ and a source-destination pair (a,b) , $\sigma _{(a,b)}$ is the sequence of packets obtained from σ by removing all packets that are not with source a and destination b .
$\sigma _{(a,b),(b,a)}$	Similar to above, but also keeps packets with source b and destination a .
$\sigma+(a,b)$	The sequence obtained by appending (a,b) to the sequence σ
$NF_r^{(X,Y)}(\sigma)$	the output of the last packet given σ to the NF configured with $r^{(X,Y)}$

The definition of keys is given as follows.

Definition 1 (Cross-connection NF). A rule r in an NF keeps cross-connection state iff for all NF instances $NF_r^{(X,Y)}$, all pairs of connections (a,b) and (c,d) such that $a,c \in X, b,d \in Y$, and $(a,b) \neq (c,d)$, there exists a sequence $\sigma \in \Sigma^{(a,b)}$, such that $NF_r^{(X,Y)}(\sigma+(c,d)) \neq NF_r^{(X,Y)}((c,d))$.

Definition 2 (Per-source NF). A rule r in an NF keeps per-source state if all its instance $NF_r^{(X,Y)}$ satisfies the three conditions:

1. for all $a \in X$ and $b \in Y$, there exists a σ over $\Sigma^{\{a\},Y}$, such that $NF_r^{(X,Y)}(\sigma+(a,b)) \neq NF_r^{(X,Y)}((a,b))$.
2. for all $a \in X$, $b \in Y$, and σ_1, σ_2 over $\Sigma^{\{a\},Y}$ such that σ_1 and σ_2 have the same length, $NF_r^{(X,Y)}(\sigma_1+(a,b)) = NF_r^{(X,Y)}(\sigma_2+(a,b))$.

$$NF_r^{(X,Y)}(\sigma_2++(a,b)).$$

3. for all $a \in X$, $b \in Y$, and σ over $\Sigma^{(X,Y)}$, $NF_r^{(X,Y)}(\sigma++(a,b)) = NF_r^{(X,Y)}(\sigma|_{(a,-)}++(a,b))$.

Definition 3 (Per-destination NF). A rule r in an NF keeps per-destination state if all its instance $NF_r^{(X,Y)}$ satisfies the three conditions:

1. for all $a \in X$ and $b \in Y$, there exists a σ over $\Sigma^{(X,\{b\})}$, such that $NF_r^{(X,Y)}(\sigma+(a,b)) \neq NF_r^{(X,Y)}((a,b))$.
2. for all $a \in X$, $b \in Y$, and σ_1, σ_2 over $\Sigma^{(X,\{b\})}$ such that σ_1 and σ_2 have the same length, $NF_r^{(X,Y)}(\sigma_1+(a,b)) = NF_r^{(X,Y)}(\sigma_2+(a,b))$.
3. for all $a \in X$, $b \in Y$, and σ over $\Sigma^{(X,Y)}$, $NF_r^{(X,Y)}(\sigma++(a,b)) = NF_r^{(X,Y)}(\sigma|_{(-b)}++(a,b))$.

Definition 4 (Per-connection NF). A rule r in an NF keeps per-connection state if all its instance $NF_r^{(X,Y)}$ satisfies the two conditions:

1. for all $(a,b) \in X \times Y \cup Y \times X$, there exists a σ over $\Sigma^{\{a\},\{b\}} \cup \Sigma^{\{b\},\{a\}}$, such that $NF_r^{(X,Y)}(\sigma+(a,b)) \neq NF_r^{(X,Y)}((a,b))$.
2. for all $(a,b) \in X \times Y \cup Y \times X$, and σ over $\Sigma^{(X,Y)} \cup \Sigma^{(Y,X)}$, $NF_r^{(X,Y)}(\sigma++(a,b)) = NF_r^{(X,Y)}(\sigma|_{(a,b),(b,a)}++(a,b))$.

Definition 5 (stateless). A rule r in an NF is called a stateless NF iff for all NF instance $NF_r^{(X,Y)}$, packet $p \in \Sigma^{(X,Y)}$, and sequence σ over $\Sigma^{(X,Y)}$, $NF_r^{(X,Y)}(\sigma++p) = NF_r^{(X,Y)}(p)$.

In addition, we assume all NFs satisfy the following consistency in the configuration space:

Definition 6 (Consistency in the configuration space). For all A, B, X, Y, σ such that $A \subset X, B \subset Y$ and σ is a sequence over $\Sigma^{(A,B)}$, $NF_r^{(X,Y)}(\sigma) = NF_r^{(A,B)}(\sigma)$.

FSM composition: The definition of FSM composition is given below.

Definition 7 (FSM composition for key learning). Given two FSMs $FSM_i = (S_i, \Sigma_i, \Delta_i, \delta_i, s_i^0)$, where S_i is the state space, Σ_i is the space of possible input symbols such that $\Sigma_1 \cap \Sigma_2 = \emptyset$, Δ_i is the set of output symbols, $\delta_i : S_i \times \Sigma_i \rightarrow S_i \times \Delta_i$ is the transition function, and $s_i^0 \in S_i$ is the initial state of FSM_i , the composite FSM of FSM_1 and FSM_2 is $FSM_{composite} = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2, \delta, s_1^0 \times s_2^0)$, where $\delta((s_1, s_2), p) = ((s'_1, s'_2), p')$ if and only if 1) $\delta_1(s_1, p) = (s'_1, p')$ and $s_2 = s'_2$; or 2) $\delta_1(s_2, p) = (s'_2, p')$ and $s_1 = s'_1$.

Proof of KeyLearning algorithm: The correctness of our KeyLearning algorithm is given in the following theorem.

Theorem 1 (Correctness of KeyLearning). Figure 10 is correct.

Proof Sketch. For brevity, we only prove the column for the per-source NF; proofs of other columns are similar. The proof for per-source NF follows from the three lemmas below. \square

Lemma 1. *All NFs that keep per-source state cannot pass Test 1.*

Proof. Let A_1 and A_2 be the FSM learned for $NF_r^{\langle\{a\},\{b\}\rangle}$ and $NF_r^{\langle\{c\},\{d\}\rangle}$ respectively (i.e., $A_1 = L^{\{(a,b)\}}(NF_r^{\langle\{a\},\{b\}\rangle})$, similarly for A_2), B be the FSM learned for $NF_r^{\langle\{a,c\},\{b,d\}\rangle}$ using packets from (a,b) and (c,d) (i.e., $B = L^{\{(a,b),(c,d)\}}(NF_r^{\langle\{a,c\},\{b,d\}\rangle})$), and C be the FSM composed of A_1 and A_2 . We only need to prove that for any sequence σ consisting of packets over $\{(a,b),(c,d)\}$, $B(\sigma) = C(\sigma)$. W.L.O.G., suppose σ ends with (a,b) . Then $B(\sigma) = NF_r^{\langle\{a,c\},\{b,d\}\rangle}(\sigma) = NF_r^{\langle\{a,c\},\{b,d\}\rangle}(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$ (condition 3), $C(\sigma) = C(\sigma|_{(a,b)}) = A_1(\sigma|_{(a,b)})$ (the first equality is by condition 3 and the second is by FSM composition). But by homogeneity in the config space, $A_1(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$. Thus, $B(\sigma) = C(\sigma)$. In other words, B is equivalent to C . \square

Lemma 2. *All NFs that keep per-source state can pass Test 2.*

Proof. Let A_1 and A_2 be the FSM learned for $NF_r^{\langle\{a\},\{b\}\rangle}$ and $NF_r^{\langle\{c\},\{b\}\rangle}$ respectively, B be the FSM learned for $NF_r^{\langle\{a\},\{b,c\}\rangle}$, and C be the FSM composed of A_1 and A_2 . By the first condition of per-source NF, there exists a σ over $\Sigma^{\langle\{a\},\{b,c\}\rangle}$, such that $B(\sigma++(a,b)) \neq B((a,b))$. By the second condition, $B(\sigma++(a,b)) = B(\sigma'++(a,b))$, where σ' is a sequence consisting of only (a,c) . Since C is composed of A_1 and A_2 , $C(\sigma'++(a,b)) = A_1((a,b))$. But by homogeneity in the config space, $A_1((a,b)) = B((a,b))$. Thus, $C(\sigma'++(a,b)) \neq B(\sigma'++(a,b))$. In other words, B is not equivalent to the composite FSM of A_1 and A_2 . \square

Lemma 3. *All NFs that keep per-source state cannot pass Test 3.*

Proof. Let A_1 and A_2 be the FSM learned for $NF_r^{\langle\{a\},\{b\}\rangle}$ and $NF_r^{\langle\{c\},\{b\}\rangle}$ respectively, B be the FSM learned for $NF_r^{\langle\{a,c\},\{b\}\rangle}$, and C be the FSM composed of A_1 and A_2 . Consider any sequence σ over $\Sigma^{\langle\{a,c\},\{b\}\rangle}$. W.L.O.G., suppose σ ends with (a,b) . Then by condition 3, $B(\sigma) = B(\sigma|_{(a,b)})$. By definition of composition, $C(\sigma) = A_1(\sigma|_{(a,b)})$. But by homogeneity in the config space, $A_1(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$. Thus, $C(\sigma) = B(\sigma)$. In other words, B and C are equivalent. \square

Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations

Wei Sun¹, Lisong Xu¹, Sebastian Elbaum², Di Zhao¹

¹*Department of Computer Science and Engineering, University of Nebraska-Lincoln
Lincoln, NE, {wsun, xu, dzhao}@cse.unl.edu*

²*Department of Computer Science, University of Virginia
Charlottesville, Virginia, selbaum@virginia.edu*

Abstract

The significant impact of TCP congestion control on the Internet highlights the importance of testing the correctness and performance of congestion control algorithm implementations (CCAI) in various network environments. Many CCAI testing questions can be answered by exploring the numerical state space of CCAIs, which is defined by a group of numerical (and nonnumerical) state variables of the CCAIs. However, the current practices for automated numerical state space exploration are either limited by the approximate abstract CCAI models or inefficient due to the large space of network environment parameters and the complicated relation between the CCAI states and network environment parameters. In this paper, we propose an automated numerical state space exploration method, called ACT, which leverages the model-agnostic feature of random testing and greatly improves its efficiency by guiding random testing under the feedback iteratively obtained in a test. Our experiments on five representative Linux TCP CCAIs show that ACT can more efficiently explore a large numerical state space than manual testing, undirected random testing, and symbolic execution based testing, while without requiring an abstract CCAI model. ACT successfully detects multiple design and implementation bugs of these Linux TCP CCAIs, including some new bugs not reported before.

1 Introduction

TCP congestion control algorithms are crucial to Internet performance and stability. We have seen many of them emerged in the last decades [1, 6, 20, 43, 50], and we have witnessed how billions of computers, servers, routers, smartphones, and other Internet devices are affected, when new TCP Congestion Control Algorithm Implementations (CCAI) are deployed, such as Linux CUBIC [20] and Windows Compound-TCP [43]. That is why a significant effort is placed in testing the correctness and performance of CCAIs in various network environments [16].

1.1 Numerical state space exploration

In this paper, we focus on how to explore the *numerical state space* \mathbf{S} ¹ of a CCAI in various network environments. \mathbf{S} is defined by a group of numerical state variables of the CCAI, such as congestion window size ($cwnd$), slow start threshold ($ssthresh$), and smoothed round-trip time (RTT, rtt). \mathbf{S} may also have some additional nonnumerical state variables, such as the Linux TCP variable ca_state whose value indicates the current status of CCAI (e.g., 0:normal, 3:recovery, 4:timeout) but does not have numerical meanings. Space \mathbf{S} contains all possible combinations of the values of the state variables, and each point in \mathbf{S} is called a *state* or state vector. Exploring \mathbf{S} aims to answer questions like the following.

Motivating Example 1: Does Linux CUBIC increase its $cwnd$ appropriately in various network environments? The aggressiveness of CUBIC is determined by its state variable $target$ [20], which is the expected congestion window size after one RTT. It is typically expected [15] that a CCAI increases its $cwnd$ less aggressively in the congestion avoidance stage (i.e., when $cwnd > ssthresh$) than in the slow start stage (i.e., when $cwnd \leq ssthresh$) where it doubles its $cwnd$ every RTT. This requirement can be tested by answering a numerical state space exploration question: does Linux CUBIC ever visit any states satisfying the condition $cwnd > ssthresh$ (i.e., congestion avoidance stage) and $target > 2 \times cwnd$ (i.e., more aggressively)?

Motivating Example 2: Does a Linux CCAI appropriately decrease its $cwnd$ during fast recovery in various network environments? It is typically expected [2] that a CCAI decreases its $cwnd$ in fast recovery when a congestion is detected (e.g., three duplicate ACKs). For example, CUBIC decreases its $cwnd$ to $0.7 * prior_cwnd$ and AIMD² [2] to $0.5 * prior_cwnd$ right after a fast recovery, where state variable $prior_cwnd$ is the congestion window size right before the fast recovery. This requirement can be roughly tested

¹ \mathbf{S} is not to be confused with the TCP connection management state space [10] such as LISTEN, SYN-SENT, and CLOSED.

²Additive Increase Multiplicative Decrease of Reno and NewReno

by answering a numerical state space exploration question: does a Linux CCAI ever visit any states satisfying the condition $\text{previous_ca_state} == 3$ and $\text{ca_state} == 0$ (i.e., just finished fast recovery) and $\text{cwnd} \geq \text{prior_cwnd}$ (i.e., no window decrease at all)?

Similar to these two motivating examples, many CCAI requirements can be tested if we can explore the \mathbf{S} of a CCAI in various network environments. Specifically, in this paper, we consider the *numerical state space exploration problem*: how to automatically sample a network environment parameter space \mathbf{P} in order to efficiently visit as many as different regions of \mathbf{S} within a given amount of testing time? Space \mathbf{P} contains the parameter values of all possible network environments that a tester needs to check, and each point in \mathbf{P} is called a *network environment* or network environment parameter vector. A region of \mathbf{S} contains a group of nearby states in \mathbf{S} , and is defined and discussed in Section 2.1.

1.2 Challenges

The numerical state space exploration problem, however, is challenging to solve. *The first challenge* is that space \mathbf{P} is usually too large to check exhaustively. For example, suppose that a tester is testing a CCAI using a simple network topology with a single link, where the packet loss rate parameter is in the range of [0%, 10%] with a granularity of 10^{-6} , the link bandwidth parameter is in the range of [0.1, 10000] Mbps with a granularity of 0.1 Mbps, and the packet delay parameter is in the range of [0, 1000] ms with a granularity of 1 ms. The \mathbf{P} of this simple example already contains about 10^{13} possible network environments (i.e., combinations).

The second challenge is that the mapping from the \mathbf{P} to \mathbf{S} of a CCAI is usually very complicated so that it is difficult to directly find a network environment in \mathbf{P} that can lead the CCAI to visit certain regions in \mathbf{S} . 1) A real-world CCAI, such as Linux CUBIC, involves multiple intertwined components contributed by tens of developers spanning tens of years. Many state variables, such as cwnd , are affected by multiple components, such as slow start, congestion avoidance, fast recovery, timeout, and undo components. 2) This is exacerbated by the fact that many states in a large \mathbf{S} can be visited only after a large number of packets. For example, thousands of packets are needed in order to increase cwnd and ssthresh to over thousands of packets. That is, the exploration path from the start state to a final state may contain thousands of intermediate states. 3) There are currently no complete abstract models (e.g., state machines, or high-order logic) of real-world TCP implementations capturing all state variables and all components of the CCAs, because they are very challenging to develop and verify. For example, a relatively complete TCP model [3] took several man-years of effort and deals with only the traditional AIMD.

Because of the unknown mapping from a large \mathbf{P} to a large \mathbf{S} , it is hard to efficiently explore \mathbf{S} by either randomly or systematically sampling \mathbf{P} and it is challenging to answer gen-

eral numerical state space exploration questions, like the motivating examples.

1.3 Our contributions

We propose an *Automated Congestion control Testing method*, called ACT, to model-agnostically and efficiently explore a general numerical state space \mathbf{S} of real-world CCAs for a given \mathbf{P} . ACT belongs to the class of feedback-guided random testing methods [28] or guided fuzzing methods [51] used in the software testing and verification community. While the general idea of feedback-guided random testing or guided fuzzing is not new, to the best of our knowledge, our work is the *first one* to use it in automatically exploring a large \mathbf{S} of a CCAI. Specifically, ACT randomly selects network environments in a large \mathbf{P} to explore a large \mathbf{S} , and the random selection of new network environments is guided by the feedback iteratively obtained from the region coverage information of previously selected network environments. We propose two novel types of feedback to explore the low-probability regions of \mathbf{S} : 1) parameter estimation to explore the low-probability regions due to the unknown non-linear mapping from \mathbf{P} to \mathbf{S} , 2) parameter concatenation to explore the low-probability regions due to the correlation among the state variables of \mathbf{S} . Intuitively ACT randomly samples in \mathbf{P} but favoring those network environments that are more likely to explore different regions of \mathbf{S} . By doing so, ACT is scalable to a large \mathbf{P} (i.e., first challenge) and does not require an abstract CCAI model (i.e., second challenge).

Our contributions are threefold. First, we propose an automated and model-agnostic method, ACT, which can efficiently explore a large \mathbf{S} for a large \mathbf{P} without requiring an abstract CCAI model, and then output the states satisfying the specified conditions along with the concrete data necessary to deterministically reproduce the detected states.

Second, we present an ACT implementation using the widely used network simulator NS3 with Direct Code Execution (DCE) [44] to execute the original Linux networking stack. It can be easily used for testing, debugging, and studying the correctness and performance of real-world CCAs in various reproducible and controllable network environments.

Third, we conduct a family of experiments on five representative Linux TCP CCAs showing that ACT can more efficiently explore different regions of \mathbf{S} than manual testing, undirected random testing, and symbolic execution based testing. ACT successfully detects multiple design and implementation bugs of these CCAs, including *several new bugs not reported before*. For example, ACT finds that Linux CUBIC (current default) sometimes misjudges the network congestion and then mistakenly aggressively increases its throughput (i.e., motivating example 1). ACT also detects that Linux AIMD (previous default) sometimes mistakenly doubles its throughput right after a fast recovery (i.e., motivating example 2) or suddenly increases its throughput to an extremely large number.

2 Design of ACT

2.1 Regions of numerical state space \mathbf{S}

Intuitively, a region of \mathbf{S} contains a group of nearby states all satisfying or not satisfying a specified condition, and a region is visited if at least one state of the region is visited. We attempt to explore different regions of \mathbf{S} instead of different individual states because of two unique properties of numerical state variables. 1) Because numerical state variables usually have a large number of possible values, the \mathbf{S} of a CCAI is usually prohibitively large (e.g., in the order of 10^{11} states in our experiments). As a result, it is impossible to visit each individual state in \mathbf{S} in any reasonable amount of testing time. 2) For a numerical state space exploration problem (e.g., the two motivating examples), there are usually one or multiple regions of nearby states (instead of only a single state) all satisfying the same condition. As long as we find at least one state in these regions (i.e., one counterexample), we can answer the exploration problem.

The shape and the size of a region might depend on the CCAIs, \mathbf{S} , \mathbf{P} , and the specified conditions. Without making any special assumption and for the sake of simplicity, we divide \mathbf{S} into equal-sized non-overlapping regions of size k . Specifically, the range of each numerical state variable is divided into equal-sized intervals with size k , and the range of each nonnumerical state variable (if any) is divided into intervals with size 1. A *region* contains all the states with each state variable in the same interval. For example, let's consider a 2-dimensional $\mathbf{S} = \{(cwnd, ssthresh) \mid cwnd \in [1, 1024], ssthresh \in [1, 1024]\}$. If $k = 512$, \mathbf{S} is divided into 4 equal-sized non-overlapping regions $\mathbf{S} = R_1(512) \cup R_2(512) \cup R_3(512) \cup R_4(512)$, where $R_i(k)$ denotes the i -th region when the region size is k . For instance, $R_1(512) = \{(cwnd, ssthresh) \mid cwnd \in [1, 512], ssthresh \in [1, 512]\}$, and $R_4(512) = \{(cwnd, ssthresh) \mid cwnd \in [513, 1024], ssthresh \in [513, 1024]\}$. In the extreme case of $k = 1$, \mathbf{S} is divided into $1024 \times 1024 = 1,048,576$ regions $R_1(1), R_2(1), \dots, R_{1048576}(1)$, each containing only one state. In another extreme case of $k = 1024$, the whole \mathbf{S} is a single region $R_1(1024) = \mathbf{S}$.

Without making any special assumptions about the CCAIs, \mathbf{S} , \mathbf{P} , and the specified conditions, we do not consider a specific region size k . Instead, we attempt to explore as many as different regions for all possible k values within a given amount of testing time.

Note that, it is reasonable to group nearby states of \mathbf{S} into regions, but it is not reasonable to group nearby network environments of \mathbf{P} . This is because even a tiny difference between two network environments may lead to significantly different CCAI behaviors. For example, two packet loss rates of 10^{-5} and 10^{-6} with a tiny difference with respect to a parameter range $[0\%, 10\%]$ lead to about six times of different throughputs for CUBIC [20].

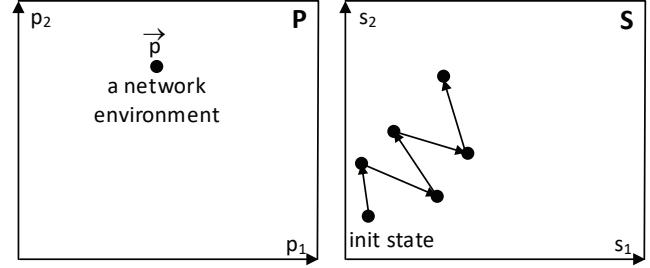


Figure 1: A network environment $\vec{p} \in \mathbf{P}$ leads a CCAI to visit a sequence of states in \mathbf{S} .

2.2 Numerical state space exploration

Each network environment $\vec{p} \in \mathbf{P}$ leads a CCAI to visit a sequence of states in \mathbf{S} starting from the initial state, as illustrated in Fig. 1 using a two-dimensional $\mathbf{P} = \{(p_1, p_2)\}$ and two-dimensional $\mathbf{S} = \{(s_1, s_2)\}$. In a network simulation, the sequence of visited states depends not only on \vec{p} but also a random seed e , which are collectively referred to as a *simulation configuration* $G = (e, \vec{p})$. The simulation results (e.g., visited states) are deterministic for a given G .

The numerical state space exploration problem is given a number N , how to select N simulation configuration G 's in order to maximize $coverage(\mathbf{S}, k)$ for any $k \geq 1$, where $coverage(\mathbf{S}, k)$ is the percentage of visited regions of \mathbf{S} when the region size is k .

$$\max_{N \text{ selected } G's} coverage(\mathbf{S}, k) \quad \forall k \geq 1 \quad (1)$$

Note that we attempt to maximize $coverage(\mathbf{S}, k)$, instead of exploring only a specific region of \mathbf{S} for a specific condition that is nevertheless very challenging too. This is because state space exploration is time consuming, and it is more convenient to explore \mathbf{S} once and then use the explored \mathbf{S} to answer multiple different questions for the same \mathbf{S} .

We say that a testing method is *more efficient* than another one, if given the same N , the $coverage(\mathbf{S}, k)$ of the former is higher than or equal to that of the latter for any $k \geq 1$. In this and next sections, we propose ACT to solve the numerical state space exploration problem, and in Section 4 we empirically evaluate the efficiency of ACT by comparing with other related exploration methods.

The design of ACT is based on the following theorem, where $|\mathbf{S}|$ denotes the total number of states in \mathbf{S} . The proof is shown in the appendix.

Theorem 1 Among all state exploration methods that visit state $i \in [1, |\mathbf{S}|]$ with probability q_i , the exploration method with $q_i = q_j$ for $\forall i, j \in [1, |\mathbf{S}|]$ maximizes $coverage(\mathbf{S}, k)$ for any $k \geq 1$.

2.3 Feedback-guided random testing

Theorem 1 shows that the optimal exploration method should uniformly visit the states. While it is hard or impossible to design such an optimal exploration method, we attempt to design a method that visits as large as a fraction of \mathbf{S} as possible within a limited testing time budget, instead of thoroughly visiting certain regions.

Our proposed ACT is based on undirected random testing [41] that randomly samples \mathbf{P} accordingly to a distribution, because it is scalable to a large \mathbf{P} and does not require an abstract model of a CCAI. Without making any special assumptions about the CCAs, \mathbf{S} , \mathbf{P} , and the specified conditions, ACT uses the simple uniform distribution for the undirected random testing. However, because of the unknown and complicated mapping from \mathbf{P} to \mathbf{S} , undirected random testing tends to repeatedly visit the high-probability regions of \mathbf{S} and thus is inefficient in covering different regions of \mathbf{S} . In other words, *uniformly sampling \mathbf{P} does not lead to uniform coverage of \mathbf{S}* .

ACT leverages the model-agnostic feature of undirected random testing, and greatly improves the region coverage of \mathbf{S} by guiding random testing under the feedback iteratively obtained in the test. Thus, ACT belongs to the class of feedback-guided random testing [28] or guided fuzzing methods [51]. We have identified two major reasons that undirected random testing has low probabilities to visit certain regions of \mathbf{S} , and correspondingly propose two types of feedback to visit these low-probability regions of \mathbf{S} : 1) parameter estimation to visit the low-probability regions due to the unknown nonlinear mapping from \mathbf{P} to \mathbf{S} , 2) parameter concatenation to visit the low-probability regions due to the correlation among different state variables of \mathbf{S} .

2.4 Parameter estimation

One reason that undirected random testing has low probabilities to visit some regions of \mathbf{S} is the unknown nonlinear mapping from \mathbf{P} to \mathbf{S} . For example, let's consider packet loss rate parameter *loss* in the range of [0%, 10%] and state variable *cwnd* in the range of [1, 1024] packets for AIMD. The average *cwnd* of AIMD is greater than 379 packets if *loss* is lower than 10^{-5} [15]. If *loss* is uniformly distributed in [0, 10%], the probability that *cwnd* > 379 is approximately lower than 0.01%, and thus the regions with *cwnd* > 379 have very low probabilities to be visited. With an unknown nonlinear mapping from \mathbf{P} to \mathbf{S} , it is impossible for undirected random testing with any specific distribution (not just uniform) to uniformly visit different states of \mathbf{S} .

Parameter estimation attempts to visit the low-probability regions due to the unknown nonlinear mapping from \mathbf{P} to \mathbf{S} . Specifically, for an unvisited state $\vec{s}^* \in \mathbf{S}$, it attempts to find a network environment \vec{p}^* such that the tested CCAI is likely to visit region $R(\vec{s}^*, k)$, which is the region of state \vec{s}^* when the region size is k . ACT starts with the smallest region size

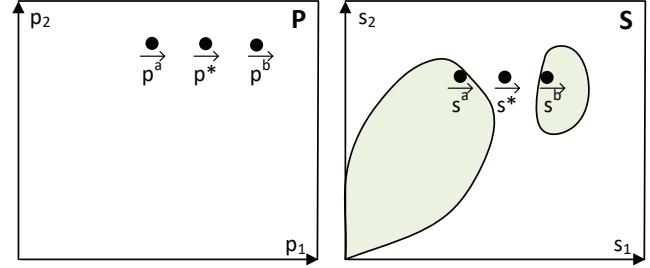


Figure 2: Interpolation finds \vec{p}^* using \vec{p}^a and \vec{p}^b to cover the unvisited gap (e.g., the region of state \vec{s}^*) between two visited regions (e.g., the regions of \vec{s}^a and \vec{s}^b).

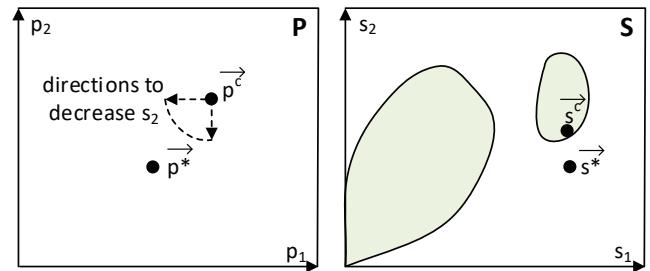


Figure 3: Extrapolation finds \vec{p}^* using \vec{p}^c to visit an unvisited corner or side of \mathbf{S} (e.g., the region of \vec{s}^* below the region of \vec{s}^c). The directions to decrease s_2 are for illustration purpose.

$k = 1$ to find \vec{p}^* , if not successful, it gradually doubles k until it finds \vec{p}^* .

Parameter estimation is illustrated in Figs. 2 and 3 where shaded areas indicate the regions already visited by undirected random testing. The pseudo-code of parameter estimation is given in Method 1. Basically, for an unvisited state \vec{s}^* , we find a new network environment \vec{p}^* using either the interpolation or extrapolation of the past selected network environments. Interpolation is used to cover the unvisited gap between two visited regions in \mathbf{S} , such as state \vec{s}^* in Fig. 2, and extrapolation is used to cover an unvisited corner or side of \mathbf{S} , such as state \vec{s}^* in Fig. 3.

To implement parameter estimation, each state $\vec{s} \in \mathbf{S}$ is associated with a pool of simulation configurations. Each simulation configuration $G = (e, \vec{p})$ contains the random seed e and the network environment \vec{p} of a simulation that visited state \vec{s} . An unvisited state has an empty pool, and a visited state may have multiple simulation configurations if it has been visited multiple times by different simulations.

As an example of interpolation, for state \vec{s}^* in Fig. 2, ACT randomly finds a pair of states \vec{s}^a and \vec{s}^b so that region $R(\vec{s}^*, k)$ lies in between $R(\vec{s}^a, k)$ and $R(\vec{s}^b, k)$ for the smallest possible k . In order to visit $R(\vec{s}^*, k)$, ACT estimates \vec{p}^*

Method 1 Parameter estimation to find a G^* for \vec{s}^*

```

1: function ESTIMATION( $\vec{s}^*$ )
2:    $e^* \leftarrow$  randomly selected random seed
3:   for region size  $k \leftarrow 1; ; k \leftarrow 2 \times k$  do
4:     // First try interpolation
5:     find a pair of states  $\vec{s}^d$  and  $\vec{s}^b$  such that  $R(\vec{s}^*, k)$  lies in
       between  $R(\vec{s}^d, k)$  and  $R(\vec{s}^b, k)$ .
6:     if find at least one pair of  $\vec{s}^d$  and  $\vec{s}^b$  then
7:       randomly and uniformly select one pair
8:        $G^a \leftarrow$  randomly and uniformly select one from the
       pool of simulation configurations associated with  $\vec{s}^d$ 
9:        $p^a \leftarrow$  the network environment in  $G^a$  for  $\vec{s}^d$ 
10:       $p^b \leftarrow$  similarly a network environment for  $\vec{s}^b$ 
11:      for  $i$  from 1 to  $\text{dim}(\mathbf{P})$  do
12:         $p_i^* \leftarrow \text{random}(p_i^a, p_i^b)$ 
13:      return  $G^* \leftarrow (e^*, \vec{p}^*)$ 
14: // If interpolation fails, then do extrapolation
15: find state  $\vec{s}^c$  such that  $R(\vec{s}^c, k)$  and  $R(\vec{s}^*, k)$  differ in
       only one state variable.
16: if find at least one state  $\vec{s}^c$  then
17:   randomly and uniformly select an  $\vec{s}^c$ 
18:    $p^c \leftarrow$  a network environment for  $\vec{s}^c$ 
19:    $j \leftarrow$  the state variable index that  $R(\vec{s}^c, k)$  and
        $R(\vec{s}^*, k)$  differ
20:   for  $i$  from 1 to  $\text{dim}(\mathbf{P})$  do
21:     if  $\frac{d\bar{s}_j}{dp_i}|_{\vec{p}^c}$  and  $s_j^* - s_j^c$  have same sign then
22:        $p_i^* \leftarrow \text{random}(p_i^c, \text{max})$ 
23:     else if different signs then
24:        $p_i^* \leftarrow \text{random}(\min, p_i^c)$ 
25:     else                                ▷ zero gradient
26:        $p_i^* \leftarrow \text{random}(\min, \text{max})$ 
27:     return  $G^* \leftarrow (e^*, \vec{p}^*)$ 

```

using the interpolation of the parameter vectors \vec{p}^d and \vec{p}^b of the pair of states. The interpolation is implemented by lines 4 to 13 of the pseudo-code. Because ACT does not make any assumption about the mapping from \mathbf{P} to \mathbf{S} , it randomly and uniformly selects a network environment \vec{p}^* within the range of \vec{p}^d and \vec{p}^b instead of possibly a linear or some other interpolations.

As an example of extrapolation, for state \vec{s}^* in Fig. 3, ACT randomly finds one state \vec{s}^c lying beside \vec{s}^* so that their regions $R(\vec{s}^c, k)$ and $R(\vec{s}^*, k)$ differ only in one state variable, say state s_j with $j \in [1, \text{dim}(\mathbf{S})]$, where $\text{dim}(\mathbf{S})$ denotes the dimension of \mathbf{S} . That is, state \vec{s}^c and \vec{s}^* have a major difference only in s_j , and have similar other state variables. In order to visit $R(\vec{s}^*, k)$, ACT estimates \vec{p}^* using the extrapolation of network environment \vec{p}^c (i.e., lines 14 to 27 of the pseudo-code). Specifically, the extrapolation estimates \vec{p}^* by

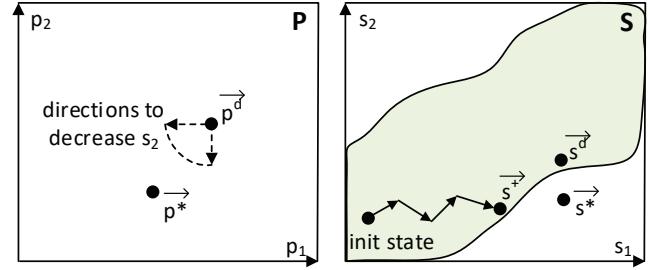


Figure 4: If s_1 and s_2 are positively correlated, \vec{p}^* estimated by extrapolation leads to not only a smaller s_2 but also a smaller s_1 , and thus visits the region of \vec{s}^d instead of \vec{s}^* .

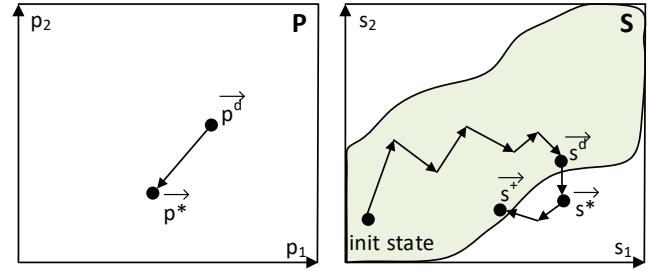


Figure 5: Parameter concatenation visits the region of \vec{s}^* by first following the path from the initial state to state \vec{s}^d using \vec{p}^d , and then the path from state \vec{s}^d to state \vec{s}^* using \vec{p}^* .

increasing or decreasing each parameter of \vec{p}^c based on the impact of that parameter on state variable s_j . The impact of a parameter p_i ($i \in [1, \text{dim}(\mathbf{P})]$) on s_j is measured using the gradient of \bar{s}_j with respect to p_i , where \bar{s}_j is the average of all visited s_j values in a simulation and is defined as $\bar{s}_j = \frac{1}{T} \int_0^T s_j(t) dt$ with T as the simulation time. The gradient at \vec{p}^c is estimated using the simulation results of undirected random testing. For example, states \vec{s}^* and \vec{s}^c in Fig. 3 differ mainly in state variable s_2 , and specifically state \vec{s}^* has a smaller s_2 than state \vec{s}^c . Then extrapolation estimates \vec{p}^* by randomly adjusting \vec{p}^c in the directions to decrease s_2 .

2.5 Parameter concatenation

We notice that some regions of \mathbf{S} have low probabilities to be visited by both undirected random testing and parameter estimation because of the correlation among the state variables of \mathbf{S} . For example, state variables *cwnd* and *ssthresh* are positively correlated due to the window reduction at each congestion event (i.e., three duplicate acknowledgements), where *ssthresh* is set to a certain fraction of *cwnd* (e.g., CUBIC: $ssthresh = 0.7 * cwnd$, AIMD: $ssthresh = 0.5 * cwnd$). Because of this positive correlation, regions with very high *cwnd* values but very low *ssthresh* values and regions with very low *cwnd* values but very high *ssthresh* values have low probabilities to be visited by both undirected random testing

and parameter estimation.

Let's use states \vec{s}^* and \vec{s}^d in Fig. 4 to illustrate why extrapolation does not work if there is a strong positive correlation between s_1 and s_2 . Because \vec{s}^* has a smaller s_2 than \vec{s}^d , extrapolation estimates p^* by randomly adjusting network environment p^d in the directions to decrease s_2 . However, because of the positive correlation between s_1 and s_2 , p^* leads to not only a smaller s_2 but also a smaller s_1 . As illustrated in Fig. 4, p^* leads the tested CCAI to visit the region of state \vec{s}^+ by following the path from the initial state to \vec{s}^+ , instead of visiting the expected region of \vec{s}^* .

Parameter concatenation attempts to visit the low-probability regions due to the state variable correlation. It is illustrated in Fig. 5 where the shaded area indicates all the region visited by the undirected random testing and parameter estimation. The pseudo-code is given in Method 2. Basically, parameter concatenation runs a network simulation with a list of network environments at different time periods in order to visit the unvisited region of state \vec{s}^* .

To implement parameter concatenation, we extend the simulation configurations used in parameter estimation. A simulation configuration associated with state \vec{s}^d is changed to $G^d = (e, \vec{p}^{d1}, t_1, \vec{p}^{d2}, t_2, \dots, \vec{p}^{dn}, t_n)$, which means state \vec{s}^d was visited by a simulation with random seed e , network environment \vec{p}^{d1} from the beginning to time t_1 , \vec{p}^{d2} to time t_2 , ..., and finally \vec{p}^{dn} visiting state \vec{s}^d at time t_n . The visiting time t_n is added to the configuration by ACT during the simulation.

Parameter concatenation runs a network simulation using both the previous network environments $\vec{p}^{d1}, \vec{p}^{d2}, \dots, \vec{p}^{dn}$ of \vec{s}^d and the new network environment \vec{p}^* estimated by extrapolation. At time t_n when the simulation just visits state \vec{s}^d , parameter concatenation changes the current network environment from \vec{p}^{dn} to \vec{p}^* . As illustrated in Fig. 5, such a list of network environments lead the tested CCAI to first visit state \vec{s}^d by following the path from the initial state to \vec{s}^d , and then visit state \vec{s}^+ by following the path from \vec{s}^d to \vec{s}^+ .

The path from \vec{s}^d to \vec{s}^+ in Fig. 5 may possibly visit new regions, such as the region of \vec{s}^* , which are not visited by the path from the initial state to \vec{s}^+ in Fig. 4 for two reasons. First, although both paths finally reach the same state \vec{s}^+ that is determined by network environment \vec{p}^* , they have different starting states and thus go through different paths.

Second, we observe that two state variables may be correlated strongly only over a long time scale but not in a short time scale. For example, over a long time scale, such as spanning multiple window reductions, *cwnd* and *ssthresh* are strongly correlated. But in a short time scale, such as within a congestion avoidance stage between two window reduc-

Method 2 Parameter concatenation to find a G^* for \vec{s}^*

```

1: function CONCATENATION( $\vec{s}^*$ )
2:   for region size  $k \leftarrow 1; ; k \leftarrow 2 \times k$  do
3:     find state  $\vec{s}^k$  such that  $R(\vec{s}^d, k)$  and  $R(\vec{s}^*, k)$  differ in
       only one state variable
4:     if find at least one state  $\vec{s}^d$  then
5:       randomly and uniformly select an  $\vec{s}^d$ 
6:        $G^d \leftarrow$  randomly and uniformly select one from the
       pool of simulation configurations associated with  $\vec{s}^d$ 
7:        $p^{dn} \leftarrow$  the last parameter vector in  $G^d$ 
8:        $j \leftarrow$  the state variable index that  $R(\vec{s}^d, k)$  and
        $R(\vec{s}^*, k)$  differ
9:       for  $i$  from 1 to  $\text{dim}(\mathbf{P})$  do
10:      if  $\frac{ds_j}{dp_i}|_{p^{dn}}$  and  $s_j^* - s_j^d$  have same sign then
11:         $p_i^* \leftarrow \text{random}(p_i^{dn}, \text{max})$ 
12:      else if different signs then
13:         $p_i^* \leftarrow \text{random}(\min, p_i^{dn})$ 
14:      else                                 $\triangleright$  zero gradient
15:         $p_i^* \leftarrow \text{random}(\min, \text{max})$ 
16:       $G^* \leftarrow$  append  $\vec{s}^*$  to end of  $G^d$ 
17:   return  $G^*$ 

```

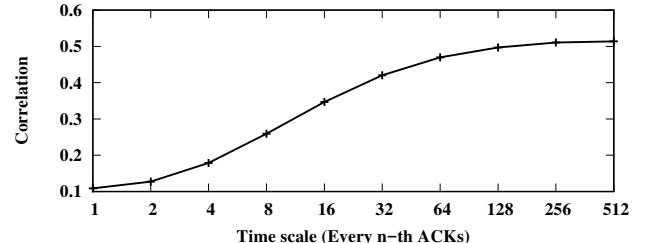


Figure 6: The longer the time scale, the stronger the positive correlation between *cwnd* and *ssthresh*.

tions, they are weakly correlated in that only *cwnd* changes and *ssthresh* remains unchanged. For example, Fig. 6 shows the positive correlation between *cwnd* and *ssthresh* becomes stronger as the time scale n increases. Specifically, the Pearson's correlation coefficient is measured in a sliding window of 10 pairs of *cwnd* and *ssthresh* sampled every n -th ACKs in a simulation and averaged over 30,000 simulations. Because of the strong correlation between s_1 and s_2 in a long time scale, both the path from \vec{s}^d to \vec{s}^+ in Fig. 5 and the path from the initial state to \vec{s}^+ in Fig. 4 reach the same state \vec{s}^+ , which has both a smaller s_1 and a smaller s_2 than \vec{s}^d . But because of the weak correlation in a short time scale, the path from \vec{s}^d to \vec{s}^+ in Fig. 5 may possibly visit the region of \vec{s}^* where only s_2 is changed.

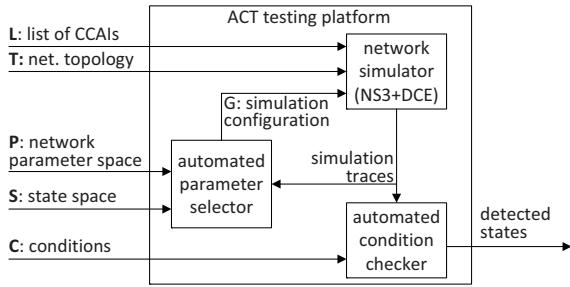


Figure 7: The ACT testing platform consists of three components: the existing network simulator NS3+DCE, our automated parameter selector, and automated rule checker.

3 Implementation of ACT

3.1 Testing platform

The ACT testing platform as illustrated in Fig. 7 takes as the input a list \mathbf{L} of CCAIs, a state space \mathbf{S} of the CCAIs, a network topology \mathbf{T} , a network environment parameter space \mathbf{P} for the topology, and a set \mathbf{C} of state conditions to check. It automatically outputs the states satisfying the conditions along with specific network environments and other data necessary to deterministically reproduce the detected states.

The platform consists of three components. 1) A network simulator simulates CCAI flows of \mathbf{L} in a network described by topology \mathbf{T} and simulation configuration G that includes a random seed e and one or multiple network environments in \mathbf{P} . We choose the widely used NS3 enabled with DCE [44], which can execute the original Linux networking stack in a reproducible and controllable network environments. The output of each simulation is a trace of the timestamped CCAI state variables. 2) The automated parameter selector automatically selects network environments in \mathbf{P} and generates the next simulation configuration G based on the feedback of the region coverage of the previously selected network environments. 3) The automated condition checker automatically checks whether any visited states satisfy the conditions in \mathbf{C} .

3.2 Test input

A test input is a 5-tuple $(\mathbf{L}, \mathbf{T}, \mathbf{P}, \mathbf{S}, \mathbf{C})$. CCAI list $\mathbf{L} = (l_1, l_2, \dots, l_m)$ with $m \geq 1$ indicates the CCAIs of a total of m tested CCAI flows, where l_i with $i \in [1, m]$ is the CCAI of the i -th tested CCAI flow. That is, ACT can be used to test not only a single CCAI flow, but also the interaction among multiple different/same CCAI flows. Network topology \mathbf{T} describes the topology (e.g., the total number of nodes and the routing information) of the tested network environments in which the m tested CCAI flows run. Leveraging the powerful NS3, our testing platform supports various types of network topologies, such as a single link, the dumbbell topology, and the parking lot topology. Network environment param-

eter space \mathbf{P} describes the parameter ranges of the network topology \mathbf{T} . Each point $\vec{p} \in \mathbf{P}$ is a network environment parameter vector $\vec{p} = (p_1, p_2, \dots)$ (also called network environment), where p_i with $i \in [1, \dim(\mathbf{P})]$ is a network environment parameter. CCAI state space \mathbf{S} describes the possible states of the tested CCAI flows. Each point $\vec{s} \in \mathbf{S}$ is a state vector $\vec{s} = (s_1, s_2, \dots)$ (also called state), where s_i with $i \in [1, \dim(\mathbf{S})]$ is a numerical or nonnumerical state variable of a CCAI in \mathbf{L} . \mathbf{C} contains a set of the conditions of the state variables of the CCAIs, and is implemented as a script that reads and analyzes the simulation traces generated by NS3.

Different CCAI tests may need different test inputs. For example, a throughput test checks only a single CCAI flow whereas a fairness test checks multiple CCAI flows, and thus their test inputs have different \mathbf{L} 's. Also the same CCAI state conditions may be used for different test inputs, for example, with different network topologies and/or parameter spaces. This paper focuses on the testing methods, and does not consider the design of comprehensive test inputs for CCAIs.

3.3 Test output

After a test, the testing platform reports all detected states satisfying the conditions. For each detected state, it outputs the corresponding simulation configuration G , which can be used to deterministically reproduce the detected state using NS3. In addition, it outputs the percentage of the regions covered in the test.

3.4 ACT method

ACT has the following four steps.

Step 1, *undirected random testing* repeatedly simulates CCAIs of \mathbf{L} in a network specified by \mathbf{T} and $G = (e, \vec{p})$ with randomly selected seed e and uniformly selected $\vec{p} \in \mathbf{P}$, until the coverage saturates. The goal of this step is not only to have an initial coverage of the state space, but also to profile the mapping from \mathbf{P} to \mathbf{S} to estimate the gradients used in parameter estimation and concatenation. Without making any assumptions for \mathbf{L} , \mathbf{T} , \mathbf{P} , and \mathbf{S} , ACT uses the simple uniform distribution for the undirected random testing.

Step 2, *parameter estimation* iteratively simulates CCAIs in a network specified by \mathbf{T} and $G^* = \text{Estimation}(\vec{s}^*)$ for a uniformly selected unvisited state $\vec{s}^* \in \mathbf{S}$, until the coverage saturates. This step is used to improve the coverage of the low-probability regions due to the unknown nonlinear mapping from \mathbf{P} to \mathbf{S} .

Step 3, *parameter concatenation* iteratively simulates CCAIs in a network specified by \mathbf{T} and $G^* = \text{Concatenation}(\vec{s}^*)$ for a uniformly selected unvisited state $\vec{s}^* \in \mathbf{S}$, until the coverage saturates. This step is used to improve the coverage of the low-probability regions due to the state variable correlation.

Step 4, *condition checking* reports all visited states in \mathbf{S} satisfying the conditions in \mathbf{C} .

ACT checks the coverage saturation using two parameters: saturation size κ and threshold δ . The coverage has reached saturation if the growth rate of $coverage(\mathbf{S}, \kappa)$ is lower than δ . Note that, these two parameters are used only to determine the total testing time, and ACT still attempts to maximize $coverage(\mathbf{S}, k)$ for all possible k values within that testing time. Smaller κ and δ increase $coverage(\mathbf{S}, k)$ for all k values but require longer testing times.

4 Experiments

4.1 General setup

We consider five representative CCAIs of Linux kernel 3.10: the traditional AIMD [2], the current Linux default CUBIC [20], HTCP [33] as a time-based CCAI, HSTCP [15] as a high-speed CCAI, and VENO [17] as a delay-based CCAI. We choose Linux kernel 3.10 for two reasons. First, this is the Linux kernel extensively tested with DCE-enhanced NS3 [44], and thus we can minimize the impact of the potential DCE-enhanced NS3 bugs on our experiments. Second, all the tested CCAIs were initially developed before 2005, and their implementations were already relatively stable in Linux kernel 3.10 that was released in 2013. For all the experiments, we use the default TCP parameters of Linux kernel 3.10, except that the maximum buffer size is increased to not limit the TCP throughput.

Each CCAI has a default test input, which is mainly used for comparing the region coverage of different testing methods, so it does not have any conditions in \mathbf{C} . The default test inputs for different CCAIs are the same, except different \mathbf{L} . For example, the default test input for CUBIC has $\mathbf{L} = (\text{CUBIC})$, and the default test input for AIMD has $\mathbf{L} = (\text{AIMD})$. In every default test input, the network topology \mathbf{T} has a single (virtual) link, which is simple and yet very powerful in simulating various network environments with random packet dynamics in terms of packet bandwidth, delay, loss, and reordering. The network environment parameter space \mathbf{P} contains all possible network environments $\vec{p} = (p_1, p_2, p_3, p_4, p_5, p_6)$, with random packet loss rate $p_1 \in [0\%, 10\%]$ with granularity 10^{-6} , link bandwidth $p_2 \in [0.1, 10000]$ Mbps with granularity 0.1 Mbps, link delay $p_3 \in [1, 1000]$ ms with granularity 1 ms, random queuing delay following a Gamma distribution [26] with shape parameter $p_4 \in [0, 20]$ and scale parameter $p_5 \in [0, 80]$ both with granularity 0.01, and application rate $p_6 \in [0.001, 10000]$ Mbps with granularity 0.1 Mbps. The ranges of the parameters are selected to cover most of possible Internet conditions.

In every default test input, the state space \mathbf{S} contains all possible states $\vec{s} = (s_1, s_2, s_3, s_4, s_5)$, where s_1 is the congestion window size variable $cwnd \in [1, 1024]$ packets with granularity 1 packet, s_2 is the slow start threshold variable $ssthresh \in [1, 1024]$ packets with granularity 1 packet, s_3 is the smoothed RTT variable $rtt \in [0, 2048]$ ms with default Linux granularity 4 ms, s_4 is the smoothed RTT deviation

variable $rttvar \in [0, 1024]$ ms with granularity 4 ms, and s_5 is the congestion avoidance state variable $ca_state \in [0:\text{normal}, 1:\text{disorder}, 2:\text{cwr}, 3:\text{recovery}, 4:\text{timeout}]$. These variables are the basic CCAI state variables, and are maintained in the tcp_sock structure in the Linux kernel. The ranges of these state variables are selected to cover most of possible TCP states in the Internet, except that $cwnd$ and $ssthresh$ could be even larger for ultra-high-speed networks. In addition to these basic state variables, more state variables can be added into \mathbf{S} depending on the tested conditions, such as congestion window size $prior_cwnd$ right before fast recovery, and CCAI-specific variables like $target$ for CUBIC.

In each experiment, each tested CCAI flow transfers a long file of size 15 MBytes, which is selected to be long enough to generate tens of thousands of packets so that all CCAIs can possibly increase their $cwnd$ and $ssthresh$ to over 1024 packets (i.e., their ranges in \mathbf{S}).

4.2 Evaluation: region coverage

We compare the region coverage of ACT with manual testing (MAN) and with other model-agnostic methods: undirected random testing (RAN) and symbolic execution based testing (SYM). We are unable compare ACT with model-guided methods, because there is no abstract model that can capture all state variables used in our experiments.

Methods: ACT: For each default test input, ACT runs DCE-enhanced NS3 simulations with the following saturation parameter values: $\kappa=128$, and $\delta=1.5\%$ per 5000 simulation runs. That is, the coverage has reached saturation if the growth rate of $coverage(\mathbf{S}, 128)$ is slower than 1.5% per 5000 simulation runs. These parameter values are selected so that ACT can finish every test in about three days.

MAN: For each default test input, MAN repeatedly runs simulations with our manually selected network environments, which are similar to those selected for the response function test in a representative CCAI test [24]. Specifically, we consider packet loss rates $p_1=0, 10^{-6}, 10^{-5}, \dots, 10^{-1}$, bandwidths $p_2=1, 10, 100, \text{ and } 250$ Mbps, link delays $p_3=8, 20, 40, 80, \text{ and } 160$ ms, queuing delay shape values $p_4=1$ and 2.5 and scale values $p_5=0, 1, \text{ and } 10$, and application rate $p_6=10000$ Mbps. There are a total of 840 network environments (i.e., combinations), and MAN repeatedly runs simulations with these network environments with different random seeds for the same total number of times as ACT.

RAN: For each default test input, RAN repeatedly runs simulations with uniformly and randomly selected network environments for the same total number of times as ACT.

SYM: Symbolic execution based testing [40, 46] executes the network simulator using symbolic execution platforms, where the packet dynamics (e.g., delay) are represented using symbolic variables with ranges defined according to \mathbf{P} . Because DCE-enhanced NS3 is a huge system where each simulated network node runs a virtualized Linux networking stack, we symbolically execute the simulations using a pow-

erful symbolic execute platform S²E [8], which is capable of symbolically testing a virtual machine. We find that SYM systematically checks all possible TCP behaviors including congestion control behaviors and non-congestion-control behaviors, such as all possible retransmissions and consecutive timeouts of the data packets, and all possible ways to establish and terminate a connection. As a result it can only test a TCP flow for a small file size of a few KBytes within three days instead of the expected 15-MByte file size, and thus it can only increase $cwnd$ by a few packets that is far below our expected 1024 packets.

Result: We show the results of only CCAI $\mathbf{L} = \text{(CUBIC)}$ in Fig. 8, and the results of other CCAIs are similar. Fig. 8 shows the $coverage(\mathbf{S}, k)$ results of ACT, RAN, and MAN, which are measured by the percentages of visited regions with size k . The region coverage of SYM is too low (lower than all others) and not shown in the figure. As k increases, the size of a region increases and the total number of regions in \mathbf{S} decreases, and thus the region coverages of all methods increase. As an extreme case, when $k = 1024$, the whole state space \mathbf{S} is treated as a single region, and thus all three methods achieve 100% coverage. It is interesting that MAN is more efficient (i.e., higher or the same coverage) than RAN for big regions but not for small ones. This is because the network environments used in MAN are representative network environments in \mathbf{P} selected by TCP experts [24], and thus MAN covers a broader range of states than RAN. As a result, MAN is more efficient than RAN for big regions (i.e., $k > 4$). However, MAN has only a limited number of network environments (i.e., 840), and thus covers a smaller number of distinct states than RAN. As a result, MAN is less efficient than RAN for small regions (i.e., $k \leq 4$). We can see that *ACT is more efficient than MAN, RAN, and SYM for all possible region sizes*. Note that ACT achieves high coverage without requiring an abstract CCAI model.

Figs. 9 and 10 show the growth of $coverage(\mathbf{S}, 2)$ and $coverage(\mathbf{S}, 128)$, respectively. When $k = 2$, there are a total of about 10^{10} regions and all three methods achieve very small coverage percentages in three days. When $k = 128$, there are a total of 2048 regions and then all three methods achieve higher coverage percentages. We can see that ACT covers slightly more small regions (i.e., $k = 2$) than RAN, but significantly more big regions (i.e., $k = 128$) than RAN. This is because ACT uniformly selects unvisited states in \mathbf{S} and thus is more likely to visit different big regions, whereas RAN uniformly selects parameter vectors in \mathbf{P} and thus is more likely to redundantly visit the same big regions. Fig. 10 shows that ACT step 2 (i.e., estimation) without requiring an abstract CCAI model already achieves a higher coverage than both MAN and RAN, and ACT step 3 (i.e., concatenation) further greatly improves the coverage.

Note that when k is small (e.g., ≤ 16), all three methods including ACT achieve low coverage (e.g., $\leq 10\%$). This is because we only run each test for three days, and there are

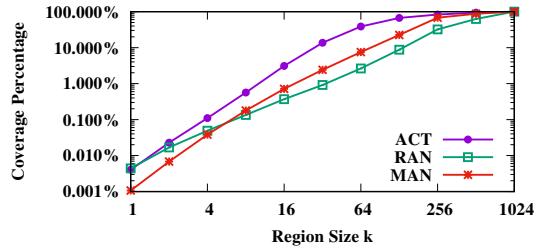


Figure 8: $coverage(\mathbf{S}, k)$ with different k values.

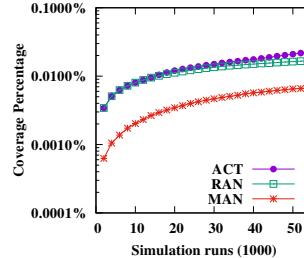


Figure 9: $coverage(\mathbf{S}, 2)$.

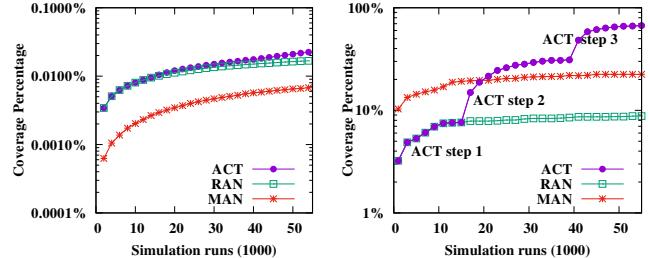


Figure 10: $coverage(\mathbf{S}, 128)$.

very large number of regions when k is small. For example, when k is 16, there are already 8,388,608 regions. In three days, RAN explores about 31,000 different regions, and ACT explores about 260,000 different regions. The coverages of all methods can be improved by running each test for a longer time by reducing parameters κ and δ . But our experiments already clearly demonstrate that ACT is significantly more efficient than MAN, RAN, and SYM giving the same amount of testing time.

4.3 Use case 1: Checking generic behaviors

We demonstrate the capability of ACT in detecting design and implementation bugs using three types of state conditions in the following three subsections, respectively: 1) a condition that checks generic CCAI behaviors, 2) a condition that checks the window increase behavior of a CCAI, 3) a condition that checks the window decrease behavior.

This group of experiments demonstrates that even a simple condition that checks generic CCAI behaviors might be useful for detecting bugs. The test inputs are the same as the default test inputs, except that \mathbf{C} contains a simple condition: $cwnd > 10^7$ packets. Intuitively, this test checks whether the $cwnd$ of a CCAI could be mistakenly larger than some upper bound, such as 10^7 packets that approximately corresponds to the throughput of a TCP flow with a rate of 100 Gbps and an RTT of 1000 ms. Note that although 10^7 is outside of the specified range [1, 1024] for $cwnd$, it is still possible for ACT to detect such states, because ACT keeps track of all the visited states, not just the states in the specified ranges. ACT with this simple condition detects *an implementation bug*. Due to a bug triggered by two consecutive undos, all tested CCAIs with *tcp_sack* disabled, except CUBIC, mistakenly

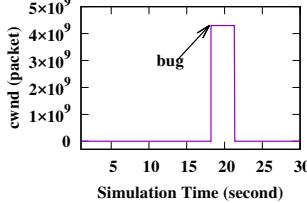


Figure 11: AIMD implementation bug: Suddenly extremely large *cwnd* after consecutive undos.

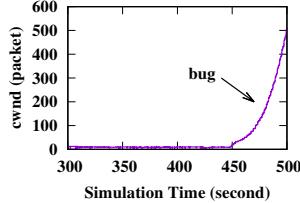


Figure 12: CUBIC design bug: Too aggressive after application rate-limited periods.

set *cwnd* to an extremely large number (i.e., 4,294,967,294 packets), as demonstrated in Fig 11. We thought that it was a new and severe bug and reported it to Linux kernel developers [38], and then were told that it was just fixed a few months ago.

4.4 Use case 2: Checking increase behavior

This group of experiments checks the first motivating example in Section 1. The test inputs are the same as the default test inputs, except that **C** contains a condition: $cwnd > ssthresh$ and $target > 2 \times cwnd$, and **S** contains additional *target*. Intuitively, this test checks whether CUBIC could be mistakenly more aggressive in congestion avoidance than in slow start. ACT detects multiple states satisfying this condition. There are three types of cases. 1) *New design bug* detected by ACT steps 1 and 2: CUBIC is designed to be a time-based congestion control algorithm, and its window increment in one RTT is a function of the duration of the RTT. As a result, in cases of extremely long propagation or queueing delays, CUBIC may set *target* to be higher than twice of the current *cwnd*, which is reasonable for long propagation delays but is questionable for long queueing delays that are possible signs of network congestion. This is an extreme case that we did not consider when we were designing CUBIC [20]. 2) *Design bug* detected by ACT step 3: Linux CUBIC mistakenly increases its *target* too aggressively after a long idle period. This bug was first reported in 2015 [25], and has been fixed in the latest Linux kernel. 3) *New design bug* shown in Fig. 12 detected by ACT step 3: Linux CUBIC mistakenly increases its *target* too aggressively after a long application rate-limited period. Both this and the previous bugs are special cases that we did not consider when we were designing and implementing CUBIC [20].

4.5 Use case 3: Checking decrease behavior

This group of experiments checks the second motivating example in Section 1. The test inputs are the same as the default test inputs, except that **C** contains a condition: $prior_ca_state == 3$, $ca_state == 0$, and $cwnd \geq prior_cwnd$, and **S** contains additional state variables used in the condition. Intuitively, this test checks whether a CCAI

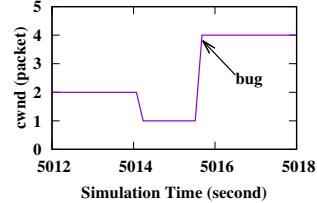


Figure 13: AIMD implementation bug: mistakenly increases *cwnd* after fast recovery.

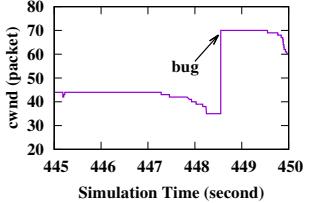


Figure 14: VENO implementation bug: mistakenly increases *cwnd* after fast recovery.

appropriately decreases its *cwnd* in fast recovery. ACT detects multiple states satisfying this condition, all by steps 1 and 2. There are two types of cases. 1) *New implementation bug* of AIMD and HTCP shown in Fig. 13. Due to a calculation boundary bug (happens only when $cwnd < 4$), AIMD and HTCP mistakenly increase *cwnd* to 4 after an undoe fast recovery. This is a new bug and was recently fixed after we reported it to Linux kernel developers [39]. *This is an important bug, because in a highly congested network where we desperately need CCAs, this bug makes the network even more congested.* 2) *Implementation bug* of VENO and HSTCP shown in Fig. 14. VENO and HSTCP mistakenly double their *cwnd* after an undoe fast recovery, because they mistakenly use the default undo function that was designed for AIMD. This bug has been reported before and was fixed in 2016 [47].

5 Discussions

What domain knowledge is required to use ACT? An ACT user needs to know the state variables of a tested CCAI (e.g., by reading the related RFC or papers) in order to define the state space **S**. In addition, currently the user needs to manually instrument the source code of CCAs and NS3 to keep track of the values of the state variables. The contribution of our work is that ACT is model-agnostic so that the user does not need to know how multiple intertwined components of CCAs change the state variables and does not need to know the complicated mapping from **P** to **S**.

An ACT user needs to know the correct behavior of a tested CCAI (e.g., by reading the related RFC or papers) in order to define the set of conditions **C**. In addition, an ACT user needs to manually analyze the outputted simulation traces with buggy behavior (i.e., satisfying the conditions) and then manually check the source code of CCAs to identify the reasons for the bugs. The contribution of our work is that ACT efficiently searches an extremely large number of possible network environments **P**, and automatically finds the specific network environments where the tested CCAs show the buggy behavior, so that the user only needs to manually analyze the specific simulation traces with buggy behavior.

What kind of CCAIs ACT can or can't test? Although we haven't evaluated ACT for all current CCAIs, we conjecture that ACT works for general current and future CCAIs for the following two reasons. First, ACT does not make any specific assumptions about the network environment parameters of \mathbf{P} and the state variables of \mathbf{S} , except that \mathbf{S} contains mainly numerical state variables and state variables should not be strongly correlated in a short time scale. Second, ACT checks the general behaviors of a tested CCAI by analyzing the impact of \mathbf{P} on \mathbf{S} , instead of checking the detailed implementations of the CCAI by analyzing its source code. While different CCAIs may have quite different implementations (e.g., loss based or delay based, expert designed or computer generated, kernel space or user space), they have same or similar general behaviors (e.g., increase or decrease *cwnd* based on network congestion). Having said that, an important future work is to evaluate the effectiveness of ACT for new CCAIs, such as BBR [6], Remy [48], and PCC Vivace [12].

What kind of bugs ACT can or can't detect? ACT can be used to detect the bugs that can be described by state variables of \mathbf{S} , like the two motivating examples. ACT does not work well for the bugs related to the specific packet behaviors, such as whether an acknowledgement packet with the correct acknowledgement number is sent right after receiving a data packet, because it is hard or impossible to describe such a behavior as a condition of state variables. In addition, ACT does not work well for bugs that happen only with certain TCP configuration parameters, because ACT does not search the large space of TCP configuration parameters.

Are there false positives and false negatives? ACT does not have false positives, because ACT can output the specific network environments and the actual simulation traces for each reported bug. However, ACT does have false negatives as it is possible that a tested CCAI satisfies a condition but ACT could not find it. This is because ACT attempts to maximize the region coverage of \mathbf{S} within a testing time budget, instead of covering all regions which requires an unrealistically long time for small k values. Intuitively, this implies that ACT can be used for bug detection but not for correctness guarantee, which is consistent with a fundamental testing principle "Program testing can be used to show the presence of bugs, but never to show their absence" [11] in the software testing community. For real-world networking systems, correctness can be verified only for special cases, such as for the abstract models of the code [34, 37], for code built on verified libraries [52], and for partial pieces of code [42].

6 Related work

6.1 TCP numerical state space exploration

Three types of methods can be potentially used to address TCP numerical state space exploration problems. 1) These problems are usually studied by *manual testing* [19, 24], where a tester manually selects some representative network

environments in \mathbf{P} to test whether a CCAI visits certain regions in \mathbf{S} . Not only is manual testing unscalable to a prohibitively large \mathbf{P} (e.g., only an order of 10^2 network environments are selected in [19, 24]), but also the effectiveness of manual testing highly depends on how much the tester knows about a CCAI.

2) *Automated and model-guided methods* such as [22] have the potential to automatically and efficiently explore a limited \mathbf{S} of a CCAI under the guidance of an abstract model of the CCAI. But the choice of \mathbf{S} is limited by the state variables captured in the abstract model. For example, the model used in [22] does not capture CUBIC state variable *target*, and thus cannot be used to explore the \mathbf{S} of CUBIC in the first motivating example. More importantly, the regions of \mathbf{S} that can be explored are limited by the CCAI components captured in the abstract model. For example, the model used in [22] does not capture the undo component of Linux CCAIs. As a result, it is unable to guide the exploration of the regions that can be reached by the undo component, and then hard to detect the bugs caused by the interference between the undo and fast recovery components in the second motivating example. However, there is currently no complete abstract model for real-world CCAIs, as described in the second challenge.

3) *Automated and model-agnostic methods*, such as undirected random testing [41] and symbolic execution based testing [40, 46], can automatically explore a general \mathbf{S} of a CCAI without requiring an abstract CCAI model. However, they are inefficient to explore different regions of a large \mathbf{S} , because they blindly visit \mathbf{S} and as a result tend to repeatedly or densely explore some regions of \mathbf{S} . Symbolic execution based testing [40, 46] groups all the network environments leading to exactly the same CCAI execution path into equivalence classes in order to improve the scalability over exhaustive testing that exhaustively tests each $\vec{p} \in \mathbf{P}$. However, it is still inefficient in exploring different regions of a large \mathbf{S} for the following reasons. First, it still blindly explores \mathbf{S} , because different equivalence classes of network environments may still repeatedly or densely explore the same regions of \mathbf{S} . Second, the number of equivalence classes of network environments is still prohibitively large, and is roughly an exponential function of the number of packets (i.e., path explosion problem [4]). As a result, it can be used to test CCAI with only a small number of packets [40, 46] or test partial code of CCAIs [42].

ACT attempts to combine the advantages of the model-guided and model-agnostic methods, that is, the efficiency of model-guided methods and the generality of model-agnostic methods. First, ACT is based on undirected random testing instead of symbolic execution based testing, so that it is scalable to a large \mathbf{P} and a large number of packets. Second, ACT guides the selection of network environments under the feedback iteratively obtained in a test, so as to select new network environments that are more likely to explore different regions. As a result, ACT does not blindly explore \mathbf{S} , and

can more efficiently explore different regions of S than undirected random testing and symbolic execution based testing.

6.2 Enhancements to random testing

The efficiency of random testing can be improved by incorporating the right guidance, such as feedback-based random testing [28] and guided fuzzing (e.g., AFL [51]), or by combining with symbolic execution in various ways (e.g., DART [18], Driller [36], MACE [9]). The major difference between these techniques and our proposed ACT is that these techniques explore the general program execution state space by maximizing the code coverage or edge coverage, whereas ACT explores the specific numerical state space of CCAIs where maximizing code coverage may not always be helpful. First, maximizing code coverage might waste the testing resources on covering code with no or little impact on congestion control, such as the TCP code related to connection management or packet formats. Second, many congestion control states can be explored only by repeatedly visiting the already visited code blocks for many times. For example, in order for AIMD to reach from a state with $cwnd = ssthresh = 500$ packets to another state with $cwnd = 1000$ packets and $ssthresh = 500$ packets, AIMD needs to repeatedly visit the same additive increase code for $500 + 501 + 502 + \dots + 999 = 374,750$ times.

The efficiency of random testing can also be improved using genetic algorithms [29], where new test inputs can be generated by recombining two existing test inputs (called crossover), or by randomly changing one existing test input (called mutation). The parameter estimation of ACT is inspired by genetic algorithms. Specifically, the interpolation is inspired by the crossover, as it generates a new network environment by combining two existing network environments. The extrapolation is inspired by mutation, as it generates a new network environment by changing one existing network environment. The major difference between ACT and genetic algorithms is the parameter concatenation of ACT that concatenates a sequence of network environments instead of combining them into a single network environment, as interpolation and extrapolation (similarly crossover and mutation) do not work well for S with correlated state variables.

6.3 General state space exploration

In addition to random testing, many automated techniques have been proposed to explore various state spaces (e.g., program execution space, TCP connection management space) of network programs.

Implementation-level model checking techniques [27, 31] recursively explore the next states from the start state by enumerating all possible events at each state. They are effective for systematically exploring a small state space, but are not scalable to a large one [28]. The path explosion problem [4] limits symbolic execution based techniques [32, 35, 40, 42] to testing only a small number of packets [40], a component of

a network protocol [42], or an abstract network model [37]. Static analysis techniques [7, 13, 45] analyze the network programs at compilation time to infer their run-time behaviors. These techniques [13] are effective at quickly checking shallow behaviors of large programs, but not at accurately checking the deep program behaviors, such as finding the exact network environments that lead a CCAI to certain states after thousands of packets. Model learning techniques [14, 21] attempt to automatically build an abstract model and then explore the state space of the model. But they work only for a small state space.

The major difference between all above techniques and our proposed ACT is that these techniques attempt to explore different individual states and are more suitable for small state spaces, such as nonnumerical state spaces (e.g., TCP connection management state space [27]) or small numerical state spaces of simple protocols (e.g., TFTP [40]), whereas ACT is specifically designed to efficiently explore different regions of an extremely large numerical state space of CCAIs where certain regions can be reached only after thousands of intermediate states (i.e., thousands of packets).

6.4 Other related TCP testing work

Pantheon [49] provides a training ground for evaluating the performance of CCAIs in real-world settings and can automatically calibrate the parameters of a network emulator to match a real network path so that a tested CCAI achieves similar average throughput and delay, whereas ACT attempts to maximize the coverage of the whole state space and then detect bugs. PacketDrill [5] is an automated TCP testing tool that checks whether TCP meets a requirement in a specific network environment \vec{P} , whereas ACT checks whether a CCAI meets a requirement in a large space \mathbf{P} of network environments. Automated trace analysis [3, 23, 30] checks the correctness of TCP packet traces against some formal models or rules mainly about the TCP connection establishment and termination, whereas ACT checks the correctness of TCP congestion control.

7 Conclusion

This paper proposes a CCAI testing tool ACT, and presents several design and implementation bugs of Linux TCP. Most of them are due to the mismatch among different TCP components, because they were designed by different researchers but their interfaces are evolving and not clearly defined. In the future, we plan to extend ACT to test other congestion control algorithms, such as those based on UDP and those in information-centric networking.

ACKNOWLEDGMENT

We thank our shepherd, Anirudh Sivaraman, and the reviewers for their constructive comments. The work presented in this paper was supported in part by NSF CNS-1526253 and NSF SHF-1718040.

Appendix

Proof: Let $I(k)$ denote the total number of regions in \mathbf{S} when the region size is k . Let $Q_i(k)$ denote the probability to visit region $R_i(k)$ with $i \in [1, I(k)]$, which is the probability that at least one state in region $R_i(k)$ is visited. In the special case when $k = 1$, we have $I(k) = |\mathbf{S}|$ and $Q_i(k) = q_i$.

Suppose that a method runs the network simulator for N times and each time visits M states in \mathbf{S} . The probability that region $R_i(k)$ is visited at least once is $1 - (1 - Q_i(k))^{N \times M}$. The expected number of visited regions is $\text{coverage}(\mathbf{S}, k) = \sum_{i=1}^{I(k)} (1 - (1 - Q_i(k))^{N \times M})$. Thus, the numerical state space exploration problem can be rewritten as follows.

$$\text{Maximize} \quad \sum_{i=1}^{I(k)} (1 - (1 - Q_i(k))^{N \times M}) \quad (2)$$

$$\text{Subject to} \quad \sum_{i=1}^{I(k)} Q_i(k) = 1 \quad (3)$$

Using the Karush-Kuhn-Tucker conditions, we can prove that the maximum coverage is achieved when $Q_i(k) = Q_j(k)$ for $\forall i, j \in [1, I(k)]$. If and only if $q_i = q_j$ for $\forall i, j \in [1, |\mathbf{S}|]$, we have $Q_i(k) = Q_j(k)$ for $\forall i, j \in [1, I(k)]$ and for any $k \geq 1$. That is, given the same amount of testing time (i.e., the same N), the uniform distribution is the only one that maximizes $\text{coverage}(\mathbf{S}, k)$ for any $k \geq 1$. ■

References

- [1] M. Alizadeh, A. Greenberg, D. Maltz, and J. Padhye et al. Data center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, New Delhi, India, August 2010.
- [2] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. *RFC 5681*, September 2009.
- [3] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of ACM SIGCOMM*, Philadelphia, PA, August 2005.
- [4] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.
- [5] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. Chu, A. Terzis, and T. Herbert. PacketDrill: Scriptable network stack testing, from sockets to packets. In *Proceedings of USENIX ATC*, San Jose, CA, June 2013.
- [6] N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. *Communications of the ACM*, 60(2):pp. 58–66, February 2017.
- [7] Q. Chen, Z. Qian, Y. Jia, Y. Shao, and Z. Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of ACM CCS*, Denver, CO, October 2015.
- [8] V. Chipounov, V. Kuznetsov, and G. Cadea. The S2E platform: design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), February 2012.
- [9] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of USENIX Conference on Security (SEC)*, San Francisco, CA, August 2011.
- [10] DARPA Internet Program. Transmission control protocol – protocol specification. *RFC 793*, September 1981.
- [11] E. Dijkstra. *Notes on Structured Programming in Book Structured Programming*. Academic Press, 1972.
- [12] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, P. Godfrey, and M. Schapira. PCC Vivace: Online-learning congestion control. In *Proceedings of USENIX NSDI*, Renton, WA, April 2018.
- [13] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proceedings of International Conference on Verification, Model Checking and Abstract Interpretation*, Venice, Italy, January 2004.
- [14] P. Fiterau-Brosteam, R. Janssen, and F. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *Proceedings of International Conference on Computer Aided Verification*, Canada, July 2016.
- [15] S. Floyd. HighSpeed TCP for large congestion windows. *RFC 3649*, December 2003.
- [16] S. Floyd and M. Allman. Specifying new congestion control algorithms. *RFC 5033*, August 2007.
- [17] C. Fu and S. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communication*, 21(2):216–228, February 2003.
- [18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of ACM Programming Language Design and Implementation*, Chicago, IL, June 2005.
- [19] S. Ha, L. Le, I. Rhee, and L. Xu. Impact of background traffic on performance of high-speed TCP variant protocols. *Computer Networks*, 51(7):1748–1762, May 2007.

- [20] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.
- [21] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *Proceedings of IEEE ICNP*, Orlando, FL, October 2008.
- [22] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proceedings of NDSS*, San Diego, CA, February 2018.
- [23] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. Network protocol system monitoring - a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, 14(2):424–437, 2006.
- [24] Y. Li, D. Leith, and R. Shorten. Experimental evaluation of high-speed congestion control protocols. *IEEE/ACM Transactions on Networking*, 15(5):1109–1122, October 2007.
- [25] P. McManus. Thanks Google for open source TCP fix, September 2015. <http://bitsup.blogspot.com/2015/09/thanks-google-tcp-team-for-open-source.html>.
- [26] A. Mukherjee. On the dynamics and significance of low frequency components of Internet load. *Internetworking: Research and Experience*, 5:163–205, December 1994.
- [27] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proceedings of USENIX NSDI*, San Francisco, CA, March 2004.
- [28] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.
- [29] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithm. *Journal of Software: Testing, Verification and reliability*, 9(4):263–282, December 1999.
- [30] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of ACM SIGCOMM*, Cannes, France, September 1997.
- [31] W. Rathje and B. Richards. A framework for model checking UDP network programs with Java Pathfinder. In *Proceedings of ACM High Integrity Language Technology (HILT) International Conference*, Portland, OR, October 2014.
- [32] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of ACM/IEEE IPSN*, Stockholm, Sweden, April 2010.
- [33] R. Shorten and D. Leith. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLD-Net*, Argonne, IL, February 2004.
- [34] M. Smith and K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Transactions on Networking*, 10(2):193–207, August 2002.
- [35] J. Song, C. Cadar, and P. Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014.
- [36] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbette, Y. Shoshtaishvili, C. Kruegel, and G. Vigna. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of NDSS*, San Diego, CA, February 2016.
- [37] R. Stoescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of ACM SIGCOMM*, Brazil, August 2016.
- [38] W. Sun. A bug report for Linux TCP congestion control algorithms, May 2017. <https://patchwork.ozlabs.org/patch/767239/>.
- [39] W. Sun. A buggy behavior for Linux TCP Reno and HTCP, July 2017. Report <https://www.spinics.net/lists/netdev/msg444955.html>, Fix <https://patchwork.ozlabs.org/patch/797520/>.
- [40] W. Sun, L. Xu, and S. Elbaum. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Santa Barbara, CA, July 2017.
- [41] W. Sun, L. Xu, and S. Elbaum. Limitations of emulating realistic network environments for correctness testing of internet applications. In *Proceedings of IEEE ICC*, pages 1–6, Kansas City, MO, May 2018.
- [42] W. Sun, L. Xu, and S. Elbaum. Scalably testing congestion control algorithms of real-world TCP implementations. In *Proceedings of IEEE ICC*, pages 1–6, Kansas City, MO, May 2018.

- [43] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proceedings of IEEE INFOCOM*, Barcelona, Spain, April 2006.
- [44] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. Direct code execution: revisiting library OS architecture for reproducible network experiments. In *Proceedings of ACM CoNEXT*, Santa Barbara, CA, December 2013.
- [45] O. Udrea, C. Lumezanu, and J. Foster. Rule-based static analysis of network protocol implementation. In *Proceedings of USENIX Security Symposium*, Vancouver, Canada, July 2006.
- [46] M. Vu, L. Xu, S. Elbaum, W. Sun, and K. Qiao. Efficient systematic testing of network protocols with temporal uncertain events. In *Proceedings of IEEE INFOCOM*, Paris, France, April 2019.
- [47] F. Westphal. TCP: make undo_cwnd mandatory for congestion modules, November 2016. <https://www.mail-archive.com/netdev@vger.kernel.org/msg138481.html>.
- [48] K. Winstein and H. Balakrishnan. TCP ex machina: computer-generated congestion control. In *Proceedings of ACM SIGCOMM*, Hong Kong, China, August 2013.
- [49] F. Yan, J. Ma, G. Hill, D. Raghavan, R. Wahby, P. Levis, and K. Winstein. Pantheon: the training ground for Internet congestion-control research. In *Proceedings of USENIX ATC*, Boston, MA, July 2018.
- [50] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE Transactions on Networking*, 22(4):1311–1324, August 2014.
- [51] M. Zalewski. American Fuzzy Lop for network fuzzing. <https://github.com/jdbirdwell/afl>.
- [52] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Canea. A formally verified NAT. In *Proceedings of ACM SIGCOMM*, 2017.

Scaling Community Cellular Networks with CommunityCellularManager

Shaddi Hasan
UC Berkeley

Mary Claire Barela
University of the Philippines, Diliman

Matthew Johnson
University of Washington

Eric Brewer
UC Berkeley

Kurtis Heimerl
University of Washington

Hundreds of millions of people still live beyond the coverage of basic mobile connectivity, primarily in rural areas with low population density. Mobile network operators (MNOs) traditionally struggle to justify expansion into these rural areas due to the high infrastructure costs necessary to provide service. Community cellular networks, networks built “by and for” the people they serve, represent an alternative model that, to an extent, bypasses these business case limitations and enables sustainable rural coverage. Yet despite aligned economic incentives, real deployments of community cellular networks still face significant regulatory, commercial and technical challenges.

In this paper, we present CommunityCellularManager (CCM), a system for operating community cellular networks at scale. CCM enables multiple community networks to operate under the control of a single, multi-tenant controller and in partnership with a traditional MNO. CCM preserves flexibility for each community network to operate independently, while allowing the mobile network operator to safely make critical resources such as spectrum and phone numbers available to these networks. We evaluate CCM through a multi-year, large-scale community cellular network deployment in the Philippines in partnership with Globe, the largest MNO in the country, providing basic communication services to over 2,800 people in 17 communities without requiring changes to the existing regulatory framework, and using existing handsets. We demonstrate that CCM can support independent community networks with unique service offerings and operating models while providing a basic level of MNO-defined service. To our knowledge, this represents the largest deployment of community cellular networks to date.

1 Introduction

Despite the global expansion of mobile networks, millions still live without access to basic voice and SMS connectivity. A key reason for this is that most countries effectively allow only a small set of nation-scale actors to hold the necessary operating and spectrum licenses, as well as interconnec-

tion infrastructure, to provide commercial mobile service. In spite of their size, incumbent mobile network operators (MNOs) are capital constrained and struggle to justify investment in rural areas with marginal business cases when compared to more profitable and lower-risk urban markets.

Community cellular networking provides one alternative approach for bringing mobile connectivity to these underserved populations [2, 28]. Community networks, networks built “by and for” the people they serve in a community-centric, often cooperative, fashion [8, 52, 58], allow for creative localized schemes for sustainable operation that work even when covering low-income, remote, or sparse communities. The strength of the community network model is its ability to flexibly adapt to local needs using local resources and local insights. A network serving one community can provide services suited to that community, and can institute their own policies that would not make sense in a nation-scale network (for example, free local SMS mailing lists). They can also tailor their network deployments to leverage local capacity, such as existing towers and power systems, lowering deployment costs. Yet even where communities could run profitable networks, they face high barriers to entry in acquiring expensive and exclusive spectrum licenses, as well as operating licenses necessary to secure phone numbers and interconnection with the global phone network.

We are left with an unfortunate market failure: incumbent MNOs hold resources and capabilities, in the form of licenses and interconnect, necessary to provide rural coverage, but can’t justify investment in underserved areas. Concurrently, community networks could provide sustainable service, but lack a combination of the commercial, legal, and technological capacity to operate in a telecommunications regime designed for large carriers. We argue that rather than standing in opposition to each other, incumbent operators and community networks have an opportunity for cooperative interaction, leveraging the strengths of each to fulfill mutually beneficial business and social goals. Fundamentally, such a partnership requires a technology stack enabling collaborative *resource management* and *provisioning of com-*

mon services between MNOs and community networks.

In this paper, we present CommunityCellularManager (CCM), a solution for operating community cellular networks at scale within the existing telecom ecosystem. CCM enables multiple community networks to operate under the control of a single, multi-tenant controller, preserving flexibility for each community network to implement their own unique services while allowing them to share MNO resources, such as spectrum and phone numbers. For MNOs, CCM resolves the administrative problems they face granting many small third parties access to their core network by aggregating their traffic behind one logical point of interconnection, while providing a point of control to mitigate the risk they take on by sharing their resources and licenses. We evaluate CCM in a deployment to expand the service of an MNO into unserved areas via community networks using the MNO’s spectrum and interconnect (Section 5).

The contributions of this paper are as follows. First, we describe the design and implementation of CommunityCellularManager, a system for deploying rural community cellular networks at scale by facilitating cooperation and resource sharing with traditional mobile network operators. Second, we present a multi-year, large-scale community network deployment in the Philippines that provides basic communications service to 17 communities and over 2,800 monthly active users, powered by CCM and in partnership with Globe, the largest MNO in the country. We further demonstrate CCM’s ability to support key use cases necessary for rural community cellular networks, such as offline operation, as well as its ability to support independent local services on a subset of the community networks in our deployment.

2 Related Work

Rural Access. Access to connectivity supports numerous services, such as health [37], education [35], and finance [9]. However, access remains unequal, with connectivity significantly better in dense, urban, and developed environments than in sparse, rural, and developing ones. In attempts to resolve this disparity, researchers have proposed a variety of novel technologies such as long-distance WiFi [47], delay tolerant networking [16], and “sneaker” nets [22, 48, 24].

Although these technologies were all successes from a research perspective, they have had limited impact on the global problem of access. These technologies exist alongside an array of other important infrastructure, and innovations at the network layer alone will be insufficient. Surana et al. [56] find that looking at the system holistically is critical for its success [55]. This includes both physical infrastructure such as power and the social structures and people surrounding the technology. We take these lessons to heart in CCM, bringing issues like local ownership and customization to the forefront of the system design.

Community Networks. Researchers have explored wireless community networks [31] for years with contributions such as specific wireless technologies [18], topologies [59, 58], and mesh protocols [5] more appropriate for their unique constraints. The above systems all use WiFi for access, primarily because of low cost and use of unlicensed spectrum. While these networks have shown to have the potential to empower local communities [14] and create more resilient networks [8], this limits these networks to higher-end devices and small coverage areas without mobility.

More recently, researchers and practitioners have built community *cellular* networks [2, 28, 51, 12]. Ideologically aligned with the goals of community networking, these networks use cellular protocols to provide wide-area coverage to basic phones at lower cost. Unfortunately, they remain limited by the technical affordances of cellular, including the need for licensed spectrum, phone numbers, and interconnect with the global phone network. CCM provides a platform for community cellular that resolves these concerns.

Mobile cores. Cellular core network standards are developed by the 3GPP [46]; recent work focused on improving scalability and reliability of core networks [50, 6, 49]. Similar to SoftCell [33], CCM is a clean-slate architecture, and was designed for public cloud environments like ECHO [42]. CCM differs in its focus on shifting administrative boundaries of the mobile core to enable cooperation between MNO and community networks.

3 Design Goals

CCM is targeted to a specific but important use case: extending mobile network service provided by an incumbent MNO in a cooperative partnership with community cellular networks. Prior work [28, 51] has shown that community cellular networks can effectively serve areas unserved by MNOs. Such cooperative arrangements [11, 39] promise to unblock regulatory and commercial barriers faced by community cellular networks, such as access to spectrum. The architectural rigidity of mobile networks, however, makes achieving our federated use case challenging, since it provides no affordance for having multiple administrative domains providing service in a loosely coupled fashion.

The 3GPP specifications [46] define the architecture of traditional mobile networks. Although these specifications continue to evolve, the network architecture prioritizes (1) mobility, (2) operator control of subscribers and network policy, and (3) efficient utilization of radio resources as first-order priorities. To do this, 3GPP networks tunnel traffic between end-user devices and a (logically) centralized “core network”, which provides services such as voice calling, messaging, and data service. These logical services are offered over the physical radio access network (RAN), which is responsible for providing users with physical connectivity to the core network and ensuring efficient use of the limited

radio resource. The two components of the network – the core and the RAN – are tightly coupled: without a core network, the RAN is not capable of providing network services to end users. This approach stands in stark contrast to that of the Internet [10], with its focus on supporting datagram services among independent networks.

Both architectural approaches – 3GPP’s centralized control and the Internet’s decentralized flexibility – have clearly enabled successful networks, but the 3GPP architecture has several significant drawbacks for community networks in rural areas. First, the strength of community cellular networks is their adaptability to their local context, providing customized services that are relevant to users in the local community. Centralized core networks make a single entity responsible for all service provisioning, preventing innovation at the local level. Second, community cellular networks are heterogeneous and administratively decentralized, with diverse IP transport and transit configurations and different sets of actors responsible for network operation and maintenance at each site. Tightly coupling the RAN equipment deployed in a community to a core network requires coordination between the community network and the MNO’s core network. Fundamentally, this requires the MNO to expose their most critical network assets – their core network – to a third party, so they need ways to mitigate the risk of doing so. As a result, this interaction incurs high administrative overhead due to the configuration required for each new site and the extensive security reviews required when connecting third-party infrastructure to MNO core networks.

These challenges suggest the following design goals:

1. Enable community networks to *autonomously deploy services* within their footprint, while providing a basic set of services from the mobile network operator.
2. Minimize the cost and risk of *adding new cell sites and community networks* for both the MNO and the community network operator.
3. Provide service at *low incremental cost* beyond that required for RAN deployment, even in small networks.

4 System Design

To achieve these goals, CCM builds upon our experience building and operating an early community cellular network [28, 27]. We implemented the first version of the software that would evolve into CCM on location during our fieldwork in Papua, Indonesia, and the constraints of that environment – reliance on satellite Internet and reliably unreliable power – shaped our design.

In CCM, *each network node can provide service autonomously*, even without a connection to the Internet. CCM is broadly divided into two components (Figure 1). The first is the CCM Client, which is co-located with a community network’s RAN equipment and fills the role typically played by a 3GPP core network, terminating the logical connection

from a user’s phone and providing services to the end user. To support voice and SMS services, we translate all 3GPP voice and SMS to SIP and SMPP at the Client, which allows us to perform call and message routing at the Client (also known as *local breakout*). By making each network node a logically independent cellular network, CCM elegantly supports autonomous service deployment in each community network and allows core network capacity to scale up and down with the deployed RAN network.

The second component, the CCM Controller, provides a set of common services across community networks, such as RAN configuration and network management tools, including a web application that serves as the primary user interface for both MNO staff and community network operators. Importantly, the Controller also manages state distribution across the different networks: rather than centralizing both policy and network state in on-path devices, we synchronize application state among network nodes. The Controller also provides interconnection between community networks and the MNO network, regulating access to one of the most important resources the MNO owns: globally routable phone numbers for voice calls and SMS.

This decomposition bridges the centralized world of traditional mobile networks with the decentralized world of community networks and IP. Network management functions – subscriber provisioning, device management, network policy, and necessarily integration with Globe (Section 4.2.1) – are all highly *centralized*. In contrast, user traffic and signalling (the data and control planes, from a 3GPP perspective) and service provision are *decentralized* where traffic doesn’t interact with Globe. This hierarchy enables scale by allowing individual community networks to take advantage of common infrastructure while preserving their ability to deploy new services. It also reflects a pattern we have observed in other scaled (non-cellular) community networks, with “second level organizations” solving expensive or complicated problems once while more local groups operate network infrastructure [3].

CCM is designed for GSM (2G) networks and provides only voice and SMS service, not data (though we do rely on IP data for management, we do not provide data service to users’ phones), as the plurality of devices in our deployment only supports 2G [54]. Similarly, we emphasize that our particular decomposition of responsibilities represents just one point in a wider design space with its own political and technical tradeoffs. Section 7 discusses extensions toward LTE and further exploration of this design space.

4.1 CCM Client

The CCM Client is responsible for all operation at the community network site. It has two key responsibilities: provide connectivity services to users at the site, and manage the site itself. Physically, the CCM Client is a suite of software tools

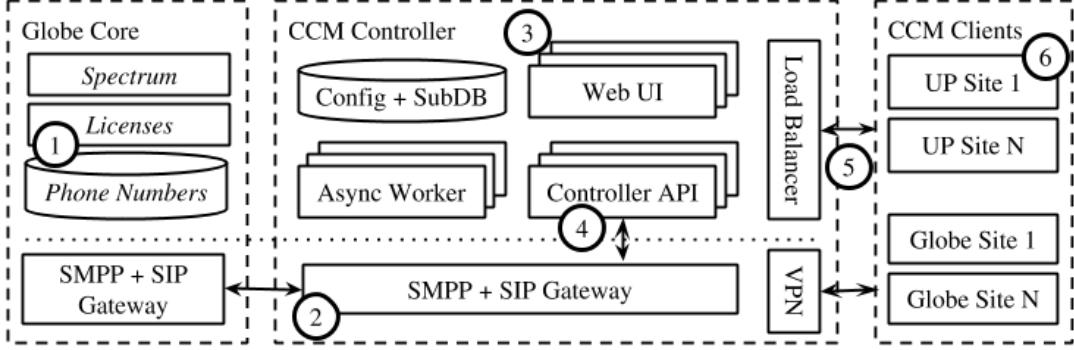


Figure 1: The CCM high level architecture, consisting of Clients, the Controller, and the interconnect with Globe. Phone numbers, licenses, and spectrum [1] are provided by Globe, with a number block manually allocated into the Controller’s subscriber database (SubDB). The CCM Controller [3] manages network configuration and status, provides a point of control for Globe to enforce global network policies and perform administrative tasks, and a web interface for network management. The CCM Controller also interconnects with Globe’s core for routing traffic between many individual community networks and Globe’s network [2]. Clients communicate with the Controller via an HTTPS API [5], which is also used to determine call and SMS routing [4]. Note that CCM utilizes local breakout at each component, so traffic may be handled as the SMPP/SIP gateways closest to the two users. CCM is multitenant, so Clients in many communities [6] can have unique local services and operate asynchronously if the backhaul is unavailable.

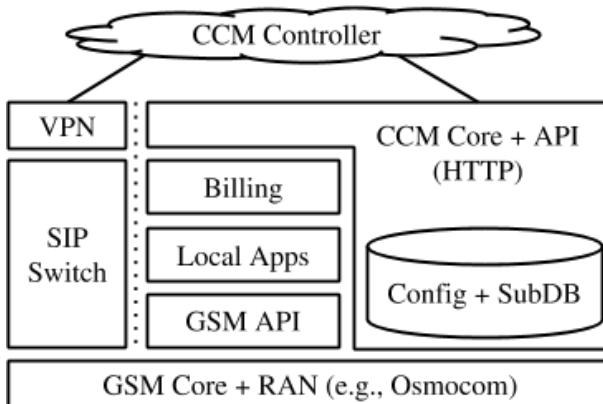


Figure 2: The CCM Client is deployed at each site and provides service even when the connection to the Internet is not available. The Client also provides an opportunity for customization of local web services and local applications.

that runs on a off-the-shelf x86 PC co-located with the radio equipment of a rural cellular site. In our deployments, this PC is either located physically within the RAN enclosure or located nearby in a separate enclosure. Figure 3 shows the components of one example install.

The Client is the primary element in our system responsible for delivering communication service (in our case, voice and SMS) to end users. We designed CCM to work with the two most popular open-source GSM network implementations, OpenBTS [44] and Osmocom [45], each of which implements a complete GSM core network and, importantly, translates cellular traffic and signalling to VOIP (SIP, RTP, and SMPP). CCM extends this translation and implements

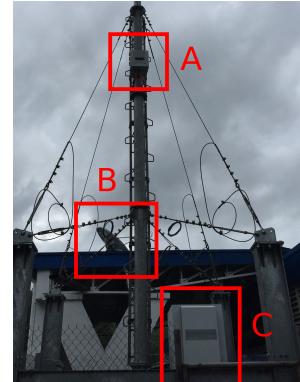


Figure 3: The CCM Client hardware components. (A) PC and GSM radio (in combined enclosure). (B) The satellite antenna used for backhaul and solar panels (not visible). (C) Closet for power system and satellite modem.

the business logic to properly route calls and SMS, implement local services, and provide basic business support services such as charging and user provisioning.

User management and billing. In GSM networks, the SIM card holds a unique IMSI number, the user’s identity on the network. Each IMSI corresponds to a particular subscriber’s profile in CCM, which associates both a phone number and an account balance with the subscriber. Similar to previous work [28], we provide basic SMS-based applications for provisioning a new SIM card (i.e., associating it with a subscriber profile and assigning a phone number) and transferring network credit between users. The network operator may add or remove credit directly to a subscriber’s account via the CCM Controller as well. CCM implements

a simple prepaid billing system in which operators can set the price of different network actions (e.g., price per minute for a voice call to a certain number prefix); time-based account balances are also supported. This user profile is one of the key elements of network state in our system, which we discuss further in Section 4.2.

Local breakout. We run a SIP softswitch [19] on each CCM Client to route calls and SMS. Because CCM leverages local breakout, this softswitch can fully route and connect local calls and messages without having any traffic (signalling or media) leaving the site. This gives CCM the ability to support arbitrary local services, including custom interactive voice response applications or SMS shortcodes. Any outbound traffic addressed to a user not currently active on a site is routed to the CCM Controller, which allows the MNO to set policy for these communications and potentially route them between different sites as well as to the global telephony network (Section 4.2.1).

Site provisioning. Community networks are built and operated by ordinary people, not specialized technical staff. As a result, we focused on automating as much of site provisioning and setup as possible: our goal was to make setup at least as easy as installing a home WiFi router. To provision a site, each CCM Client generates and assigns itself a unique identifier during a “manufacturing” phase; we expect that either the vendor or experts assembling the unit will perform this step. The unique identifier is printed on the outside of the hardware to be deployed at the site.

Devices start in an unprovisioned state; CCM prevents the radio equipment from transmitting while unprovisioned. To provision the device, installation technicians or community members register the device by inputting the unique identifier into the CCM Controller’s web interface. Once the device is powered on and connected to the Internet, it connects to the CCM Controller to provision itself. If the device has been registered with the CCM Controller, the device will receive VPN configuration information and a secret API token for future requests to the CCM Controller.¹ Using the API token, the site can begin a process of generating secondary credentials required for operation, such as a VPN keypair. Once the Client establishes secure connectivity to the Controller, configuration and network state information is synchronized and the device is ready to begin operation.

Under nominal conditions, the only user interaction re-

¹The gap between when a device is registered with the CCM Controller and when it actually connects could allow an attacker with knowledge of the unique ID to register a fake device with the CCM Controller. To mitigate this, we instruct users to register the device only after they’ve verified it is powered on and connected to the Internet. In practice, this is done in a “staging” lab environment prior to field installation, thus minimizing the window of opportunity for an attacker. This attack could be completely mitigated if the device-specific secret API tokens or a keypair were pre-provisioned with a CCM Controller prior to associating a device with a user’s account; however, doing this would require coordination between device manufacturers and the entity operating the CCM Controller, which was not feasible in our deployment.



Figure 4: A screenshot of the CCM web UI.

quired to provision a site is entering the unique ID into the web UI (see Figure 4). No prior coordination is required between equipment providers, the CCM Controller operator, and the community network team actually deploying the site. This is in contrast to traditional 2G networks, which typically require RAN, core, and network management software from the same vendor to be manually provisioned by one entity.

4.2 CCM Controller

The CCM Controller is responsible for managing traffic among the collection of community networks it manages, as well as with the outside world. Moreover, while in practice CCM supports geographically distinct cell sites without overlapping coverage, users can move between networks and CCM must be able to coordinate state across networks; the Controller also handles this task.

4.2.1 Voice and SMS Interconnection

Internally, CCM routes all calls and SMS between SIP switches at each CCM Client site. Within the administrative domain of a single CCM Controller, we can assign phone numbers to subscribers much like an enterprise can assign extensions to internal users or RFC1918 addresses for private networks. However, for users of CCM networks to make and receive communication from the global phone network, we need to interconnect CCM sites with an entity that has been assigned globally-routeable phone numbers and has the ability to route voice and SMS communication. CCM is capable of interconnecting with wholesale VOIP providers (such as Nexmo [40] or Bandwidth [4]) as well as an MNO’s VOIP infrastructure; we will focus on the latter in this work, though the mechanism is similar for both.

CCM interconnects with an MNO through its VOIP gateway (typically used for supporting enterprise customers) as well as the MNO’s SMPP gateway, an industry standard for SMS exchange. In this arrangement, the MNO allocates the CCM Controller blocks of phone numbers to be used exclusively by users of CCM sites. On the CCM side, we run a

corresponding SIP switch and SMPP gateway, as well as any VPN infrastructure required for connecting to the MNO’s systems. The CCM SIP and SMPP infrastructure is stateless, and determines inbound call routing by querying the CCM Controller API to determine subscriber location; all billing and charging is handled on the CCM Client.

The integration architecture we use has two consequences. First, since the CCM Controller handles multi-tenancy, there is a single point of interconnection between the MNO and potentially many community networks. This is crucial: setting up our initial integration with Globe took just over four months from start to finish, and required the team we were working with at Globe to obtain approvals and request configuration changes with a number of different teams within their organization. Further, the technical work required to set up this interconnection, while simple in theory, is complicated by the fact that MNOs rely on custom and legacy systems that must be carefully managed to prevent downtime. Going through this process for dozens of community networks would be impossible, so while our integration approach creates a single point of failure, the benefits of a one-time integration outweigh the risks.

Second, this tightly ties CCM networks to the MNO who assigns phone numbers: the MNO can at any point shut off service to these users; users would need new phone numbers if the partner MNO changed. This is advantageous from the perspective of the MNO, since they retain ultimate control over their users, which was important for gaining approval from Globe to allow community networks to operate under their spectrum and operating licenses. While other approaches [25, 17, 51] enable community cellular without hard dependencies on third parties, our MNO partnership approach required no regulatory change; the project would not have been feasible in the Philippines in the timeframe we’ve taken otherwise.

4.2.2 State Management

System state in CCM consists of per-site configuration, network policy, and subscriber authentication, billing and location data. In order to support disconnected operation, CCM needs to ensure this state is distributed across all sites administered by a CCM Controller.

Checkin. The fundamental mechanism for state distribution in CCM is the “checkin”. At least once per minute, each CCM Client sends a HTTPS request to an API endpoint on the Controller. The content of this request includes the site ID, usage logs (also known as call data records, or CDRs), diagnostic information (such as CPU utilization), and subscriber data to the Controller. Based on the computed configuration for the site, the Controller’s response to this checkin request is the desired state of the system’s configuration and the set of subscribers a site should be able to serve.

Both the request and response are JSON dictionaries di-

vided into independent sections: for example, billing records are stored in the `usage` section. Only changed state is transferred in the server response after the initial checkin. The Client and Controller each maintain a shared context of previously received configuration by tracking a hash of the contents of each checkin section; the Client includes the last hash of each section it has received in its subsequent requests. If the Client hash matches the last recorded Controller hash, it only sends the difference between the last state and the current one. Otherwise, it sends the full contents of the section. Because most configuration changes rarely, this minimizes received checkin response sizes.

Subscriber data. Subscriber data consists of authentication information, location (i.e., which Client a user is currently attached to), and billing information. The first two of these are straightforward to handle. Authentication information consists of SIM card keys as well as whether a subscriber is allowed or not allowed on the network; the network operator can add users via the Controller or through an SMS-based short code application in the field. In both cases, this information is directly written to the Controller and replicated to all sites in the network. Each CCM Client reports the list of subscribers currently attached to the site during checkin; this allows the Controller to have a global view of subscriber location for routing inbound traffic and traffic between sites.

Billing state is more complicated, as it can be mutated both at any particular site (e.g., decremented after a subscriber makes a call) as well as at the CCM Controller (e.g., when an operator adds credit to a subscriber’s account directly). A site may also be disconnected from the CCM Controller for arbitrary lengths of time, while still providing communications services within its coverage area. Nevertheless, we need to be able to apply network policies around charging to local calls while disconnected from the tower.

To achieve this, we restrict modifications to subscriber balance to commutative operations (add or subtract), and then represent subscriber balances as a CRDT [38]. Subscriber balances are synchronized across the network during the checkin process: as each site checks in, the Controller merges its subscriber balance state with what the site reports, and provides the merged state back to the site in the response if it differs. This enables each node of the system to both read and write subscriber balance independently – when a site is disconnected, users can continue to communicate locally and transfer balance between each other, and balance will converge once the site comes back online. Users can also move between multiple offline sites and communicate locally on each. In our current implementation, this raises the possibility of double spending, but our partners made the business decision to allow this to facilitate usage. The risk of this particular attack is low in practice because of the substantial distances between different sites, and could be further mitigated by setting a threshold for spending during

disconnected operation at each site.

Site Management All other configuration state is similarly managed by the Controller. We provide a web-based UI for viewing network-wide state (including user activity) and defining network policy and configuration. While some information must be manually defined by the network operator (such as pricing plans), other information is automatically generated to simplify operation (such as the radio channel that each site uses). Outside this controller-based interface, we do not support any other means of configuring devices.²

The UI provides fine-grained permissions and control, allowing the operator of the Controller (Globe) to determine which types of users may access different aspects of the UI. In our deployment, spectrum usage is controlled by Globe, and access to Globe sites is restricted to their staff. Globe has allocated a separate “network” for a set of sites run directly by communities, but has restricts what administrators of these networks can do. The superuser has view and edit access to change configuration across sites, networks, and subscribers, as well as the ability to adjust credit and download activity reports. The superuser can also create additional users based on predefined roles: business analyst (view-only), partner (view-only + manage subscriber) and loader (view-only + manage subscriber + adjust credit). The system allows Globe to adjust specific permissions per user.

5 Deployments

We deployed CCM as part of a long-term partnership between the researchers, the University of the Philippines and Globe Telecom [20], the largest MNO in the Philippines with over sixty million subscribers. The researcher-driven portion of the project, conducted in the Aurora province, is funded by the Philippine Government and focuses on bringing the benefits of community cellular to remote parts of the Philippines; we refer to these as the “UP sites”. The MNO created two different administrative domains for their subnetworks, one owned entirely by Globe and another for the UP sites. For both of these networks, users are using phone numbers from Globe, though the service is branded separately from their main service bundles to make clear to users that the services and quality expectations on these extremely rural sites are different from Globe’s main network. In this section we describe each of these subnetworks including their pricing, interconnect, and context. For site deployment dates and locations, please see the Appendix.

5.1 UP Community Networks

UP selected the province of Aurora for their CCM installations. Aurora is a coastal province of Luzon, the main island

²Each Client site does have a command line interface, but this is only used for debugging or emergency recovery purposes.



Figure 5: Components of a typical UP CCN installation: solar panels (top left), battery bank and network hardware (middle left), VSAT antenna (bottom left), outdoor GSM radio system, tower and shelter (right).

of the Philippines, on the Pacific Ocean and the Philippine Sea. The capital, Baler, is a relatively affluent town with readily available connectivity and robust land routes to urban areas such as Metro Manila. Further south down the coast, dozens of fishing communities lie outside of any existing connectivity. In these areas, people travel by boat to Baler for any needs that cannot be met in their home communities, including device repair and connectivity. Despite the distance from coverage, initial site surveys done in 2016 found that mobile phones were prevalent and demonstrated demand by uncovering a variety of existing (though complex) connectivity solutions [7]. These communities were selected by the UP team for (1) their relative proximity to Metro Manila, (2) lack of connectivity, and (3) connection to a local university for assistance in deployments. These deployments began in late 2017, with all seven planned sites launched as of February 2019. The community GSM sites provide voice and SMS access to over 1,500 subscribers.

Structure and Context. The Aurora networks are organized in line with most definitions of community networks [58]. The operation and maintenance of the community network is handled by local cooperatives, in partnership with the local government units (LGUs), the local state college and the UP research team. These partners were identified during the initial site surveys. The LGU facilitated the legal appropriation of land for towers and helped expedite the permits and clearances required for civil works at each site. In addition, the LGUs extended assistance to mobilize local labor for installation and deployment activities.

The cooperatives conduct the day-to-day operations, business management, and maintenance of the network. They are in charge of pre-paid credit distribution from Globe to the community retailers, who are mostly existing *sari-sari* (general merchandise) store owners. The cooperative orders credit from Globe at least once a month. After payment has been sent through a bank deposit and verified, Globe tops

up the cooperative’s accredited mobile number via the CCM Controller web interface. From here, the cooperative distributes it to their authorized retailers via a locally-hosted SMS-based credit transfer application. The cooperative receives a wholesale discount from Globe, part of which is passed on to the retailers. The retailers earn additional income by charging a small “convenience” fee per sale. Finally, the gross revenue from all charged calls and texts on the network is split based on a revenue sharing scheme between Globe and the cooperative, where the cooperative gets 80% and 20% goes to Globe. Earnings are used by the cooperative to pay personnel and as savings.

In terms of maintenance, the researchers employ a three-tier support system. The first tier, L1, is composed of local maintenance personnel hired by the cooperative and residing in the community. They are assigned to do daily upkeep and basic troubleshooting of the cell site. Any issues that are not resolved at the L1 tier are escalated to the local state college and/or the LGU, the L2 tier, which can provide intermediate-level technical assistance. Finally all other issues that are not resolved at the L1 or L2 tiers are escalated to the research team, which provides L3 support.

Prior to launch, the researchers facilitated social enterprise training sessions with cooperatives that had no prior experience in conducting business-related activities pertaining to operations of a community network, and technical training with community maintenance personnel. During the network launch, the researchers held a forum with the broader community where the unique properties of their networks were explained (e.g., lack of roaming to Globe’s main networks) to the subscribers. The event was also a venue for the community to raise questions and concerns regarding the network and services, and distribute SIM cards. The distribution of SIMs in the UP networks is currently tightly controlled as part of an ongoing randomized control trial [15] on the impact of cellular networks on rural communities.

5.2 Globe’s Community Networks

Concurrent to the installation of the UP sites, Globe also installed CCM Client access points in eleven other rural communities throughout the Philippines. Of these, the first two were “proof of concept” (PoC) sites installed in Tanay, Rizal province (60km from Manila) and another in Talisay, Quezon province (130km from Manila). Following the successful trials of CCM in these two PoC sites, a further eight networks were deployed in Eastern Visayas. All of the sites are rural and lack any existing network coverage, with populations ranging from one to five thousand people. Though census information for rural areas in the Philippines is spotty, the demographics and economics of Tanay have been described in detail in other work [54]. Another twenty networks are planned with rollouts expected throughout 2019.

Structure and Context. The MNO’s sites consist of two

groups: the PoC sites in Luzon and partner sites in Visayas. Both are organized in a more traditional fashion with Globe handling marketing, credit distribution, and installation. For the PoC sites, Globe agents conducted selection interacting with the LGUs to procure locations and timings amenable to the local community. For the Visayas sites, Globe instead partnered with a non-governmental organization (NGO) to find partner communities and negotiate installations. This NGO has a long history of projects in Visayas and was able to find suitable communities in rural areas as well as assist in the day-to-day operations of the sites. The NGO also handles the SIM card and pre-paid credit distribution from Globe to the community retailers.

The installations were done by Globe’s tower deployment team using their standard site equipment, aside from the custom RAN equipment itself. The sites use a two-tiered maintenance system with one level of lightweight local support and the main support provided by Globe staff in Manila. One local community member is selected by the deployment team and tasked with using the network to send messages back to Globe’s technical team in case of ongoing issues. If any failures disable the network, Globe’s engineers use the CCM Controller’s web interface to observe the network failures and send engineering staff out to resolve the issue. Credit sellers buy their credits by depositing money into Globe’s account at a nearby (but outside of the community) bank. They then take a small premium when distributing these credits throughout the community.

5.3 Project Evolution

Rural connectivity projects do not lend themselves to neat stories, and this deployment was no exception. Our deployment of CCM originated from discussions between Globe and the researchers’ company, Endaga, in early 2015, and implementation work began in the summer of 2015. Originally, Endaga operated both the CCM Controller and directly supported all field equipment running the CCM Client software. After Endaga joined Facebook in late 2015, CCM was released by Facebook as an open source project to enable continued development by Globe and others. At the same time, we interconnected the CCM Controller with Globe’s core and then transferred administrative control of the CCM Controller to Globe in mid-2017 after a successful early proof-of-concept. While the researchers continued to be involved in the development and aspects of the deployment of CCM, day-to-day operation of the service passed to Globe; this continues today. In practice, this means that Globe controlled access to the CCM Controller (including software updates), and the researchers could only deploy local services on the sites we directly had access to: the UP sites. While this wasn’t the deployment configuration we originally designed CCM for, CCM was able to continue providing an interconnection abstraction towards different groups of com-

munity networks, even while the administrative boundaries between system elements shifted.

6 Evaluation

Our team completed interconnection between CCM and Globe’s network in mid-2016, with two trial sites launched in early 2017. After a year of evaluation, Globe launched nine additional sites and granted permission to the UP to launch seven community networks that year (launch dates are provided in the appendix), of which six are live as of December 2018. While a total of 17 sites have been launched throughout the project, only 9 were in operation as of December 2018 due to hardware failures; the failed sites are being replaced with new hardware that is currently undergoing testing. The UP sites used a mix of hardware (including some self-assembled), and their deployments have not been impacted as severely as Globe’s sites, which used a single vendor for their deployments. Supporting heterogeneous hardware was not an explicit goal of CCM, but its ability to do so proved useful in our challenging rural context.

6.1 Usage

As of December 2018, CCM supports a total of about 2,800 monthly active users across 17 launched sites (Figure 6). Spikes in the number of active users are due to site launches, which are typically accompanied by marketing campaigns to raise awareness for the launch of service in a new town.

Table 1 provides an overview of traffic volumes across the network. Across all sites, we observed that inbound call traffic is much more common than outbound call traffic, and is in fact the predominant form of usage on both networks. This is indicative of “call-me” or “flashing” [13] behavior as subscribers are aware that they can save money by letting their contacts call them instead; on all networks in our deployment, subscribers are only charged for user-initiated calls and SMS. Narrowing our analysis to only user-initiated traffic, we find that SMS is the predominant form of communication, with roughly 7x more messages sent than minutes spent on calls, and over 13x the number of calls made. This is in line with the fact that SMS rates are significantly lower than per-minute rates and extremely popular in the Philippines (the “texting capital of the world” [23]).

We observed more local traffic among the UP sites than the Globe sites. Interviews conducted by the team suggest that one contributing factor is the fact that communities served by the UP sites can be clustered into two groups where the sites are located relatively near each other, and because the community networks support existing locally relevant services. In the UP sites in San Luis, Aurora, locals frequently conduct trade activities and have personal connections with residents from the other sites, relationships which existed before the arrival of the community cellular network.

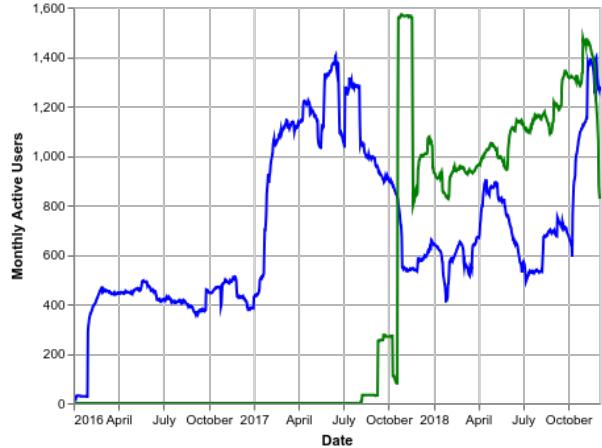


Figure 6: Monthly active users for Globe (blue) and UP (green) networks.

Service Type	Volume		
	UP	Globe	Total
Calls Out (min)	55,459	114,294	169,754
Calls In (min)	1,128,849	1,886,642	3,015,492
SMS Out	440,767	713,017	1,153,784
SMS In	367,212	701,538	1,068,750

Table 1: Volume of usage by service type. Call volumes are reported as duration in minutes, while SMS volume is number of messages sent. Inbound communication is free.

In another example, the only high school in the area is located in Dikapinisan and students from the nearby sites need to relocate temporarily for their studies. Parents used the community cellular network to call their children and get updated on other current events. We note though that for these “inter-cove” (the sites are located in a series of coves) transactions, although they are classified as local traffic for billing purposes, these communications are routed through the Controller’s SIP switch. Community members also told our team that they used the network for local events, such as a community beauty contest that used SMS for voting.

6.2 Disconnected Operation

Figure 7 depicts the launch date and uptime for each site. Downtime was a common case in our extremely rural sites. All sites use satellite backhaul, and even where grid power is available, its poor reliability necessitates battery backups or fully off-grid solar systems; some sites were also turned off on a nightly basis to conserve power. Overall, the mean site uptime is only 35% across all sites, with a median of 27% for

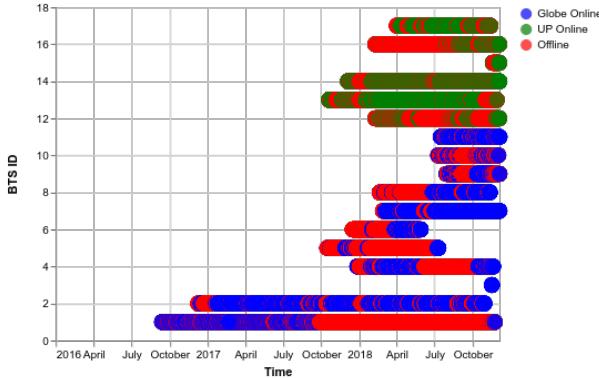


Figure 7: Site uptime. Sites 1-11 are Globe operated, while 12-17 are UP operated. Each point is one hour.

Globe sites and 40% for UP sites.³ This reality motivated our need to prioritize disconnection and downtime as common cases to be handled by CCM.

To understand whether users benefited from CCM’s disconnected operation support, we examined CDRs from time periods while sites were offline. We only considered user-initiated “local” communications and credit transfers, since communication out of a site is not possible while it does not have a connection to the CCM Controller. This is still a conservative assumption, since communication between two sites in the same network (such as between two different UP sites) is also considered “local”. We include credit transfer as well since it is a crucial utility used by all users of the network to buy service: if credit transfer breaks, sales are directly impacted, hurting network sustainability.

We find that overall, offline local traffic accounts for 16% of the total local traffic, comprising 7% of local traffic in the MNO sites and 23% of local traffic in the UP sites. This accounts for 4,678 minutes of calls, 81,434 SMS messages, and 11,970 credit transfers representing approximately US\$4,600 worth of transfer activity. These credit transfers are largely top-up sales to end users and themselves represent 14% of the transfer volume across all sites, benefiting Globe, retailers, and the community networks at large. Without effective support for offline operation, these sites would have been completely down during backhaul outages, further complicating the already difficult business and sustainability cases for these rural network sites.

The primary cost CCM incurs for supporting offline operation is the overhead of state synchronization, carried by the checkin protocol. Figure 8 shows the distribution of checkin request and response sizes before and after our optimization. The median checkin request and response is 2.6kB and 283B, respectively, corresponding to median “unoptimized” request and response sizes of 14.4kB and 283B. Our optimization is much more effective at the tails: 99th percentile request and responses sizes are reduced by 86.3% and 91.2%

³Not all sites are unreliable. Three UP and two Globe sites achieve >60% uptime, even despite planned nightly downtime to conserve power.

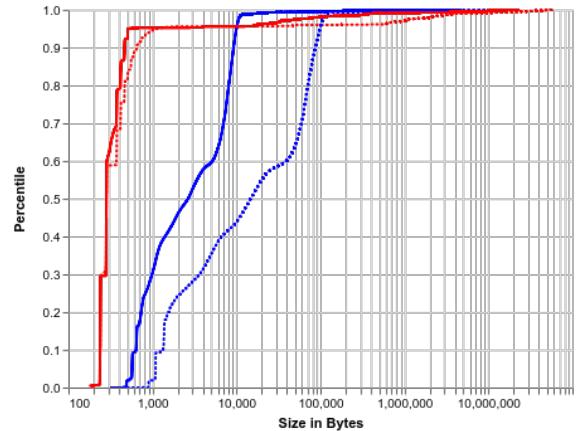


Figure 8: CDF of checkin sizes. Blue are requests from the Client to the Controller, and red are responses from the Controller to the Client. Dashed lines are raw data sizes and solid lines are size after optimization for transport.

respectively. The optimized checkins place minimal burden on our networks: even at the 99th percentile, a checkin consumes less bandwidth than a single 6 second call (using a 64kbps codec). We note that these sizes are prior to *gzip* compression over the wire, further reducing overhead.

6.3 Local Services

A consequence of CCM’s decentralized architecture, and one that differentiates it from traditional centralized cellular networks, is that it allows customization of the individual networks to local needs, requirements, and desires. During our deployment, we and our partners took advantage of this capability to implement a number of unique services in the UP sites. Specifically, we implemented (1) a local repair support tool aiming to empower lay actors from within the community to conduct routine maintenance and repair, (2) a custom local billing solution to allow our team to explore the demand curves for rural communication access without requiring costly changes to Globe’s billing systems, and (3) a local “outage hotline” connecting users directly to the UP team. These applications demonstrate CCM’s ability to support flexible and dynamic community networks.

Repair. Repair is central to sustainable rural networking interventions [56, 29]. Further, in rural areas, we have the unique advantage that rural users are natural repairers [30]. While Globe installations use traditional maintenance practices, the UP community networks sought to address network repair and maintenance through a local-only repair support service that leveraged latent skills and abilities present in the community [32]. The researchers implemented a set of services leveraging the fact that community networks *interact directly with users* and can help guide them in conducting repair. The service consisted of digital “repair manuals” embedded into the community cellular infrastructure. Network

components are labeled with small codes (e.g., “ANT” for antenna) and an SMS shortcode (e.g., 777) that, when texted, provides background and debugging information about that particular system element. Labels are printed on a large poster in a community building near the tower. For instance, a user texting “hot” to the shortcode will receive a message instructing them to turn off the system to allow it to cool. This system was implemented on the Client’s local softswitch (see Figure 2), and provides a unique, local mechanism for repair that is particularly appropriate for the UP sites, which don’t have dedicated commercial technicians.

Billing. The UP sites also have unique requirements for their billing system. Though CCM handles tariffs between Globe and the community network, the community sets their prices within their domain for their users. For the UP systems, the billing system was extended to enable *promos*, a well-known local pricing scheme among Filipino MNOs. Unlike the per-minute and per-SMS fees charged by default, users pay up-front for a set of network functions, usually a discrete volume of SMS and voice minutes. These services were implemented in CCM as Client-only databases storing current promo offerings and promo counts for each subscriber. The implementation also allows the administrator to grant promos to subscribers, similar to a rewards system.

Promos exist at a per-site level and are not synchronized across sites; quotas or discounted tariffs are stored locally. These are defined when the network administrator creates the promos via a web form or a CSV file upload. The system supports multiple promos per user. It also offers a SMS interface for users to check their promo status and usage.

Outage Hotline. The UP sites also offer an SMS-based outage hotline. This free service accepts questions, comments, reports and other service-related inquiries from the communities. The hotline logs received messages to a file on the CCM Client’s storage, which are synchronized to a remote server (distinct from the Controller) at regular intervals. The hotline also helps the researchers in the detection of technical issues in the field as community members can easily report any problems that they may experience on the ground. This is also the mechanism that local maintenance personnel utilize to send network status updates.

6.4 Reliability

Network downtime is common in our deployment. According to one user at the start of our deployment, “I’m OK even if [the network] is turned off at night. At least now, we have something that we can use to communicate. Unlike before, we totally have none.” This attitude changed during the course of the deployment; despite the utility of disconnected operation, users still came to expect continuous, reliable operation from these networks: “Why don’t we have signal during the night? They shouldn’t turn it off during the night, because it is still important in case something bad

Cause	Percentage	Example
Backhaul	42%	VSAT offline due to weather
Power	24%	Discharged batteries
Site Hardware	21%	Overheating
Site RF	12%	Broken antenna cable

Table 2: UP site outages by duration. This data comes from manual record keeping by site caretakers and is approximate; we do not have ground truth for all site outages.

happens here.” In our deployments, outages occurred at the boundaries of the network’s physical infrastructure: power systems, backhaul networks, and site hardware. While we were able to mitigate these to an extent, we regard reliability in community cellular networks as an open challenge.

All sites use satellite backhaul, and unreliable or non-existent grid power requires use of battery backups or off-grid solar systems. While we do not have ground truth for all site outages, outage records from the UP sites provide an approximate distribution of outage causes (Table 2). At these sites, the most common cause of outage was backhaul failure, followed by power outages, collectively accounting for about 66% of the downtime, with hardware or RF issues accounting for the remainder. Backhaul failures had minor impact since they tended to be transient and since CCM gracefully handles offline operation, but power failures were more serious, often resulting in extended downtime. Improving resilience to power failures is hard, as increasing battery capacity adds to site cost; approaches like “virtual coverage” [26] address this challenge to an extent, but require non-standard hardware. To reduce long-term outages, sites are shut down at night to reduce power utilization during off-peak hours, either manually or via an automatic switch.

All significant offline periods at Globe sites (Figure 7) were due to hardware failure; this impacted 9 of 11 launched sites. When Globe took on operation of the CCM Controller, they selected a new hardware vendor to provide equipment for Globe sites. This vendor included the CCM Client into their existing mature rural base station product, which promised to allow Globe sites to use proven hardware already being produced at scale, reducing cost. This hardware used an SD card as a disk, which was used in read-only mode in the vendor’s standard product offering; this is a best practice for rural networking hardware [56]. To support CCM, which requires mutable persistent storage to support offline operation and local applications, the vendor continued to use the SD card, but in a writeable mode. This led to a pernicious failure mode where devices would fail after several months’ successful operation in the field due to disk corruption driven by power failures and excessive writes. Community networks benefit from applications like CCM and those outlined in Section 6.3 that require mutable persistent storage, but supporting these applications in rural environments remains a challenge.

7 Discussion

Towards LTE. The rapidly maturing ecosystem of open source telecom software and newly-affordable RAN equipment designed for rural cellular networks makes community cellular networks technically and financially feasible. As noted, CCM only supports GSM (2G) service because basic GSM-only phones are still the plurality in the rural areas we target [54]. However, CCM’s architecture represents a framework for MNOs and community networks to partner and share resources. We expect the shift towards LTE (4G) to make community LTE networks viable in the coming years and we are actively pursuing this agenda [53].

The shift to LTE creates new opportunities for community networks. While GSM networks are essentially deployed in four spectrum bands globally, LTE devices support dozens of band combinations, some of which support spectrum sharing (such as Band 48 [17]) and others that fall within “digital dividend” bands (e.g., Band 71); the latter is ideal for rural networks due to improved propagation of low band spectrum. Even without shared spectrum, demand for LTE spectrum for capacity in urban areas is likely to result in substantial underutilization in rural geographies, opening up opportunities for partnership and spectrum re-use as we’ve done in our deployment. Driven by use cases like private LTE networks for enterprises and industrial “Internet of Things” deployments, as well as applications like mobile edge computing, the LTE base station ecosystem is already larger than that for GSM, lowering costs for community networks. Radio equipment of varying degrees of openness is under development [57, 1], and multiple open source software core networks, analogous to OpenBTS and Osmocom, are available [41, 21, 43].

Given this progress, community LTE networks could provide broadband service independently of MNOs, connecting to the Internet just as a small ISP would. Recognizing this, we recently proposed a *decentralized* LTE architecture [34] that does not require a telecom’s (or any centralized organization’s) participation. We expect a fully distributed architecture should ease deployment and empower community networks at the cost of increased difficulty scaling. Exploration of the political and technical tradeoffs across the centralized/decentralized and telecom partnership/independent network design spaces remain open research questions.

Fault diagnosis. Community cellular networks rely on a collection of systems to provide service to end users; just to send an SMS, a user’s traffic interacts with a radio implementation (proprietary hardware, or a software radio), a GSM stack (OpenBTS or Osmocom), a SIP engine (Freeswitch [19]), an SMPP gateway (Kannel [36]), and of course CCM. Even for experts, diagnosing faults in the mobile stack is challenging due to the need to manage state across layers and components. We relied on regular automated end-to-end health checks of the CCM Client to identify and rectify faults; automated failure diagnosis will be

essential for community networks.

Sustainability. As of today, only a few of the UP sites (and none of the MNO sites) are financially sustainable, taking into account the ongoing operating costs for the sites. This was expected for the unsustainable UP sites, as they were too small to provide enough revenue. We are hopeful that future research endeavors into novel business models and cost structures will resolve this issue. The MNO sites were designed to all be sustainable but poor system reliability hurt usage. We remain hopeful that stability improvement will increase revenue to sustainable levels.

Repeatability. One of our broader objectives is to develop a scalable and repeatable model for community cellular networks. We were fortunate that our project did not require any significant regulatory changes and that spectrum sublicensing was permissible under existing Filipino regulations. This is not always the case, and where MNOs are unable to allow third parties to use their spectrum, they may not be able to engage in this particular business model. Nevertheless, CCM reduces the challenge of starting many community networks to a *commercial* negotiation rather than a *regulatory* discussion (or even *legislative* action), often a much lower bar.

8 Conclusion

For the millions who live outside of basic mobile coverage, community cellular networks present a way to sustainable mobile coverage. Finding ways to remove the regulatory and commercial barriers to their growth is key to realizing this potential. Working with MNOs provides a straightforward, if not necessarily easy, path to doing this that requires neither major regulatory changes in many jurisdictions nor any of the actors involved to act contrary to their first order interests. The challenge that remains is building the platform to connect these different entities together.

Our work shows that these challenges are surmountable, and provides an example of ways to deploy community networks at scale. We identified critical design goals for such a system – autonomous services, minimal transaction costs and risk, and minimal absolute deployment costs – and implemented CCM to realize these goals. Through a large-scale deployment, we demonstrate CCM’s ability to effectively support these community network - MNO partnerships, connecting 17 communities and thousands of users. Our system is open source,⁴ and we hope others will find it useful for replicating this model.

Acknowledgments

We thank the anonymous reviewers and our shepherd Keith Winstein for their helpful comments which improved the quality of this work. CCM was developed over many years at

⁴<https://github.com/co-cell/ccm>

Endaga, Facebook, UC Berkeley, University of Washington, and University of the Philippines, Diliman. We are grateful to the many individuals who contributed to CCM over the course of its development, especially Omar Ramadan, Matt Ball, Steve Muir, Evgeniy Makeev. This work was supported by funding from Endaga, Facebook, the Philippines Commission on Higher Education's PCARI program, USAID, and the NSF GRFP under Grant No. DGE-1762114. We also thank the UP Diliman VBTS team, notably Philip Martinez, Ronel Vincent Vistal, Maria Theresa Perez, Maria Theresa Cunanan, Giselle Dela Cruz, Clarisse Aquino, Luigi Morata, and Joshua Dalmacio for their tireless dedication in the field, and anonymous individuals in communities throughout the Philippines who worked to make this deployment possible. Finally, we thank the Konekt team at Globe for their willingness to try something new and their commitment to expanding rural connectivity in the Philippines.

References

- [1] ALI, K., AND HEIMERL, K. Designing Sustainable Rural Infrastructure Through the Lens of OpenCellular. *Communications of the ACM* 61, 8 (2018), 22–25.
- [2] ANAND, A., PEJOVIC, V., BELDING, E. M., AND JOHNSON, D. L. VillageCell: Cost Effective Cellular Connectivity In Rural Areas. In *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development* (Atlanta, Georgia, 2012), ICTD '12, ACM, pp. 180–189.
- [3] BAIG, R., ROCA, R., FREITAG, F., AND NAVARRO, L. Guifi.net, a Crowdsourced Network Infrastructure Held in Common. *Computer Networks* 90 (2015), 150–165.
- [4] BANDWIDTH. <https://www.bandwidth.com>. Retrieved 9/2019.
- [5] BANDYOPADHYAY, S., HASUIKE, K., HORISAWA, S., AND TAWARA, S. An Adaptive MAC Protocol for Wireless Ad Hoc Community Network (WACNet) using Electronically Steerable Passive Array Radiator Antenna. In *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE* (2001), vol. 5, IEEE, pp. 2896–2900.
- [6] BANERJEE, A., MAHINDRA, R., SUNDARESAN, K., KASERA, S., VAN DER MERWE, K., AND RANGARAJAN, S. Scaling the LTE Control-Plane for Future Mobile Access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (2015), ACM, p. 19.
- [7] BARELA, M. C., BLANCO, M. S., MARTINEZ, P., PURISIMA, M. C., HEIMERL, K., PODOLSKY, M., BREWER, E., AND FESTIN, C. A. Towards Building a Community Cellular Network in the Philippines: Initial Site Survey Observations. In *Proceedings of the Eighth International Conference on Information and Communication Technologies and Development* (2016), ACM, p. 55.
- [8] BELUR, S. B., KHATURIA, M., AND RAO, N. P. Community-led Networks for Sustainable Rural Broadband in India: the Case of Gram Marg. In *Community Networks: the Internet by the People, for the People*. Association for Progressive Communications, 2017, p. 193.
- [9] BLUMENSTOCK, J. E., CALLEN, M., GHANI, T., AND KOEPKE, L. Promises and Pitfalls of Mobile Money in Afghanistan: Evidence from a Randomized Control Trial. In *Conference Proceedings of the Seventh International Conference on Information and Communication Technologies and Development* (NY, USA, 2015), New York.
- [10] CLARK, D. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review* 18, 4 (1988), 106–114.
- [11] CONGRESO DE LA REPUBLICA DEL PERU. Ley No. 30083. <http://www.leyes.congreso.gob.pe/Documentos/Leyes/30083.pdf>, 2013.
- [12] DHANANJAY, A., TIERNEY, M., LI, J., AND SUBRAMANIAN, L. WiRE: a New Rural Connectivity Paradigm. In *SIGCOMM* (2011), pp. 462–463.
- [13] DONNER, J. The Rules of Beeping: Exchanging Messages Via Intentional "Missed Calls" on Mobile Phones. *Journal of Computer-Mediated Communication* 13, 1 (2007).
- [14] DUARTE, M. E. *Network Sovereignty: Building the Internet Across Indian Country*. University of Washington Press, 2017.
- [15] EARL, S., SINHA, C., AND SMITH, M. L. Innovations in Evaluating ICT4D Research. *Connecting ICTs to Development* (2013), 241.
- [16] FALL, K. A Delay-Tolerant Network Architecture for Challenged Internets. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2003), ACM, pp. 27–34.
- [17] FEDERAL COMMUNICATIONS COMMISSION. Citizens Broadband Radio Service (FCC 15-47). <https://docs.fcc.gov/public/attachments/FCC-15-47A1.pdf>, 2015.
- [18] FRANGOURDIS, P. A., POLYZOS, G. C., AND KEMERLIS, V. P. Wireless Community Networks: an Alternative Approach for Nomadic Broadband Network Access. *IEEE Communications Magazine* 49, 5 (2011).
- [19] FREESWITCH. <http://www.freeswitch.org/>. Retrieved 2/2019.
- [20] GLOBE TELECOM, INC. <https://www.globe.com.ph/>. Retrieved 9/2018.
- [21] GOMEZ-MIGUELEZ, I., GARCIA-SAAVEDRA, A., SUTTON, P. D., SERRANO, P., CANO, C., AND LEITH, D. J. srsLTE: an Open-Source Platform for LTE Evolution and Experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization* (2016), ACM, pp. 25–32.
- [22] GRAY, J., CHONG, W., BARCLAY, T., SZALAY, A., AND VANDENBERG, J. TeraScale SneakerNet: Using Inexpensive Disks for Backup, Archiving, and Data Exchange. *arXiv preprint cs/0208011* (2002).
- [23] GSMA INTELLIGENCE. Country Overview: Philippines Growth Through Innovation. <https://www.gsmaintelligence.com/research/?file=141201-philippines.pdf>. Retrieved 9/2018.
- [24] GUO, S., FALAKI, M. H., OLIVER, E. A., OLIVER, E. A., RAHMAN, S. U., RAHMAN, S. U., SETH, A., ZAHARIA, M. A., AND KESHAV, S. Very Low-Cost Internet Access Using KioskNet. *SIGCOMM Comput. Commun. Rev.* 37, 5 (2007), 95–100.
- [25] HASAN, S., HEIMERL, K., HARRISON, K., ALI, K., ROBERTS, S., SAHAI, A., AND BREWER, E. A. GSM Whitespaces: An Opportunity for Rural Cellular Service. In *DySPAN* (2014), pp. 271–282.
- [26] HEIMERL, K., ALI, K., BLUMENSTOCK, J., GAWALT, B., AND BREWER, E. Expanding Rural Cellular Networks with Virtual Coverage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (Lombard, Illinois, 2013), NSDI'13, USENIX Association.
- [27] HEIMERL, K., AND BREWER, E. The Village Base Station. In *Proceedings of the 4th ACM Workshop on Networked Systems for Developing Regions* (San Francisco, California, 2010), NSDR '10, ACM, pp. 14:1–14:2.
- [28] HEIMERL, K., HASAN, S., ALI, K., BREWER, E., AND PARikh, T. Local, Sustainable, Small-Scale Cellular Networks. In *Proceedings of the Sixth International Conference on Information and Communication Technologies and Development* (Cape Town, South Africa, 2013), ICTD '13, ACM.

- [29] JACKSON, S. J., POMPE, A., AND KRIESHOK, G. Things Fall Apart: Maintenance, Repair, and Technology for Education Initiatives in Rural Namibia. In *iConference* (2011).
- [30] JACKSON, S. J., POMPE, A., AND KRIESHOK, G. Repair Worlds: Maintenance, Repair, and ICT for Development in Rural Namibia. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (New York, NY, USA, 2012), CSCW '12, ACM, pp. 107–116.
- [31] JAIN, S., AND AGRAWAL, D. P. Wireless Community Networks. *Computer* 36, 8 (2003), 90–92.
- [32] JANG, E., BARELA, M. C., JOHNSON, M., MARTINEZ, P., FESTIN, C., LYNN, M., DIONISIO, J., AND HEIMERL, K. Crowdsourcing Rural Network Maintenance and Repair via Network Messaging. In *Conference on Human Factors in Computing Systems (CHI)* (2018).
- [33] JIN, X., LI, L. E., VANBEVER, L., AND REXFORD, J. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (2013), ACM, pp. 163–174.
- [34] JOHNSON, M., SEVILLA, S., JANG, E., AND HEIMERL, K. dLTE: Building a more WiFi-like Cellular Network: (Instead of the Other Way Around). In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15–16, 2018* (2018), pp. 8–14.
- [35] KAM, M., MATHUR, A., KUMAR, A., AND CANNY, J. Designing Digital Games for Rural Children: A Study of Traditional Village Games in India. In *Conference on Human Factors in Computing Systems (CHI)* (2009).
- [36] KANNEL. <https://www.kannel.org/>. Retrieved 2/2019.
- [37] KUMAR, N., PERRIER, T., DESMOND, M., ISRAEL-BALLARD, K., KUMAR, V., MAHAPATRA, S., MISHRA, A., AGARWAL, S., GANDHI, R., LAL, P., AND ANDERSON, R. Projecting Health: Community-Led Video Education for Maternal Health. In *Seventh International Conference on Information and Communication Technologies and Development* (2015).
- [38] LETIA, M., PREGUIÇA, N., AND SHAPIRO, M. CRDTs: Consistency without Concurrency Control. *arXiv preprint arXiv:0907.0929* (2009).
- [39] MARTÍNEZ FERNÁNDEZ, A., VIDAL MANZANO, J., SIMÓ REIGADAS, J., PRIETO EGIDO, I., AGUSTÍN DE DIOS, A., PACO, J., AND RENDÓN, Á. The TUCAN3G Project: Wireless Technologies for Isolated Rural Communities in Developing Countries Based on 3G Small-Cell Deployments. *IEEE Communications Magazine* 54, 7 (2016), 36–43.
- [40] NEXMO. <https://www.nexmo.com/>. Retrieved 2/2019.
- [41] NEXTEPC. <http://nextepc.org/>.
- [42] NGUYEN, B., ZHANG, T., RADUNOVIC, B., STUTSMAN, R., KARAGIANNIS, T., KOCUR, J., AND VAN DER MERWE, J. ECHO: A Reliable Distributed Cellular Core Network for Hyper-Scale Public Clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (2018), ACM, pp. 163–178.
- [43] NIKAEIN, N., KNOPP, R., KALTENBERGER, F., GAUTHIER, L., BONNET, C., NUSSBAUM, D., AND GHADDAB, R. OpenAirInterface: an Open LTE Network in a PC. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking* (2014), ACM, pp. 305–308.
- [44] OPENBTS. <http://openbts.org>. Retrieved 9/2018.
- [45] OSMOCOM. <https://osmocom.org/projects/cellular-infrastructure>. Retrieved 9/2018.
- [46] PATEL, G., AND DENNETT, S. The 3GPP and 3GPP2 Movements Toward an All-IP Mobile Network. *IEEE Personal Communications* 7, 4 (2000), 62–64.
- [47] PATRA, R., NEDEVSCHI, S., SURANA, S., SHETH, A., SUBRAMANIAN, L., AND BREWER, E. WiLDNet: Design and Implementation of High Performance WiFi Based Long Distance Networks. In *4th USENIX Symposium on Networked Systems Design and Implementation* (2007).
- [48] PENTLAND, A. S., FLETCHER, R., AND HASSON, A. DakNet: Rethinking Connectivity in Developing Nations. *Computer* 37, 1 (2004), 78–83.
- [49] QAZI, Z. A., PENUMARTHI, P. K., SEKAR, V., GOPALAKRISHNAN, V., JOSHI, K., AND DAS, S. R. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *Proceedings of the Symposium on SDN Research* (2016), ACM, p. 2.
- [50] QAZI, Z. A., WALLS, M., PANDA, A., SEKAR, V., RATNASAMY, S., AND SHENKER, S. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 348–361.
- [51] RHIZOMATICA. <http://rhizomatica.org/>. Retrieved 2/2019.
- [52] SALDANA, J., ARCIA-MORET, A., BRAEM, B., PIETROSEMOLI, E., SATHIASEELAN, A., AND ZENNARO, M. Alternative Network Deployments: Taxonomy, Characterization, Technologies, and Architectures. Tech. rep., 2016.
- [53] SEVILLA, S., KOSAKANCHIT, P., JOHNSON, M., AND HEIMERL, K. Building Community LTE Networks with CoLTE. In *The community network manual : how to build the Internet yourself*, L. Belli, Ed. FGV Direito Rio, 2018.
- [54] SHAH, K., MARTINEZ, P., CRUZ, G. M. D., BLUMENSTOCK, J. D. J., AND HEIMERL, K. An Investigation of Phone Upgrades in Remote Community Cellular Networks. In *International Conference on Information and Communication Technologies and Development* (2017), P. Lahore, Ed.
- [55] SURANA, S., HO, M., PATRA, R., AND SUBRAMANIAN, L. Designing Healthcare Systems in the Developing World: The Role of Computer Science Systems Research. Tech. rep., New York University.
- [56] SURANA, S., RABIN PATRA, S. N., RAMOS, M., SUBRAMANIAN, L., BEN-DAVID, Y., AND BREWER, E. Beyond Pilots: Keeping Rural Wireless Networks Alive. In *5th USENIX Symposium on Networked Systems Design and Implementation* (2008).
- [57] TELECOM INFRA PROJECT. OpenRAN Project Group. <https://telecominfraproject.com/openran>.
- [58] VEGA, D., BAIG, R., CERDÀ-ALABERN, L., MEDINA, E., MESEGUER, R., AND NAVARRO, L. A Technological Overview of the Guifi.net Community Network. *Computer Networks* 93 (2015), 260–278.
- [59] VEGA, D., CERDÀ-ALABERN, L., NAVARRO, L., AND MESEGUER, R. Topology Patterns of a Community Network: Guifi.net. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on* (2012), IEEE, pp. 612–619.

9 Appendix

Site launch dates. Sites were launched throughout the three year duration of the project. While initial testing at “proof of concept” sites began as early as 2016, the bulk of sites were deployed in late 2017 and throughout 2018. We note that while the UP sites were deployed at a rate of less than once per month, Globe sites were deployed in bursts, with as many as four sites being deployed within the same week.

Site Name	Type	Commercial Launch
Sabang-Limbok	UP	Sept 13, 2017
Dikapinisan	UP	Oct 25, 2017
Dibut	UP	Feb 1, 2018
Diotorin	UP	May 30, 2018
Bacong-Market	UP	Aug 29, 2018
Dianao	UP	Oct 17, 2018
Tanay	Globe	Jan 29, 2016
Talisay	Globe	Jan 21, 2017
Binobohan	Globe	Feb 2, 2018
Ginulagan	Globe	Apr 3, 2018
Balogo	Globe	Apr 3, 2018
Casalaan	Globe	Apr 5, 2018
Banat-i	Globe	Apr 5, 2018
Mayaposi	Globe	Jun 30, 2018
Golden Valley	Globe	Aug 10, 2018
San Mariano	Globe	Aug 11, 2018
Binucayan	Globe	Aug 11, 2018

Table 3: Site launch dates as of December 2018.

Table 3 depicts launch dates and locations for each site, and Figure 9 shows the locations of deployed sites.

Site costs. The below tables provide an example of site fixed costs and operating expenses for a typical site in our deployment. Note that prices are in USD, and are approximate. These prices are estimates that reflect cost of equipment once it has already cleared customs and is in country. Similarly, civil and telecom works costs can vary depending on the difficulty of access to a particular site.

Globe preferred to have a single vendor for each component, with the goal of standardizing their deployments and reducing costs. In contrast, the UP installations used different setups and vendors due to a combination of funding limitations, procurement difficulties and delays. For example, we experienced challenges in the procurement and importation of GSM radio hardware. As such, some sites used equipment assembled from spare components we already had in stock.

The monthly backhaul subscription constitutes the bulk of the OpEx costs. Since the sites are very remote, the only feasible option is VSAT (a satellite Internet technology) which is also expensive. Backhaul prices vary significantly depending on the provider; Globe has existing bulk contracts for capacity as well as their own VSAT hub and network infrastructure, lowering costs compared to end-to-end VSAT providers. Other OpEx components include transportation costs by the local cooperative for credit distribution and collection, and monthly honoraria for maintenance staff.

Infrastructure re-use – for example, using an existing tower or building to mount equipment – can reduce costs significantly when possible. For the UP sites, the lack of

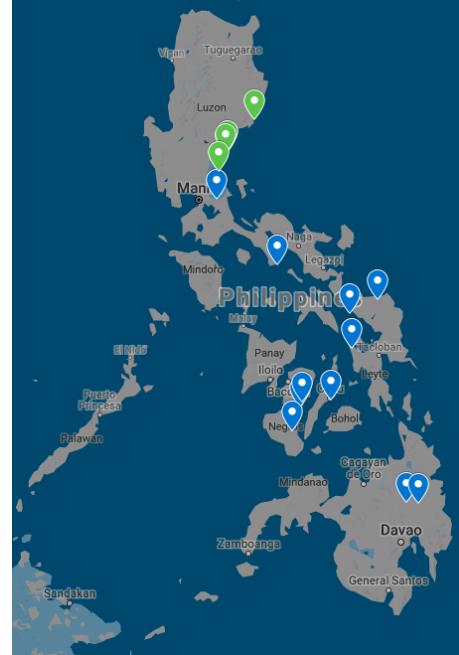


Figure 9: Map of deployed sites. UP sites are green, Globe sites are blue.

existing towers or other high structures required constructing new towers from scratch. We identified a local metal-worker to fabricate of towers, which we expect will reduce transportation costs compared to shipping tower components from Manila. Moreover, while grid power is provided in Aurora by a local electric cooperative, the grid infrastructure was deemed unreliable by the locals, who recommended that we use an off-grid solar power system instead. The local cooperative also favored this to avoid paying for the site's electrical consumption.

In our deployments, we used equipment that ran on several different voltages: 24VDC (common for low-power wireless equipment), -48VDC (common for telecom equipment), and 220VAC (grid voltage in the Philippines). This not only required additional equipment to perform the necessary conversions, but also led to decreased power efficiency for the entire site, driving up the cost of power, the second-largest component of site CapEx after the radio itself.

These costs also do not include fees for permits, as they were waived by the partner local government. As the project's main intention is research and not profitability, the local government units recognized the project's potential to help their constituents. While we do not have direct knowledge of what these fees cost, anecdotally we understand that they can both vary significantly by municipality and can constitute a significant portion of site costs; the permitting process is a point of leverage for local governments. Although fees were waived in our case, we still have to submit requisite documents such as construction plans and electrical plans.

Item	Cost	Notes
GSM Radio	US\$ 5,200	Combined GSM Radio + CPU for CCM Client, 2x10W
GSM Installation Accessories	US\$ 480	Includes cables, mounting brackets, etc.
GSM Antennas	US\$ 570	Two high-gain omnidirectional GSM900 antennas.
GSM Radio spare	US\$ 125	Budget per-site for spare radios.
VSAT System	US\$ 1,790	Includes installation (subcontracted).
Tower	US\$ 1,550	10m pole.
Lightning Protection	US\$ 200	
Telecom Works	US\$ 1,100	Installation for networking and power.
Civil Works	US\$ 3,100	Transport and construction of site infrastructure.
Site Survey and Testing	US\$ 800	Pre- and post-installation evaluation.
Power System (Solar)	US\$ 3,800	Two day backup power for off-grid sites.
Power System (Grid)	US\$ 3,100	Inverter + batteries for locations with grid power.
Total CapEx (lower bound)	US\$ 18,015	
Total CapEx (upper bound)	US\$ 18,715	

Table 4: An example breakdown of a deployed site cost for a Globe site. Two different power systems are considered, one for fully off-grid sites and another for sites with grid power. The radio vendor is anonymous due to a non-disclosure agreement with Globe.

Item	Cost	Notes
GSM Radio (Endaga CCN1)	US\$ 4,700	Locally-assembled, 2x10W.
GSM Radio (NuRAN LiteCell 1.5)	US\$ 6,000	2x10W.
x86 Computer	US\$ 300	
GSM Antenna and Accessories	US\$ 340	9dBi omni antenna, RF cables, connectors, grounding.
Networking Equipment	US\$ 200	Off-the-shelf switch and cables.
Power System (Solar)	US\$ 4,220	Three day standby power. Includes 800W panels, batteries, controller, inverter and other accessories.
VSAT System	US\$ 2,000	VSAT modem and antenna; includes installation.
Civil Works	US\$ 3,370	12m tower, equipment shelter, foundation, fencing.
Installation	US\$ 2,000	Includes personnel, transport and community training.
Total CapEx (lower bound)	US\$ 17,160	
Total CapEx (upper bound)	US\$ 18,760	

Table 5: CapEx cost breakdown for a UP site in Aurora.

Item	Cost (monthly)	Notes
VSAT service	US\$ 100-400	512x512kbps, Price varies across vendors.
Transportation	US\$ 40	For credit distribution, remittance or technical visits.
Local maintenance	US\$ 40	Two maintenance personnel at \$20 each. May be subsidized by LGU.
Total OpEx (lower bound)	US\$ 180	
Total OpEx (upper bound)	US\$ 480	

Table 6: OpEx cost breakdown for a UP site in Aurora.

TrackIO: Tracking First Responders Inside-Out

Ashutosh Dhekne^{† *}

Ayon Chakraborty*

Karthikeyan Sundaresan*

Sampath Rangarajan*

[†]*University of Illinois at Urbana-Champaign, *NEC Labs America, Inc.*

Abstract

First responders, a critical lifeline of any society, often find themselves in precarious situations. The ability to track them real-time in unknown indoor environments, would significantly contribute to the success of their mission as well as their safety. In this work, we present the design, implementation and evaluation of TrackIO—a system capable of accurately localizing and tracking mobile responders real-time in large indoor environments. TrackIO leverages the mobile virtual infrastructure offered by unmanned aerial vehicles (UAVs), coupled with the balanced penetration-accuracy tradeoff offered by ultra-wideband (UWB), to accomplish this objective directly from outside, without relying on access to any indoor infrastructure. Towards a practical system, TrackIO incorporates four novel mechanisms in its design that address key challenges to enable tracking responders (i) who are mobile with potentially non-uniform velocities (e.g. during turns), (ii) deep indoors with challenged reachability, (iii) in real-time even for a large network, and (iv) with high accuracy even when impacted by UAV’s position error. TrackIO’s real-world performance reveals that it can track static nodes with a median accuracy of about 1–1.5 m and mobile (even running) nodes with a median accuracy of 2–2.5 m in large buildings in real-time.

1 Introduction

Tracking first responders: First responders are integral to the safety and security of any community and to the society at large. However, they often find themselves in precarious and unknown environments, which poses a threat to their own safety (e.g. “entrainment” [16] faced by fire-fighters). Being able to accurately track first responders in indoor environments, allows a commander outside to better visualize and direct his responders appropriately. This not only helps address the situation efficiently but also ensures safety of the responders themselves—the latter can now view and track their own location with respect to the rest of the team.

Applicability of current solutions: The topic of indoor localization has seen many solutions in the past decade [4, 9, 53]. These can be broadly categorized under those that rely on indoor infrastructure (e.g. multitude of access points, RF/acoustic/infrared beacons, etc.) and those that do not (e.g. leveraging cellular BSs, GPS satellites, IMUs, etc.). While the latter can be applied to our target environment, they either offer less-than-desirable accuracies (e.g. tens of meters with cellular BSs), or are not functional indoors (e.g. GPS). Inertial sensors carried by responders are a possibility, but suffer from

poor accuracy as well ($\approx 10 - 50$ m, due to drift over time), without periodic calibration and resetting to known indoor reference points. Further, the lack of access to multiple stationary APs/BSs, prevents these solutions from accurately tracking *mobile* responders in real-time. Hence, notwithstanding the plethora of prior solutions, our target environment requires a new, robust, (indoor) *infrastructure-free* solution that can accurately ($\approx 1-2$ m) track *mobile* responders in unknown, indoor environments from outside.



Figure 1: *Left:* The TrackIO setup with a DJI Phantom 4 UAV carrying a master UWB node and the Raspberry Pi controller units, *Right:* 4 UWB equipped helmets for first responders.

Key design choices: This motivates us to design a localization-tracking system from scratch, paving the way for two key design choices: (i) modality of localization, and (ii) wireless technology for localization. The lack of indoor infrastructure support, and the need to quickly deploy and localize responders in 3D, across multiple floors of a building (from outside), makes UAV (unmanned aerial vehicle) an ideal platform for the task. The UAV can serve as a virtual mobile infrastructure that is deployed on-demand, outside the building to localize the responders inside. For the choice of wireless technology, we summarize their pros-cons in Table 1. We refer to only techniques that allow long distance localization that would be applicable in our application. While lower frequencies (e.g. LTE) offer better indoor penetration/coverage (e.g. 1 Km) from outside, they are limited by available bandwidths (tens of MHz) and hence accuracy (tens of meters [35]). Higher frequencies (e.g. mmWave, > 20 GHz) offer high accuracies (tens of cm) from large (GHz) bandwidths, but suffer from high attenuation (does not work with blockages, not accounted for in Table 1). Ultra-wideband (UWB) operates in 3 – 10 GHz and offers a 1 GHz bandwidth, thereby striking a good balance between accuracy (tens of cm) and indoor penetration (tens of meters). Further, its low power design accompanied by a standardized high-resolution, ranging protocol between peer UWB nodes, makes it a synergistic choice for deployment on the UAV. Thus, our objective is *to localize and track responders (carrying UWB nodes) in*

*The work was performed during an internship at NEC Labs America.

	LTE [35]	WiFi [25]	UWB [43]	mmW [36]
Accu	> 20m	5m	10cm	1cm
Range	> 1km	100m	50m	40m

Table 1: UWB offers the best tradeoff between accuracy and penetrability among all RF localization technologies.

real-time even if they are deep indoors, with the help of a UAV (also carrying a UWB node) flying outside.

Challenges: One might wonder if deploying multiple UAVs to effectively serve as stationary BSs/APs outside can help solve the indoor tracking problem. We argue in Section 2.1 that having multiple UAVs outside does not guarantee access to multiple (three or more) of them by a given indoor responder, not to mention the need for their synchronization. More importantly, we show that when available, multiple UAVs need to be efficiently deployed to cover (localize) responders in different sections of the building simultaneously, rather than to serve as stationary APs/BSs. Thus, at the core of our problem, we must *localize and track indoor responders in real-time by a single UAV using its key degree of freedom, namely mobility*. This in turn poses several non-trivial challenges. *(i) Mobility of responders:* The mobility of the UAV is used to create a synthetic aperture *over time*, which serves to provide reference points for localizing an indoor node¹ through multi-lateration. However, such temporal dependency, makes multi-lateration approaches fail significantly (accuracy of about 10m, Section 3), when the indoor node is also mobile. *(ii) Indoor coverage:* While UWB’s penetration capabilities are better than mmWave, they are still limited to tens of meters and hence cannot guarantee reachability to all nodes. While deploying multiple UAVs, outside different sections of the building can alleviate coverage issues, it still cannot ensure reachability to those that are deep inside the building. *(iii) Real-time tracking:* The UWB protocol provides the basic two-way ranging primitive between a UWB node-pair (UAV and responder in our case). However, employing its TDMA operational structure to collect sufficient ranging measurements to all UWB nodes from the UAV will not be scalable for real-time tracking in a large network. *(iv) Absolute location fix:* Since the UAV localizes the responders with respect to its own position, to get their absolute location fix, we need to accurately estimate the UAV’s position as well. Whereas high-end UAVs employ multiple GPS receivers along with inertial sensor fusion to provide position accuracy to under a meter, lower-end UAVs provide accuracies of only around 2-3 m, thereby limiting the accuracy of the overall system.

TrackIO: Towards addressing these challenges, we build TrackIO – a UAV-UWB based system that is capable of localizing and tracking mobile responders to within 1-2 m accuracy from a single UAV outside in real-time, even in deep indoor environments. When multiple UAVs are available, TrackIO

deploys them on different sections of the building for wider, simultaneous coverage. In realizing this, TrackIO incorporates four novel elements in its design.

(i) Trajectory Tracking: TrackIO adopts a first-principles approach to directly estimate the trajectory of the mobile node, rather than just its location. TrackIO analytically instruments multi-lateration formulation to not only estimate the location but also the velocity of the responder. It incorporates intelligent mechanisms for adaptively varying the size and choice of the synthetic aperture (anchor points used for localization) to address responders with non-uniform velocity (e.g. those turning corners, etc.).

(ii) Multi-hop Localization Paradigm: TrackIO enables a multi-hop localization paradigm for extended indoor coverage, where, responders directly reachable from the UAV (hop_1), are localized first. Then, they serve as anchors for localizing nodes (hop_2) that are reachable by them but not by the UAV. Nodes are able to dynamically estimate their own hop status based on their reachability to the UAV and overheard ranging messages from neighboring nodes. TrackIO alleviates the deterioration in accuracy over hops (due to iterative localization), by selecting only upstream nodes with accurate location estimates as anchors for downstream localization.

(iii) Concurrent Ranging Protocol: To enable real-time tracking even for a large, multi-hop network of nodes (e.g. big buildings), TrackIO transforms UWB’s sequential ranging protocol into an *efficient, concurrent* one. It leverages the broadcast nature of the wireless medium to (a) parallelize the ranging measurements within each hop, and (b) efficiently multiplexes ranging measurements between hops, while also eliminating redundant message transmissions. TrackIO achieves a $3\times$ speed-up, resulting in a location update frequency of 6 Hz that allows for real-time tracking.

(iv) Reverse Location Look-up: Instead of the UAV serving as the anchor, TrackIO now estimates the location of the UAV itself, by leveraging UWB again. It accomplishes this by using four static UWB beacons, deployed on the roof corners of a responder service vehicle, as anchors. One of these UWB beacons is also fitted with a GPS receiver, whose stationary estimates over time are highly accurate. This coupled with known inter-beacon distances, allows for accurate localization of the UAV to within a meter despite mobility.

TrackIO’s performance: We have built a complete version of TrackIO using a DJI Phantom 4 [13] as the UAV, and Decawave DW1000 [11] as the UWB node. The ranging estimates collected at the UAV are transferred to a ground service vehicle, where TrackIO’s algorithms estimate the position and trajectory of all the responders in real-time. Our real-world deployment and evaluation across multiple floors of a mid-size office building (2500 sq.m.) reveal that TrackIO is able to track indoor static nodes with a median accuracy of about 1–1.5 m and mobile (even running) nodes with a median accuracy of 2–2.5 m. A demo of TrackIO is available at <http://www.nec-labs.com/trackio>.

¹Responders are synonymously referred to as nodes.

Broader applicability: While TrackIO leverages UAV and UWB as its modality for enabling real-time tracking of first responders, we would like to note that TrackIO’s core mechanisms of trajectory tracking and multi-hop localization can be equally applicable to other localization modalities (e.g. WiFi) as well. Hence, TrackIO’s contributions can also benefit other potential indoor localization and tracking applications.

2 Challenges in Building a Practical System

The UAV flies outside to create a synthetic aperture of anchor points, from where it ranges with each of the indoor nodes using UWB, thereby allowing for their subsequent localization through multi-lateration. Albeit straight-forward in principle, realizing this in practice faces several challenges, some fundamental, and others practical that we now outline. **Brief Primer on UWB Ranging:** UWB nodes employ a protocol known as *two way ranging* (TWR) to estimate the distance between each other. This standard protocol [20], involves exchanging a specific set of messages (Figure 2) that cancels the effect of clock offsets between nodes. Performed in hardware with precise clocks and coupled with a 1GHz wireless bandwidth, TWR allows accurate time-of-flight estimates (even in presence of multi-path), resulting in accurate ranging (≈ 10 cm). One TWR exchange takes around 16.7 ms on a widely used UWB chip (DW1000 [10]).

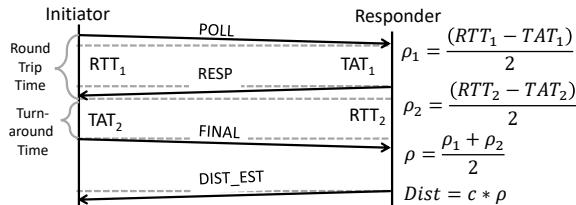


Figure 2: The original 802.15.4 TWR protocol is designed for ranging between two devices.

2.1 Impact of Responder Mobility

Fig. 3(a) shows a typical synthetic aperture where three representative $\langle \text{location}, \text{range} \rangle$ tuples are chosen to solve for a stationary node’s location using trilateration. The solution is reasonably accurate as all the three $\langle \text{location}, \text{range} \rangle$ tuples are consistent with respect to a unique location of the stationary node. Contrast this with figure 3(b), where the node moves with an uniform velocity. In this case, the three $\langle \text{location}, \text{range} \rangle$ tuples are no longer consistent with respect to any particular node location. Different portions of the synthetic aperture now correspond to different node locations. In other words, the node has changed its position significantly by the time the UAV started and completed building the aperture. This can affect localization accuracy by as much as 10 m, as shown in Fig. 9.

Can we alleviate the impact of mobility? A natural approach is to figure out if multiple $\langle \text{location}, \text{range} \rangle$ tuples can be gathered from distinct UAV locations “simultaneously”.

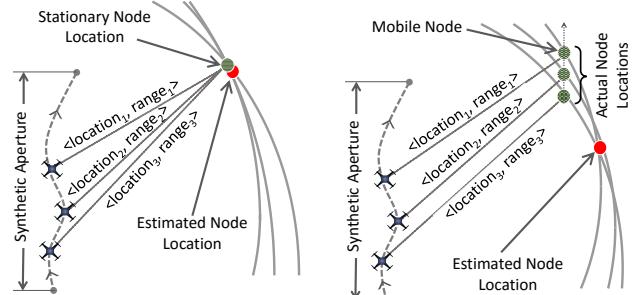


Figure 3: (a) Localization of a static node through trilateration (b) Naive trilateration fails for mobile nodes

Using multiple UAVs: Multiple UAVs form a spatial aperture that can simultaneously collect range estimates in principle. This is however, difficult to realize in practice for the following reasons: (a) It is unlikely that a particular indoor node is simultaneously reachable from multiple UAV locations, (b) Synchronizing the different UAVs as well as their corresponding range estimates in real-time becomes extremely challenging, and (c) Operating multiple UAVs in close vicinity requires sophisticated path planning to be done apriori. Multiple UAVs have a role to play in the broader system (for improving building coverage, as we discuss later). However, they are less useful to solve the problem of node mobility, which motivates us to address the problem with a single UAV.

Increasing the UAV’s speed relative to responders: Another approach to counter node mobility can be to increase the UAV’s velocity. Figure 4 shows the limited benefit of moving the UAV faster. Even when the UAV is traveling at 10m/s it cannot completely compensate for the node’s mobility. Moreover, moving the UAV too fast causes the channel to change very rapidly, resulting in ranging errors.

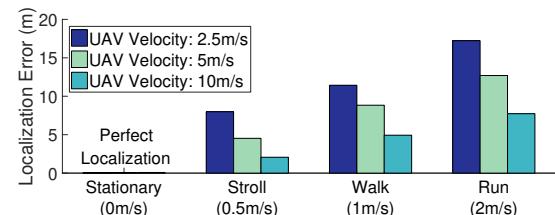


Figure 4: Localization accuracy improves with UAV speed, yet falls short of the target accuracy.

2.2 Insufficient Building Coverage

The FCC power emission limit for UWB transmitters is -41.3 dBm/MHz [20] that severely restricts the communication range between two UWB nodes. With the UAV located outside the building and limited indoor penetrability, some nodes that are relatively deep indoors are not directly reachable. We perform elaborate measurement studies to characterize the communication range in such indoor environments. Figure 5 shows the packet-loss percentage as the distance between the nodes increases in a cluttered indoor space. Such ranges could vary from about 30 m (50% loss) in very dense/cluttered indoor environments (e.g., rooms with concrete walls) to about

60 m in relatively open indoor spaces (e.g., office, library, shopping malls etc.).

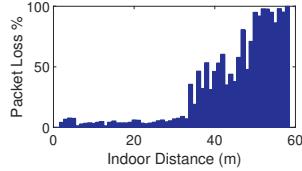


Figure 5: UWB packet-loss over various distances in a cluttered office environment.

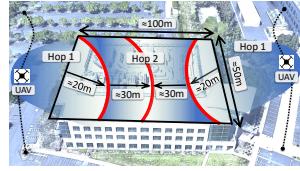


Figure 6: UAVs on the outside may not be able to cover nodes deep indoors. A multi-hop solution is necessary.

Multiple UAVs can improve but not solve the coverage problem. With the UAV flying approximately 10 m away from the face of the building, nodes that are about 30 m inside are directly reachable. This limits the indoor coverage area to a great extent. Note that even flying multiple UAVs along the four faces of a medium-sized building (floor area ≈ 10000 sq.m) only improves coverage in the building’s periphery but not in the deep interiors that account for about 20% of the indoor space (see figure 6). Given the criticality of the application, complete indoor coverage is of paramount importance. This necessitates the design of a multi-hop localization network, where range estimates and hence localization can be achieved from nodes of a given hop to nodes belonging to the next hop and so on.

Challenges with multi-hop localization. Realizing multi-hop localization is challenging for several reasons: (i) iterative localization leads to cascading errors and hence poor accuracy across hops; (ii) nodes need to identify their reachability status (e.g. hop_1 , hop_2 , etc.) to other nodes to help track a dynamic, multi-hop topology; (iii) orchestration of ranging measurements across hops becomes critical for ensuring real-time tracking of the multi-hop network.

2.3 Inability to Track Real-time

In a large network of nodes spanning multiple hops, a *time division* (TDMA) scheme needs to be designed that runs TWR across relevant pairs of nodes to estimate their range fast enough to relatively localize all nodes in the network. Clearly, executing a TWR across all pairs of nodes is not suitable: unreachable links will waste time, and in a size N network, one round of (range) data collection will require $O(N^2)$ time slots. Hence, for a network consisting of several tens of nodes, collecting a single set of range information might take several hundreds of milliseconds. With several such sets needed to position the indoor nodes, the total delay can be several seconds. Further, with the mobility of nodes resulting in highly dynamic topologies, it becomes very challenging to track nodes in real-time with such an update rate of measurements.

2.4 Imprecise UAV Localization

Note that the UAV’s location measurements need to be as precise as possible in order to leverage the highly precise range estimations offered by the UWB technology. Unfortunately,

UAV location estimates obtained out-of-the-box is at least an order of magnitude less precise compared to UWB ranges. For instance, consider the location estimates obtained from a GPS device. In an open field, such locations have minimal errors ($\approx 2\text{--}3\text{m}$). However, in scenarios, where the UAV moves along the periphery of a building, the GPS signal reception can be significantly hampered resulting in the error to escalate to as high as 15–20m [50].

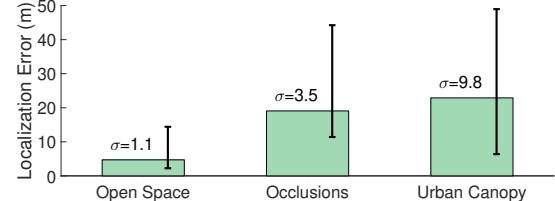


Figure 7: Effect of UAV’s GPS errors on localization accuracy

In figure 7 we show the impact on localization accuracy of a static node using simple trilateration in three different deployment settings; from a relatively open space to locations having partial occlusions and urban canopies. Note that even for a static outdoor node, slightly erroneous GPS locations of the UAV can be detrimental for its eventual localization. Assuming GPS corrections and inertial sensor fusion applied by the UAV, the errors could be at best, 1 – 2 m even when the node is outside and static; localizing a mobile node indoors would only lead to significantly degraded accuracies.

3 System Design

We now present TrackIO – a UAV-UWB based system that is capable of localizing and *tracking mobile* responders to within 1-2 m accuracy from a single UAV outside in *real-time*, even in *deep indoor* environments. TrackIO accomplishes this without the necessity or dependence of any infrastructure deployed indoors. TrackIO can almost instantly be functional from the time of launch (under a minute). This is achieved by employing a host of algorithmic and architectural changes to the underlying multilateration and ranging protocols.

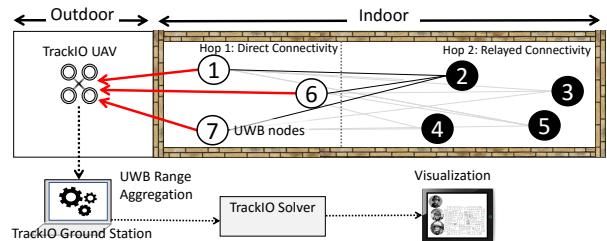


Figure 8: High level system design

3.0.1 Overview

Fig. 8 shows a snapshot of TrackIO in action along with its various architectural components. The UAV flying outside the building’s periphery is equipped with a UWB *master node* that collects range information from the *client nodes* inside the building. The nodes are possibly worn by personnel (e.g.,

firefighters, military troops, emergency responders etc.) who are tracked through our system. Client nodes that are directly reachable from the UAV’s master node are designated as hop_1 nodes, additional nodes are referred to as hop_2 nodes, hop_3 nodes and so on based on subsequent reachability. In Fig. 8, the UAV directly ranges the nodes in hop_1 , who in turn range the nodes in hop_2 and relay information back to the UAV. The UAV offloads range information to a ground control station that solves for the locations of all client nodes. Additionally, we develop a mobile application that can be used to visualize the tracking information on a map with sub-second latency.

When multiple UAVs are available, they are deployed on different sections of the building and/or at different altitudes of the same section (for tall buildings) for wider, simultaneous coverage. Since each UAV would execute TrackIO in parallel, we focus on a single UAV’s operation in the rest of this section. Also, for easier exposition, we focus on the UAV localizing responders in a single floor (horizontal plane) by fixing its altitude appropriately. How the UAV scans floors and identifies the appropriate altitude (z^*) is covered in Section 3.5. Some results presented in this section are obtained from simulation studies, which are intended for highlighting the intricate aspects of our system design. Nonetheless, sections 4 and 5 present extensive evaluation results from experiments carried out in real testbeds.

3.1 Tracking Trajectory of Mobile Nodes

3.1.1 Estimating Velocity through Synthetic Aperature

Recall that when a node is mobile, the $\langle \text{location}, \text{range} \rangle$ tuples measured by the UAV do not uniquely map to a single location, resulting in poor localization accuracy of multilateration solvers. Instead of alleviating the impact of mobility, TrackIO adopts a first-principles approach to directly estimate the trajectory (speed and heading) of the mobile node, rather than just its position. To accomplish this, TrackIO analytically instruments the multilateration formulation to estimate both the initial location (x, y) as well as the velocity vector $(V_x \hat{i} + V_y \hat{j})$, where \hat{i} and \hat{j} are unit vectors along positive X and Y axes respectively of the node. Using these, the node’s traversed path can be traced. This assumes that human mobility can be approximated with uniform velocity, which is reasonable within the short time-scales (few seconds) of the UAV’s synthetic aperture. This assumption is relaxed in Section 3.1.2, where we show how non-uniform mobility (e.g. turning corners, etc.) can also be addressed in this framework.

Suppose we have ranging measurements from n consecutive UAV locations – n is called the aperture size and is essentially a moving window of n historical measurements. For any time instant T_i ($i = [1..n]$), the UAV records the mapping $\langle \text{location}_i, \text{range}_i \rangle$, where location_i is the UAV’s 3D location and range_i is the distance estimate of the mobile node from the UAV. The mobile node is located at an unknown location (x_i, y_i, z_i) . We denote the UAV’s 3D-location

as (Cx_i, Cy_i, Cz_i) . The measured range is given by:

$$\text{range}_i = \sqrt{(Cx_i - x_i)^2 + (Cy_i - y_i)^2 + (Cz_i - z_i)^2} \quad (1)$$

Assuming we know which building floor the responder is currently occupying, we do not need to solve for z_i ($= z^*$). Yet, this is a single equation with two unknowns—we cannot directly solve for (x_i, y_i) . Even if we obtain multiple such ranges, each equation will add a new set of unknowns. However, the new unknowns are not independent, but related through the node’s velocity. Hence, assuming the node is moving at a constant velocity, there are inherently only four unknowns $(x_1^*, y_1^*, V_x^*, V_y^*)$ that do not increase with additional ranges, thereby allowing us to solve for them.

We can reformulate this as an unconstrained minimization problem that attempts to find the best fit, i.e. location and velocity that minimize the following error function:

$$(x_1^*, y_1^*, V_x^*, V_y^*) = \arg \min_{(x_1, y_1, V_x, V_y)} f \\ f = \sum_{i=1}^n ((Cx_i - x_i)^2 + (Cy_i - y_i)^2 + (Cz_i - z_i)^2 - \text{range}_i^2)^2 \quad (2)$$

The various (x_i, y_i) are obtained from the intial (x_1, y_1) and velocity (V_x, V_y) based on kinematic equations:

$$x_i = x_{i-1} + V_x \cdot \Delta T_i = x_1 + V_x \sum_{j=1}^i \Delta T_j \\ y_i = y_{i-1} + V_y \cdot \Delta T_i = y_1 + V_y \sum_{j=1}^i \Delta T_j \quad (3)$$

where ΔT_i denotes the time between measurements. Since (x_i, y_i) are generated based on the initial location (x_1, y_1) and the velocity vectors (V_x, V_y) , by minimizing equation 2, we obtain the closest approximation of both, location and velocity vectors for the node. The first output from this solver is obtained only after n measurements (typically a few seconds worth of data) have been recorded. Thereafter, a location update is obtained for every round of range measurements. Thus, the system’s steady state update rate depends only on the duration of one range measurement round, and does not depend on the aperture size.

We now analyze the improvement in localization achieved by incorporating velocity vectors over simple multilateration. Our simulation framework mimics a UAV and a set of indoor UWB nodes that follow predetermined trajectories at any desired speed. We introduce an empirically derived range estimation error to the ranges. Fig. 9 shows how simple multilateration results in higher localization errors with increasing node velocities. The UAV is assumed to move at a fixed 5 m/sec velocity. Note that even for human walking speeds the error could be as high as 10 m. On the contrary, the velocity-based solver is least impacted by increasing velocity of the mobile node.

3.1.2 Adaptive Apertures for Non-uniform Velocity

The above approach assumes that the node does not change its velocity (speed or direction) during the course of one aperture window (say, 4 secs). However, this assumption is broken if

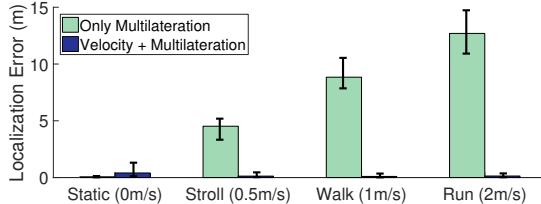


Figure 9: Localization error remains minimal when using velocity vectors even under fast human mobility

the node turns, accelerates or halts. In principle, this could be solved by adding higher order derivatives of the node’s location (e.g., acceleration, jerk) into the kinematics equations employed by our location solver. However, our analysis indicates that such an approach is rather contrived. It makes the solver prone to overfitting and extremely sensitive to range errors. Further, given the short time scale of the aperture window, we find that the approximation of uniform velocity does not hurt the performance much during acceleration and halting but does induce significant errors during *turns*, which we now address. We propose to utilize the solver’s confidence in the estimated location to infer non-uniform velocity and when detected, trigger an aperture reset that eliminates measurements prior to the turn.

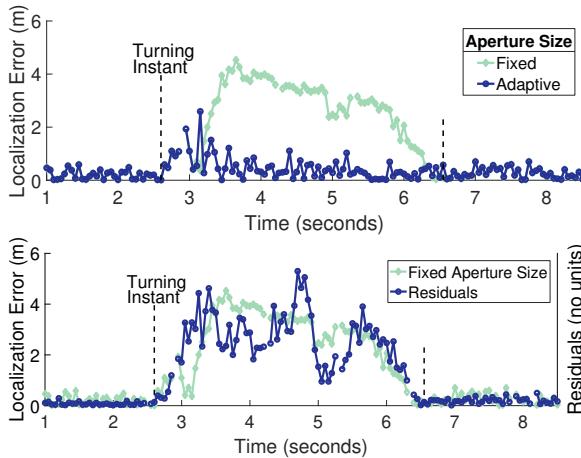


Figure 10: (a) Localization error is high during turns. Resetting the aperture helps curtail the loss in localization accuracy. (b) The high error-residual also indicates low solver confidence in the location estimate providing a hint for turn detection.

Impact of turns. Fig. 10(a) shows the impact of sudden turns on the localization error (green line). We simulated a node moving in a straight line, then taking a 90° turn, and continuing again in a straight line. An aperture of 4 seconds—UAV locations and the corresponding ranges of the past 4 seconds—are used to estimate the current node location. Observe how the localization error (grey line) starts to increase from the point where the aperture’s *head* crosses the turning position (first dashed vertical line) and falls back to its pre-turn values after the aperture’s *tail* has crossed the turning position (second dashed vertical line).

Adaptive aperture to address turns. If we have a mecha-

nism to detect turns, we could potentially eliminate historic measurements till the turn and restart constructing the aperture. To understand the benefit, we introduce the notion of an adaptive aperture in the above simulation. At the time of the turn, we remove all history and restart estimating location after a short history has built up². Just after resetting history, the localization error is indeed high (dark blue line just after the “turning instant” in Fig. 10(a)) but quickly recovers and becomes acceptable once the aperture fills up with relevant measurements after the turn. In comparison, if a fixed aperture size is used, the effects of a turn last for the entire duration of the aperture (green line).

Triggering an adaptive aperture. During turns, the solver is unable to provide a reasonable answer since no single velocity estimate can represent all the measurements. This results in larger residual errors after solving Equation 2. Observe in Fig. 10(b) that the solver’s residuals (in arbitrary units) are highly correlated with localization error. Thus, a sudden increase in the residuals helps identify non-uniform velocity events such as turns. We use Gaussian Mixture Models on the residuals to identify a changing trend in them and captures such events.

In summary, localization of mobile nodes, even those with non-uniform velocity is possible through a combination of joint location-velocity solving and by adaptively resetting the aperture size. At any given instant, our solver uses different aperture sizes that are appropriate for each node.

3.2 Multi-hop Tracking for Coverage

TrackIO is designed to function even if some nodes are beyond the UAV’s direct range. TrackIO allows such unreachable nodes to range with other nodes in the vicinity which can in-turn reach the UAV and/or have already been localized. Thus, a multi-hop topology is dynamically created with nodes belonging to different hops based on their reachability characteristics. The UAV’s synthetic aperture localizes first hop (directly reachable) nodes. These hop_1 nodes then act as anchors for localizing hop_2 nodes. This process iterates across hops. TrackIO employs several mechanisms to ensure that mobile nodes can be accurately localized even across multiple hops.

Dynamic estimation of hop membership. Nodes that are within the UAV’s UWB communication range, directly receive ranging messages initiated by the UAV, and classify themselves as hop_1 nodes. Those that do not receive messages from the UAV but receive some of the response messages sent by hop_1 nodes, classify themselves as hop_2 nodes and so on. Thus, nodes can determine their own hop membership in a decentralized manner.

Anchor selection for iterative localization. Two components contribute to the final localization error of hop_m nodes:

²During the short period that new history is being built, the system continues to output results from the previous aperture.

1. relative localization error of hop_m nodes with respect to hop_{m-1} nodes, and 2. localization error of hop_{m-1} nodes chosen as anchors. Without loss of generality, hop_{m-1} nodes can be assumed to be spaced far apart compared to the synthetic aperture formed by the UAV. This increased spacing between anchors, *improves* hop_m localization, compared to that of hop_1 nodes (w.r.t. the UAV). However, the localization error of the hop_{m-1} nodes and recursively that of upstream hop nodes, will cumulatively contribute to the error of hop_m nodes. Thus, the choice of anchors in hop_{m-1} , has a cascading impact on the localization accuracy of downstream nodes (i.e. $\text{hops} \geq m$).

Nodes that are static or moving with a uniform velocity in hop_1 inherently have better localization accuracy than those with non-uniform velocity. Hence, by leveraging the solver's ability to identify such nodes (those with high residuals), TrackIO avoids selecting them as anchors for localizing hop_m nodes, curtailing the cascading effect across hops.

Instantaneous mobility tracking beyond hop_1 . In contrast to the first hop nodes which are localized through a *temporal* aperture created by the UAV's motion, hop_m ($m > 1$) nodes are localized through a *spatial* aperture formed from a diverse placement of hop_{m-1} nodes. This decoupling (from UAV's mobility), allows for *instantaneous* localization of hop_m nodes from previously obtained hop_{m-1} locations. The time scale of such localization is in milliseconds within which the nodes move a negligible distance. As a result of the spatial aperture employed, hop_m ($m > 1$) nodes can use conventional multilateration approaches (without need for velocity vectors) even when they are mobile.

Localizing hop_1 nodes with non-uniform velocity using downstream spatial apertures. Unlike hop_1 nodes, mobility is not a concern for hop_m ($m > 1$) nodes as they are instantaneously localized using a *spatial* aperture formed from high-confidence (low residual) hop_{m-1} nodes. Hence, hop_2 nodes can in-turn, form a spatial aperture (serve as anchors) and correct the location of hop_1 nodes, which are currently experiencing non-uniform velocity (low confidence, high residual). Fig. 11 shows the localization accuracy of a turning hop_1 node using ranges from hop_2 nodes. Observe how the turn gets localized precisely using the spatial aperture from hop_2 . Thus, TrackIO is able to eliminate most of the impact of non-uniform velocity of hop_1 nodes.

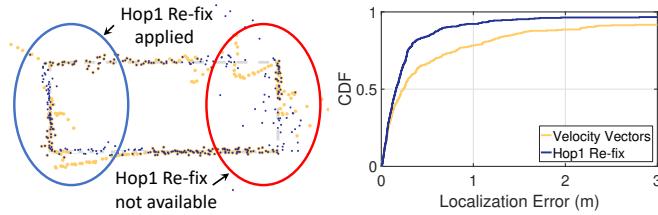


Figure 11: Re-fixing hop_1 nodes using hop_2 nodes improves localization even during turns. But, it may not be always available.

Note that this downstream spatial aperture technique is opportunistic – it can be used when enough hop_2 nodes exist.

In contrast, the mechanism of adaptive (temporal) aperture of the UAV described in Section 3.1, provides benefits even when no other nodes exist in the topology. Therefore, TrackIO incorporates both these techniques to address non-uniform velocity of hop_1 nodes.

Leveraging multiple UAVs vs. multiple hops. While multi-hop localization allows for coverage of even large buildings using a single UAV, the localization error of its downstream hop nodes (i.e. $m > 2$) will increase and might not satisfy our desired target of 1-2m. Hence, TrackIO leverages the multi-hop paradigm to primarily reach deep interiors of buildings (where even multiple UAVs cannot help), while employing multiple UAVs to provide non-overlapping, peripheral coverage for large buildings. Thus, using a combination of multiple UAVs and hops, TrackIO is able to cover large buildings with just two hops from a single UAV.

Handling hop_2 disconnections. In rare circumstances, if a node goes out of range from all hop_1 nodes, its localization must rely on an IMU-based dead reckoning system. This approximate location estimate will then be communicated with the UAV using alternative communication modes (such as WiFi or cellular data). Adding support for such eventualities is left to future work.

3.3 Concurrent Ranging

A fast and reliable ranging protocol is essential to create a real-time localization system. Since the UWB ranging protocol is designed for ranging between a pair of nodes, it does not broadcast messages. This leads to a sequential ranging of every node in a hop, which is not scalable for real-time operation, especially in a multi-hop network. The key idea in TrackIO is to leverage the broadcast nature of wireless signals to communicate and hence *concurrently* range with multiple nodes using a single transmission. To this end, TrackIO makes appropriate modifications to the underlying protocol (Fig. 2) to create a concurrent ranging scheme (Fig. 12). We describe the scheme for the first two hops; subsequent hops are similar.

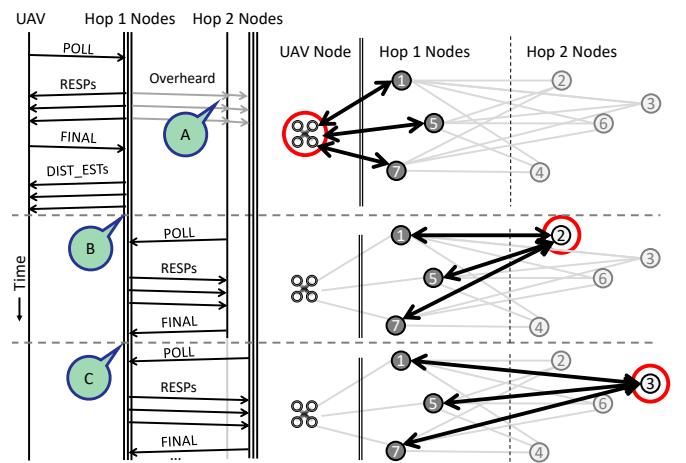


Figure 12: Progress of the protocol in a 2-hop example topology.

Concurrent ranging at hop_1 . The UAV simultaneously ini-

tiates ranging with all reachable hop_1 nodes by broadcasting a single POLL. Each node that receives this message, takes turns (based on its hard-coded NodeID) to send a RESP message. After collecting the timings from all the RESPs, the UAV broadcasts a single FINAL message containing information for all hop_1 nodes. On receiving the FINAL, all hop_1 nodes calculate their distance from the UAV and send it back to the UAV (DIST_EST messages).

Concurrent ranging at hop_2 . Identical to hop_1 nodes, hop_2 nodes listen to the channel for messages. However, being outside the direct communication range of the UAV, they cannot receive the POLL message. Instead, they only overhear the messages sent by nearby hop_1 nodes in response to the UAV's POLL (point A in Fig. 12). After all hop_1 nodes have completed sending their DIST_EST messages (point B in Fig. 12), the first hop_2 node initiates a full sequence of POLL-RESPs-FINAL simultaneously with all hop_1 nodes in the vicinity. hop_2 nodes follow the same protocol as the UAV with one subtle difference. hop_1 nodes do not send DIST_ESTs back to hop_2 nodes. Instead, hop_1 nodes calculate and locally store all the hop_1 - hop_2 ranges, which are piggybacked on the subsequent DIST_EST message. This saves unnecessary network overhead, speeding up the collection of range estimates. All hop_2 nodes take turns (point C in Fig. 12), followed by the UAV starting the next round.

Efficient multiplexing of ranging between hops. Initially the UAV is not aware of the topology. Hence, it waits for all the nodes in the network to send a RESP. Once it has received the last RESP (or, after a timeout), the UAV creates a bitmap (Fig. 13) indicating which nodes are deemed to be in hop_1 (setting the corresponding bit to one) based on the responses. The UAV sends this bitmap in its FINAL message. When hop_1 nodes send a DIST_EST message it also contains a copy of this bitmap. A node that receives such a DIST_EST, but not the POLL from the UAV, would see its bit cleared and know that it belongs to hop_2 . Also, it would know how many other hop_1 nodes are expected to send their DIST_ESTs, and the order of all other hop_2 nodes. This allows hop_2 nodes to efficiently take transmission turns without collision, even when they are not in communication range of each other and the UAV. The UAV generates the bitmap dynamically in every round to track topology dynamics due to node mobility.

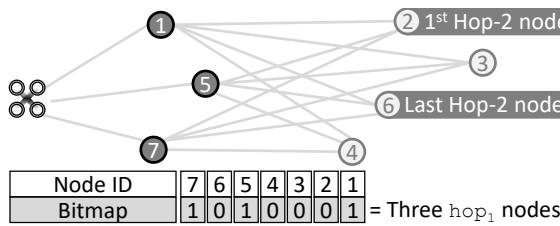


Figure 13: The bitmap constructed by the UAV and sent in the FINAL message enables a collision free hop_2 communication.

Finally, the variable length DIST_EST sent by hop_1 nodes piggybacks their distance from all hop_2 nodes obtained in the

previous round, along with their own *current* range estimates to the UAV. The UAV aggregates all the information received in the DIST_ESTs and forwards to a ground control center for further processing.

3.4 Reverse Lookup for UAV Location Fix

Obtaining the UAV's precise GPS location is critical to TrackIO's end-end accuracy. This can be challenging since off-the-shelf GPS receivers have multi-meter location errors [24, 50]. High-End UAVs already employ GPS chips with better precision and higher update rate [14], and improve the precision further by incorporating IMU data as well. Some UAVs [14] also support custom, albeit expensive RTK solutions [39] that promise location accuracy within a few cm, but require precise GPS transmitters in the vicinity.

In cases, where such precise UAV location estimates are not possible, TrackIO leverages UWB to also localize the UAV. It places four *static* UWB nodes as anchors at known locations on the ground. One of these anchors is also fitted with a GPS receiver. The known, exact, pairwise distances between the anchors enables TrackIO to accurately determine the GPS coordinates of all the static anchors. These static anchors in turn allow for accurate localization of the UAV itself. We envision that these anchors can be permanently mounted at the four corners of a service vehicle (at different heights to provide vertical diversity). The service vehicle can use sophisticated GPS techniques [19, 21, 22, 26, 38, 39] to achieve better accuracy for the ground anchors.

3.5 TrackIO's Operations in a Nutshell

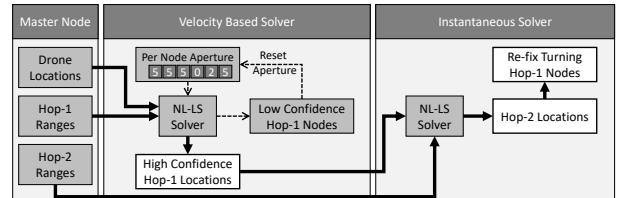


Figure 14: TrackIO System Design

When a UAV is launched to cover a section of the building, the static UWB anchors on the ground start localizing the UAV to get its precise location estimate. The UAV performs its flight trajectory to start creating a continuous, moving window of synthetic apertures (of 4 secs each). Within each of its aperture window, it performs the following. It executes its concurrent ranging protocol to help classify nodes into various hops based on their reachability. It then localizes the hop_1 nodes first using its location-velocity solver that estimates both the location and velocity. Using the error residuals of the solver, it employs only high-confidence hop_1 nodes (static or uniform velocity) as anchors for localizing the hop_2 nodes. For the latter, it employs a conventional multi-lateration solver to obtain only the location of hop_2 nodes, which being instantaneous, is sufficient. Finally, it uses a spatial aperture of hop_2 nodes (anchors), along with an adaptive (temporal)

aperture from the UAV, to refine the location estimate of the hop_1 nodes that have non-uniform velocity.

Altitude Considerations. So far, we have only focused on the horizontal plane and assumed that TrackIO is aware of the nodes’ altitude. However, in cases, when TrackIO is not aware of the floor where the service personnel currently are, the horizontal localization error can be significant (since the algorithm will not take into account the additional vertical offset the signals have to travel). We address such situations by detecting the appropriate altitude (and hence floor) through a special one-time maneuver of the UAV. We move the UAV up and down through a short vertical distance that spans the target floors. During this movement, as the UAV approaches the horizontal plane of the nodes, its range estimates to the nodes should start to decrease, reach a minimum when it is on the plane, and increases when it moves away from the plane. TrackIO records the altitude (z^*) as that corresponding to the minimum range estimates and hence determines the floor of interest. TrackIO then uses this altitude to execute its localization process for the target floor.

4 Implementation and Testbed Setup

We build a custom payload consisting of a Decawave DW1000 UWB module and a Raspberry Pi 3 used as the TrackIO controller. The payload, weighting about 200 grams is mounted onboard a DJI Phantom 4 UAV platform. A fully charged UAV flight with our current prototype lasts for about 20 mins (≈ 25 mins without payload). In the following we describe the key hardware/software components that form TrackIO.

4.1 TrackIO Components

UWB Modules: The UWB module mounted on the UAV acts as the master node and is responsible for collecting ranging information from the client nodes. Alongside the DW1000 RF chip, the UWB module houses an ARM based microcontroller that runs our multi-hop ranging protocol (implemented in about 3000 lines of C code). The latter collects inter-node ranging information (at about 6 Hz) which is read by the controller Raspberry Pi and forwarded to the ground station through a WiFi interface.

Ground Station: Ground station refers to the compute node responsible for collecting ranging information obtained from the UAV and running TrackIO’s localization algorithms. First, it localizes the UAV using the four fixed client nodes on the ground. These nodes are placed at different heights (vertical diversity) on four vertices of a $5\text{ m} \times 5\text{ m}$ square to emulate a service vehicle housing the ground station. One node is equipped with a GPS receiver for an absolute location fix. Second, the UAV’s location is fed along with the rest of the range information that simultaneously solves all client node locations. We implement the solver algorithms in *Python* that run in real time on the ground station compute node (a Core i7 Lenovo laptop). We also implement an Android application that shows client node locations on a map.

Flight Automation: Automating the flight offers flexibility to programmatically control the flight’s trajectory as well as its speed. We use the Android Mobile SDK [12] provided by DJI to program two candidate trajectories for our UAV to follow: (a) STRAIGHT, a straight line path of length 30 m, and (b) WAVY, a sinusoidal path of length 30 m with an amplitude of 5 m (see figure 17). Note that such automation also helps us to re-run/repeat flights for controlled experiments, which would have been otherwise impossible in case of manually controlled flights.



Figure 15: Snapshots of trajectories marked with RFID tags. A volunteer is shown walking along the track with the RFID reader stick in her hand.

4.2 Testbed Setup

We deploy TrackIO with a single UAV in the third and fourth floor of our department building spanning approximately 1250 sq. meters (half the building’s floor area). Nine client nodes are placed indoors that mimic static or mobile first responders. Out of these nine nodes, six are in hop_1 (3 static, 3 mobile) and remaining three in the hop_2 (2 static, 1 mobile).

Obtaining location groundtruth: For static indoor nodes, the node location is accurately estimated with the help of a laser ranger. For tracking the mobile nodes’ groundtruth positions, we deploy fixed RFID tags on the ground, one every meter, along predefined trajectories. We create portable *RFID reader sticks* equipped with a ThingMagic M6E-Nano readers (Fig. 15). We adjust the reader’s transmit power as well as the antenna orientation to limit the reading range to about 50 cm. The volunteers are instructed to move along the trajectories while holding the stick vertically. The stick also hosts a Raspberry Pi that controls the reader and logs timestamped entries of RFID tags along with RSS/phase information (≈ 10 Hz) it reads along the trajectory. We post-process such logs to obtain accurate estimations of position (within 20 cm) and velocity of the mobile node at granular timescales.

Trajectories: We lay out the trajectories within our office area spanning multiple rooms, cubicles, hallways and open spaces. Specifically we construct four different trajectories, three in the first hop and one in the second hop. We create the trajectories with increasing number of turns in them. The first three trajectories are (a) *LINE*, a linear trajectory of length 20 m, (b) *TRIANGLE*, a triangular trajectory with a perimeter of 30 m and (c) *RECT*, a rectangular trajectory with a perimeter

of 40 m. The trajectory in the second hop is roughly a 30 m long sinusoidal path (SINU).

5 Evaluation

We present evaluation results from experiments conducted in our real testbed discussed in §4.2. Recall that we use 6 nodes (3 static, 3 mobile) in hop_1 and 3 nodes (2 static, 1 mobile) in hop_2 . Four volunteers (mobile nodes) are simultaneously instructed to move along their designated trajectories at different speeds. Combined, we accumulate over 2 hours worth of traces accounting for over 10+ Kms of total trajectory length. Evaluating TrackIO’s performance (w.r.t. groundtruth) through controlled experiments requires us to do trace-driven analysis of the ranging information logged by the ground station compute node. However, our system receiving range information at 6 Hz is capable of *real time* operations. In §5.4, we highlight end-to-end latency of our system for various node distributions. Fig. 16 shows the median localization error for both hop_1 and hop_2 nodes, the latter localized using static or mobile hop_1 nodes. While static nodes are localized with an accuracy of 1 – 1.5 m, note that even for mobile nodes, the median localization error is a little less than 2 m (hop_1) to around 2.5 m (hop_2). In extreme cases, where the hop_2 nodes are localized using all mobile hop_1 nodes and the latter do not offer a good spatial diversity (e.g., all hop_1 nodes are in close vicinity), TrackIO still offers a localization accuracy of about 4 m (top 10 percentile). However such situations can be avoided by judiciously selecting nodes in hop_1 that offer spatial diversity.

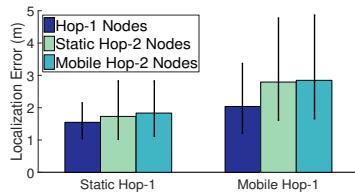


Figure 16: Overall localization accuracy for static and mobile hop_1 and hop_2 nodes

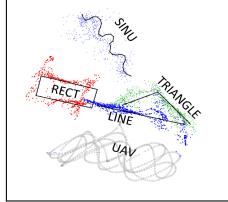


Figure 17: Scatter plot of estimated trajectories for all mobile nodes

5.1 Hop1 Localization Performance

We now evaluate the hop_1 localization error over the dimensions of node speed, trajectory, and turns.

Effect of node speed. Since TrackIO jointly solves for both location and velocity of each node, ideally, the localization error should be independent of the velocity for nodes moving at a constant velocity. However, when moving at a brisk pace, the human body performs a complex set of movements, including bobbing of the head, which strains the constant velocity assumption. To evaluate these practical limitations, a volunteer moved along LINE at different speeds—a stroll, walk, and a run. Fig. 18 shows the resulting localization accuracy. The reported speeds are the average speed obtained from ground-truth. The median localization accuracy is around 1.5 m during the stroll, whereas it is around 2.8 m during the

run. The gains over simple multilateration are above $3\times$ for all the velocities.

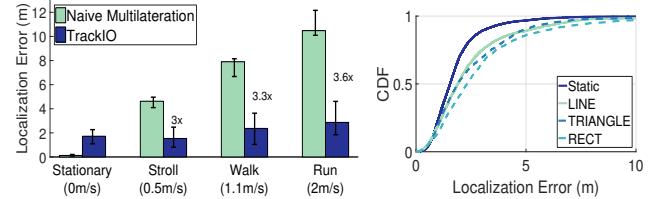


Figure 18: TrackIO gains significantly over simple multilateration by solving for velocity vectors as well. Figure 19: The three trajectories have similar localization error.

Effect of trajectory shapes. Fig. 19 shows the impact of different trajectory shapes (LINE, TRIANGLE, RECT) on the localization accuracy. While using naive multilateration, the localization errors can spike upto 10 m, TrackIO makes the system resilient to turns (median ≈ 2 m for all trajectories). Minor differences do exist which can be explained by the increasing number of turns present in the respective trajectories.

Effect of Turns. Human motion mostly comprises of straight lines interspersed with turns of various degrees. Fig. 20 shows the localization errors during turning events versus that during traversal in straight line segments. Note that the errors can significantly spike during such turning events ($3\times$ at 80 %ile).

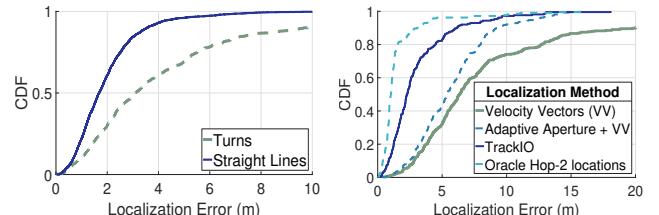


Figure 20: Localization error worsens during turns

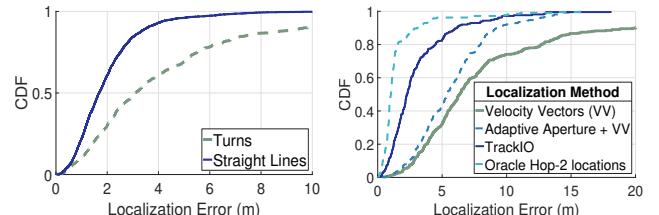


Figure 21: Benefit of spatial aperture vs. adaptive aperture

TrackIO tackle such cases through adaptively changing the aperture and spatial aperture offered by hop_2 node positions to fix the erring hop_1 nodes as described in §3. Fig. 21 shows the effectiveness of these approaches in our testbed and will be referred to in the following analysis.

Benefit of Adaptive Aperture. Fig. 22 shows a time series of localization error when a turning event occurs. The localization error increases sharply and remains large while the aperture slowly moves over this point in time. Our adaptive aperture dynamically resets historical measurements in case it detects turning events. It improves localization accuracy by a factor of $1.8\times$, and also reduces the time (*early recovery* ≈ 5 secs) it takes to stabilize the localization performance.

Benefit of spatial aperture from hop_2 nodes. Opportunistic re-fixing of a turning hop_1 node, T might be possible if there are at least three hop_2 nodes that do not depend on T for their localization. Fig. 21 shows the reduction in localization error when a fast moving and turning node is subjected to hop_2 guided re-fix. As an example, we use the entire 2 m/s

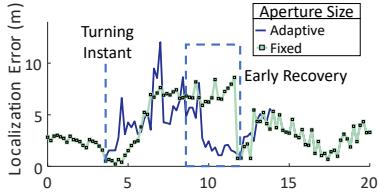


Figure 22: Adaptive aperture recovers from the effects of a turn earlier than fixed aperture.

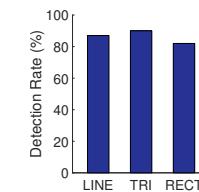


Figure 23: TrackIO correctly detects more than 80% of turns.

run shown in Fig. 18 *including the turns* at either ends. The velocity-vector based localization performs poorly as seen from Fig. 21. We use 3 other static hop_1 nodes to localize 3 static hop_2 nodes and then use their locations to solve for the location of the running node, T . We observe about $2.8 \times$ improvement in the median localization accuracy after re-fixing. This re-fixing accuracy is affected by two factors: (a) ranging error, and, (b) imprecise location of the hop_2 nodes. We can eliminate the effect of imprecise locations and hypothetically study just the ranging error effect by assuming ground truth hop_2 node locations are known—as if given by an Oracle. Fig. 21 shows this error to be within 2m at the 75%ile. While extremely promising, the re-fix approach may not always be available depending on the current topology. In comparison, the adaptive aperture technique is always available for any hop_1 node. Fig. 21 puts both these approaches (adaptive aperture and re-fix) into perspective.

Effect of drone trajectory and velocity. Fig. 24 shows the localization precision of two different trajectories, STRAIGHT and WAVY, at two different drone speeds. In general, geometric diversity of measurements helps obtain better localization. Therefore, the faster the drone moves, the better is the localization. Similarly, a WAVY pattern of drone movement also helps in obtaining better localization even for lower speeds. We therefore fly the drone in a WAVY pattern for this evaluation.

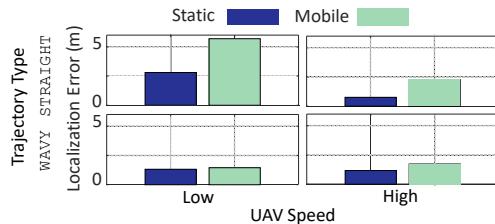


Figure 24: A WAVY trajectory provides better localization accuracy due to increase in spatial diversity.

5.2 Hop₂ Localization Performance

Effect of hop₁ mobility. We perform instantaneous localization³ of hop_2 nodes based on 3 hop_1 nodes. By selecting which 3 nodes to use for this purpose, we obtain a combination of static and mobile hop_1 nodes—ranging from all static to all mobile. Fig. 25 shows the impact on localization errors as we allow an increasing number of hop_1 nodes to

be mobile. These results show the error-span between using only static hop_1 nodes (1.83 m) to using only mobile hop_1 nodes (2.84 m). If multiple hop_1 nodes are available, choice of anchors influences hop_2 localization error (Fig. 26). Due to instantaneous localization of hop_2 nodes, the accuracies for both static and mobile hop_1 nodes are similar (see Fig. 16).

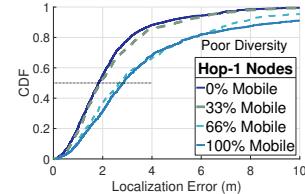


Figure 25: Localization error for poor diversity as a function of the number of mobile hop_1 nodes.

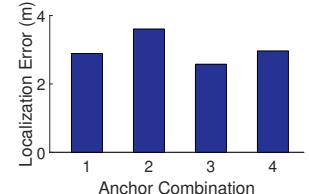


Figure 26: Selection of an anchor combination influences localization error.

Effect of hop₁ diversity. The relative locations of hop_1 nodes also affect the localization accuracy. We consider random static snapshots of hop_1 node locations moving on the three trajectories (LINE, TRIANGLE, and RECT) and further localize hop_2 nodes. Figure 27 shows a long tail indicating that some hop_1 position combinations perform poorly. A further analysis of such failing combinations reveals that hop_1 nodes are nearly collinear⁴ in such cases, causing very high dilution of precision [8, 34, 56] (Fig. 28). We expect such situations to be minimal and short lived in real-life.

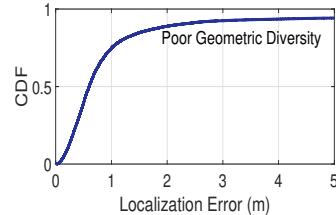


Figure 27: hop_2 localization error as a function of various hop_1 locations. Localization fails about 10% times.

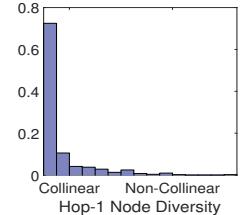


Figure 28: Most failure cases are caused by highly collinear hop_1 nodes.

5.3 UAV Localization

Note that the final localization accuracy of the mobile nodes is tied to absolute location fixes for the UAV. We study the impact of such accuracy as a function of the different modalities we can localize the UAV through (viz, COTS GPS receiver [1], UAV’s GPS with sensor fusion [13], UWB based). Fig. 29 shows the error in each of these modalities compared to laser ranger based groundtruth (accurate to 1mm). Fig. 30 shows the improvement in localization error of a static indoor node (1 m using UWB versus 2+ m using GPS) when UWB based drone localization is used. Note that the UAV’s large trajectory and the low vertical diversity in ground-UWB anchors degrades the drone’s location accuracy. Yet, in GPS chal-

³Instantaneous localization does *not* depend on UAV’s synthetic aperture created in time.

⁴We define collinearity as the ratio of the height and the base of the triangle formed by the three hop_1 nodes.

lenged situations, such as in a dense urban space, UWB based drone localization will remain valuable.

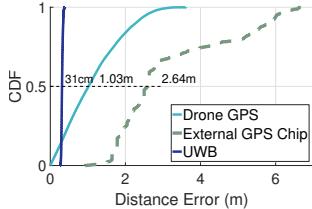


Figure 29: Range error between two fixed locations on the ground for different modalities.

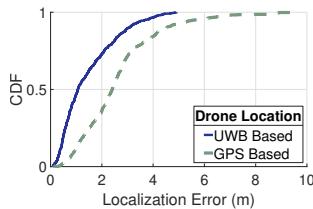


Figure 30: Effect of UAV localization modalities on static hop_1 localization accuracy.

5.4 TrackIO End-to-End Latency

We present TrackIO’s end-to-end latency for different topologies in Figure 31. Each topology shown assumes an additional 4 hop_1 nodes on the ground for UAV localization. Thus, topology A consists of $4 + 6 = 10 \text{ hop}_1$, and 3 hop_2 nodes. With even 20 nodes TrackIO leaves room for real-time operations.

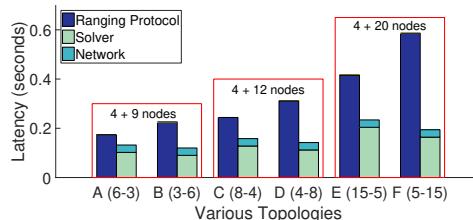


Figure 31: Protocol latency as a function of the network topology

5.5 Adding IMU: A What-If Analysis

Adding an IMU to our implementation might improve TrackIO’s performance due to availability of another estimation for velocity and direction. We show in Fig. 32 and Fig. 33 that while improvement in performance are possible when accurate direction and velocity information is available, presence of small errors in those estimates substantially reduce the gain over TrackIO. We leave more sophisticated IMU-based implementation to future work.

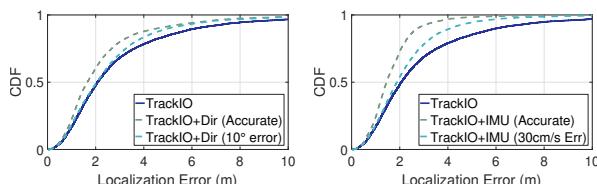


Figure 32: Improvement using direction from IMU

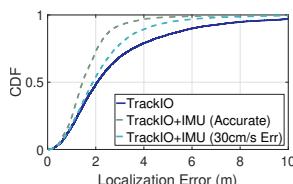


Figure 33: Improvement using velocity from IMU

6 Related Work

Indoor localization. A significant amount of work exists in indoor localization [5, 33, 42, 51, 54, 55, 57], but most of it relies on indoor infrastructure and fingerprinting. Both of these are not available in our target application. Techniques that use commodity WiFi [28, 31] rely on the difference in subcarrier phases. However, subcarrier phase wraps after a short distance ($7.5 - 15$ m) rendering them unsuitable in our application.

Localizing from outside a building could be performed with RF sensing [2, 3], however, we require a system that would be robust to changing multipath, fast human mobility, and would penetrate deep into a building. Use of inertial sensors (IMU) for tracking human motion [42, 44, 52] has been extensively studied. However, most of these systems suffer from drifts and saturation introduced by the IMU sensors [18, 30, 46]. In contrast to IMU-based tracking, the ranging approach we take in TrackIO is not based on dead-reckoning and instead provides instantaneous location.

UWB based localization. UWB radios are increasingly being used for localization solutions in a variety of applications from positioning [37, 40], to tracking industrial objects [17, 47], to sports analytics [18]. UWB is particularly resistant to indoor multipath [41, 43, 45] due to its 1ns time resolution (1GHz bandwidth). This makes it suitable for use in indoor spaces where multipath can be rampant. Different UWB platforms are commercially available today [7, 10, 48], and we chose Decawave Trek1000 UWB platform for its superior performance [27, 43]. Most of these works assume some static UWB anchors. In our application however, we have no pre-deployed anchors, but create a synthetic aperture over time by flying an anchor on a UAV. Some recent works [32, 49] have explored use of the multipath profile as virtual anchors localize using a single UWB device. However, they assume the location of all strong reflectors are known, making it prone to issues when multipath could change, due to moving people or objects. In contrast, our technique does not depend on the knowledge of the floor-plan, and is robust to changing multipath profile.

Localization of UAV. UAV localization has been extensively studied and approaches range from using a single GPS [13, 14], to using differential GPS [19], to using complex motion models based on the drone’s IMU data. UAV localization using UWB has been proposed in [6, 29]. We incorporate UAV localization into TrackIO protocol. Authors of [15, 23] also consider UAVs as a vehicle for fire-fighting, though they do not discuss the outdoor-indoor localization problem.

7 Conclusion

Indoor localization without any support from the building’s infrastructure is a challenging yet important problem. Particularly of importance to first responders, continuous real-time tracking can be a life saver in many everyday situations. TrackIO uses a UAV to create the missing infrastructure *outside* the building and performs continuous ranging with indoor nodes. Through numerous algorithmic, architectural, and engineering modifications to trilateration and ranging protocols we obtain promising results localizing mobile indoor nodes accurate to about 2m from twenty meters outside the building. We believe TrackIO is a promising first step in active localization from outside the building. While TrackIO provides a fully working system where none exists today, we plan to continue to explore avenues to further improve accuracy, resilience, and redundancy.

References

- [1] Adafruit ultimate gps breakout. <https://www.adafruit.com/product/746>.
- [2] ADIB, F., HSU, C.-Y., MAO, H., KATABI, D., AND DURAND, F. Capturing the human figure through a wall. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 219.
- [3] ADIB, F., KABELAC, Z., AND KATABI, D. Multi-person localization via rf body reflections. In *NSDI* (2015), pp. 279–292.
- [4] BAHL, P., AND PADMANABHAN, V. N. Radar: An in-building rf-based user location and tracking system. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2000), vol. 2, Ieee, pp. 775–784.
- [5] BAHL, P., AND PADMANABHAN, V. N. Radar: An in-building rf-based user location and tracking system. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2000), vol. 2, Ieee, pp. 775–784.
- [6] BENINI, A., MANCINI, A., AND LONGHI, S. An imu/uwb/vision-based extended kalman filter for mini-uav localization in indoor environment using 802.15. 4a wireless sensor network. *Journal of Intelligent & Robotic Systems* 70, 1-4 (2013), 461–476.
- [7] BE SPOON. Precise location rtls. <http://bespoon.com/technology/precise-location-rtls/>.
- [8] BORRE, K., AKOS, D. M., BERTELSEN, N., RINDER, P., AND JENSEN, S. H. *A software-defined GPS and Galileo receiver: a single-frequency approach*. Springer Science & Business Media, 2007.
- [9] CHINTALAPUDI, K., PADMANABHA IYER, A., AND PADMANABHAN, V. N. Indoor localization without the pain. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking* (2010), ACM, pp. 173–184.
- [10] DECAWAVE. Decawave. <http://www.decawave.com/>.
- [11] DECAWAVE. DW1000 User Manual. <https://decawave.com/content/dw1000-user-manual>.
- [12] DEVELOPER, D. Mobile sdk. <https://developer.dji.com/mobile-sdk/documentation/introduction/index.html>.
- [13] DJI. DJI Phantom 4 Specs.
- [14] DJI. Mavic pro specs. <https://www.dji.com/mavic/info#specs>.
- [15] DUEWEL, E., AND STODDARD, S. After dark, drone is 'best friend a firefighter could have'. <https://www.usnews.com/news/best-states/oregon/articles/2018-08-11/after-dark-drone-is-best-friend-a-firefighter-could-have>.
- [16] FAHY, R. F., LEBLANC, P. R., AND MOLIS, J. L. Firefighter fatalities in the united states - 2017. *NFPA* (2018).
- [17] FERNANDEZ-MADRIGAL, J.-A., CRUZ-MARTIN, E., GONZALEZ, J., GALINDO, C., AND BLANCO, J.-L. Application of uwb and gps technologies for vehicle localization in combined indoor-outdoor environments. In *ISSPA* (2007), pp. 1–4.
- [18] GOWDA, M., DHEKNE, A., SHEN, S., CHOUDHURY, R. R., YANG, L., GOLWALKAR, S., AND ESSANIAN, A. Bringing iot to sports analytics. In *NSDI* (2017), pp. 499–513.
- [19] GOWDA, M., MANWEILER, J., DHEKNE, A., CHOUDHURY, R. R., AND WEISZ, J. D. Tracking drone orientation with multiple gps receivers. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking* (2016), ACM, pp. 280–293.
- [20] GUTIERREZ, J. A., CALLAWAY, E. H., AND BARRETT, R. *IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks*. IEEE Standards Office, New York, NY, USA, 2003.
- [21] HEDGE COCK, W., MAROTI, M., LEDECZI, A., VOLGYESI, P., AND BANALAGAY, R. Accurate real-time relative localization using single-frequency gps. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems* (2014), ACM, pp. 206–220.
- [22] HEDGE COCK, W., MAROTI, M., SALLAI, J., VOLGYESI, P., AND LEDECZI, A. Regtrack: a differential relative gps tracking solution. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services* (2013), ACM, pp. 475–476.
- [23] HRABIA, C.-E., HESSLER, A., XU, Y., BREHMER, J., AND AL-BAYRAK, S. Effeu project: Efficient operation of unmanned aerial vehicles for industrial fire fighters. In *Proceedings of the 4th ACM Workshop on Micro Aerial Vehicle Networks, Systems, and Applications* (New York, NY, USA, 2018), DroNet'18, ACM, pp. 33–38.
- [24] HUGHES, W. J. T. C. N. T., AND TEAM, E. Global positioning system (gps) standard positioning service (sps) performance analysis report. *WAAS FAA*, 101 (2018).
- [25] IBRAHIM, M., LIU, H., JAWAHAR, M., NGUYEN, V., GRUTESER, M., HOWARD, R., YU, B., AND BAI, F. Verification: Accuracy evaluation of wifi fine time measurements on an open platform. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (2018), ACM, pp. 417–427.
- [26] IRISH, A., ISAACS, J., ILAND, D., HESPAÑA, J., BELDING, E., AND MADHOW, U. Shadowmaps, the urban phone tracking system. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 283–286.
- [27] JIMÉNEZ, A. R., AND SECO, F. Comparing decawave and bespoon uwb location systems: Indoor/outdoor performance analysis. In *IPIN* (2016), pp. 1–8.
- [28] JOSHI, K. R., BHARADIA, D., KOTARU, M., AND KATTI, S. Videow: Fine-grained device-free motion tracing using rf backscatter. In *NSDI* (2015), pp. 189–204.
- [29] KEMPKE, B., PANNUTO, P., AND DUTTA, P. Polypoint: Guiding indoor quadrotors with ultra-wideband localization. In *Proceedings of the 2nd International Workshop on Hot Topics in Wireless* (2015), ACM, pp. 16–20.
- [30] KOK, M., HOL, J. D., AND SCHÖN, T. B. Using inertial sensors for position and orientation estimation. *arXiv preprint arXiv:1704.06053* (2017).
- [31] KOTARU, M., JOSHI, K., BHARADIA, D., AND KATTI, S. Spotfi: Decimeter level localization using wifi. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 269–282.
- [32] KULMER, J., HINTEREGGER, S., GROSSWINDHAGER, B., RATH, M., BAKR, M. S., LEITINGER, E., AND WITRISAL, K. Using decawave uwb transceivers for high-accuracy multipath-assisted indoor positioning. In *Communications Workshops (ICC Workshops), 2017 IEEE International Conference on* (2017), IEEE, pp. 1239–1245.
- [33] KUMAR, S., GIL, S., KATABI, D., AND RUS, D. Accurate indoor localization with zero start-up cost. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (2014), ACM, pp. 483–494.
- [34] MASSATT, P., AND RUDNICK, K. Geometric formulas for dilution of precision calculations. *Navigation* 37, 4 (1990), 379–391.
- [35] NI, L., WANG, Y., TANG, H., YIN, Z., AND SHEN, Y. Accurate localization using lte signaling data. In *2017 IEEE International Conference on Computer and Information Technology (CIT)* (Aug 2017), pp. 268–273.
- [36] PALACIOS, J., CASARI, P., AND WIDMER, J. Jade: Zero-knowledge device localization and environment mapping for millimeter wave systems. In *INFOCOM 2017-IEEE Conference on Computer Communications*, IEEE (2017), IEEE, pp. 1–9.
- [37] PANNUTO, P., KEMPKE, B., CHUO, L.-X., BLAAUW, D., AND DUTTA, P. Harmonium: Ultra wideband pulse generation with band-stitched recovery for fast, accurate, and robust indoor localization. *ACM Trans. Sen. Netw.* 14, 2 (June 2018), 11:1–11:29.
- [38] PARKINSON, B., AND ENGE, P. Differential gps. In *Global Positioning System: Theory and Applications* (1996), American Institute of Aeronautics and Astronautics, pp. 3–50.

- [39] PIX4D. Do rtk/ppk drones give you better results than using gcps? <https://pix4d.com/rtk-ppk-drones-gcp-comparison/>.
- [40] PROROK, A., ARFIRE, A., BAHR, A., FARSEROTU, J. R., AND MARTINOLI, A. Indoor navigation research with the khepera iii mobile robot: An experimental baseline with a case-study on ultra-wideband positioning. In *2010 International Conference on Indoor Positioning and Indoor Navigation* (Sept 2010), pp. 1–9.
- [41] PROROK, A., TOMÉ, P., AND MARTINOLI, A. Accommodation of nlos for ultra-wideband tdoa localization in single- and multi-robot systems. In *2011 International Conference on Indoor Positioning and Indoor Navigation* (Sept 2011), pp. 1–9.
- [42] RAI, A., CHINTALAPUDI, K. K., PADMANABHAN, V. N., AND SEN, R. Zee: Zero-effort crowdsourcing for indoor localization. In *Proceedings of the 18th annual international conference on Mobile computing and networking* (2012), ACM, pp. 293–304.
- [43] RUIZ, A. R. J., AND GRANJA, F. S. Comparing ubisense, bespoon, and decawave uwb location systems: Indoor performance analysis. *IEEE Transactions on Instrumentation and Measurement* 66, 8 (Aug 2017), 2106–2117.
- [44] RUIZ, A. R. J., GRANJA, F. S., HONORATO, J. C. P., AND ROSAS, J. I. G. Accurate pedestrian indoor navigation by tightly coupling foot-mounted imu and rfid measurements. *IEEE Transactions on Instrumentation and measurement* 61, 1 (2012), 178–189.
- [45] SCZYSLO, S., SCHROEDER, J., GALLER, S., AND KAISER, T. Hybrid localization using uwb and inertial sensors. In *2008 IEEE International Conference on Ultra-Wideband* (Sept 2008), vol. 3, pp. 89–92.
- [46] SHEN, G., CHEN, Z., ZHANG, P., MOSCIBRODA, T., AND ZHANG, Y. Walkie-markie: Indoor pathway mapping made easy. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (2013), USENIX Association, pp. 85–98.
- [47] SPIEKER, S., AND ROHRIG, C. Localization of pallets in warehouses using wireless sensor networks. In *2008 16th Mediterranean Conference on Control and Automation* (June 2008), pp. 1833–1838.
- [48] UBISENSE. Ubisense solutions. <http://www.ubisense.net/ubisense-solutions>.
- [49] VAN DE VELDE, S., AND STEENDAM, H. Cupid algorithm for cooperative indoor multipath-aided localization. In *Indoor Positioning and Indoor Navigation (IPIN), 2012 International Conference on* (2012), IEEE, pp. 1–6.
- [50] VAN DIGGELEN, F., AND ENGE, P. The worlds first gps mooc and worldwide laboratory using smartphones. In *Proceedings of the 28th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2015)* (2015), pp. 361–369.
- [51] VASISHT, D., KUMAR, S., AND KATABI, D. Decimeter-level localization with a single wifi access point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 165–178.
- [52] WANG, H., SEN, S., ELGOHARY, A., FARID, M., YOUSSEF, M., AND CHOUDHURY, R. R. No need to war-drive: Unsupervised indoor localization. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 197–210.
- [53] WANG, H., SEN, S., MARIAKAKIS, A., ROY CHOUDHURY, R., ELGOHARY, A., FARID, M., AND YOUSSEF, M. Unsupervised indoor localization. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 499–500.
- [54] XIONG, J., AND JAMIESON, K. Arraytrack: a fine-grained indoor location system. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 71–84.
- [55] YANG, Z., WU, C., AND LIU, Y. Locating in fingerprint space: wireless indoor localization with little human intervention. In *Proceedings of the 18th annual international conference on Mobile computing and networking* (2012), ACM, pp. 269–280.
- [56] YARLAGADDA, R., ALI, I., AL-DHAHIR, N., AND HERSHY, J. Gps gdop metric. *IEE Proceedings-radar, sonar and navigation* 147, 5 (2000), 259–264.
- [57] YOUSSEF, M., AND AGRAWALA, A. The horus wlan location determination system. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (2005), ACM, pp. 205–218.

3D Backscatter Localization for Fine-Grained Robotics

Zhihong Luo

Qiping Zhang

Yunfei Ma

Manish Singh

Fadel Adib

MIT Media Lab

Abstract – This paper presents the design, implementation, and evaluation of TurboTrack, a 3D localization system for fine-grained robotic tasks. TurboTrack’s unique capability is that it can localize backscatter nodes with sub-centimeter accuracy without any constraints on their locations or mobility. TurboTrack makes two key technical contributions. First, it presents a pipelined architecture that can extract a sensing bandwidth from every single backscatter packet that is three orders of magnitude larger than the backscatter communication bandwidth. Second, it introduces a Bayesian space-time super-resolution algorithm that combines time series of the sensed bandwidth across multiple antennas to enable accurate positioning. Our experiments show that TurboTrack simultaneously achieves a median accuracy of sub-centimeter in each of the x/y/z dimensions and a 99th percentile latency less than 7.5 milliseconds in 3D localization. This enables TurboTrack’s real-time prototype to achieve fine-grained positioning for agile robotic tasks, as we demonstrate in multiple collaborative applications with robotic arms and nanodrones including indoor tracking, packaging, assembly, and handover.

1 Introduction

The emergence of agile and miniature robots has led to a novel set of capabilities and sensing tasks. Nanodrones that can fit in your palm are used for mapping indoor environments and are deployed in swarms for emergency response and hazard detection in urban settings [29, 67, 43]. Dexterous robotic arms have shifted manufacturing automation from assembly lines that consist of dozens of robots, each of which is dedicated to a single task, to a pair of multi-functional robots that can collaborate on picking up, assembling, and packaging items [16, 25]. Personal robots like the Roomba are already cleaning our homes, and their roles are expected to expand to folding clothes, washing dishes, and helping with other daily routines [4, 47, 21].

A fundamental challenge that still faces agile robots, however, is their ability to operate in highly cluttered settings [28, 42, 27]. While highly-trained vision systems can perform accurate classification and tracking tasks, their performance suffers in cluttered environments, and fails if the object of interest is fully occluded, e.g., if a robot must pick up an item from under a pile [74, 48]. Moreover, tracking individual nanodrones in a swarm is challenging even in line-of-sight settings due to their

constrained size and payload, which prevent instrumenting them with visually identifiable markers [33, 44].

RF-based identification and localization offers an alternative sensing modality that is highly robust to visual clutter, providing an attractive solution and a complementary sensing capability. Motivated by the recent advances in RF-based localization by the networking community [68, 45, 60, 50, 72], in this paper, we set out to build a system for RF-based identification and 3D localization for fine-grained robotic tasks. Building such a system requires meeting requirements along three fronts:

- **Accuracy:** To enable agile manipulation tasks, like grasping and packaging, we need to achieve sub-centimeter localization accuracy [57, 18]. Such accuracy is needed to enable a robot to align its grip with an object for item grasping and manipulation tasks.
- **Mobility:** Robotic arms and nanodrones are in constant mobility as they perform sensing and localization. Hence, a localization system for fine-grained robotic tasks must be fast enough to track them and deal with random mobility patterns.
- **Scalability:** Since robots are expected to manipulate everyday items, we need cost-effective solutions that scale to hundreds or thousands of items, even in relatively confined areas like homes or small businesses.

Unfortunately, no system exists today that can realize all three goals simultaneously. On one hand, WiFi and Bluetooth-based solutions [34, 73, 68] are not scalable in our context since it is not feasible or cost-effective to tag every item with a Bluetooth or WiFi radio. On the other hand, billions of manufactured items are already tagged with few-cent RFIDs, making RFID-based localization attractive from a scalability standpoint. However, RFID localization solutions are limited either in their accuracy or in their ability to deal with mobility. Techniques like Tagoram [75] and RFIDraw [71] can work with mobile RFIDs, but they have decimeter-scale accuracy in obtaining a tag’s exact position;¹ this prevents using them for tasks like grasping or manipulation. Others like MobiTagBot [60] and RFCompass [69] achieve high accuracy but require the tag to remain static for multiple seconds as they perform their localization; this prevents them from tracking nanodrones or enabling agile tasks

¹These techniques track *changes in distance* so they can accurately recover the shape of a trajectory, but relatively low accuracy in obtaining the exact position.

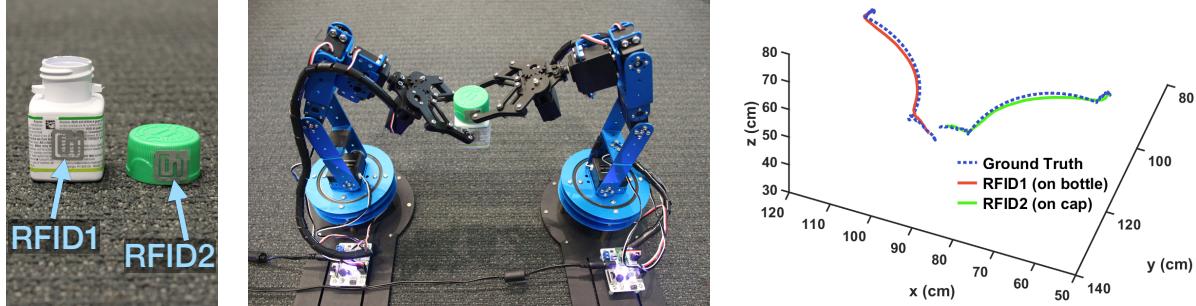


Figure 1: **TurboTrack in an Assembly Task.** Left figure shows a bottle and its cap tagged with RFIDs. Middle figure shows two robots collaborating on assembly, where one robot puts a cap on a bottle carried by another. Right figure shows TurboTrack’s tracking output as the robots move the cap (in green) and bottle (in red) into position for assembly and compares it to a ground-truth vision-based system (in blue) to show its accuracy.

where a robot needs to move and manipulate the object of interest simultaneously.

The main contribution of this paper is to build an RF localization system that can achieve all the above requirements. Our system, TurboTrack, introduces two main innovations that together allow it to achieve the high accuracy and unconstrained mobility needed to deliver fine-grained robotic tasks:

(a) One-Shot Wideband Estimation: TurboTrack’s first component allows it to estimate an RFID’s response over a wide bandwidth – one that is three orders of magnitudes larger than the backscatter communication bandwidth. The large bandwidth can be used to mitigate reflections from other objects in the environment and isolate the RFID’s response. This component is inspired by past work that performs frequency hopping for wideband estimation [45]. In contrast to this past work, which needs every RFID to remain static as it repeatedly queries it dozens of times while hopping frequencies, TurboTrack can estimate the wide bandwidth in one shot from every single RFID response. To do so, it introduces a wideband localization helper which acts like a radar. The helper transmits a wideband signal, and measures its reflection off different objects in the environment, including the RFID. In §3, we describe how TurboTrack constructs the helper’s wideband signal to be compatible with the RFID protocol, and how it synchronizes the helper with an RFID reader to isolate the RFID’s reflection and estimate its wideband channel from every single response. This one-shot estimation component enables TurboTrack to operate correctly with moving targets.

(b) Bayesian Space-Time Super-Resolution: In principle, if one could estimate multiple GHz of bandwidth off an RFID using the above technique, then we could apply standard ultra-wideband ranging methods to directly localize a tag. Unfortunately, the bandwidth that can be estimated from an RFID remains limited by the RFID’s antenna response and impedance matching circuitry. Specifically, because of their designs that optimize for energy harvesting efficiency, the ability to sense an RFID’s channel significantly degrades beyond a cou-

ple hundred MHz, even if we perform frequency hopping [45]. Such bandwidth is still an order of magnitude lower than that required for sub-centimeter localization using ultra-wideband techniques [56, 14].

TurboTrack’s second innovation is a space-time super-resolution algorithm that overcomes this challenge. Its key insight is that while few hundred MHz of bandwidth cannot enable sub-centimeter positioning, they narrow down the potential locations of an RFID to a handful of candidates. By combining these candidates over space (multiple antennas) and time, TurboTrack zooms in on the exact location. In §4, we formalize this as a Gaussian mixture problem and incorporate it into a Bayesian framework that fuses spatio-temporal bandwidth measurements. Further, to deal with the nonlinear nature of the measurements, we introduce an approximate inference algorithm that exploits RF and geometric properties of the underlying estimators to design a computationally efficient solution for highly accurate positioning.

We built a prototype of TurboTrack using USRP X310 software radios and tested it with off-the-shelf, battery-free RFIDs. Our evaluation with over a million location measurements demonstrates that TurboTrack can achieve sub-centimeter localization accuracy in each of the x/y/z dimensions. TurboTrack’s 99th percentile error remains lower than 2-cm in each of the dimensions, while retaining a 99th percentile latency smaller than 7.5 milliseconds in 3D localization. Further, we compared its performance to two state-of-the-art proposals, RFind [45] and RFIDraw [71]. Our results show that TurboTrack achieves two to three orders of magnitude improvement in localization accuracy of moving targets.

Finally, to demonstrate TurboTrack’s capability in mobile and accurate positioning, we tested it in two classes of agile and fine-grained robotic tasks. First, we show how it can accurately track collaborative tasks between robotic arms including packaging, handover, and assembly. Second, we demonstrate how it can accurately track nanodrones as they fly in indoor environments.

Contributions. TurboTrack’s contributions are:

- The first system architecture that performs one-shot

- wideband estimation from every backscatter packet, enabling low-latency and high-precision localization.
- A spatio-temporal Bayesian framework for RF localization that fuses time series of bandwidth and phase measurements across multiple antennas.
 - An approximate inference algorithm with fast convergence time for RF localization. The algorithm achieves computational efficiency by incorporating the properties of RF signals and the geometric nature of its measurements.
 - A real-time prototype implementation and evaluation demonstrating the system’s ability to track fine-grained robotic tasks performed using robotic arms and nanodrones.

We note that our current implementation inherits some of the limitations of RFIDs. Most importantly, its range of operation is limited by a reader’s ability to power up battery-free tags, which is typically within less than 10 m. However, this limitation is not inherent to our design since both TurboTrack’s architecture and algorithms are general to any backscatter sensor and do not stop at RFIDs. For example, they could work with battery-assisted or solar-powered tags, which would enable long-range communication [56, 55]. Such tags may be attached to nanodrones or robotic arms to track the robots themselves over tens to hundreds of meters and not just the items they manipulate.

2 Design Overview

TurboTrack is a system that enables ultra-low latency, very high accuracy localization of backscatter sensors for fine-grained robotic applications. TurboTrack’s localization works both in line-of-sight and through occlusions. Further, TurboTrack can operate with inexpensive backscatter sensors like off-the-shelf RFIDs without requiring any hardware modifications, and it is fully compatible with today’s standard UHF RFID protocol. The sensors can be attached as stickers to objects of interest (e.g., manufacturing items) for object manipulation or to miniature robots like nanodrones for tracking.

Architecturally, TurboTrack combines a standard RFID reader with a wideband localization helper as shown in Fig. 2. The helper transmits wideband signals, and captures their reflections off different objects in the environment. The combination of a standard reader with a wideband helper enables both identification (through the RFID’s identifier) and accurate localization (using the wideband signals). To deliver both of these tasks, TurboTrack’s centralized controller synchronizes the helper’s signals with the RFID reader at the physical layer, and, at the protocol level, it incorporates the helper’s operation into the reader’s finite state machine.

Algorithmically, TurboTrack leverages the wideband channel estimates for localization. It fuses estimates

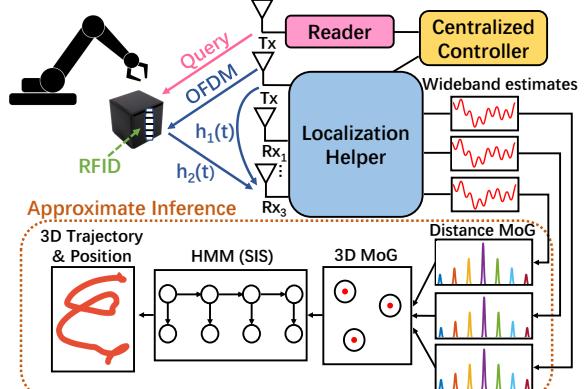


Figure 2: System Architecture. TurboTrack combines a reader with a localization helper to obtain wideband estimates. It fuses the estimates across space and time via a Bayesian framework, and solves for accurate 3D positions using approximate inference. MoG refers to Mixture of Gaussians, and SIS refers to Sequential Importance Sampling.

across multiple receive antennas of the localization helper and across time through a spatio-temporal Bayesian framework. It models each antenna’s measurements as a Gaussian mixture, and solves for each tag’s location and trajectory through an approximate inference algorithm customized for backscatter localization. The algorithm linearizes and approximates the antenna measurements in 3D and propagates their beliefs through a particle filter. The resulting accuracy is high enough to enable TurboTrack to track fine-grained robotic tasks.

The next sections describe TurboTrack’s operation at the architectural (§3) and the algorithmic levels (§4).

3 One-Shot Wideband Estimation

In this section, we describe how TurboTrack’s helper can obtain wideband estimates from every single (narrow-band) RFID response. The one-shot estimation enables it to track changes in the channel at very high speeds. Then, in §4, we describe how it uses the wideband estimates for accurate localization.

3.1 Primer on Backscatter Modulation

In backscatter networking, a wireless device called a reader starts a communication session by sending a signal on the downlink. A backscatter sensor, e.g., an RFID, harvests energy from this signal and powers up. To communicate with the reader, the sensor switches between two states: reflective and non-reflective, to transmit bits of zeroes and ones.

Our past work has observed that backscatter modulation is frequency agnostic [45], meaning that RFIDs not only modulate the reader’s signal but also all transmitted signals in the environment. This enables us to estimate an RFID’s channel out-of-band. In particular, as an RFID backscatters the reader’s signal, we can transmit an unmodulated wave at another sensing frequency and estimate the RFID’s channel at that frequency. By hopping the sensing frequency across successive RFID

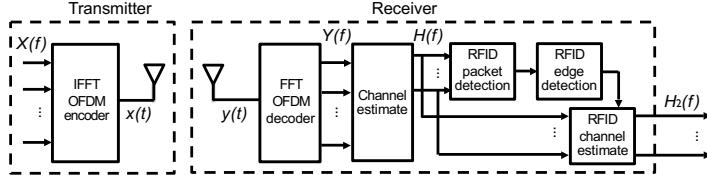


Figure 3: **One-shot Estimation Architecture.** The helper transmits OFDM symbols. The receiver demodulates these symbols by performing an FFT followed by an overall channel estimation step. The overall channel is fed into an RFID packet and edge detection module that allow discovering the RFID’s state transitions. Using the output of the edge detection and elimination block, TurboTrack can extract reliable wideband channel estimates.

responses, we can emulate a large bandwidth – $1000 \times$ larger than typical RFID backscatter bandwidths of few 100 kHz. But, such an approach needs to query the same RFID many times before it can sense a wide bandwidth. Thus, it requires the RFID to remain static, reducing the reader’s throughput and increasing the tracking latency.

TurboTrack’s first component focuses on estimating a wide bandwidth in a single shot, i.e., from every single RFID response. In principle, one could do that by simply transmitting a wideband signal from the localization helper simultaneously with the reader, and estimating the channel across its bandwidth. However, this approach is complicated by two main factors. First, the RFID’s switching process introduces a fast fading channel for the wideband signals; ignoring such fading corrupts the wideband estimates. Second, by spreading its transmitted signal over a wide bandwidth rather than a single frequency, the helper’s signal-to-noise ratio (SNR) significantly degrades, which limits its ability to detect the RFID’s response and precludes channel estimation.

The rest of this section describes how TurboTrack’s helper overcomes these challenges. On the transmit side, the process involves constructing backscatter-aware wideband transmissions. And on the receive side, it involves robustly detecting the RFID’s response for accurate channel estimation.

3.2 Backscatter-Aware Wideband Transmissions

To simplify channel estimation over a wide bandwidth, TurboTrack borrows the OFDM (Orthogonal Frequency Division Multiplexing) modulation technique from WiFi and LTE systems. At a high level, OFDM divides a wideband channel into an array of narrowband channels, and performs modulation in the frequency domain.

For the purpose of this paper, we do not need to delve into the details of how OFDM operates beyond the FFT and IFFT blocks of Fig. 3. Specifically, an OFDM modulator encodes information in the frequency domain as $X(f)$ then takes an IFFT before transmitting the signal over the air. The receiver demodulates the signal by taking an FFT as shown in Fig. 3, and can estimate the channel $H(f)$ by dividing the FFT’s output by $X(f)$.

So, how can we construct OFDM symbols to be

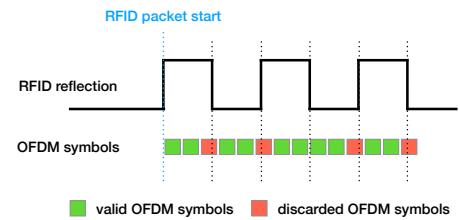


Figure 4: **Two Modulations.** RFIDs communicate by switching between reflective and non-reflective states. TurboTrack’s localization helper constructs and decodes its symbols to accommodate the backscatter switching.

friendly with backscatter modulation? To see why OFDM construction is important, consider Fig. 4 which shows both the backscatter modulation and the OFDM symbols over time. OFDM channel estimation assumes that the entire OFDM symbol lies within a channel coherence time (i.e., that the channel does not change during the estimation process). However, as a backscatter sensor switches its impedance, it causes an extremely fast-fading channel and corrupts the entire OFDM channel estimate. Hence, if we choose long OFDM symbols, then all of them will be corrupted with the backscatter switching process. On another hand, if we choose very short symbols, we cannot pack many frequencies into them, which would prevent us from estimating the wideband channel at sufficient frequency resolution to deal with frequency selectivity.

To address this challenge and obtain both non-corrupted and fine-grained wideband estimates, TurboTrack’s helper exploits information from the RFID reader about the backscatter switching rate. Specifically, the reader communicates that rate (called backscatter link frequency or BLF) in its downlink command to the backscatter sensor. Hence, by coordinating with the RFID reader, the localization helper can use the BLF to construct its OFDM symbols.

In particular, to ensure that the channel is not corrupted, an OFDM symbol must lie within a specific RFID reflection mode (i.e., transition-free region). Hence, the OFDM symbol duration T_{symbol} must be smaller than half the period of an RFID switching period $T_{switching}$:

$$T_{symbol} < \frac{T_{switching}}{2} = \frac{1}{2BLF}$$

Knowing that an OFDM symbol consists of N samples (i.e., N subcarriers), each with time period T_{sample} , and denoting the overall bandwidth of the helper’s OFDM transmission B , this means that we should choose N such that $N < \frac{B}{2BLF}$. For example, if the helper’s OFDM bandwidth is 100MHz and the backscatter link frequency is 500 KHz, N must be less than 100.

3.3 Robust Wideband Channel Estimation

Now that we know how the localization helper constructs its transmitted symbols, we switch our focus to how it

can obtain robust channel estimates on the receive side. Recall that the difficulty in wideband estimation arises from the low SNR at each subcarrier since the power is spread across frequencies. This complicates packet detection² and reduces the reliability of the sensed channel.

(a) Robust Packet Detection: First, to compensate for the reduced power and robustly estimate the beginning of a backscatter packet, TurboTrack exploits the frequency agnostic property of backscatter modulation. In particular, since the different OFDM subcarriers undergo the same backscatter modulation, we can incoherently average their estimates. Such averaging would enable us to reliably observe changes in the overall reflected power and use them to detect the beginning of the RFID response. Specifically, for every OFDM symbol at time n , we compute $H_{\text{combined}}(n) = \sum_f |H(f, n)|$.

Next, we leverage our knowledge of the RFID packet's preamble to detect the packet start. Specifically, every RFID packet payload is preceded by a known preamble $p(n)$. Hence, the localization helper correlates the averaged channel estimates $H_{\text{combined}}(n)$ with the preamble to detect packet start. We can write this correlation as:

$$D(\Delta) = \sum_{n=1}^T p^*(n) H_{\text{combined}}(n + \Delta)$$

where T is the preamble length and Δ is the time instance where correlation is performed. The helper identifies the packet beginning when D rises above a threshold.

(b) Edge Flip Elimination: Next, TurboTrack proceeds to eliminating corrupted OFDM symbols. Recall from §3.2 that TurboTrack constructs the OFDM symbols to accommodate for the backscatter reflection rate. While this ensures that at least one whole symbol is in a reflective or non-reflective state, it does not ensure that all OFDM symbols are non-corrupted.

By leveraging the knowledge from the previous step – namely when an RFID packet starts as well as the RFID's switching frequency – TurboTrack's localization helper can automatically detect and discard erroneous channel estimates (marked in red in Fig. 4). In doing so, it only retains the channel estimates that are obtained when the RFID is reflecting or not reflecting, while eliminating estimates corrupted by RFID state transitions.

(c) Channel Estimation: Now that we have eliminated erroneous OFDM channel estimates, we can proceed to estimating the RFID channels at each of the subcarriers. Note that the OFDM channel estimates $H(f)$ not only consist of the RFID's reflection but also the direct path between the helper's transmit and receive antennas as well as other reflections in the environment. To estimate the RFID's channel, TurboTrack exploits that the difference between the reflective and non-reflective states is

²The helper cannot simply rely on the reader's packet detection, since the reader uses much lower bandwidth and lower sampling rate.

due to the RFID, and subtracts them from each other to obtain the RFID's channel.

Mathematically, assume that after discarding the erroneous channel estimates from the preamble, the helper is left with L symbols where the RFID is non-reflective and M symbols where it is reflective, we can estimate these channels at each subcarrier f as:

$$\widehat{H}_2(f) \propto \frac{1}{L} \sum_{f \in \text{reflective}} H(f) - \frac{1}{M} \sum_{f \in \text{non-reflective}} H(f)$$

Note that in the above equation, we only average over the reflective states of a *single* RFID response, enabling one-shot wideband estimation.

Finally, to improve the efficiency of the wideband estimation process, TurboTrack incorporates the localization helper into the finite state machine of the RFID reader. In doing so, the helper only needs to perform OFDM processing (packet detection, edge elimination, etc.) over the short interval of time during which it expects the RFID's response rather than over the entire duration of the reader's communication session. Specifically, the receiver opens a short time window immediately after the reader finishes transmitting its query command. In our implementation, this window is 300 μ s-long in comparison to the 2ms-long communication session. As a result, this synchronous architecture saves significant computational resources (by 6.6 \times), allowing TurboTrack to achieve ultra-high frame rates and ultra-low latency.

A few additional points are worth noting about TurboTrack's use of OFDM for channel estimation:

- Because the helper's transmitter and receiver are connected to the same oscillator, the estimated channels do not have any carrier frequency offset (CFO) or sampling frequency offset (SFO). Hence, unlike WiFi or LTE, we do not need to correct for them.
- Since the helper transmits the same OFDM symbol back-to-back to estimate the channel, each OFDM symbol acts a cyclic prefix for the subsequent one.
- Since the helper's transmit and receive antennas are co-located, the line-of-sight would dominate the channel estimate, and we do not expect any sampling offset (or packet detection delay) between the transmitted and received OFDM symbols. To correct for any sample offsets introduced by the hardware channel, we perform a time-domain correlation that allows us to detect the beginning (i.e., first sample) of the first OFDM symbol. Moreover, we perform a one-time calibration with a known RFID location in order to eliminate other over-the-wire hardware channels.
- Similar to standard OFDM receivers, the localization helper drops the DC subcarrier as it is less robust to noise, and since dropping it improves the dynamic range of the ADC (Analog-to-Digital Converter).
- To further improve its signal-to-noise ratio (SNR), TurboTrack employs an Exponential Moving Average

(EMA) on the channel estimates. The EMA provides more robust channel estimates and higher accuracy without sacrificing frame rate.

4 Bayesian Space-Time Super-Resolution

So far, we have discussed how TurboTrack can estimate an RFID’s channel over a wide bandwidth. In this section, we describe how it uses this wide bandwidth to estimate and track an RFID’s 3D location.

4.1 Bootstrapping Localization

Before we describe TurboTrack’s localization algorithm, we start by asking whether backscattering a large bandwidth would be sufficient for precise localization. In principle, if one could backscatter a very large bandwidth off an RFID, then we could directly use that bandwidth to localize the tag in a manner similar to ultra-wideband (UWB) ranging systems [58]. In particular, UWB systems leverage their large bandwidth to compute the time-of-flight – i.e., the time it takes their signals to travel between a transmitter and a receiver; they then map this time-of-flight to the distance traveled by multiplying it by the speed of propagation. Because time and frequency are inversely related, their distance resolution is inversely proportional to their bandwidth:

$$\text{resolution} = \text{speed}/\text{bandwidth}$$

Since RF signals travel at the speed of light, obtaining sub-centimeter resolution using UWB ranging would require a bandwidth of 30 GHz.

Unfortunately, backscattering 30 GHz off RFID tags is neither feasible nor desirable for multiple reasons. First, backscatter devices have antennas and impedance matching circuits that are optimized to harness energy within a specific frequency band. Hence, signals backscattered significantly outside their optimal frequency band have very poor SNR, making channel estimation infeasible. Second, even if one could backscatter such a wide bandwidth off the tags, generating it would require very costly RF radios and processing its reflections would be compute intensive as it would require processing 30 GS/s.

To achieve higher accuracy without such a large bandwidth, one could leverage the phase of the backscattered signal since it is very sensitive to changes in distances. Specifically, in the noiseless case, we can express the phase ϕ as:³

$$\phi = 2\pi \frac{d}{\lambda} \bmod 2\pi$$

where d is the distance traveled and λ is the wavelength.

The difficulty in using the phase, however, is that it wraps around every wavelength. Hence, it allows us to accurately recover a fractional distance d_{frac} modulo the wavelength. Said differently, we would know that the actual distance is $d_{frac} + n\lambda$, but n is an unknown integer.

³In presence of multipath, we can use a large bandwidth to obtain “sanitized phases” [45] by projecting on the direct path after identifying it.

Since TurboTrack obtains a wide bandwidth and the phase from every antenna, it can combine them to narrow down the candidate locations to a handful. For example, if we consider a bandwidth of 100 MHz and a wavelength of 33 cm, this leaves us with only $\text{resolution}/\lambda = 9$ potential candidate locations for the tag. Hence, TurboTrack still needs a mechanism to resolve ambiguity and identify the correct candidate.

4.2 Bayesian Formulation

To localize tags, TurboTrack employs a Bayesian framework that fuses measurements across space and time:

- *Spatially*, each distance candidate maps to an ellipse whose foci are the transmit and receive antennas. This is because TurboTrack’s antennas measure the round-trip distance from the transmitter to the backscatter tag, and back to the receiver. Since a given transmit-receive pair has multiple distance candidates, this leads to multiple confocal ellipses as shown in Fig. 5(a). Adding more receive antennas would create other sets of ellipses. However, due to noise, we do not expect the correct ellipses from all antennas to intersect at the same point; hence, we cannot simply rely on voting to identify the correct candidate.
- *Temporally*, as the tag moves, each of the candidate locations traces a different trajectory. The intuition of using temporal series is that because noise is random, we expect the intersection points that correspond to the actual location to be closer to each other across time, providing opportunities to identify them.

Hidden Markov Model. Given the above intuition, we formulate the localization problem as a Hidden Markov Model (HMM), as shown in Fig. 5(b). HMMs form a class of powerful Bayesian inference models with hidden states and observed variables. In our context, the hidden states correspond to the actual locations of the RFID over time, and the observations are the candidate distances obtained from the different receive antennas.

Most importantly, TurboTrack’s HMM has a nonlinear Gaussian observation model and a linear Gaussian transition model: the distance observations are nonlinear in the state variables (the coordinates) as can be seen with the *dist* function, while state transitions are linearly related due to motion. Unfortunately, the nonlinearity prevents us from adapting common solutions like Kalman Filters to model the distributions [35].

TurboTrack’s goal is to find the most likely trajectory $\mathbf{x}_1 \dots \mathbf{x}_T$ given the observations (distance candidates) $\mathbf{y}_1 \dots \mathbf{y}_T$. Formally, it needs to solve for the maximum a posteriori (MAP):

$$\mathbf{x}_{0:T}^* = \arg \max_{\mathbf{x}_{0:T}} p(\mathbf{x}_{0:T} | \mathbf{y}_{0:T})$$

where \mathbf{x}_t is a d dimensional coordinate vector of the tag, \mathbf{y}_t is a k dimensional vector of the estimated dis-

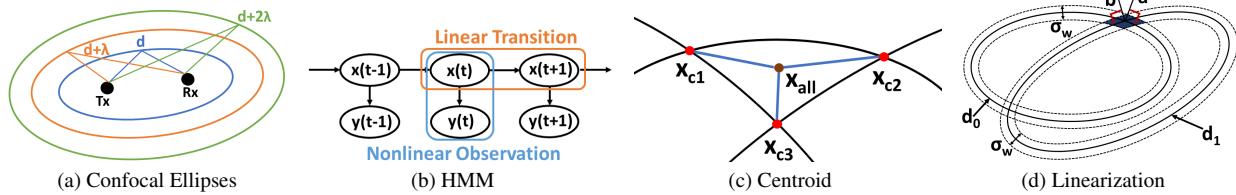


Figure 5: **TurboTrack exploits RF and geometric properties for estimation.** (a) shows that distances map to confocal ellipses. (b) shows the HMM with nonlinear observations. (c) shows centroid approximation. (d) shows geometric linearization

tances from each receiver, and $p(x|y)$ denotes the posterior probability distribution.

Solving the above MAP inference problem requires us to model the likelihood function $p(y_i | x_i)$ and the prior $p(x_{0:T})$. In particular, per Bayes' rule, we can write the posterior as: $p(x_{0:T} | y_{0:T}) \propto p(y_{0:T} | x_{0:T})p(x_{0:T})$

Next, we describe how we model the likelihood function and prior given the geometric nature of the problem and the underlying wireless properties of the estimators.

Likelihood Function. We first model TurboTrack's likelihood function. Recall from §4.1 that the distance from a given transmit-receive antenna pair exhibits as multiple candidates. We formulate the likelihood as a mixture of Gaussian (MoG), where each Gaussian is centered at a different integer wavelength. In particular, given the true distance d , we approximate the distance estimates d_{est} with the following distribution:

$$z(d_{est}|d; \sigma_w) = \sum_{i=-N}^N w(i)\mathcal{N}(d + i\lambda, \sigma_w^2) \quad (1)$$

$$\sum_{i=-N}^N w(i) = 1$$

where N is determined by the total number of candidates, i is integer wavelength, and σ_w corresponds to the standard deviation of Gaussian noise. The weights $w(n)$ denote the discrete distribution of different integers.⁴

The above formulation models the likelihood distribution for one receiver. Since TurboTrack employs multiple receivers, each provides a different set of distance estimates. Because estimates from different antennas are independent, the overall likelihood $p(y_t | x_t)$ is a product of those from different receivers:

$$p(y_t | x_t) = \prod_{j=1}^k z(y_t[j] | dist(x_t)[j]; \sigma_w) \quad (2)$$

where $dist : \mathbf{R}^d \rightarrow \mathbf{R}^k$ returns the round trip distance to the RFID, k denotes the total number of receivers, and j is the index of each receiver.

Transition Model. Next, we model the prior $p(x_{1:T})$. Since the prior consists of a Markov time series of the RFID's location, we use a simple transition model:

$$x_t = x_{t-1} + v_t, v_t \sim \mathcal{N}(\mathbf{0}, \Sigma_v) \quad (3)$$

⁴These weights could correspond to the value of the fractional Fourier transform or a MUSIC projection [32]. In our implementation, we found the overall algorithm performance is the same in both cases, so we used the fractional Fourier as it is less computationally expensive.

where v_t 's correspond to linear position changes, drawn from a Gaussian distribution with zero mean and covariance matrix Σ_v . Together, Eqs. 2 and 3 form our HMM.

4.3 Approximate Inference via Particle Filters

Due to the nonlinear observation model, we cannot solve the MAP problem analytically. So, TurboTrack resorts to approximate inference solutions, specifically particle filters. Instead of propagating beliefs through analytical probability distributions, these models approximate distributions with a set of particles [30]. Approximating distributions with particles requires us to answer two main questions: First, how can we choose particles so that their discrete distributions accurately represent the analytical distributions? And second, how can we update these particles with new observations (distance candidates)?

In the rest of this section, we describe how TurboTrack's algorithm exploits the RF and geometric structures of our measurements to answer these questions and develop a computationally efficient solution. For simplicity, our exposition focuses on 2D localization using three antennas, but the technique can naturally generalize to 3D with an arbitrary number of antennas.

4.3.1 Initial Sampling Function

Standard particle filters bootstrap the sampling process by populating the entire d -dimensional space with exponentially many particles [63]. In contrast, TurboTrack leverages its first observation to bootstrap its inference with a relatively small number of particles. In particular, recall from §4.1 that each receive antenna discovers a finite number of distance candidates, which can be mapped to a set of ellipses in 2D as shown in Fig. 5(a).

To obtain an initial set of particles, one option is to consider all intersection points between ellipses from different antennas. But, this is undesirable for three reasons. First, any such two-way intersection point leaves out important information about distance measurements from other receive antennas. Second, it would result in a polynomially large number of intersection points.⁵ Third, such a set of intersection points would not be representative of the distribution as they are sparsely distributed rather than concentrated at the most likely locations.

⁵The total number of intersection points is the number of distance candidates raised to the power of the number of receive antennas.

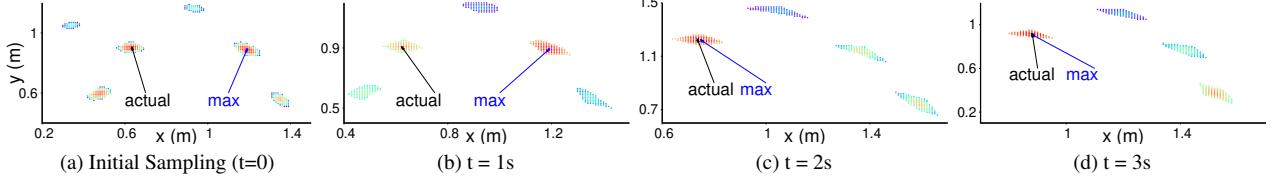


Figure 6: **TurboTrack’s Particle Filter.** We show the particles in 2D space over time. Colors indicate particle weights, with red being highest and blue lowest. We label the actual location and that with highest weight. Over time, the highest weight particle converges to the correct one.

Instead, TurboTrack first identifies the most likely intersection points then uses them to sample particles that are representative of the distribution. At a high level, it exploits the fact that each antenna’s distance measurements can be modeled as a Mixture of Gaussians as per §4.2 in order to (1) prune out unlikely candidates, and (2) linearize the intersection points into a mixture of 2D Gaussians. Subsequently, it can sample from a mixture of 2D Gaussians then adapt the particle weights as per standard importance sampling [62].

Step 1: Pruning. In the pruning step, TurboTrack uses the distances between intersection points to identify the most likely candidates. In particular, given a tuple of distance measurements from three antennas, $[d_0, d_1, d_2]$, we can plot three ellipses and identify the three closest intersection points between them as shown in Fig. 5(c).⁶ In the noiseless case, the three points coincide. Otherwise, we compute the average distance between them and discard all points whose average is above a threshold.

Formally, if we denote the intersection points as $\{\mathbf{x}_{01}^s, \mathbf{x}_{12}^s, \mathbf{x}_{02}^s\}$. TurboTrack will only keep tuple s ⁷ in the set $C := \{s \mid r(s) \leq T\}$ where

$$r(s) := \frac{1}{3}(\|\mathbf{x}_{01}^s - \mathbf{x}_{12}^s\|_2 + \|\mathbf{x}_{01}^s - \mathbf{x}_{02}^s\|_2 + \|\mathbf{x}_{12}^s - \mathbf{x}_{02}^s\|_2)$$

Step 2: Geometric Linearization. Next, TurboTrack approximates each intersection point as a 2D Gaussian using the original distance distributions of the corresponding ellipses. To see why this linearization is possible, consider two intersecting ellipses in Fig. 5(d), whose intersection point we denote as \mathbf{x}_{01}^s . Since the noise on the distance measurements is much smaller than the size of the ellipses, the curvature of the ellipses around the intersection point is relatively flat. This allows us to approximate the distribution as a 2D Gaussian centered around \mathbf{x}_{01}^s and whose axes lie along the normals to the two ellipses at the intersection point.

Formally, if we denote the two axes by the normal vectors: \mathbf{a}, \mathbf{b} , and define $A = [\mathbf{a}, \mathbf{b}]$, then we obtain R_{ij} as a Gaussian distribution in the 2D coordinate system:

$$R_{ij} \sim \mathcal{N}(\mathbf{x}_{01}^s, \Sigma_{01}^s = AVA^T) \text{ s.t. } V = \begin{pmatrix} \sigma_w^2 & 0 \\ 0 & \sigma_w^2 \end{pmatrix} \quad (4)$$

⁶Three ellipses can have at most six intersection points, from which we choose the closest three.

⁷ s is integer tuple $[s_0, s_1, s_2]$ for $[d_0 + s_0\lambda, d_1 + s_1\lambda, d_2 + s_2\lambda]$.

where V denotes the diagonal covariance matrix, which represents the fact that the distance errors from the different antennas are independent.

Step 3: Centroid Approximation. So far, we have identified the most likely tuples of intersection points, and we have linearized each of the intersection points as a 2D Gaussian. Next, TurboTrack fuses every surviving tuple into a single candidate. This step is important because it combines measurements from all antennas (while the previous step considered antennas only in pairs). To do so, it approximates any given candidate \mathbf{x}_0 as the centroid of its three corresponding intersection points (from Step 1). For simplicity, we assume mutual independence between the intersection points, which results in the following Gaussian distribution $f(\mathbf{x}_0; \mathbf{y}_0, \mathbf{s})$:

$$\begin{aligned} f(\mathbf{x}_0; \mathbf{y}_0, \mathbf{s}) &\sim \mathcal{N}(\mu_s, \Sigma_s) \\ \mu_s &= \frac{1}{3}(\mathbf{x}_{01}^s + \mathbf{x}_{12}^s + \mathbf{x}_{02}^s) \\ \Sigma_s &= (\frac{1}{3})^2(\Sigma_{01}^s + \Sigma_{12}^s + \Sigma_{02}^s) \end{aligned} \quad (5)$$

This provides us with a set of 2D Gaussian distributions, each centered at a likely candidate position.

Step 4: Weighting and Initial Sampling. In the final step, we combine the set of Gaussian distributions into a single 2D mixture of Gaussians. To do so, we assign weights proportional to the product of weights of individual integers to tuples in C . This provides us with the initial sampling distribution $q_0(\mathbf{x}_0; \mathbf{y}_0)$:

$$\begin{aligned} q_0(\mathbf{x}_0; \mathbf{y}_0) &= \sum_{s \in C} \frac{w(s)}{w} f(\mathbf{x}_0; \mathbf{y}_0, \mathbf{s}) \\ w(\mathbf{s}) &= w(s_0)w(s_1)w(s_2), w = \sum_{s \in C} w(s) \end{aligned} \quad (6)$$

Given this distribution, we can now efficiently sample a small number of particles $\{\mathbf{x}_0^{(i)}\}$ and use them to approximate the distribution. After sampling, we normalize every particle’s weight to the total weights.

Fig. 6(a) shows an example output of initial sampling in 2D space. Each particle’s color indicates its normalized weight, where red and blue indicate highest and lowest probability. The particles are concentrated in several clusters which correspond to the fused intersection points. While the actual location is in one of the clusters, the one with the highest weight is in another cluster (due to noise). This shows the power of fusing across antennas and emphasizes the need to resolve ambiguity by exploiting target motion and fusing over time.

4.3.2 Sequential Sampling

So far, we have described how TurboTrack can obtain an initial set of particles by fusing RF observations from multiple antennas at a single point in time $t = 0$. Next, we discuss how TurboTrack can update its particles through new observations over time. To do so, it adapts Sequential Importance Sampling (SIS), a well-known particle update filter, to our problem domain.

As its name indicates, SIS operates through a sequential process. To select representative particles for the probability distribution at some time t , it uses the (new) observations from time t and the probability distribution from the previous state $t-1$. Hence, for every particle i , it is optimal to sample from the following distribution [15]:

$$p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{y}_t) \propto p(\mathbf{y}_t \mid \mathbf{x}_t)p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)})$$

Recall that we have already modeled both terms on the right hand side. In particular, we know that $p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)})$ is a Gaussian as per the motion model in Eq. 3. Moreover, $p(\mathbf{y}_t \mid \mathbf{x}_t)$ can be approximated as a Gaussian as per Eq. 5. Because the product of two Gaussians is Gaussian, the overall sampling function is both Gaussian and approximately optimal.⁸

Now that we know the sampling distribution for a given particle i , SIS samples a single particle from that distribution for time t . It then repeats the same process for all the particles. After obtaining all the samples at time t , it normalizes their weights then moves to the next time step $t+1$ and repeats the same process.

Few additional points are worth noting about TurboTrack’s inference algorithm:

- *MAP Inference*: At every point in time, TurboTrack can make a decision about the most likely location by choosing the particle with the highest weight. It can also update the past trajectory since every particle can backtrack its entire history. This allows us to continuously update the trajectory with new observations.
- *Resampling*: It is known that even with optimal approximate sampling functions, the SIS algorithm can degenerate to a small number of particles [30, 38]. To avoid degeneracy, TurboTrack follows a standard resampling approach. In particular, at every time t , it estimates the effective sample size $\widehat{N_{eff}} = \frac{1}{\sum_{i=1}^N (w_t^{(i)})^2}$, and resamples when it is lower than a threshold.
- *Computational Efficiency*: In the sample update step, TurboTrack exploits that many of the particles i come from the same Gaussian in the Gaussian mixture because they share the same tuple of intersection points. Hence, to optimize computation, it computes the sequential sampling function once for each cluster and reuses it for all particles in that cluster.

⁸Note that $p(\mathbf{y}_t \mid \mathbf{x}_t)$ is a mixture of Gaussians as per Eq. 6, but we approximate it to the Gaussian nearest to $\mathbf{x}_t^{(i)}$ in the mixture.

- *Outlier Detection & Recovery*: The SIS filter accounts for Gaussian noise but does not incorporate mechanisms to deal with wireless interference or strong leakage from multipath.⁹ To deal with such scenarios, TurboTrack employs outlier detection and recovery. We identify two scenarios for outlier detection. The first is when the centroid at t is far from that at $t-1$. The second is when all particles are assigned near-zero weights, indicating that there is no valid position given current distance estimation. To recover from such outliers, TurboTrack leverages its high frame rate and extrapolates the previous estimates.

Finally, Alg. 4.1 summarizes TurboTrack’s approximate inference algorithm and Fig. 6 shows how the algorithm converges to the right candidate over time.

Algorithm 4.1 TurboTrack’s Approximate Inference

Input: Distance estimations: $\mathbf{y}_{0:T}$

Initialization: At time $t = 0$

- ▷ Compute mixture of Gaussian function $q_0(\mathbf{x}_0; \mathbf{y}_0)$
 - (a) Prune unlikely intersection points of \mathbf{y}_0
 - (b) Linearize unpruned tuples as Gaussians per Eq. 4
 - (c) Approximate Gaussians per Eq. 5
 - (d) Assign weights to tuples per Eq. 6
- ▷ Sample and assign weights

For $i = 1, \dots, N$:

- (a) Sample $\mathbf{x}_0^{(i)}$ from $q_0(\mathbf{x}_0; \mathbf{y}_0)$
- (b) Assign weights: $\hat{w}_0^{(i)} = p(\mathbf{y}_0 \mid \mathbf{x}_0^{(i)})$

▷ Normalize weights: $w_0^{(i)} = \hat{w}_0^{(i)} / \sum_{j=0}^N \hat{w}_0^{(j)}$

Iteration: For time $t = 1, \dots, T$

For $i = 1, \dots, N$:

▷ Compute Gaussian function $f(\mathbf{x}_t; \mathbf{y}_t, \mathbf{s}^{(i)})$

- (a) Identify the tuple $\mathbf{s}^{(i)}$ corresponding to the intersection point¹⁰ closest to $\mathbf{x}_{t-1}^{(i)}$
- (b) Linearize tuple as per Eq. 4
- (c) Approximate Gaussian as per Eq. 5

▷ Sample and assign weights

- (a) Sample $\mathbf{x}_t^{(i)}$ from $\pi(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{y}_{0:t}) \propto p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)})f(\mathbf{x}_t; \mathbf{y}_t, \mathbf{s}^{(i)})$
- (b) Add to history: $\mathbf{x}_{0:t}^{(i)} = (\mathbf{x}_{0:t-1}^{(i)}, \mathbf{x}_t^{(i)})$
- (c) Assign weights:

$$\hat{w}_t^{(i)} = w_{t-1}^{(i)} \frac{p(\mathbf{y}_t \mid \mathbf{x}_t^{(i)})p(\mathbf{x}_t^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{\pi(\mathbf{x}_t^{(i)} \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{y}_{0:t})}$$

▷ Normalize weights: $w_t^{(i)} = \hat{w}_t^{(i)} / \sum_{j=0}^N \hat{w}_t^{(j)}$

▷ Compute effective sample size $\widehat{N_{eff}}$.

If $\widehat{N_{eff}} < N_{thres}$, resampling based on $w_t^{(i)}$

Output: MAP Trajectory: $\mathbf{x}_{0:T}^* = \arg \max_{\{\mathbf{x}_{0:t}^{(i)}, w_{0:t}^{(i)}\}} w_{0:t}^{(i)}$

⁹Recall that we mitigate (but not eliminate) multipath by projection.

¹⁰Recall that the intersection point can be computed from $\mathbf{y}_t + \mathbf{s}^{(i)} \lambda$.

5 Implementation & Evaluation

Localization Helper: We implement TurboTrack’s localization helper on USRP X310 software radios, all synchronized to the same external clock [11]. We use three USRPs with two UBX daughterboards each. Since each USRP can support two chains, we use one chain at a transmitter and four as receivers. Each transmit/receive chain is connected to a circularly polarized patch antenna [8]. All antennas are arranged in a single plane, with a separation of 40 cm between any two adjacent pairs. USRPs sample at 100 MSps and send data over Ethernet to a computer using the Intel Converged Network Adapter X520-DA2 [3] to support high data rates. The computer runs Ubuntu 16.04 and has an 4-core 8-thread 64-bit Intel Core i7 processor and 16GB RAM.

RFID Reader: We adapt a USRP RFID reader developed by past work [36] and implement it on USRP N210 software radios [12] with SBX daughterboards. The reader implements the EPC Gen2 protocol [2]. The reader powers up and communicate with the RFID tags, setting various parameters such as the BLF and tag selection in multi-tag scenarios.

Real-time Processing. We implement the algorithms described in §3-§4 directly into the USRP’s UHD driver in C++. Our implementation consists of a multi-threaded pipelined architecture with worker pools. On the transmit side, we use two threads in total: one for the RFID reader and another one for the OFDM transmitter. On the receive side, we use one thread for extracting buffer-sized packets from the USRPs. In addition, every antenna has a worker pool of two threads; each worker extracts and processes individual RFID responses from the shared buffer, computes and timestamps the wideband estimates, and feeds them to an output buffer. Finally, we aggregate the threads, combining the wideband estimates from all the antennas to perform 3D localization.

Our implementation runs in real-time, transmitting and receiving OFDM symbols, decoding, estimating channels, performing super-resolution and localization ($\Sigma_v = 5\text{mm}$)¹¹. Our OFDM symbols use $N = 20$ subcarriers over a bandwidth of 100 MHz to increase resilience to RFID state-transitions. In contrast, the RFID BLF is set to 40 kHz.

Baselines: We implemented two baselines:

- *RFind* [45]: represents a state-of-the-art system for centimeter-scale RFID positioning. It emulates a large bandwidth by frequency hopping. As per the implementation described in the original paper, the hopping process takes few seconds (for channel acquisition).

¹¹In principle, one could learn the HMM parameters, but setting a constant standard deviation worked well for our applications.

We reproduced the implementation and wrote the code directly into the UHD driver of the USRP (as we did for TurboTrack).

- *RF-IDraw* [71]: represents a state-of-the-art system for high-accuracy RFID tracking.¹² As per the authors evaluation, RF-IDraw can achieve high tracking accuracy but decimeter-scale accuracy in exact positioning. It combines various antenna patterns. We faithfully reproduced the authors’ implementation on Thingmagic m6e [6] RFID readers.

RFID tags: We evaluated TurboTrack with commercial, off-the-shelf, passive UHF RFID tags. We tested it with multiple tags including Avery Dennison AD-238u8 [1], Alien Squiggle [17], and Smartrac [10]. Each tag costs about 5-10 cents. Because our tags are vertically polarized, we program the robots to maintain their orientation to minimize phase changes from orientation changes (for both TurboTrack and the baselines). Alternatively, one could use circularly polarized tags to avoid this problem, or incorporate an orientation-phase model into TurboTrack’s inference algorithm.

Robots: We tested TurboTrack with three kinds of moving robots. For 2D evaluation, we attached an RFID to a Roomba [4]. For 3D evaluation, we attached the RFID to an item carried by LeArm 6DoF robotic arms [5] (as shown in Fig. 1) and to A20 minidrones from Potensic [7] (as shown in Fig. 12).

Ground truth: We use the OptiTrack system [9] to obtain ground truth location measurements. The OptiTrack is an optical tracking system which consists of an array of tripod-mounted infrared cameras that can achieve millimeter-scale accuracy by relying on infrared-reflective markers placed on the objects of interest. Since the OptiTrack can only operate in line-of-sight, in our non-line-of-sight evaluation, we ensure that while the RFIDs of interest are occluded from our antennas, they remain in LOS of the OptiTrack cameras.

Evaluation Environment: We evaluated TurboTrack in a standard office building, fully furnished with tables, chairs, computers. We tested it in both line-of-sight and non-line-of-sight settings, where RFIDs are within 6m of the antennas. We performed NLOS testing similar to past work [45, 71] by blocking the visible LOS path between an RFID and TurboTrack’s antenna using standard office cubicle dividers made of wood.

¹²Mathematically, RF-IDraw’s tracking algorithm is similar to Tagoram’s [75] method for tracking movement with unknown track and both systems achieve comparable tracking and localization accuracy.

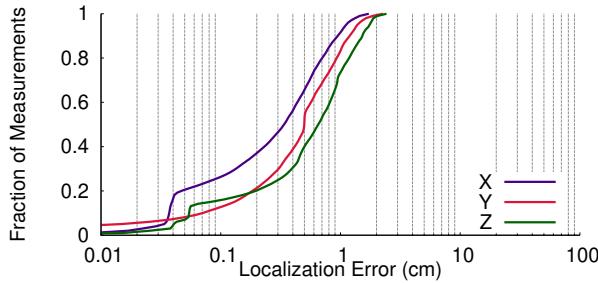


Figure 7: **3D Localization Accuracy in Line-of-Sight.** The figure plots the CDF of TurboTrack’s localization error in LOS along each of the x/y/z dimensions.

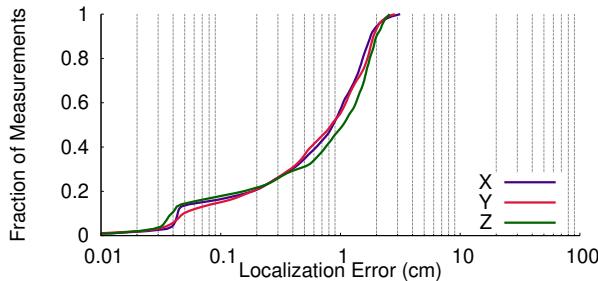


Figure 8: **3D Localization Accuracy in Non-Line-of-Sight.** The figure plots the CDF of TurboTrack’s localization error in NLOS in the x/y/z dimensions.

6 Results

6.1 Performance Evaluation

We first evaluated TurboTrack’s performance quantitatively. Specifically, we evaluated its localization accuracy, latency, and performance with moving targets.

(a) 3D Localization Accuracy: We tested TurboTrack’s localization accuracy in LOS and NLOS settings. We performed 5,000 experimental trials, each lasting between 30 seconds and one minute. This allowed us to collect more than 20 million location measurements. In each trial, we performed different arbitrary movements with the robotic arm or the drone carrying the RFID. We compute the tracking error as the difference between TurboTrack’s location estimate and the ground truth from OptiTrack.

Figs. 7-8 plot the CDFs of the location errors in each of the x, y, and z dimensions for both LOS and NLOS settings. Our results show that TurboTrack achieves a median error less than 1 cm and a 90th percentile error is less than 2 cm in each dimension. Further, we note that the accuracy in LOS is slightly better than its accuracy in NLOS settings. This is expected since the SNR is higher in LOS, resulting in higher accuracy.

(b) Latency & Frame Rate: Next, we evaluated the latency of TurboTrack’s pipelined architecture in both 2D and 3D localization. Our 2D trials were performed using

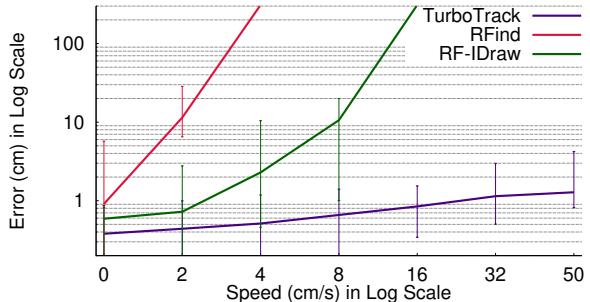


Figure 9: **Tracking Error vs. Speed.** We plot the median 2D tracking accuracy of TurboTrack (violet), RF-IDraw (green), and RFFind (red) vs speed. Error bars indicate 10th and 90th percentiles.

an RFID attached to a Roomba. The latency is computed as the time difference between the time the USRP obtains an RFID response and the time it outputs a location. We ignore the time-of-flight of the wireless signal since it is only few nanoseconds for the distances of testing (few meters) hence negligible for our latency measurements.

Our results shows that the 99th percentile latency measurements for 2D and 3D tracking are 6.6 ms and 7.3 ms, respectively. This latency is primarily limited by the processing time of the computer rather than the latency of the RFID’s response or acquisition. Specifically, in both the 2D and 3D experiments, TurboTrack’s receivers all obtain the RFID responses at the same time, yet the difference in latency is due to the difference in processing speed. We also note that across these experiments, TurboTrack could achieve a frame rate of 300 frames/second in both 2D and 3D localization. This high frame rate owes to TurboTrack’s pipelined architecture which decouples the frame rate from the latency, enabling it to achieve both high frame rates and low latency.

(c) Performance with Motion: TurboTrack’s high accuracy, low latency, and high frame rate aim at enabling it to track objects in continuous motion, as necessary for robotic manipulation and tracking tasks. Next, we evaluated TurboTrack’s accuracy with motion and compare it to our baselines. In fairness to RF-IDraw, we focused on 2D tracking since the system was only evaluated in 2D.

We perform 400 experimental trials, each time varying the speed at which an RFID moves. We varied the speed by attaching the RFID on a mobile Roomba whose speed can be controlled. Fig. 9 plots the CDF of tracking accuracy for RF-IDraw, RFFind, and TurboTrack. It is important to note that the figure is *in log-log scale* to demonstrate how much TurboTrack is more capable in maintaining its accuracy despite high speed motion.

Our results show that TurboTrack outperforms both RF-IDraw and RFFind across all speeds. Even at speeds of 50 cm/s, which is the maximum speed of Roomba [4], its error remains under 2 centimeters. This is due to

three reasons: First, it has significantly higher frame rates than RFInd. Second, it has more resilience to multipath than RF-IDraw due to its larger bandwidth. And lastly, its Bayesian framework boosts its accuracy and robustness.

We also note that RFInd suffers the most with motion, even at speeds as low as 2 cm/s. This owes to its frequency hopping process, which requires the object to remain static for 3 s. On the other hand, while RF-IDraw can deal with some movements, its ability to accurately track the phase diminishes beyond speeds of few cm/sec.

In the above result, we eliminated the initial position error as per RF-IDraw’s implementation. In fairness to RFInd, we run 100 additional experiments with a static RFID and compute RFInd and RF-IDraw’s accuracy. We summarize our results in the table below.

	RFInd	RF-IDraw	TurboTrack
Median (cm)	0.87	19	0.51
90 th percentile (cm)	2.3	63	1.1

Table 1: Positioning Accuracy.

The table shows that both TurboTrack and RFInd outperform RF-IDraw, which has a median accuracy of 19 cm. This result is expected since RF-IDraw is designed for high tracking accuracy rather than high localization accuracy. In the RF-IDraw paper [71], the authors call this the initial position error. We also note that even though RFInd has larger overall bandwidth than TurboTrack (around 200 MHz vs TurboTrack’s 100 MHz), TurboTrack takes advantage of motion and fuses measurements across multiple antennas to achieve a 90th percentile error of around 1.1 cm.

6.2 Microbenchmarks

Next, we would like to understand the effectiveness of each of TurboTrack’s sub-components. To do so, we ran micro-benchmarks with partial implementations of the system as well as with simplified variants. This enables us to gain deeper understanding into the importance of each component as well as the effectiveness of TurboTrack’s design choices from two main perspectives: localization accuracy and computational efficiency.

(a) Decomposing TurboTrack’s Gains. We would like to quantify the accuracy gains arising from TurboTrack’s space-time super-resolution algorithm and the different sub-components of this algorithm. Hence, we implemented three variants of the algorithm and compared their localization accuracy; we focus on 2D localization for simplicity. All three schemes are given the same distance estimates obtained from TurboTrack’s one-shot wideband estimation algorithm over 100 MHz of bandwidth. The schemes differ in how they perform localization based on these distance estimates:

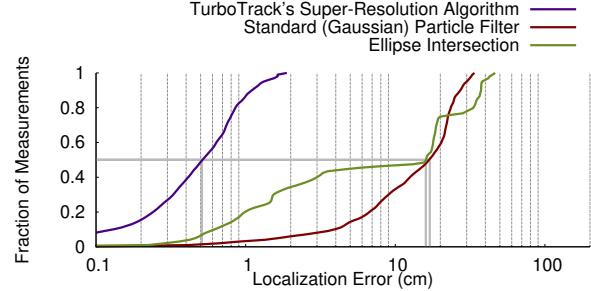


Figure 10: Comparing Partial Implementations of TurboTrack. The figure plots the CDF of the 2D localization errors of three different localization schemes, all of which use the output of TurboTrack’s one-shot wideband estimation.

1. *Ellipse Intersection:* Recall from §4 that each distance estimate maps to an ellipse whose foci are the transmit and receive antennas. A simple localization method would consider the most likely distance estimate from each receive antenna, map it to an ellipse, and solve for the intersection point of the ellipses in order to identify the tag’s location [14, 13]. To implement this scheme, we select the distance estimate with the highest weight from each receive antenna (where the weights are obtained from the amplitude of the interpolated FFT in a manner similar to [13]). Since TurboTrack employs more antennas than the number of ellipses needed for 2D localization, we solve for all the intersection point and assign the one closest to the ground truth as the tag’s location. Doing so provides this scheme with more information than we provide TurboTrack’s super-resolution algorithm. Hence, TurboTrack’s ability to outperform this scheme is a stronger demonstration of its effectiveness.
2. *Standard (Gaussian) Particle Filter:* The second localization scheme employs a standard Gaussian particle filter. The main difference between this scheme and TurboTrack’s algorithm is that its likelihood function (i.e., Eq. 1) is a single Gaussian (centered around the most likely distance estimate and with a large standard deviation) rather than a mixture of narrow Gaussians. This scheme also incorporates the SIS filter.
3. *TurboTrack’s Super-Resolution Algorithm:* The final scheme implements TurboTrack’s space-time super-resolution algorithm, which incorporates the proposed Mixture of Gaussians (MoG) likelihood function and SIS filter.

Fig. 10 plots the CDFs of the localization error for each of the three schemes. We observe the following:

- All three schemes achieve a median error less than 20 cm. This decimeter-scale error of moving RFIDs is smaller than that of state-of-the-art systems (as per in §6.1(c)), and is possible because of TurboTrack’s one-shot wideband estimation technique.

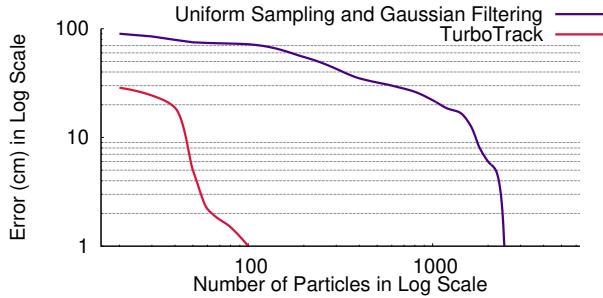


Figure 11: **Tracking Error vs. Number of Particles.** The figure plots the median error of 2D localization as a function of the number of particles for TurboTrack’s sampling method and compares it to a uniform sampling method. The figure shows that TurboTrack can converge to sub-centimeter accuracy using $25\times$ less particles.

- The error distribution of the ellipse intersection method exhibits multiple discontinuities. This is because there are multiple candidate distance estimates (as described in §4.2), and the scheme picks one of them independently in every frame. Thus, while the error is small if the correct candidate is chosen, the error is typically larger than a wavelength (33 cm) if the incorrect candidate has a higher weight (e.g., due to noise). We note that this scheme’s results are in line with the reported errors in [45] when using the same bandwidth with frequency hopping for static RFIDs.
- Interestingly, the ellipse intersection method has slightly smaller median error (16 cm) than that of a standard particle filter-based method (17 cm). However, the 90th percentile error of the particle filter is smaller than that of the intersection-based method. This is due to two reasons. First, the ellipse intersection scheme is given more information since we allow it to choose the intersection point closest to the ground truth. And second, the particle filter smoothes the trajectory, which increases its median error but enables it to achieve better tail performance.
- Finally, TurboTrack’s full implementation significantly outperforms both partial schemes, achieving at least $30\times$ improvement in median accuracy (5 mm median error).

(b) Complexity Gain of Approximate Inference. Next, we would like to quantify the efficiency (complexity) gains arising from TurboTrack’s approximate inference algorithm and its initial sampling scheme. To do so, we compare the algorithm to a baseline implementation of a standard Gaussian particle filter. The baseline uniformly samples 2D space and updates particles weights using the Gaussian filtering method described in §6.2(a).

The complexity of a particle filter is a direct function of the number of particles N used to represent the distribution. Hence, to compare the two schemes, we ran both particle filters (the baseline and TurboTrack’s) with the same number of particles over 30-second-long trajec-

tories, and we computed the localization accuracy. We repeat this process multiple times, each time with a different number of particles.

Fig. 11 plots the median error (on the y-axis) as a function of the number of particles (on the x-axis) in log-log scale for each of the two schemes. The figure shows that TurboTrack’s inference algorithm can converge to sub-centimeter accuracy with only 100 particles; in comparison, to achieve the same accuracy, the uniform sampling-based scheme requires around 2500 particles. Since the complexity of the particle filters is $\mathcal{O}(N)$, this demonstrates that TurboTrack’s algorithm is more computationally efficient.

6.3 Qualitative Performance

Finally, we evaluated TurboTrack qualitatively in fine-grained robotic tasks. Our results, shown in Fig. 12-13, demonstrate the ability to track nanodrones docking, maneuvering, and even flying simultaneously. The results also show that TurboTrack’s fine-grained tracking can be an enabler for collaborative packaging and handover between robotic arms.

7 Related Work & Conclusion

(a) RF-based Localization is a long studied problem in the networking community. Early work relied on measuring the received signal strength (RSS) [54, 76, 24, 26], the angle of arrival (AoA) [49, 77, 20, 40], and the received signal phase [19, 41]. These proposals could operate correctly in line-of-sight but not in the presence of multi-path since constructive and destructive interference make the strength, angle, and the phase of the received signal unpredictable.

Unfortunately, state-of-the-art proposals that can deal with multi-path cannot deliver on the mobility or accuracy requirements for fine-grained robotic tasks. In particular, solutions that achieve high accuracy require the target to remain static for seconds as their antennas move over multiple meters, collecting measurements from different spatial locations then combining them to localize [70, 60, 69, 51]. Others, like RFind [45], achieve high accuracy without requiring antenna motion, but they still require the object of interest to remain static for few seconds as they perform frequency hopping over a large bandwidth. As demonstrated in §6, this leads to large errors in mobile settings. Fundamentally, even if one could hop frequencies faster, such systems would still suffer from a range-Doppler ambiguity [46] and have lower accuracy, frame rate, and throughput than TurboTrack.

To avoid this latency problem, researchers have looked into recovering the shape of the trajectory while sacrificing exact positioning [71, 75, 61, 39]. These proposals focus on tracking *changes in distances* and can achieve

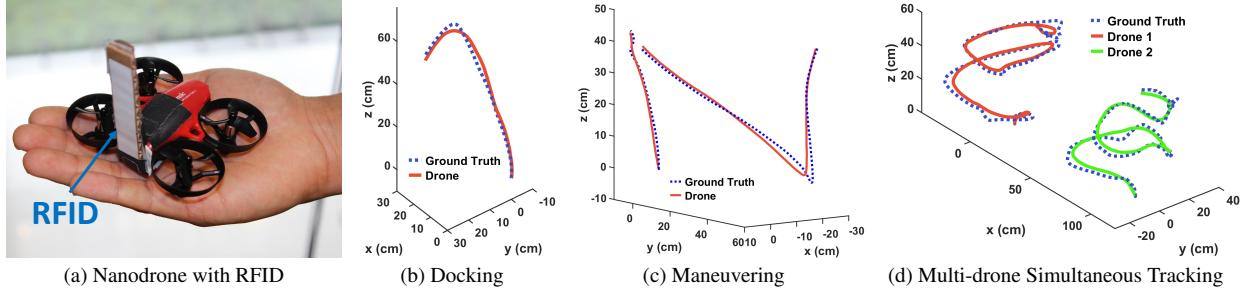


Figure 12: **Tracking Nanodrones.** (a) shows the nanodrone we use in our experiments. (b), (c), and (d) show TurboTrack’s output and the ground truth (dotted blue) in tracking docking, maneuvering, and two drones flying simultaneously.

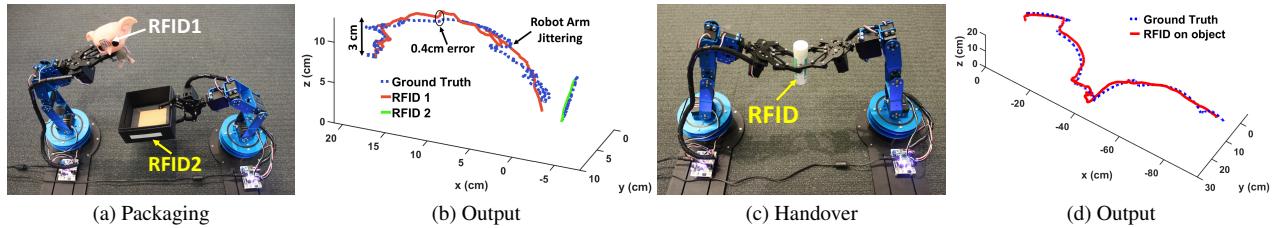


Figure 13: **Robotic Manipulation.** (a) shows two robotic arms collaborating on packaging an RFID-tagged item in an RFID-tagged box. (c) shows one robotic arm handing over an RFID-tagged item to another. (b) and (d) show TurboTrack’s output for tracking the RFID tagged items.

high accuracy over short periods of time, however, they have decimeter-scale accuracy (i.e., tens of centimeters) in computing the exact location. As a result, while they may recover the shape of a trajectory of an RFID or a WiFi device, they cannot enable precise positioning tasks like grasping or manipulation. In contrast to all of these proposals, because TurboTrack computes the *wideband estimates from every single RFID response*, it can achieve precise positioning and at low latency.

(b) Super-resolution algorithms have been extensively studied both theoretically and practically. Most past work falls in the imaging community, where the goal is to obtain a higher-resolution image by combining images of the same scene across multiple viewpoints [65, 23, 37, 64] or across time frames in a video [31, 22, 52, 53]. While TurboTrack is inspired by this body of work, it differs in two key aspects. First, in contrast to imaging systems which rely only on pixel intensity, TurboTrack also has access to phase information and utilizes it in its super-resolution algorithm. Second, its formulation is unique in how it models and linearizes distances obtained from wideband RF measurements and how it incorporates them into a computationally efficient particle filter.

The RF community has also taken interest in super-resolution algorithms, with famous algorithms like MUSIC [32], smoothed MUSIC [59], and ESPRIT [66]. TurboTrack builds on this body of work as well, and to the best of our knowledge introduces the first Bayesian spatio-temporal framework that combines bandwidth and phase measurements to achieve this level of accuracy.

Naturally, TurboTrack also relates to a growing liter-

ature on object manipulation in the robotics community. The majority of past work relies on vision-based or optical systems which, unlike TurboTrack, cannot operate in visually occluded settings [48, 74, 42]. Finally, we note that some of the RF localization solutions mentioned above [69, 60] have been explored in this context as well, but they lacked the localization accuracy and/or the low latency required to deliver on these tasks. TurboTrack is inspired by this work and builds on it to enable highly accurate tracking and identification for fine-grained robotic tasks, particularly in cluttered or occluded settings.

Acknowledgments. We thank Nick Selby for his help in early developments of the system and in the baseline evaluation. We also thank our shepherd, Lili Qiu, and the anonymous NSDI reviewers for their feedback and insights. This research is partially supported by the MIT Media Lab and NSF CPS Award CNS-1739723.

References

- [1] Avery denison. <http://rfidaverydennison.com>, 2018. Avery Denison.
- [2] EPC UHF Gen2 Air Interface Protocol. <http://www.gs1.org/epcrfid/epc-rfid-uhf-air-interface-protocol/>, 2-0-1, 2018.
- [3] Intel X520. <https://ark.intel.com>, 2018. Intel X520.

- [4] irobot roomba. <https://www.irobot.com>, 2018. irobot Roomba.
- [5] Lewansoul learn. <http://www.lewansoul.com>, 2018. LewanSoul LeArm.
- [6] M6e. <http://www.thingmagic.com>, 2018. ThingMagic Inc.
- [7] mini-drone a20. <http://www.ipotensic.com/>, 2018. Potensic mini-drone.
- [8] MTI RFID antenna. <http://www.mtiwe.com>, 2018. MTI Wireless Edge.
- [9] Optitrack. <http://www.optitrack.com>, 2018.
- [10] Smartrac RFIDs. <https://www.smartrac-group.com>, 2018. Smartrac.
- [11] UBX daughterboard. <http://www.ettus.com>, 2018. ettus inc.
- [12] usrp n210. <http://www.ettus.com>, 2018. ettus inc.
- [13] F. Adib, Z. Kabelac, and D. Katabi. Multi-person localization via rf body reflections. In *Usenix NSDI*, 2015.
- [14] F. Adib, Z. Kabelac, D. Katabi, and R. C. Miller. 3d tracking via body radio reflections. In *Usenix NSDI*, 2014.
- [15] H. Akashi and H. Kumamoto. Construction of discrete-time nonlinear filter by monte carlo methods with variance-reducing techniques. *Systems and Control*, 19(4):211–221, 1975.
- [16] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi-robot cooperation in the martha project. *IEEE Robotics Automation Magazine*, 5(1):36–47, March 1998.
- [17] Alien Technology Inc. ALN-9640 Squiggle Inlay, 2018. www.alientechnology.com.
- [18] P. K. Allen, A. Timcenko, B. Yoshimi, and P. Michelman. Automated tracking and grasping of a moving object with a robotic hand-eye system. *IEEE Transactions on Robotics and Automation*, 9(2):152–165, April 1993.
- [19] D. Arnitz, K. Witrisal, and U. Muehlmann. Multi-frequency continuous-wave radar approach to ranging in passive uhf rfid. *IEEE Transactions on Microwave Theory and Techniques*, 57(5):1398–1405, 2009.
- [20] S. Azzouzi, M. Cremer, U. Dettmar, R. Kronberger, and T. Knie. New measurement results for the localization of uhf rfid transponders using an angle of arrival (aoa) approach. In *IEEE RFID*. IEEE, 2011.
- [21] J. M. Beer, C.-A. Smarr, T. L. Chen, A. Prakash, T. L. Mitzner, C. C. Kemp, and W. A. Rogers. The domesticated robot: design guidelines for assisting older adults to age in place. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 335–342. ACM, 2012.
- [22] C. M. Bishop, A. Blake, and B. Marthi. Super-resolution enhancement of video. In *AISTATS*, 2003.
- [23] N. K. Bose and N. A. Ahuja. Superresolution and noise filtering using moving least squares. *IEEE Transactions on Image Processing*, 15(8):2239–2248, 2006.
- [24] M. Bouet and A. L. Dos Santos. Rfid tags: Positioning principles and localization techniques. In *Wireless Days, 2008. WD'08. 1st IFIP*, pages 1–5. IEEE, 2008.
- [25] L. Chaimowicz, T. Sugar, V. Kumar, and M. F. M. Campos. An architecture for tightly coupled multi-robot cooperation. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 3, pages 2992–2997 vol.3, May 2001.
- [26] K. Chawla, C. McFarland, G. Robins, and C. Shope. Real-time rfid localization using rss. In *Localization and GNSS (ICL-GNSS), 2013 International Conference on*, pages 1–6. IEEE, 2013.
- [27] B. Choi and J. Lee. Mobile robot localization in indoor environment using rfid and sonar fusion system. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2039–2044, Oct 2009.
- [28] C. M. Costa, H. M. Sobreira, A. J. Sousa, and G. M. Veiga. Robust and accurate localization system for mobile manipulators in cluttered environments. In *2015 IEEE International Conference on Industrial Technology (ICIT)*, pages 3308–3313, March 2015.
- [29] Defense IQ. Small is beautiful: Nano drone tech is advancing. <https://www.defenceiq.com/defence-technology/articles/nano-drone-tech-is-advancing>.
- [30] A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and computing*, 10(3):197–208, 2000.

- [31] M. Elad and A. Feuer. Superresolution restoration of an image sequence: adaptive filtering approach. *IEEE Transactions on Image Processing*, 8(3):387–395, 1999.
- [32] B. Friedlander. The root-music algorithm for direction finding with interpolated arrays. *Signal processing*, 30(1):15–29, 1993.
- [33] M. Hassanalian and A. Abdelkefi. Classifications, applications, and design challenges of drones: A review. *Progress in Aerospace Sciences*, 91:99 – 131, 2017.
- [34] K. Joshi, S. Hong, and S. Katti. Pinpoint: Localizing interfering radios. In *Usenix NSDI*, 2013.
- [35] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.
- [36] N. Kargas, F. Mavromatis, and A. Bletsas. Fully-coherent reader with commodity sdr for gen2 fm0 and computational rfid. *IEEE Wireless Communications Letters*, 4(6):617–620, 2015.
- [37] K. I. Kim and Y. Kwon. Single-image super-resolution using sparse regression and natural image prior. *IEEE transactions on pattern analysis & machine intelligence*, (6):1127–1133, 2010.
- [38] A. Kong, J. S. Liu, and W. H. Wong. Sequential imputations and bayesian missing data problems. *Journal of the American statistical association*, 89(425):278–288, 1994.
- [39] M. Kotaru and S. Katti. Position tracking for virtual reality using commodity wifi. *IEEE CVPR*, 2017.
- [40] R. Kronberger, T. Knie, R. Leonardi, U. Dettmar, M. Cremer, and S. Azzouzi. Uhf rfid localization system based on a phased array antenna. In *Antennas and Propagation (APSURSI), 2011 IEEE International Symposium on*, pages 525–528. IEEE, 2011.
- [41] X. Li, Y. Zhang, and M. G. Amin. Multifrequency-based range estimation of rfid tags. In *RFID, 2009 IEEE International Conference on*, pages 147–154. IEEE, 2009.
- [42] M.-Y. Liu, O. Tuzel, A. Veeraraghavan, Y. Taguchi, T. K. Marks, and R. Chellappa. Fast object localization and pose estimation in heavy clutter for robotic bin picking. *The International Journal of Robotics Research*, 31(8):951–973, 2012.
- [43] G. Loianno, C. Brunner, G. McGrath, and V. Kumar. Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu. *IEEE Robotics and Automation Letters*, 2(2):404–411, 2017.
- [44] C. Luis and J. L. Ny. Design of a trajectory tracking controller for a nanoquadcopter. *CoRR*, abs/1608.05786, 2016.
- [45] Y. Ma, N. Selby, and F. Adib. Minding the billions: Ultrawideband localization for deployed rfid tags. *ACM MobiCom*, 2017.
- [46] B. R. Mahafza. *Radar systems analysis and design using MATLAB*. Chapman & Hall, 2013.
- [47] J. Maitin-Shepard, M. Cusumano-Towner, J. Lei, and P. Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2308–2315. IEEE, 2010.
- [48] J. Maitin-Shepard, M. Cusumano-Towner, J. Lei, and P. Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *2010 IEEE International Conference on Robotics and Automation*, pages 2308–2315, May 2010.
- [49] G. Mao, B. Fidan, and B. D. Anderson. Wireless sensor network localization techniques. *Computer networks*, 51(10):2529–2553, 2007.
- [50] W. Mao, J. He, and L. Qiu. Cat: High-precision acoustic motion tracking. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking, MobiCom ’16*, pages 69–81, New York, NY, USA, 2016. ACM.
- [51] R. Miesen, F. Kirsch, and M. Vossiek. Holographic localization of passive uhf rfid transponders. In *RFID (RFID), 2011 IEEE International Conference on*, pages 32–37. IEEE, 2011.
- [52] B. Narayanan, R. C. Hardie, K. E. Barner, and M. Shao. A computationally efficient super-resolution algorithm for video processing using partition filters. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(5):621–634, 2007.
- [53] M. K. Ng, H. Shen, E. Y. Lam, and L. Zhang. A total variation regularization based super-resolution reconstruction algorithm for digital video. *EURASIP Journal on Advances in Signal Processing*, 2007(1):074585, 2007.

- [54] L. M. Ni, Y. Liu, Y. C. Lau, and A. P. Patil. Landmarc: indoor location sensing using active rfid. *Wireless networks*, 10(6):701–710, 2004.
- [55] Z. Nitzan, D. Lavee, and G. Guri. Battery-assisted backscatter rfid transponder, July 1 2008. US Patent 7,394,382.
- [56] P. Pannuto, B. Kempke, and P. Dutta. Slocalization: sub- μ w ultra wideband backscatter localization. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 242–253. IEEE Press, 2018.
- [57] J. R. Cannon and E. Miles. Utilizing human vision and computer vision to direct a robot in a semi-structured environment via task-level commands. In *Intelligent Robots and Systems, IEEE/RSJ International Conference on(IROS)*, volume 01, page 366, 08 1995.
- [58] Z. Sahinoglu, S. Gezici, and I. Gvenc. *Ultra-wideband Positioning Systems: Theoretical Limits, Ranging Algorithms, and Protocols*. Cambridge University Press, New York, NY, USA, 2011.
- [59] T.-J. Shan, M. Wax, and T. Kailath. On spatial smoothing for direction-of-arrival estimation of coherent signals. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 1985.
- [60] L. Shangguan and K. Jamieson. The design and implementation of a mobile rfid tag sorting robot. In *ACM MobiSys*, pages 31–42, 2016.
- [61] L. Shangguan and K. Jamieson. Leveraging electromagnetic polarization in a two-antenna whiteboard in the air. In *ACM CoNEXT*, 2016.
- [62] R. Srinivasan. *Importance Sampling: Applications in Communications and Detection*. Springer Berlin Heidelberg, 2013.
- [63] S. Thrun. Particle filters in robotics. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 511–518. Morgan Kaufmann Publishers Inc., 2002.
- [64] B. C. Tom and A. K. Katsaggelos. Reconstruction of a high-resolution image by simultaneous registration, restoration, and interpolation of low-resolution images. In *icip*, page 2539. IEEE, 1995.
- [65] R. Tsai. Multiframe image restoration and registration. *Advance Computer Visual and Image Processing*, 1:317–339, 1984.
- [66] U. Tureli, H. Liu, and M. D. Zoltowski. Ofdm blind carrier offset estimation: Esprit. *IEEE Transactions on communications*, 48(9):1459–1461, 2000.
- [67] US Department of Defense. Department of Defense Announces Successful Micro-Drone Demonstration. <https://dod.defense.gov/News/News-Releases/News-Release-View/Article/1044811/department-of-defense-announces-successful-micro-drone-demonstration/>.
- [68] D. Vasisht, S. Kumar, and D. Katabi. Decimeter-level localization with a single wifi access point. In *Usenix NSDI*, 2016.
- [69] J. Wang, F. Adib, R. Knepper, D. Katabi, and D. Rus. RF-Compass: Robot Object Manipulation Using RFIDs. In *ACM MobiCom*, 2013.
- [70] J. Wang and D. Katabi. Dude, where’s my card? rfid positioning that works with multipath and non-line of sight. In *ACM SIGCOMM*, 2013.
- [71] J. Wang, D. Vasisht, and D. Katabi. Rf-idraw: virtual touch screen in the air using rf signals. In *ACM SIGCOMM*, 2015.
- [72] T. Wei and X. Zhang. Gyro in the air: tracking 3d orientation of batteryless internet-of-things. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 55–68. ACM, 2016.
- [73] J. Xiong and K. Jamieson. ArrayTrack: a fine-grained indoor location system. In *Usenix NSDI*, 2013.
- [74] H. Yang and A. Kak. Determination of the identity, position and orientation of the topmost object in a pile: Some further experiments. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, pages 293–298, April 1986.
- [75] L. Yang, Y. Chen, X.-Y. Li, C. Xiao, M. Li, and Y. Liu. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. In *ACM MobiCom*, 2014.
- [76] J. Zhou and J. Shi. Rfid localization algorithms and applicationsa review. *Journal of intelligent manufacturing*, 20(6):695–707, 2009.
- [77] J. Zhou, H. Zhang, and L. Mo. Two-dimension localization of passive rfid tags using aoa estimation. In *Instrumentation and Measurement Technology Conference (I2MTC), 2011 IEEE*, pages 1–5. IEEE, 2011.

Many-to-Many Beam Alignment in Millimeter Wave Networks

Suraj Jog, Jiaming Wang, Junfeng Guan, Thomas Moon, Haitham Hassanieh, Romit Roy Choudhury

University of Illinois at Urbana Champaign

{sjog2, jw27, jguan8, tmoon, haitham, croy}@illinois.edu

Abstract

Millimeter Wave (mmWave) networks can deliver multi-Gbps wireless links that use extremely narrow directional beams. This provides us with a new opportunity to exploit spatial reuse in order to scale network throughput. Exploiting such spatial reuse, however, requires aligning the beams of all nodes in a network. Aligning the beams is a difficult process which is complicated by indoor multipath, which can create interference, as well as by the inefficiency of carrier sense at detecting interference in directional links. This paper presents BounceNet, the first many-to-many millimeter wave beam alignment protocol that can exploit dense spatial reuse to allow many links to operate in parallel in a confined space and scale the wireless throughput with the number of clients. Results from three millimeter wave testbeds show that BounceNet can scale the throughput with the number of clients to deliver a total network data rate of more than 39 Gbps for 10 clients, which is up to $6.6 \times$ higher than current 802.11 mmWave standards.

1 Introduction

Millimeter wave (mmWave) is emerging as the de facto technology for next generation wireless networks [24, 35]. The abundance of bandwidth available in mmWave frequencies (above 24 GHz) led to the design of wireless radios that can operate at several Gbps [2, 39, 56] and the wireless industry is constantly pushing towards incorporating these radios in wireless products [7, 8, 24, 25, 27, 50]. Hence, mmWave will significantly change the future of wireless LANs by delivering links at fiber-like speed. This will allow wireless LANs to handle the surge in IoT and mobile devices. Furthermore, it will enable new applications like multi-user wireless VR for education, professional training, and multiplayer games, where high bandwidth data must be streamed to each user in real-time [1, 11, 29]. It will also enable large scale robotic factory automation where many robots stream continuous real-time video back to servers that run AI algorithms and generate decisions to coordinate the robots [34, 57].

Enabling the above vision, however, requires scaling mmWave networks from a single communication link to a network of many links without compromising the throughput of each user. Fortunately, mmWave radios use very directional steerable narrow beams to focus their power. This presents a significant new opportunity for exploiting dense spatial reuse

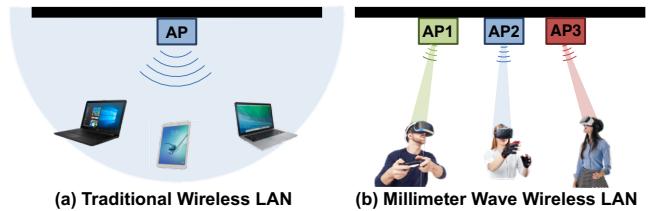


Figure 1: Spatial reuse in traditional WiFi vs mmWave networks.

to enable many links to simultaneously communicate at multi-Gbps data rates without interfering. Consider the example shown in Fig. 1. In the current broadcast model for 802.11 WLANs, whenever a node is transmitting, all other nodes must stay silent to avoid interference. With more users, the throughput is divided since the entire medium is shared. In contrast, the use of very narrow beams in mmWave networks allows several APs and clients to transmit and receive simultaneously on the same channel without interfering as shown in Fig. 1(b). Hence, mmWave can potentially scale the network throughput with the number of users by adding more APs.

The directional nature of communication, however, brings its own new challenges. Millimeter wave APs and clients need to align their narrow beams towards each other in order to communicate at very high data rates. Past mmWave research focused on developing algorithms and protocols to quickly find the best direction to align the beams for a single communication link [1, 23, 41, 49, 53, 64]. However, in a network with multiple links, selfishly choosing the best alignment for each AP-client pair independent of other APs and clients can create interference that severely harms the throughput of interfering links. First, due to multipath reflections, even if two nodes are transmitting in completely different directions, their packets might still collide. The problem is further complicated by the fact that carrier sense is ineffective at detecting interference since the narrow beams prevent mmWave radios from hearing nearby transmissions unless these transmissions are specifically directed towards them. Hence, we can rely on neither carrier sense nor the direction in which the nodes transmit to avoid interference.

In this paper, we introduce BounceNet, the first many-to-many millimeter wave beam alignment protocol that efficiently aligns the beams of many APs and clients in a manner that allows them to simultaneously communicate without interfering. To achieve this, we must address two key questions:

(1) *How does BounceNet align the beams of all the APs*

and clients in 3D space to densely pack as many links as possible? The challenge arises from the fact that the choice of beam alignment at any node is intertwined with the choices at other APs and clients. To address this, BounceNet leverages the sparsity in the mmWave channel. There is much past work that shows that mmWave signals travel along a small number of paths, e.g., 2 or 3 paths [5, 48]. This means that there is a small number of paths connecting any two nodes in the network. BounceNet leverages this sparsity to reformulate the problem as a signal level routing problem at the physical layer where wireless signals are routed along different “air paths” in a manner that avoids interference and maximizes network throughput. Routing physical signals is possible in mmWave due to the lack of scattering effects at such high frequencies which ensures the signal reflects off obstacles and does not scatter in many directions [48]. Hence, BounceNet can choose to route the signal along an isolated path by aligning the narrow beam towards that path.

By choosing a combination of direct and reflected paths to route the wireless signals, BounceNet can align the beams of all APs and clients in the network. While this allows it to maximize the number of links that can operate concurrently without interfering, it forces some APs and clients to communicate along reflected paths which typically achieve lower data rates. To address this issue, BounceNet generates several combinations of beam alignments and schedules them in different time slots; i.e., the transmissions of the links are routed along different paths in each time slot to ensure that each client gets high data rate while still maximizing the number of links that can operate simultaneously. BounceNet jointly solves the alignment and scheduling problems. We also model paths belonging to the same link as a supernode in a multi-layer conflict graph and weight them by the SNR of the path. This ensures that paths which deliver higher data rates are used more often as we describe in detail in section 6.

(2) *How does BounceNet quickly learn the paths and interference patterns in order to adapt the beam alignment in dynamic and mobile environments?* In dynamic environments, the propagation paths and the interference patterns constantly change. Thus, we must periodically perform a beam search to learn the directions of the paths along which an AP and client can communicate.¹ BounceNet must also learn the propagation paths that can result in interference between two links and, hence, needs to perform the beam search between all APs and clients in the network to learn all the possible paths. Past work has shown how to leverage sparsity to quickly learn the paths without scanning all directions and reduce the search time to a millisecond [23, 49]. However, for a network of N APs and clients, this process must be performed $O(N^2)$ times. For $N = 10$, even with fast algorithms like [23, 49], the overhead is 100 ms which is prohibitively expensive especially at multi-Gbps data rates.

¹Typically, the beam search is repeated every 100 ms in current standards like 802.11ad in order to track mobile users and maintain alignment.

Instead of performing the search independently for all APs and clients, BounceNet redesigns the beam search protocol to jointly find all the paths between the nodes. BounceNet coordinates the APs’ transmissions and then shares their measurements over the Ethernet which allows it to amortize the cost of the search and reduce it to $O(N)$. Since the beam search is inherent to mmWave and is required to maintain connectivity between clients and APs, BounceNet’s design does not introduce additional overhead compared to current standards. This allows BounceNet to quickly learn the paths and reconfigure the beam alignment to maintain high throughput as we describe in detail in section 5.

Implementation & Results: We have designed BounceNet to be backward compatible with the current mmWave wireless LAN standard 802.11ad/ay making it easy to integrate into future standards. Our design also addresses several practical challenges like side-lobe leakage from imperfect beam patterns and interference estimation. We have implemented BounceNet by using extensive real measurements from three indoor wireless testbeds:

- A 60 GHz testbed with 3° beam directional antennas.
- A 60 GHz testbed with 12° beam directional antennas.
- A 24 GHz testbed with 8-element phased arrays.

For a testbed with 10 APs and clients packed in an area of 860 sq.ft., our results show that BounceNet can scale the overall network data rate with the number of clients delivering over 39 Gbps for 10 clients. Furthermore, compared to the current 802.11ad standard that exploits spatial reuse, BounceNet can increase the average client throughput by $6.6\times$, $5\times$, and $3.1\times$ for each of the above testbeds respectively. Compared to a baseline that aligns the beams of each link independent of other links, BounceNet increases the average client throughput by $1.27\times$, $2.7\times$, and $3.4\times$ for each of the above testbeds respectively. BounceNet also improves the minimum data rate among all clients by up to $13.5\times$ compared to the baseline which can create interference that severely harms some clients. Finally, Fig. 2 shows an example snapshot of a time slot where BounceNet exploits multipath to enable all 10 APs and clients, in the 60 GHz testbed with 12° beams, to communicate at the same time without interfering, hence demonstrating BounceNet’s ability to enable extreme spatial reuse.

Contributions: We make the following contributions:

- We present the first many-to-many beam alignment protocol that can efficiently align the beams of a network of APs and clients to maximize the number of links that can operate concurrently.
- We demonstrate the opportunity of routing physical signals along different paths that bounce off the environment to improve the spatial reuse of the network. We harness this opportunity to design new algorithms that maximize network throughput while maintaining a lower bound of fairness for each client.

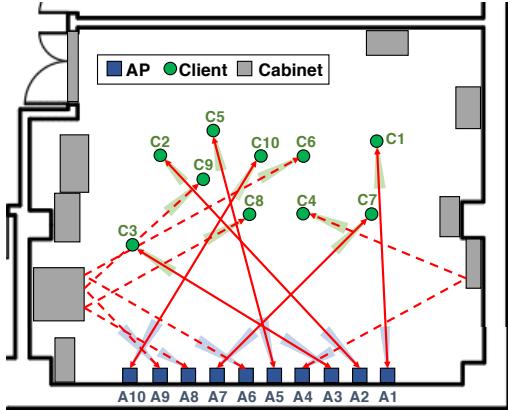


Figure 2: Example of BounceNet’s signal routing in practice.

- We extensively evaluate our system through micro-benchmark measurements, trace-driven simulations, and experiments using 3 testbeds. Our results demonstrate the first design of a wireless LAN that can deliver more than 39 Gbps to 10 clients.

2 Related Work

Millimeter Wave Networks: BounceNet is related to recent work on increasing the speed and robustness of beam alignment in mmWave networks to enable mobility [19, 20, 23, 41, 42, 49, 53, 63–65] and avoid blockage [1, 31, 32, 41, 52, 54, 61]. All this work, however, focuses on a single communication link. BounceNet is the first to demonstrate many-to-many beam alignment. It is complementary to these systems and can benefit from faster beam search to discover the paths between nodes.

BounceNet also builds on past work in mmWave that uses 60 GHz wireless links in data centers [12, 21, 66] and leverages reflections off the ceiling to improve the throughput and avoid blockage [12]. Data centers, however, have static and known topologies with predictable interference models [12], and this does not hold in 802.11 LANs where the clients can move.

Our work is also related to recent mmWave work that deploys multiple APs to deal with blockage [60, 62]. [60] leverages multiple APs and allows clients to switch between them whenever blockage occurs in VR applications. However, it requires brute-force training to map all reflectors in the environment and relies on sensors in VR headsets to track the direction of users. [62] addresses blockage by having multiple APs jointly transmit the same signal to the clients. However, the method works only for downlink traffic and requires phase and frequency synchronization to ensure the signals sum up coherently. Achieving such level of synchronization is difficult and adds significant complexity to the design [22, 46]. BounceNet opts for a simpler design that scales the throughput of the network for both downlink and uplink traffic without

requiring phase, frequency or packet level synchronization. It also learns the reflected paths in real-time.

Some recent simulation-based work for mmWave wireless PANs (Personal Area Networks) [3, 4, 17, 18, 44, 58] and mmWave mesh networks [38] tries to exploit spatial reuse. However, these solutions assume that the exact locations of the nodes are known a priori and can be used to compute the interference between links while ignoring multipath. BounceNet, on the other hand, designs and empirically tests a system that can work in the presence of multipath without prior assumptions of the clients’ locations.

Finally, [14, 16] use MU-MIMO in mmWave and demonstrate concurrent transmissions to two clients from one MU-MIMO AP. BounceNet’s beam alignment algorithm is complementary to MU-MIMO and can benefit from having APs that support MU-MIMO to further scale the gains.

Enterprise WiFi and WLANs with Directional Antennas: Past work has designed protocols for mobile ad-hoc networks and WLANs with directional antennas [9, 10, 28, 33]. However, past work can support only large cone beams (e.g. 45° and 60° cones) at data rates of at most tens of Mbps. The scale of the problem is far more extreme in mmWave with narrow pencil beams of few degrees to sub-degree beamwidth at data rates of multi-Gbps. Hence, the overhead of past protocols can be prohibitively expensive in mmWave. Moreover, most of these protocols assume the locations of the nodes are known and ignore multipath [9, 10, 28].

The closest to our work is [33] which leverages directional phased arrays at 2.4 GHz to increase spatial reuse. However, [33] assumes only APs to have directional antennas which simplifies the problem since the clients can easily perform interference detection in the omnidirectional mode. Furthermore, the scheduling algorithm in [33] is exponential in the number of APs and hence is only shown to work for 3.

Past work had designed centralized scheduling algorithms for enterprise WiFi networks [51]. However, WiFi networks are omni-directional. Extending past algorithms to deal with directionality is non-trivial since the interference or conflict graph used for scheduling is itself dependent on the choices of beam alignment and there is a combinatorial number of choices as we discuss in section 5. BounceNet jointly solves the beam alignment and scheduling problems to deliver an efficient algorithm.

3 Background

BounceNet is designed to be backward compatible with 802.11 millimeter wave standards for indoor wireless LANs. In this section, we provide a brief overview of the 802.11ad standard for 60 GHz networks [26, 40].²

²Note that another standard in the works is 802.11ay. However, it fully inherits the same PHY and MAC structure of 802.11ad. The main difference is the introduction of MIMO [15].

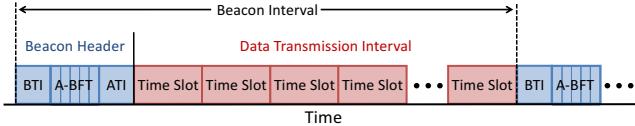


Figure 3: 802.11ad/ay Beacon Interval Structure.

The standards divide time into transmission cycles typically referred to as Beacon Intervals (BI) which consist of two phases shown in Fig. 3. The first is the association phase which is referred to as the Beacon Header Interval (BHI). It is used to associate the clients with the AP and perform beam alignment. The second is the transmission phase which is referred to as Data Transmission Interval (DTI) where time slots are allocated for communication between the AP and associated clients. We will first describe these phases for the case of a single AP and multiple clients. We will then extend our description to multiple APs.

A. Association Phase:

The beacon header shown in Fig. 3 is used to associate the clients with the AP and perform beam alignment so that both the clients and the AP know which direction they should point their beam during data transmission.

The beacon header starts with a Beacon Transmission Interval (BTI) where the AP transmits announcement frames in all directions by sequentially sweeping its narrow beam along different sectors. During this time, the clients listen to the channel in all directions using a quasi-omnidirectional beam pattern so that they can receive packets from all paths. The announcement frames are marked with the sector ID along which they are sent allowing each client to discover the directions which the AP can use to send it data packets.

BTI is then followed by Association Beamform Training (A-BFT) which reverses the above operation. The AP uses a quasi-omnidirectional beam pattern so that it can hear clients from all directions while the clients sweep their narrow beam along different sectors. This allows the AP to discover the beam directions which the client can use to send its data packets and send it feedback to inform it of these directions. A-BFT is divided into multiple slots. Each client selects a random slot to perform its sweep. If two clients collide in an A-BFT slot, they will not get feedback from the AP and they can try again in another random slot.

The above process enables the AP and client to align their beams towards each other so that they can boost their SNR and use very high data rates for data transmission. However, during this association phase and before aligning their beams, the AP and clients use a control PHY with a low data rate of 27.5 Mbps to ensure the frames can be decoded correctly at low SNR. The beacon header finally ends with Announcement Transmission Interval (ATI), where the AP and associated clients exchange control frames such as information regarding time slots that have already been allocated to the client.

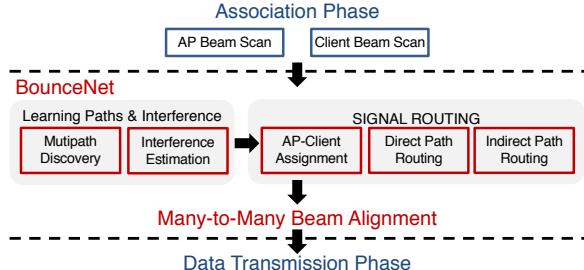


Figure 4: BounceNet’s System Architecture.

B. Transmission Phase:

The data transmission interval (DTI) is divided into time slots. The AP either uses TDMA to allocate each slot to a certain client or it allows the clients to contend for each time slot using CSMA. CSMA, however, does not work for directional networks [9, 33]. Hence, TDMA is more commonly used especially for video streaming applications where clients require dedicated slots in every beacon interval to ensure high quality and reliability.

For data transmission, the standard provides 32 different modulation and coding schemes (MCS) including single carrier modulation and OFDM modulation. Commercial products, however, adopt single carrier modulation due to the high power consumption of OFDM [45, 55]. Hence, in this paper, we will focus on single carrier: MCS1 to MCS12 which provide data rates between 385 Mbps and 4.62 Gbps [26].

C. Multiple Access Points:

In the case of multiple APs, a lead AP is selected. The lead AP divides the beacon interval into smaller beacon intervals called beacon service periods (BSP). Each BSP has its own beacon header and data transmission period and it is allocated to one AP. All other APs must stay silent during this service period. In order to enable spatial reuse, the lead AP can allocate a service period to two APs and request that they measure mutual interference and report back. If no interference occurs, it allocates the same service period to these APs in subsequent beacon intervals. Unfortunately, our results show that such a greedy mechanism for exploiting spatial reuse is unable to scale the network throughput with the number of clients.

4 BounceNet Overview

BounceNet’s goal is to align the beams of all APs and clients in the network in a manner that maximizes spatial reuse. This allows WLANs to add additional APs to quickly scale their throughput with the number of clients.

We have designed the BounceNet protocol to support *independent flows*. This means that for an AP-client pair that is assigned to communicate along a path in a time slot, its link flow runs independently of other links for that time slot. The AP and client can transmit packets on the uplink or downlink without interfering with other links. The pair does not have to share any data packets or synchronize the individual packet transmissions with other APs or clients.

BounceNet is also backward compatible with 802.11ad/ay. It maintains the same high-level structure. BounceNet's architectural flow is shown in Fig. 4. It uses a controller that sits between the association phase and the data transmission phase of the protocol. BounceNet uses association phase to learn the paths and interference in the network and then runs its signal routing algorithm which dictates the many-to-many beam alignment in the data transmission phase.

BounceNet starts with an association phase similar to 802.11 where the APs and clients sweep their beams to collect information about the directions in which their signals can reach other APs and clients. This information is then aggregated at the APs, and fed to the BounceNet controller which allows it to discover all the paths connecting any two nodes in the network. We refer to this as multipath discovery (Section 5.1). BounceNet then uses the phased array beam patterns and the learned paths to estimate the interference created by routing signals along each path (Section 5.2).

BounceNet uses the results to route physical signals along propagation paths in a manner that maximizes the number of AP-clients pairs that can communicate simultaneously. Ideally, we would have liked to treat all APs as one large AP with many paths to all clients and find the optimal routing. However, this significantly increases the complexity of the problem and will require very fast handoff between APs to allow clients to switch APs within a beacon interval.³ Hence, BounceNet assigns a single AP to each client for communication during the entire Beacon Interval.

To reduce the complexity of the system and ensure fairness, BounceNet performs signal routing in three stages:

- **Stage 1:** Associate each client to communicate with one AP for the duration of the entire beacon interval. (Section 6.1)
- **Stage 2:** Route the signal of each AP-client pair along their direct or highest throughput path in a manner that maximizes the number of links that can communicate in a given time slot without interfering. (Section 6.2)
- **Stage 3:** Route additional signals of AP-client pairs along their indirect paths to increase throughput without interfering with existing transmissions. (Section 6.3)

The above signal routing results in several beam alignments that are used for transmissions between APs and clients during each time slot of the data transmission phase. The entire process is repeated every beacon interval to adapt to changes in the environment and accommodate client mobility.

5 Learning Paths & Interference

BounceNet must first map all the paths between all nodes in the network and discover the potential interference between paths. Typically, for a network with N APs and N clients, this would require collecting $O(N^2)$ measurements. BounceNet

³Such fast handoffs are not feasible in mmWave networks because they require transferring the buffer at one AP to another AP at the time scale of few ms which would overwhelm the backhaul.

Algorithm 1 BounceNet Multipath Discovery

```

 $N \leftarrow$  Number of APs
 $\forall$  Clients  $\rightarrow$  Set quasi-omnidirectional beam
 $\forall$  APs  $\rightarrow$  Set quasi-omnidirectional beam
Begin BTI:
for  $m \in \{1, \dots, N\}$  do
    AP( $m$ )  $\rightarrow$  Set directional beam
    for  $\theta \in$  Sectors do
        AP( $m$ )  $\rightarrow$  Transmit frame in direction  $\theta$ 
         $\forall$  Clients & APs
        if Frame Received then
            Paths.AP( $m$ ) $\{\theta\} \leftarrow SNR$ 
    AP( $m$ )  $\rightarrow$  Set quasi-omnidirectional beam
Begin A-BFT:
Repeat the above process for clients.
Report Paths back to APs in transmitted frames.

```

instead redesigns the 802.11ad/ay protocol and exploits its beam alignment phase to extract all the paths from $O(N)$ measurements that are already part of the standard protocol.

5.1 Multipath Discovery

As described earlier, in case of multiple APs, the current standard divides the beacon interval into smaller beacon intervals and dedicates each interval to one AP. Instead, BounceNet aggregates them into one beacon interval with one beacon header and one data transmission interval. In particular, BounceNet only expands the BTI, shown in Fig. 3, to allow all APs to perform their beam scan of sequentially sweeping all sectors. While an AP is performing a sweep, all other clients and APs set their antenna to a quasi-omnidirectional mode and record the sector IDs of the frames they receive along with the SNR of the signals. A-BFT is then performed by assigning each client to a slot. While some client is performing its sweep, all other clients and APs set their beam to quasi-omnidirectional and record the sector IDs and SNRs of the frames received from the client. Algorithm 1 shows pseudocode for BounceNet's association phase.

The above process recovers a list of directions from which any node (AP or client) in the network can reach any other node. However, this might not be sufficient for discovering the paths between an AP and a client. Consider the example shown in Fig. 5(a) where there are three paths between an AP and a client. During BTI, we discover that the AP can reach the client by transmitting in one of three directions: 30°, 60° or 150° as shown in Fig. 5(b). During A-BFT, we discover that the client can reach the AP by transmitting in one of three directions: 30°, 110° or 150° as shown in Fig. 5(c). Unfortunately, since we do not know the position and orientation of the client, we do not know which direction at the AP corresponds to which direction at the client.

To address this, BounceNet needs to match the directions corresponding to the same paths by correlating the SNRs

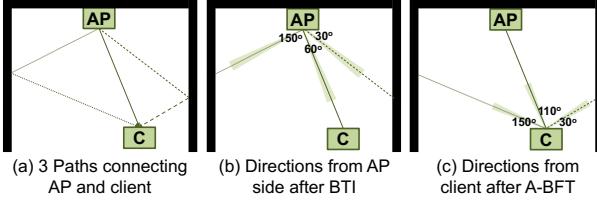


Figure 5: Multipath Discovery in BounceNet.

recorded from the client side and from the AP side. For instance, the directions corresponding to the direct path can be easily identified since typically the direct path delivers significantly higher SNR compared to indirect paths as we empirically show in Fig. 14(a) in section 8. However, in some cases, there could be two indirect paths that show similar SNR values (within 1 dB of each other). In such situations, correlation might lead to erroneous matching due to the inherent noise in SNR measurements. Fortunately though, as we show in section 8, the number of reflected paths between a pair of nodes in millimeter wave is quite small, e.g. 1 to 2 paths [5, 48]. Hence, at most, only two paths would remain ambiguous after the correlation step. BounceNet can then leverage the beam refinement option in 802.11ad which allows AP-client pairs to test pairwise directions to resolve such ambiguity. This incurs four more measurements. However, these measurements are taken while both AP and client beams are directional. Hence, they are transmitted at high data rate and incur negligible overhead.

5.2 Interference Estimation

Once we have discovered all the paths between the nodes in the network, we can estimate the interference caused by using any two paths simultaneously. BounceNet defines interference between paths as opposed to between nodes. If two paths interfere, then signals cannot be simultaneously routed along these two paths. We would like to keep the flows independent and avoid synchronization. Hence, at any point in time, both paths can be used to transmit uplink traffic, downlink traffic, or one path is used on the uplink while the other is used on the downlink. Consider a path between AP 1 and client 1 and another path between AP 2 and client 2 as shown in Fig. 6. Interference can occur in one of four cases: between AP 1 and AP 2, client 1 and client 2, AP 1 and client 2, or AP 2 and client 1 if there is a path connecting any of these pairs.

Formally, each path is defined by two angles corresponding to the direction from which it leaves one node and arrives at another node. We distinguish two types of paths:

- *Communications Paths*: defined as $(\theta_{AP_i}, \theta_{Ci})$ between AP 1 and client 1 as well as between AP 2 and client 2.
- *Interference Paths*: defined as (ϕ_{AP_i}, ϕ_{Cj}) between AP 1 and client 2 or AP 2 and client 1. They can also be defined as $(\phi_{AP_i}, \phi_{AP_j})$ or (ϕ_{Ci}, ϕ_{Cj}) .

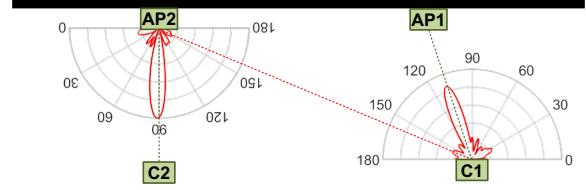


Figure 6: Estimating Interference using phased array beam patterns.

Ideally, it would be sufficient to check the directions of the paths to discover if interference occurs. Suppose AP 1 and client 1 can communicate along the path $(\theta_{AP_1}, \theta_{C1})$ and AP 2 and client 2 communicate along the path $(\theta_{AP_2}, \theta_{C2})$. In this case, for example, AP 2 will create interference at client 1 only if there exists an interference path (ϕ_{AP_2}, ϕ_{C1}) where ϕ_{AP_2} is in the direction of θ_{AP_2} and ϕ_{C1} is in the direction of θ_{C1} . A similar rule can be used to detect interference between the other pairs.

Unfortunately, such a simple interference detection scheme will not work in practice. This is because the antenna beam patterns are not ideal cones. They have side lobes and can leak signal in other directions. Consider the example in Fig. 6, while AP 2 is transmitting in direction $\theta_{AP_2} = 90^\circ$, its signal might leak along another direction $\phi_{AP_2} = 160^\circ$ and reach client 1. To address this, BounceNet incorporates the phased array transmit and receive beam patterns into its interference estimation.⁴ Specifically, to estimate interference between any pair of nodes, we consider all the interference paths between the two nodes and weight them by the beam pattern gains. Formally, when AP 2 directs its beam towards client 2 in the direction θ_{AP_2} , it will have a beam pattern of $B_{\theta_{AP_2}}(\phi)$. Similarly, when client 1 directs its beam towards AP 1 in the direction θ_{C1} , it will have a beam pattern of $B_{\theta_{C1}}(\phi)$. The interference created by AP 2 on client 1 due to an interference path $P = (\phi_{AP_2}, \phi_{C1})$ can be calculated as:

$$B_{\theta_{AP_2}}(\phi_{AP_2}) \cdot B_{\theta_{C1}}(\phi_{C1}) \cdot SNR(P)$$

where $SNR(P)$ is the normalized SNR⁵ of the path P from AP 2 to client 1 measured during multipath discovery.

The maximum interference AP 2 causes can then be estimated as the constructive sum of leakage along all paths between AP 2 and client 1:

$$INR = \sum_{P=(\phi_{AP_2}, \phi_{C1})} B_{\theta_{AP_2}}(\phi_{AP_2}) \cdot B_{\theta_{C1}}(\phi_{C1}) \cdot SNR(P)$$

where INR is the interference-to-noise ratio. BounceNet repeats this estimation eight times: from AP 1 to AP 2 and client 2, from AP 2 to AP 1 and client 1, from client 1 to AP 2 and

⁴Such patterns can be modeled or measured to account for imperfections in the mmWave phased arrays.

⁵The SNR is normalized by the antenna beam patterns used during the measurement of the SNR value in the multipath discovery phase.

client 2 and from client 2 to AP 1 and client 1. BounceNet then defines the INR between the two communication paths as the maximum INR of all these 8 values.

Two points are worth noting:

- The above interference estimation does not assume to know the location or orientation of the APs or the clients. It also does not rely on knowing the room geometry or the use of ray tracing. It only requires the direction of the propagation paths (ϕ_1, ϕ_2) between nodes in the network and the associated signal strength along the paths.
- BounceNet is able to constantly maintain an up-to-date view of the multipath and interference pattern in the network since it obtains fresh measurements from the AP and client sweeps at the start of every Beacon Interval (which is approximately 100 ms). This feature allows BounceNet to deal with dynamic network conditions and accommodate for client mobility.

6 BounceNet’s Signal Routing

Once BounceNet knows all the paths connecting the nodes and all the interference between the paths, it can route signals to/from clients in a manner that maximizes the number of AP-client pairs that can communicate in parallel. The choice of routing will govern the many-to-many beam alignment. BounceNet simplifies the problem by dividing it into three stages: *AP-Client Association*, *Direct Path Routing*, and *Indirect Path Routing*. We will elaborate on each stage below.

6.1 AP-Client Association

In the first stage, our goal is to associate each client to one AP for communication during the subsequent Data Transmission Phase of the Beacon Interval. Each client can associate with one AP, whereas each AP can serve multiple clients. Hence, for a network with N APs and N clients, we have N^N possible assignments. Trying all assignments is computationally infeasible. Thus, we develop an algorithm that sequentially assigns the clients to APs, with the objective of increasing throughput while minimizing the interference in the network. The intuition behind our algorithm is based on the following observations:

- In indoor settings, clients can typically achieve the highest data rate if they have a direct line-of-sight path to an AP. Hence, to ensure fairness, we should assign each client to an AP with a direct line-of-sight path.
- To maximize spatial reuse and throughput, we should avoid assigning multiple clients to the same AP unless the client cannot find any unassigned AP with a direct path.

Our algorithm works as follows. For each client, BounceNet keeps a list of best APs which have a direct path (high SNR path) to that client. BounceNet starts with the client with the least number of best APs and assigns it to one of the APs in its best AP list. It then adds this AP-Client pair

to a list of already assigned links. For every subsequent client, BounceNet finds an AP from its best AP list such that: (1) the AP has not yet been assigned to a client, and (2) when communicating along their direct path, the AP-Client pair creates the minimum amount of interference on the direct paths of the already assigned links.⁶ If no such AP exists, BounceNet simply picks the AP from the client’s best AP list that creates the least interference.

The above algorithm is a best effort algorithm to assign each client to an AP with a direct path that creates the least amount of interference between the links. In the worst case, the best AP list of each client contains N APs. Then, while assigning the i^{th} client, BounceNet must compute the interference created by choosing one of the $N - i$ remaining APs on the i assigned links. Hence, the complexity is: $\sum_{i=1}^N (N - i)i = O(N^3)$. This reduces the complexity from exponential $O(N^N)$ to polynomial $O(N^3)$.

6.2 Direct Path Routing

Once each client is assigned to an AP, we will have N unique direct paths. BounceNet starts by routing signals to/from clients along these direct paths. Decoupling the signal routing along the direct and reflected paths simplifies the problem and allows us to ensure fairness among links when it comes to routing signals through their highest throughput paths, i.e. their direct paths. In the next section, we will show how BounceNet routes additional signals along indirect paths to enhance throughput.

A. Scheduling of Direct Paths

BounceNet uses graphs to solve the problem. It starts by building the *Direct Path Conflict Graph*: $G(V, E)$. V represents the set of vertices in the graph. Each vertex v corresponds to a direct path between an AP-client pair. E represents the set of edges in the graph. An edge $e_{u,v}$ exists between vertices u and v if the corresponding paths interfere. We use the estimation from section 5.2 to compute the interference between paths, and if the $INR > 0$ dB, we assign the paths as interfering.

In each time slot, BounceNet’s goal is to schedule routing signals along as many paths as possible. Traditionally scheduling is modeled and solved as a minimum graph coloring problem on the conflict graph [30, 36, 47, 59]. This finds the minimum number of colors required to color the graph such that no two vertices connected by an edge share the same color. Thus, paths corresponding to vertices of the same color can be scheduled and used concurrently in the same time slot. This will minimize the number of time slots needed to schedule the paths while ensuring that each path gets one time slot to route signal to/from the client. Fig. 7(a) shows a possible minimum coloring of a graph which requires 3 colors. This

⁶The amount of interference is estimated as the sum of the INRs computed in Section 5.2.

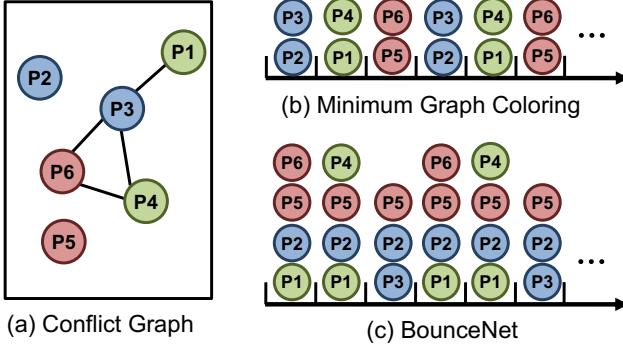


Figure 7: Scheduling of Direct Paths.

means that we can schedule all paths within 3 time slots as shown in Fig. 7(b). Since there are 6 paths, this will give 2× higher throughput than a scheduling which does not utilize spatial reuse and routes signals only along one path at any point in time.

B. Fairness in Millimeter Wave Networks

The above formulation can leverage spatial reuse to increase throughput while ensuring that each client gets an equal share of the time on the channel. This notion of fairness, however, is suboptimal in mmWave networks and needlessly wastes throughput. Due to the use of very directional beams in mmWave networks, the medium is no longer “equally” shared among all clients. Consider the example in Fig. 7(a). Paths 2 and 5 do not interfere with any other path and hence we should route signals through these paths in every time slot. Not doing so would reduce the throughput without benefiting anyone in the network. On the other hand, paths 4 and 6 share their medium with two other paths since they interfere with two other paths. Hence, a path should get a share of the medium which is at least a fraction of the number of paths it shares its medium with. For example, we should route signals through paths 4 and 6 in 1/3 of the time slots, whereas we should route signals through paths 2 and 5 in all time slots since they interfere with no one.

Formally, if a path interferes with d other paths, it shares its medium with these d paths and hence should get a share of at least $1/(d+1)$. In the conflict graph G , d will correspond to the degree of the vertex, i.e. the number of edges that the vertex has. Using this new notion of fairness, we develop an algorithm to route signals through direct paths in a manner that achieves higher throughput while maintaining fairness.

C. BounceNet’s Algorithm

BounceNet starts by trying to maximize the number of paths that can be used in each time slot. Maximizing the number of paths is theoretically equivalent to solving a maximum independent set problem. The maximum independent set refers to the maximum number of vertices that do not share any edges. For example, in Fig. 7(a), the maximum indepen-

Algorithm 2 BounceNet Scheduling of Direct Paths

```

 $G(V, E) \leftarrow$  Direct Path Conflict Graph
 $M \leftarrow$  Number of time slots in beacon interval
 $F_1(u) = M \forall u \in V$ 
for  $t \in \{1, \dots, M\}$  do
     $W_t \leftarrow$  WEIGHTEDMAXINDEPENDENTSET( $G, F_t$ )
    for  $u \in W_t$  do
        if  $F_t(u) > 2(d(u) + 1)$  then
             $F_{t+1}(u) = F_t(u) - (d(u) + 1)$ 
        else
             $F_{t+1}(u) = 0$ 

```

dent set can be formed of paths 1, 2, 4, and 5 since none of these paths share edges, i.e. none of them interfere. Routing signals through these paths in every time slot will achieve the highest possible throughput. However, it will result in starvation of some clients whose paths are never included in the maximum independent set, e.g. Path 3 in Fig. 7(a).

Instead, BounceNet uses a variant of the same problem referred to as the Weighted Maximum Independent Set. The idea is to give each vertex u a weight $F(u) \geq 0$. We then find the set of vertices W that maximize the sum of weights such that no two vertices in W share an edge. More formally, we find the set W that satisfies:

$$\text{maximize } \sum_{u \in W} F(u) \text{ such that } \forall u, v \in W, e_{u,v} \notin E \quad (1)$$

BounceNet solves the above optimization problem for every time slot and schedules to route paths corresponding to the vertices in W to each of the time slots. After each time slot, BounceNet decrements the weights of each of the vertices in W by an amount proportional to the interference it creates in the network, i.e. the degree of the vertex d . Hence, if we initialize all the weights equally, then for the first time slot, BounceNet will pick a Maximum Independent Set. However, as the algorithm proceeds, the weights of the scheduled paths keep getting decremented, and eventually paths that interfere with the paths in the Maximum Independent Set start to get picked in W , and in turn get scheduled.

Pseudocode of this algorithm is shown in Algorithm 2. Fig. 7(c) shows an example of the output of BounceNet’s direct path routing. In this example, BounceNet’s algorithm achieves 3.66× higher throughput while ensuring fairness, i.e. each path gets scheduled at least $1/(d+1)$ of the time.

D. Analysis

If BounceNet wishes to schedule the nodes into M slots, it initializes all the weights to M . Then, every time a vertex u is picked, its weight is decremented by $d(u) + 1$ where $d(u)$ is the degree of this vertex. After this vertex has been picked up $M/(d(u) + 1)$ times, its weight becomes 0. Once the weight of a vertex becomes zero, its inclusion in W can no longer help maximize the sum of weights, and hence it does not get

picked up (or in our context, the path is no longer used) after that. However, by the time the weight of the vertex reaches 0, it has already been scheduled in $1/(d(u) + 1)$ of the time slots and hence fairness is achieved. For example, if a vertex has degree $d = 0$, i.e. it does not interfere with anyone, it will be picked up every time since it will always help maximize the sum of weights. Every time it is picked, its weight is decremented by 1. Its weight will reach 0 only after it has been scheduled M times which means it has been scheduled in all time slots. In Appendix A, we prove the following lemma:

Lemma 6.1 *If $t = O(M \log(NM))$, then $F_t(u) = 0 \forall u \in V$*

Algorithm 2, however, requires solving a Weighted Maximum Independent Set problem which is NP-hard [13]. This would require an exponential time algorithm to find the optimal solution, which would be infeasible for any real-time implementation. We use the approximation algorithm from [13] to solve this problem. Empirically we find that the algorithm is at most two timeslots worse than optimal. However, in many cases, the algorithm achieves the optimal. This is because the sparsity renders the *Direct Path Conflict Graphs* in mmWave networks as chordal with very high probability. Chordal graphs are graphs in which all cycles of four or more vertices have a *chord*. For such graphs, [13] is optimal.

6.3 Indirect Path Routing

In this section, we will show how BounceNet will route additional signals along indirect multipath routes to increase the throughput without creating interference to signals being routed along the direct path.

BounceNet's indirect path routing is best understood through an example. Let us consider the direct path scheduling result shown in Fig. 7(c). During the first time slot, paths 1, 2, 5 and 6 were scheduled. Hence, clients 1, 2, 5 and 6 can communicate on their direct paths during this time slot. Note that a client can route its signal through only one path during any time slot. As a result, we only need to consider whether we can route signals through multipath for clients 3 and 4.

To this end, BounceNet forms an *Indirect Path Conflict Graph*. This graph includes vertices corresponding to the direct paths that have been scheduled as well as vertices corresponding to indirect paths of AP-client pairs that have not been scheduled in this time slot. Fig. 8(a) shows an example of this graph where client 3 has two indirect paths to its AP and client 4 has three indirect paths to its AP. Indirect path vertices corresponding to the same client are always in conflict since the client can use only one of those indirect paths. Hence, vertices corresponding to indirect paths of the same client form a fully connected subgraph which we will refer to as a *supernode*. We then estimate the interference that the indirect paths can create on direct paths that are already scheduled as well as other indirect paths.

Direct paths have already been scheduled and hence they are locked. Any indirect path that interferes with the direct

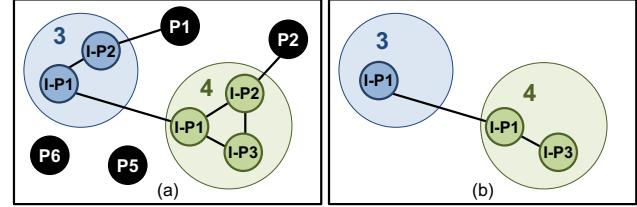


Figure 8: Indirect Path Conflict Graph before & after pruning.

path cannot be used in this time slot and hence can be eliminated from the indirect path conflict graph. Thus, BounceNet prunes the graph by removing all vertices that interfere with direct paths as well as vertices corresponding to direct paths as shown in Fig. 8(b). The resulting graph is typically much smaller and formed only of supernodes and vertices corresponding to indirect paths. BounceNet can route signals through any of the remaining indirect paths without interfering with signals being routed through the direct paths.

In order to schedule indirect paths, BounceNet uses the same algorithm as before where it maximizes throughput by solving a maximum weighted independent set problem on the *Indirect Path Conflict Graph*. However, BounceNet has to take into account two key differences:

- Unlike direct paths where there is small variance in SNR, the SNR of indirect paths can vary significantly as we will show in section 8. Hence, BounceNet should give indirect paths with higher SNR more weight. To do so, BounceNet gives each *supernode* a weight of M and divides this weight among its indirect path vertices in a manner proportional to the data rate that each indirect path can achieve. For example, if *supernode* 4 in Fig. 8 has indirect paths with SNRs 3 dB, 5 dB, and 7 dB, then it can deliver data rates of around 1.1 Gbps, 1.9 Gbps, and 2.5 Gbps respectively. Hence, its indirect paths will be weighted as $0.2M$, $0.35M$, and $0.45M$. This ensures that the higher data rate paths have a higher chance of getting picked.
- The degree d of a vertex no longer corresponds to the number of other clients it shares the medium with since vertices of the same *supernode* belong to the same client. Hence, instead of decrementing the weight of the node by $d + 1$, we decrement it by $d - s + 1$ where s is the number of other vertices that remain in the supernode after pruning the graph. For example, in Fig. 8(b) the indirect path in *supernode* 3 has $s = 0$ whereas in *supernode* 4 have $s = 1$.

7 Testbed and Implementation

We evaluated BounceNet using three indoor testbeds that operate at 60 GHz and 24 GHz. The 60 GHz testbeds used Pasternack PEM009 radios [43] shown in Fig. 10(a). One testbed is equipped with directional antennas with beamwidth 3° and the other with 12° antennas shown in Fig. 10(b). The 60 GHz Pasternack modules are connected to USRP software defined radios through a Balun circuit to sample the signal.



Figure 9: Indoor Experimental Space: (a) Lecture Hall (b) Atrium (c) Lounge (d) Empty Room (e) Lab (f) Office Space.

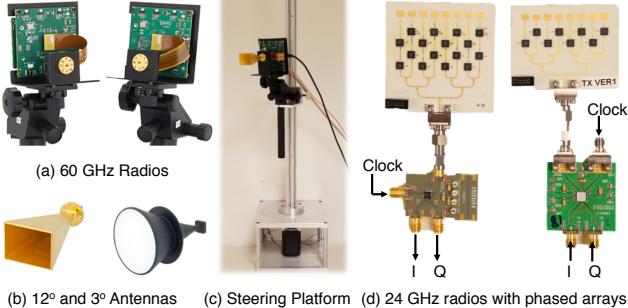


Figure 10: Experimental hardware used to evaluate BounceNet.

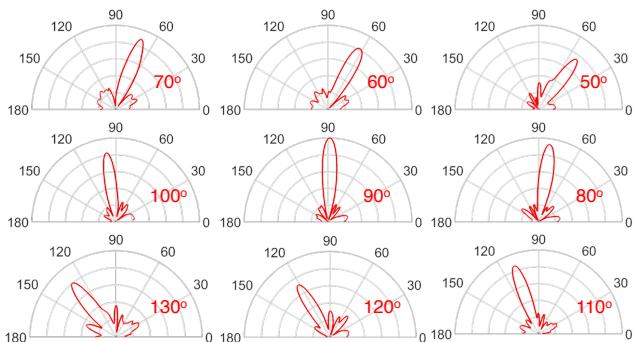


Figure 11: Example beam patterns of the 24 GHz phased arrays.

They are also mounted on a steerable platform shown in Fig 10(c) controlled through an Arduino.

The 24 GHz testbed used two radios, each equipped with an 8-element phased array shown in Fig. 10(d). The radios use HMC815B and HMC977 IQ up/down converters from Analog Devices which operate between 21 GHz and 27 GHz with 3.75 GHz of bandwidth. The integrated boards shown in Fig. 10(d) also include RF amplifiers and a frequency doubler. The boards are fed a clock in the range 10.5 GHz to 14.5 GHz from a TI LMX2594 PLL which is doubled to the 24 GHz range. The I and Q signals are connected to USRP software defined radios where the signals are collected. Fig. 11 shows examples of the beam patterns of the phased array that we obtain from our own empirical measurements. Note that while the beam patterns from some commercial phased arrays have much larger side lobes, we are able to achieve beam patterns as shown in Fig. 11 by leveraging the online algorithm for phased array calibration presented in [37].

We use the Tektronix DPS77004SX oscilloscope which

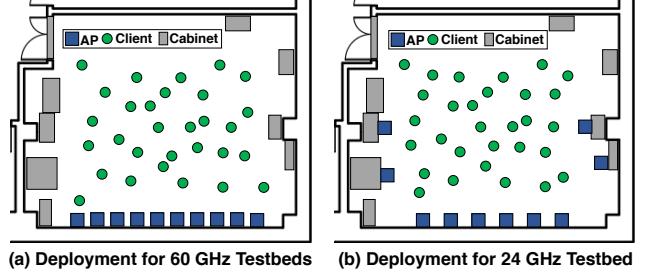


Figure 12: Placement of APs in the 60 GHz and 24 GHz testbeds.

samples at 200 GS/s and has a bandwidth of 70 GHz to calibrate the transmitted power of both 60 GHz and 24 GHz radios to match FCC regulations. We also use it to calibrate the measured power and noise floor of the USRPs.

Due to the large overhead of real-time processing and the limited bandwidth of USRPs, we use the software radios to measure interference and signal-to-noise ratio, which we map to the minimum achievable data rate using the receiver sensitivity table of 802.11ad [26] with 1% packet loss rate. We then used these testbed measurements to run trace-driven simulations using an 802.11ad ns3 library that takes phased array beam patterns into account [6]. We also modified this library to implement BounceNet. We then empirically verified the results by testing the interference and making sure any pair of paths used in a given time slot does not interfere. We then report the data rates per client as well as the overall network data rate. Finally, we also study the impact of our system when integrated with higher layer protocols like TCP and UDP and report application level throughput results.

We collected measurements in different rooms in order to evaluate the level of multipath and verify that BounceNet can exploit this multipath to maximize the number of links. We tested in six different types of rooms shown in Fig. 9: a lecture hall, an atrium, a lounge, a completely empty room, a lab space, and an office space. The full BounceNet protocol was evaluated in the lab which is 860 sq.ft. of space. The APs were deployed along the walls of the lab with the clients scattered across the room as shown in Fig. 12. We vary the number of APs and clients from 1 to 10. In every run, the clients are assigned randomly to these locations. We tested 5000 different configurations of locations. To emulate mobility, we move the clients in 5 cm steps along a path where we run scans and collect measurements for each step in the path.

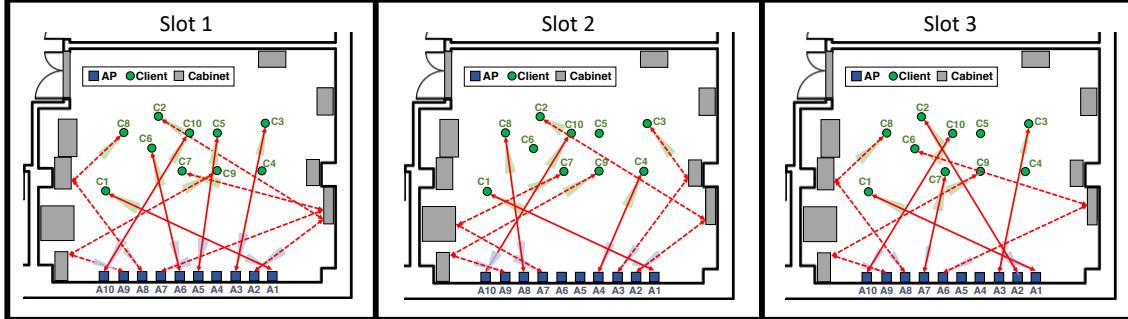


Figure 13: Beam Alignments computed by BounceNet for 12° beam testbed.

8 Microbenchmark Results

We start our evaluation with a few microbenchmarks that provide insights into the working of the system as well as the characteristics of mmWave networks before we present the evaluation results.

A. Multipath in mmWave Networks:

BounceNet leverages multipath in mmWave networks to maximize the number of links that can operate at the same time. Table 1 shows the distribution of the number of reflected multipath per link in each of the six rooms shown in Fig. 9. The results show that for all rooms except the atrium, in about 80% of the cases the client has 1 to 2 reflected paths through which it can route its signal to the AP. This is expected as the atrium is a large open space with limited reflectors. The results also show that very few clients see 3 or 4 indirect paths due to sparsity in mmWave.

Fig. 14(a) shows the CDF of the SNRs of the direct and reflected paths respectively measured from our testbeds. We observe that direct paths always provide sufficient SNR to support the highest data rate of 4.62 Gbps. The variation in direct path SNRs is small and the median SNR of direct paths is 15 dB larger than the median SNR of reflected paths which motivates BounceNet’s design to split routing signals along direct and indirect paths into two stages. Furthermore, the SNRs of indirect paths can vary between 5 dB to 20 dB and hence it is important to take the SNR of indirect paths into account when deciding which indirect path to route signals through as we have described in section 6.3.

B. Accuracy of Interference Estimation:

Here, we evaluate the accuracy of BounceNet’s ability to correctly estimate interference. We choose 100 different pairs of links from our testbed and measure the ground truth interference between every pair. For each pair, we consider both the direct path and indirect paths. To obtain the interference estimates from BounceNet, we perform the association phase using the experimental setup. Then, we use the measurements to find all the paths and compute the INR as described in section 5.2. Fig. 14(b) shows the CDF of the absolute error between the ground truth interference measurements and the

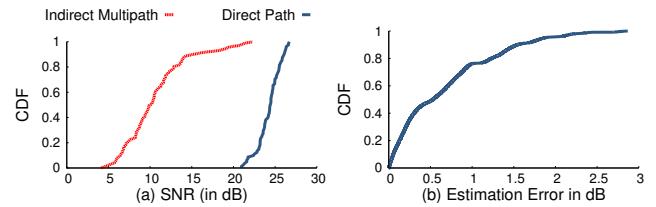


Figure 14: **Microbenchmarks:** (a) SNR of indirect vs. direct paths. (b) Interference estimation error.

Table 1: Percentage of Links with n Reflected Paths

Room	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$
Lecture Hall	0	20	46.6	26.6	6.6
Atrium	5	95	0	0	0
Lounge	0	46.6	50	3.3	0
Empty Room	0	21.0	52.6	26.4	0
Lab	0	37.4	41.4	21.2	0
Office Space	0	30	45	15	5

estimated values from BounceNet. BounceNet’s median error is 0.52 dB and 90th percentile error is 1.54 dB which is within the 3 dB tolerance for various mmWave MCSs. BounceNet is able to achieve such high accuracy in predicting the interference in the network because it accounts for both the multipath in the environment as well as the imperfections in antenna beam patterns. Furthermore, it is able to do this using only a linear number of measurements $O(N)$, therefore avoiding the need to explicitly measure interference between every pair which would be $O(N^2)$.

C. BounceNet’s Signal Routing

In Fig. 13, we present additional examples of BounceNet’s beam alignments in the 12° testbed. We pick one client configuration and plot the beam alignments computed by BounceNet for the first three time slots. We can see that BounceNet makes use of both direct and reflected paths in order to squeeze in as many links as possible for communication during the time slot. Furthermore, over the three time slots, BounceNet schedules the direct paths for different clients, thus clients get a chance to use their direct paths in different time slots. Clients that create less interference such as C1 and C10 get to use their direct paths in all time slots whereas clients that create more interference such as C2 or C7 get to use it once.

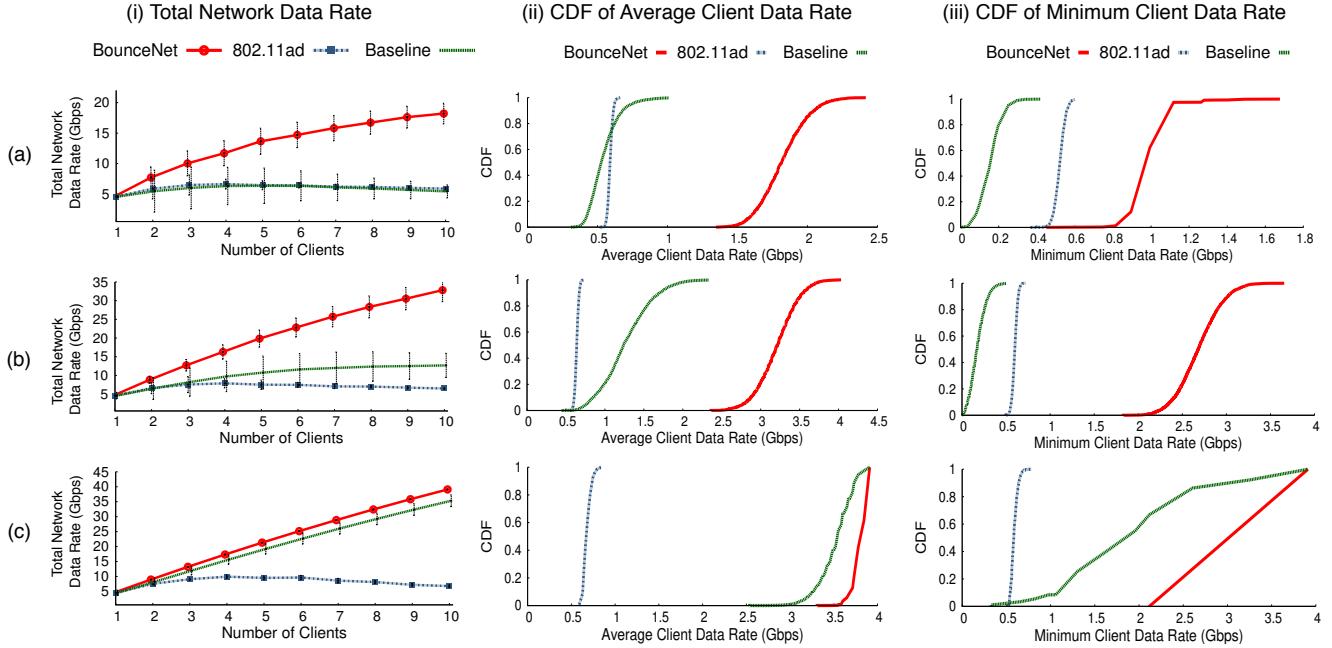


Figure 15: Data rates in BounceNet, 802.11ad and baseline for (a) 24 GHz phased array (b) 60 GHz with 12° beams (c) 60 GHz with 3°.

9 Evaluation Results

We will present our main evaluation results here. We will start by describing our baselines and evaluation metrics.

A. Compare Schemes: We compare BounceNet to:

- (1) **802.11ad with Spatial Reuse:** As described in section 3, the current standard provides a greedy mechanism for exploiting spatial reuse by measuring pairwise mutual interference and merging links that do not interfere into the same slots. If the nodes detect changes in the interference in the network, they reset to transmitting in exclusive time slots.
- (2) **Baseline:** Our baseline will consider independently aligning the beams of each AP and client and letting them transmit. To give the baseline an edge, we assume that the APs and clients can perform their beam search without creating any interference. Hence, they can find the right alignment in $O(N)$ and then use it for data transmission.

B. Metrics: We evaluate BounceNet using these metrics:

- **Total Network Data Rate:** The aggregate data rate of all the clients in the network.
- **Average Client Data Rate:** The average data rate of the clients in the network.
- **Minimum Client Data Rate:** The minimum data rate among all clients in the network.
- **Fraction of Time on the Channel:** The fraction of time slots a client gets to transmit in; used to evaluate fairness.
- **Average Client Throughput:** The average application layer throughput of a client using TCP or UDP flows.

C. BounceNet Data Rate Gain:

We start by evaluating the gains in total network data rates. Fig. 15(i) shows the total network data rate as a function of the number of clients in a network with 10 APs for BounceNet, 802.11ad, and the baseline. As the number of clients increases, BounceNet is able to scale the total network data rate with the number of clients to deliver a total of 39.2 Gbps and 32.8 Gbps data rates for 10 clients using 60 GHz with 3° and 12° beams respectively. For 24 GHz, BounceNet is able to achieve 18.2 Gbps for 10 clients. This is expected as sidelobe leakage of phased arrays creates more interference in the network which limits spatial reuse.

802.11ad, on the other hand, is unable to properly exploit spatial reuse and shows limited gains. Specifically, for the case of 10 clients, BounceNet achieves 6.6×, 5×, and 3.1× gain in network throughput as compared to 802.11ad for 3° beam, 12° beam, and the phased array respectively. This is due to 802.11ad's inefficiency which stems from requiring pairs of links to measure mutual interference during data transmission and merge these links during the following beacon interval only if they do not interfere. The baseline can exploit spatial reuse for 3° beam since the interference in this case is very limited. Hence, for 10 clients with 3° beam, BounceNet only achieves 1.27× gain over the baseline. This gain, however, increases to 2.7× and 3.4× for 12° beam and the phased array respectively where there is more interference. In fact, the baseline is unable to exploit spatial reuse and scale network throughput in such cases.

In Fig. 15(ii) we plot the CDF of the average data rate achieved by the clients across all the runs with 10 clients in

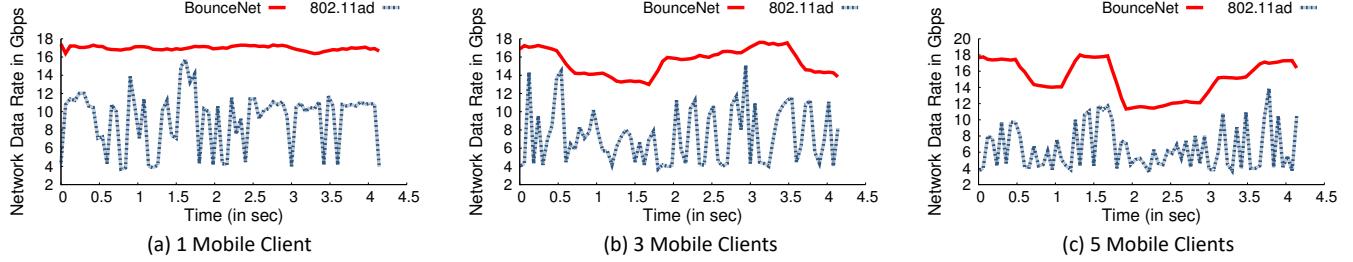


Figure 16: Mobility: This figure shows that BounceNet can adapt to changing and mobile clients whereas 802.11ad is unable to exploit spatial reuse in mobile networks.

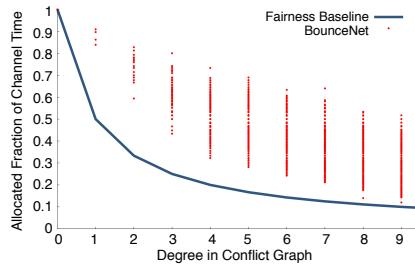


Figure 17: Client's share of time on the channel.

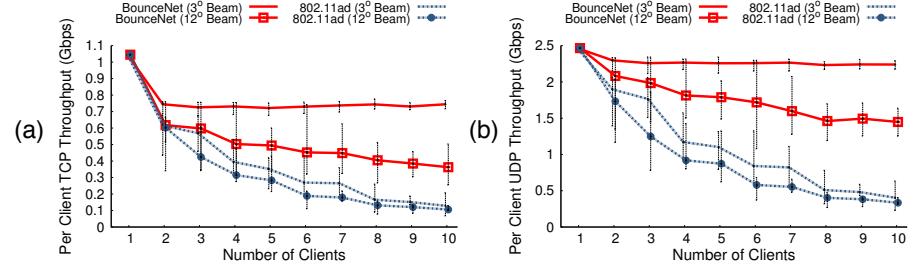


Figure 18: BounceNet's Application Level Average Throughput Under (a) TCP and (b) UDP.

the network. A client in BounceNet can achieve a 50th percentile average data rate of 3.8 Gbps for 3° beam, 3.25 Gbps for 12° beam, and 1.81 Gbps for the phased array. Whereas in 802.11ad, the 50th percentile average data rate is around 0.6 Gbps in all three cases. The baseline, however, shows high average data rate of 3.4 Gbps for 3° beam which decreases to 1.26 Gbps for 12° and 0.5 Gbps for the phased array. Hence, with wider beams, simply ignoring interference would result in an even worse performance than 802.11ad.

Two points are worth noting. First, each of the 10 clients in BounceNet can achieve a 90th percentile average data rate of 3.9 Gbps for 3°, 3.7 Gbps for 12°, and 2 Gbps for the phased array. This is a small deviation from the median data rate which shows that BounceNet is fair in dividing the rate across the clients. Second, while BounceNet scales the network throughput, the overhead of beam alignment starts to kick in. This, however, can be addressed by employing faster beam alignment protocols [23, 49, 64].

We also plot the CDF of the minimum data rate among all clients in Fig. 15(iii), across all the runs with 10 clients in the network. The figure shows that BounceNet can significantly improve the minimum and benefit worst case clients which can suffer from interference. BounceNet can improve the minimum data rate of any client in the network by 13.5× for 12° beam and 7.5× for phased arrays as compared to the baseline. This is because the baseline does not try to avoid interference, and hence clients that suffer from interference can really benefit from BounceNet.

In Appendix B, we present additional results when there are only 5 APs in the network. This allows us to evaluate

BounceNet in scenarios where clients outnumber the APs.

D. Adapting to Changes and Mobile Clients:

To understand BounceNet's ability to adapt to mobile clients, we examine what happens to the total network data rate as clients move for both BounceNet and 802.11ad. As the baseline does not actively try to optimize for spatial reuse, we expect the total network data rate to remain smooth albeit lower than BounceNet.

We run an experiment where there are five clients in the network and we vary the number of clients that are moving. Fig. 16 shows the total network data rate versus time, when one client, three clients or five clients are moving. This figure shows that BounceNet can continue to maintain a high data rate as the clients move. For one client moving, BounceNet achieves almost a constant data rate. As more clients move, the interference patterns in the network change, and hence the maximum achievable data rate changes. The figure shows that BounceNet can quickly adapt to changes and continue to exploit spatial reuse.

On the other hand, the data rate in 802.11ad fluctuates significantly and keeps falling back to the case of no spatial reuse. This is because 802.11ad merges AP-client pairs only after measuring the mutual interference during the data transmission phase. Hence, it takes 802.11ad several beacon intervals ($\approx 100\text{ms}$) to exploit spatial reuse. By that time, the client has moved and the interference patterns have changed. Even if one client moves, it can affect the interference patterns of many links. Fig. 16 shows that as more clients move, the interference patterns change faster, and hence 802.11ad is unable to properly exploit spatial reuse.

E. BounceNet Fairness:

Recall from section 6.2 that fairness in mmWave networks depends on how much each client interferes with other clients. If a client interferes with d other links, it should get at least a fraction of $1/(d+1)$ of time on the channel. For each of our 5000 experiments, we compute the fraction of channel time that a client interfering with d other links in the network obtains as a result of BounceNet’s algorithm. Fig. 17 plots this fraction for all clients against their degree in the conflict graph (equivalent to their number of interfering links). The figure shows that the algorithm guarantees that all points lie above the line denoted by $Fraction = 1/(d+1)$. Hence, every link gets at least its fair share of channel time in BounceNet.

F. Application Level throughput in BounceNet:

In order to understand whether BounceNet’s gains translate to higher layer network throughput, we evaluated the application level throughput achieved using BounceNet and 802.11ad under TCP and UDP traffic flows in ns3. Fig. 18 shows the throughput versus the number of clients. BounceNet’s scaling properties are maintained with roughly the same gain over the 802.11ad standards. For 10 links, BounceNet can achieve a UDP throughput of 1.44 Gbps for 60 GHz with 12° beamwidth and 2.23 Gbps for 3° beamwidth. As expected, the application level throughput is lower than the MAC data rates due to the overhead of headers. For TCP the throughput is even lower with 360 Mbps for 12° beamwidth and 740 Mbps for 3° beamwidth. This is expected as TCP has larger overhead and does not perform well in wireless networks.

G. Results Summary:

802.11ad requires multiple beacon intervals to detect interference in the network and schedule concurrent transmissions. While this would work in completely static scenarios where the paths do not change, it is inefficient in mobile or dynamic environments. Our results show that in such cases, 802.11ad keeps resetting to a configuration with no spatial reuse. BounceNet, on the other hand, is able to maintain an up-to-date view of the paths and interference every Beacon Interval which allows it to achieve significant gains especially for narrower beams (e.g. 3°) where the potential for spatial reuse is very high.

The baseline, on the other hand, performs well with narrow beams (e.g. 3°) and on average achieves comparable results to BounceNet. However, the tail of the distribution is very long. Specifically, clients that experience interference would achieve significantly lower data rates than both BounceNet and 802.11ad. The performance quickly degrades for wider beams where there is more interference between links. BounceNet can achieve the best of both worlds by combining efficient path learning and interference estimation algorithms with signal routing and beam alignment. Hence, BounceNet can exploit spatial reuse for both very narrow beams and wide beams and can perform well in both static and mobile environments.

10 Limitations and Discussion

Few points are worth noting.

- Our current evaluation is limited by today’s hardware which makes it infeasible to implement a full-fledged real-time version of our system. Cheap commercial mmWave devices [2, 39, 56] do not provide access to the lower layers: PHY and MAC. On the other hand, the hardware we used costs around \$14,000 for the RF front end of one TX/RX pair, making it prohibitively expensive to scale the implementation. Note, however, that our simulations are not based on ray-tracing or any channel modeling. Rather, they are based on actual measurements of SNRs and beam scanning through a labor-intensive study that generated over 5000 configurations. We have also used two pairs of links to verify that our interference estimates are accurate. Our results show a significant opportunity to scale the throughput in mmWave networks and we believe the protocol can be implemented on cheap commercial devices if the chip manufacturers open up the firmware.
- BounceNet’s protocol is mainly designed for continuous traffic in applications like VR, 3D video streaming, and Robotics. To deal with bursty traffic, one can leverage the polling mechanism available in 802.11ad [26] to obtain a *real-time* view of the traffic demands for different clients during the Beacon Interval, and adjust the conflict graph based on the traffic.
- BounceNet’s interference estimation relies on accurate measurements of the SNR. The high directionality in mmWave networks reduces multipath fading and channel fluctuations which allows us to achieve accurate estimates as we show in section 8. However, to address the case of noisy and unstable SNR measurements, we take a more conservative approach for determining when two links interfere (Section 6.2.A). The threshold to determine interference can be adjusted as a trade-off between robustness to noisy SNR estimates and maximizing spatial reuse.

11 Conclusion

In this paper, we introduced BounceNet, the first many-to-many millimeter wave beam alignment system that can efficiently align the beams of many APs and clients in a manner that allows them to simultaneously communicate without interfering. We evaluated BounceNet using three experimental testbeds and demonstrated that it can enable dense spatial reuse and scale the total network throughput with the number of APs and clients.

Acknowledgments: We would like to thank our shepherd, Prof. Lin Zhong, and the reviewers for their comments. We would also like to thank Piotr Indyk for his input on the proof. Lastly, we also thank the Systems and Networking Group (SyNRG) at UIUC for their feedback. This work is funded in part by NSF Award CNS-1750725.

References

- [1] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling high-quality untethered virtual reality. In *NSDI*, 2017.
- [2] Acer. TravelMate P6 TMP658-M-70S3 Laptop.
- [3] Xueli An and Ramin Hekmat. Directional MAC protocol for millimeter wave based wireless personal area networks. In *Vehicular Technology Conference, 2008. VTC Spring 2008. IEEE*, pages 1636–1640. IEEE, 2008.
- [4] Xueli An, Shuang Zhang, and Ramin Hekmat. Enhanced mac layer protocol for millimeter wave based WPAN. In *2008 IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5. IEEE, 2008.
- [5] Christopher R. Anderson and Theodore S. Rappaport. In-Building Wideband Partition Loss Measurements at 2.5 and 60 GHz. *IEEE TWC*, 2004.
- [6] Hany Assasa and Joerg Widmer. Extending the ieee 802.11ad model: Scheduled access, spatial reuse, clustering, and relaying. In *Proceedings of the Workshop on Ns-3, WNS3 ’17*, pages 39–46, New York, NY, USA, 2017. ACM.
- [7] Matt Branda. Qualcomm Research demonstrates robust mmWave design for 5G. Qualcomm Technologies Inc., November 2015.
- [8] Neeraj Choubey and Ali Yazdan Panah. Introducing Facebook’s new terrestrial connectivity systems-Terragraph and Project ARIES, Facebook Research, 2016.
- [9] Romit Roy Choudhury and Nitin H Vaidya. Deafness: A MAC problem in ad hoc networks when using directional antennas. In *Proceedings of the 12th IEEE International Conference on Network Protocols.*, pages 283–292, 2004.
- [10] Romit Roy Choudhury, Xue Yang, Ram Ramanathan, and Nitin H Vaidya. On designing MAC protocols for wireless networks using directional antennas. *IEEE Transactions on Mobile Computing*, 5(5):477–491, 2006.
- [11] Adam Connor-Simons. Enabling wireless virtual reality, MIT News, 2016.
- [12] Yong Cui, Shihan Xiao, Xin Wang, Zhenjie Yang, Chao Zhu, Xiangyang Li, Liu Yang, and Ning Ge. Diamond: Nesting the Data Center Network with Wireless Rings in 3D Space. In *NSDI*, 2016.
- [13] András Frank. Some polynomial algorithms for certain graphs and hypergraphs. In *Proceedings of the Fifth British Combinatorial Conference*, pages 211–226, 1975.
- [14] Y. Ghasempour, M. K. Haider, and E. W. Knightly. Decoupling beam steering and user selection for MU-MIMO 60-GHz WLANs. *IEEE/ACM Transactions on Networking*, pages 1–14, 2018.
- [15] Yasaman Ghasempour, Claudio RCM da Silva, Carlos Cordeiro, and Edward W Knightly. IEEE 802.11ay: Next-generation 60 GHz communication for 100 Gb/s Wi-Fi. *IEEE Communications Magazine*, 55(12):186–192, 2017.
- [16] Yasaman Ghasempour, Muhammad K Haider, Carlos Cordeiro, Dimitrios Koutsonikolas, and Edward W Knightly. Multi-stream beam-training for mmwave mimo networks. In *ACM MobiCom*, 2018.
- [17] Michelle X Gong, Dmitry Akhmetov, Roy Want, and Shiwen Mao. Directional CSMA/CA protocol with spatial reuse for mmWave wireless networks. In *Global Telecommunications Conference (GLOBECOM)*, pages 1–5. IEEE, 2010.
- [18] Michelle X Gong, Robert Stacey, Dmitry Akhmetov, and Shiwen Mao. A directional CSMA/CA protocol for mmWave wireless PANs. In *Wireless Communication and Networking Conference*, pages 1–6. IEEE, 2010.
- [19] Muhammad Kumail Haider, Yasaman Ghasempour, and Edward W Knightly. Search light: Tracking device mobility using indoor luminaries to adapt 60 GHz beams. In *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 181–190. ACM, 2018.
- [20] Muhammad Kumail Haider, Yasaman Ghasempour, Dimitrios Koutsonikolas, and Edward W Knightly. Listeer: mmwave beam acquisition and steering by tracking indicator leds on wireless aps. 2018.
- [21] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *ACM SIGCOMM*, 2011.
- [22] Ezzeldin Hamed, Hariharan Rahul, Mohammed A Abdelghany, and Dina Katabi. Real-time Distributed MIMO Systems. In *Proceedings of ACM SIGCOMM*, 2016, pages 412–425.
- [23] Haitham Hassanieh, Omid Abari, Michael Rodriguez, Mohammed Abdellahany, Dina Katabi, and Piotr Indyk. Fast millimeter wave beam alignment. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2018.
- [24] Kelly Hill. A look at Verizon’s fixed millimeter wave testing. RCR Wireless News, May 2017. <http://www.rcrwireless.com/20170501/test-and-measurement/verizon-fixed-millimeter-wave-testing-tag6>.
- [25] Huawei. Huawei to bring 73GHz mmWave Mu-MIMO live demo to Deutsche Telekom, Press Release, 2016.
- [26] IEEE Standards Association. IEEE Standards 802.11ad-2012: Enhancements for Very High Throughput in the 60 GHz Band, 2012.
- [27] Intel Inc. Intel accelerates path to 5G, Press Release, 2016.
- [28] Gentian Jakllari, Wenjie Luo, and Srikanth V Krishnamurthy. An integrated neighbor discovery and MAC protocol for ad hoc networks using directional antennas. *IEEE Transactions on Wireless Communications*, 6(3):1114–1024, 2007.
- [29] Suraj Jog, Jiaming Wang, Haitham Hassanieh, and Romit Roy Choudhury. Enabling Dense Spatial Reuse in mmWave Networks. In *ACM SIGCOMM*, 2018.
- [30] Sanjeev Khanna and Krishnan Kumaran. On wireless spectrum estimation and generalized graph coloring. In *Proceedings of INFOCOM*, volume 3, pages 1273–1283. IEEE, 1998.
- [31] Sungoh Kwon and Joerg Widmer. Relay selection for mmwave communications. In *Personal, Indoor, and Mobile Radio Communications (PIMRC), 2017 IEEE 28th Annual International Symposium on*, pages 1–6. IEEE, 2017.
- [32] Sungoh Kwon and Joerg Widmer. Multi-beam power allocation for mmwave communications under random blockage. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–5. IEEE, 2018.
- [33] Xi Liu, Anmol Sheth, Michael Kaminsky, Konstantina Papagiannaki, Srinivasan Seshan, and Peter Steenkiste. DIRC: Increasing indoor wireless capacity using directional antennas. *ACM SIGCOMM Computer Communication Review*, 39(4):171–182, 2009.
- [34] Barry Manz. 5G Cellular Networks Are the Future of Robotics, Mouser Electronics, 2016.
- [35] Markets and Markets. Millimeter Wave Technology Market worth 4,632.8 Million USD by 2022, Press Release, 2017.
- [36] Arunesh Mishra, Suman Banerjee, and William Arbaugh. Weighted coloring based channel assignment for WLANs. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(3):19–31, 2005.
- [37] Thomas Moon, Junfeng Guan, and Haitham Hassanieh. Online millimeter wave phased array calibration based on channel state information. In *IEEE VLSI Test Symposium, 2019. VTS’19*. IEEE, 2019.
- [38] Raghuraman Mudumbai, SK Singh, and Upamanyu Madhow. Medium access control for 60 GHz outdoor mesh networks with highly directional links. In *INFOCOM 2009, IEEE*, pages 2871–2875. IEEE, 2009.
- [39] NetGear. NIGHTHAWK X10 R9000 Wi-Fi Router.
- [40] Thomas Nitsche, Carlos Cordeiro, Adriana B Flores, Edward W Knightly, Eldad Perahia, and Joerg C Widmer. IEEE 802.11 ad: directional 60 GHz communication for multi-gigabit-per-second Wi-Fi. *IEEE Communications Magazine*, 52(12):132–141, 2014.

- [41] Thomas Nitsche, Adriana B Flores, Edward W Knightly, and Joerg Widmer. Steering with eyes closed: mm-wave beam steering without in-band measurement. In *Conference on Computer Communications (INFOCOM)*, pages 2416–2424. IEEE, 2015.
- [42] Joan Palacios, Guillermo Bielsa, Paolo Casari, and Joerg Widmer. Communication-driven localization and mapping for millimeter wave networks. *IEEE INFOCOM*, 2018.
- [43] Pasternack Enterprises Inc. 60 GHz Transmitter/Receiver Development System. www.pasternack.com.
- [44] Jian Qiao, Lin X Cai, Xuemin Shen, and Jon W Mark. STDMA-based scheduling algorithm for concurrent transmissions in directional millimeter wave networks. In *Communications (ICC), 2012 IEEE International Conference on*, pages 5221–5225. IEEE, 2012.
- [45] Qualcomm. QCA9500 specifications. <https://www.qualcomm.com/products/qca9500>.
- [46] Hariharan Shankar Rahul, Swarun Kumar, and Dina Katabi. Jmb: scaling wireless capacity with user demands. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 235–246. ACM, 2012.
- [47] Krishna N Ramachandran, Elizabeth M Belding-Royer, Kevin C Almeroth, and Milind M Buddhikot. Interference-Aware Channel Assignment in Multi-Radio Wireless Mesh Networks. In *Infocom*, volume 6, pages 1–12, 2006.
- [48] Sundeep Rangan, Theodore S Rappaport, and Elza Erkip. Millimeter-wave cellular wireless networks: Potentials and challenges. *IEEE*, 2014.
- [49] Maryam Eslami Rasekh, Zhinus Marzi, Yanzi Zhu, Upamanyu Madhow, and Haitao Zheng. Noncoherent mmwave path tracking. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, pages 13–18. ACM, 2017.
- [50] Wonil Roh, Ji-Yun Seol, Jeongho Park, Byunghwan Lee, Jaekon Lee, Yungsoo Kim, Jaeweon Cho, Kyungwhoon Cheun, and Farshid Aryanfar. Millimeter-wave beamforming as an enabling technology for 5G cellular communications: Theoretical feasibility and prototype results. *IEEE Communications Magazine*, 52(2):106–113, 2014.
- [51] Vivek Shrivastava, Nabeel Ahmed, Shravan Rayanchu, Suman Banerjee, Srinivasan Keshav, Konstantina Papagiannaki, and Arunesh Mishra. Centaur: realizing the full potential of centralized wlans through a hybrid data path. In *Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 297–308. ACM, 2009.
- [52] Sumit Singh, Federico Ziliotto, Upamanyu Madhow, E Belding, and Mark Rodwell. Blockage and directivity in 60 GHz wireless personal area networks: From cross-layer model to multihop MAC design. *IEEE Journal on Selected Areas in Communications*, 27(8):1400–1413, 2009.
- [53] Sanjib Sur, Vignesh Venkateswaran, Xinyu Zhang, and Parameswaran Ramanathan. 60 GHz Indoor Networking through Flexible Beams: A Link-Level Profiling. In *SIGMETRICS*, 2015.
- [54] Sanjib Sur, Xinyu Zhang, Parameswaran Ramanathan, and Ranveer Chandra. BeamSpy: Enabling Robust 60 GHz Links Under Blockage. In *NSDI*, 2016.
- [55] Tensorcom. Product specifications. <http://tensorcom.com/products-1/>.
- [56] TP-Link. Talon AD7200 Multi-Band Wi-Fi Router.
- [57] Victoria Turk. These Supermarket Warehouse Robots Have Their Own Mobile Network, Vice Motherboard, 2016.
- [58] Kiran Venugopal, Matthew C Valenti, and Robert W Heath. Interference in finite-sized highly dense millimeter wave networks. In *Information Theory and Applications Workshop (ITA), 2015*, pages 175–180. IEEE, 2015.
- [59] Weizhao Wang, Yu Wang, Xiang-Yang Li, Wen-Zhan Song, and Ophir Frieder. Efficient interference-aware TDMA link scheduling for static wireless networks. In *Proceedings of the 12th annual international conference on Mobile computing and networking*, pages 262–273. ACM, 2006.
- [60] Teng Wei and Xinyu Zhang. Pose Information Assisted 60 GHz Networks: Towards Seamless Coverage and Mobility Support. In *MobiCom’17*, 2017.
- [61] Teng Wei, Anfu Zhou, and Xinyu Zhang. Facilitating Robust 60 GHz Network Deployment By Sensing Ambient Reflectors. In *NSDI*, 2017.
- [62] Ding Zhang, Mihir Garude, and Parth H Pathak. mmChoir: Exploiting joint transmissions for reliable 60 GHz mmwave WLANs. In *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 251–260. ACM, 2018.
- [63] Anfu Zhou, Leilei Wu, Shaqing Xu, Huadong Ma, Teng Wei, and Xinyu Zhang. Following the shadow: Agile 3-d beam-steering for 60 GHz wireless networks. *IEEE INFOCOM*, 2018.
- [64] Anfu Zhou, Xinyu Zhang, and Huadong Ma. Beam-forecast: Facilitating Mobile 60 GHz Networks via Model-driven Beam Steering. In *INFOCOM*, 2017.
- [65] Anfu Zhou, Xinyu Zhang, and Huadong Ma. Beam-forecast: Facilitating mobile 60 GHz networks via model-driven beam steering. In *INFOCOM 2017-IEEE Conference on Computer Communications*, IEEE, pages 1–9. IEEE, 2017.
- [66] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *ACM SIGCOMM*, 2012.

Appendix

A Proof of Lemma 6.1

Suppose we are given a graph $G(V, E)$ where $|V| = N$ and $d(u)$ denotes the degree of u . Consider the following process which iteratively assigns weights (in the range $\{0 \dots M\}$) to the vertices. The initial assignment is F_0 such that $F_0(v) = M$ for all $v \in V$. We compute F_t as follows:

- Compute a Weighted Max Independent Set W_{t+1} in the weighted graph induced by G and F_t .
- If $u \in W_{t+1}$, then $F_{t+1}(u) = F_t(u) - (d(u) + 1)$ if $F_t(u) > 2(d(u) + 1)$ and $F_{t+1}(u) = 0$ otherwise.
- If $u \notin W_{t+1}$, then $F_{t+1}(u) = F_t(u)$.

Lemma A.1 *If $t = O(M \log(NM))$, then $F_t(u) = 0 \forall u \in V$*

Proof Consider the potential function $T_t = \sum_u F_t(u)$.

Claim A.2 $T_{t+1} \leq T_t(1 - 1/M)$.

Proof Consider the set of vertices S_t containing u 's such that $F_t(u) > 0$. Since the maximum value of $F_t(u)$ is M , it follows that

$$|S_t| \geq T_t/M \quad (2)$$

Consider now the set W_{t+1} , and w.l.o.g. assume that $W_{t+1} \subset S_t$. Observe that W_{t+1} must be a maximal independent set, i.e., we cannot add any $u \in S_t - W_{t+1}$ to W_{t+1} without violating the independence property. Since the total number of nodes

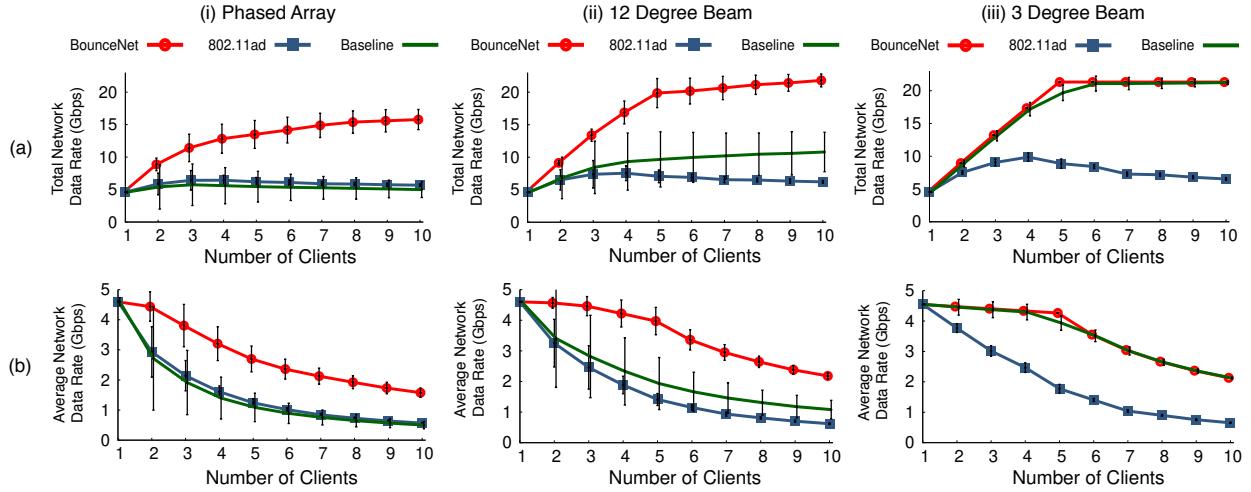


Figure 19: Data rates in BounceNet, 802.11ad and baseline for the case of 5 APs in network (a) Total Network Data Rates (b) Average Client Data Rates.

with an edge to a node in W_{t+1} (including self-loops) is at most $\sum_{w \in W_{t+1}} d(w) + 1$, it follows that

$$\sum_{w \in W_{t+1}} d(w) + 1 \geq |S_t| \quad (3)$$

However, the left-hand side in the above expression is upper bounded by the amount by which we reduce the potential, i.e., by the difference $T_t - T_{t+1}$ (the reduction in potential could be higher, because we round all weights smaller than $d + 1$ to 0). From Equations 2 and 3 we have

$$T_t - T_{t+1} \geq \sum_{w \in W_{t+1}} d(w) + 1 \geq |S_t| \geq T_t/M$$

and the lemma follows.

Since T_t has integral values, it follows that after $O(M \log(T_0))$ steps we have $T_t = 0$, and therefore $F_t(u) = 0$ for all u .

B Data Rate Gains for 5 APs

In Fig. 19, we present results for the case when there are 5 APs in the network. This allows us to evaluate BounceNet’s performance in scenarios where the number of clients is greater than the number of APs. In such scenarios where the clients outnumber the APs, two or more clients could be assigned to the same AP, following the algorithm presented in Section 6.1. Since clients that share an AP can essentially be considered as interfering links, the corresponding nodes in the conflict graph will have edges between them. We can then apply BounceNet’s signal routing algorithm (Section 6.2 and 6.3) to this modified conflict graph.

Fig. 19(a) shows the total network data rate, and Fig. 19(b) shows the average network data rate per client, as a function

of the number of clients in the network. BounceNet is able to deliver a total of 21.33 Gbps, 20.81 Gbps and 15.78 Gbps data rates for 10 clients in the 3° beam, 12° beam and the phased array testbeds respectively. The baseline performs almost as well as BounceNet for the 3° beam since the interference in this case is very limited, and as a result, the baseline is able to exploit spatial reuse. However, as the amount of interference increases, the performance of the baseline deteriorates, with BounceNet achieving 2.2× and 3.2× gain in network throughput over the baseline for the case of 10 clients in the 12° beam, and the phased array testbeds respectively. Since the baseline does not account for interfering links, it leads to frequent packet collisions, and as a result, inefficient use of the channel.

Compared to 802.11ad, BounceNet achieves 3.26×, 3.35×, and 2.78× gain in network throughput for the case of 10 clients in the 3° beam, 12° beam, and the phased array testbed respectively. One should note that for 802.11ad, the gains with 5 APs are smaller as compared to the gains observed in Section 9.C, where there were 10 APs in the network. This is because BounceNet’s strength over 802.11ad comes primarily from its ability to exploit spatial reuse efficiently, and with only 5 APs in the network, the potential for spatial reuse is reduced, and therefore the gains that BounceNet can provide over the standard will be smaller. Hence, to achieve significant gains in throughput, BounceNet advocates for dense AP deployments with narrow directional antenna beams in mmWave networks.

Finally, the following points are worth noting.

- With the 3° beam in the 60 GHz testbed, we see that the total network data rate for BounceNet saturates after 5 clients as can be seen in Fig. 19(a)(iii), achieving 21.33 Gbps and 21.29 Gbps for 10 clients and 5 clients respectively. This

is expected, since at any given time at most 5 clients can be communicating simultaneously in the network. Such saturation can also be observed in the other two testbeds.

- It may seem counter-intuitive that the total network data rate for BounceNet in the 12° and the phased array testbeds continues to grow even when there are more than 5 clients in the network. This happens because as the number of clients increases in the network, the total number of propagation paths (direct and reflected) between APs and clients

increases as well. Therefore, now it is more likely that BounceNet can find a set of five propagation paths that can coexist in the network, and consequently, BounceNet can schedule more clients in every time slot. However, one should note that the rate of growth of the network data rate reduces as the number of clients increases beyond five, and correspondingly, the average per-client data rates start to drop more sharply beyond five clients as can be seen in Fig. 19(b).