



The Advanced Computing Systems Association

In cooperation with  
ACM SIGCOMM and ACM SIGOPS

conference  
proceedings

# 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)

Renton, WA, USA

April 4–6, 2022



Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)

Renton, WA, USA April 4–6, 2022

ISBN 978-1-939133-27-4

# NSDI '22 Sponsors

## Platinum Sponsor



## Diamond Sponsor



## Gold Sponsor



## Silver Sponsors



## Bronze Sponsors



## Open Access Sponsor



# USENIX Supporters

## USENIX Patrons

Amazon • Ethyca • Google • Meta  
Microsoft • NetApp • Salesforce

## USENIX Benefactors

AuriStor • Bloomberg • Discernible • Goldman Sachs • IBM  
Shopify • Thinkst Canary • Transcend • Two Sigma

## USENIX Partner

Blameless • Lightstep • Top10VPN

## Open Access Supporter

Google

## Open Access Publishing Partner

PeerJ



**USENIX Association**

**Proceedings of the  
19th USENIX Symposium on  
Networked Systems Design and Implementation**

**April 4–6, 2022  
Renton, WA, USA**

© 2022 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 978-1-939133-27-4

## Conference Organizers

### Program Co-Chairs

Amar Phanishayee, *Microsoft Research*  
Vyas Sekar, *Carnegie Mellon University*

### Program Committee

Sangeetha Abdu-Jyothi, *University of California, Irvine, and VMware Research*  
Fadel Adib, *Massachusetts Institute of Technology*  
Behnaz Arzani, *Microsoft Research*  
Anirudh Badam, *Microsoft Research*  
Mahesh Balakrishnan, *Facebook*  
Aruna Balasubramanian, *Stony Brook University*  
Hitesh Ballani, *Microsoft Research*  
Sujata Banerjee, *VMware Research*  
Theo Benson, *Brown University*  
Matthew Caesar, *University of Illinois at Urbana–Champaign*  
Vijay Chidambaram, *The University of Texas at Austin*  
Asaf Cidon, *Columbia University*  
Angela Demke Brown, *University of Toronto*  
Fahad Dogar, *Tufts University*  
Giulia Fanti, *Carnegie Mellon University*  
Rodrigo Fonseca, *Microsoft Research*  
Manya Ghobadi, *Massachusetts Institute of Technology*  
Soudeh Ghorbani, *Johns Hopkins University*  
Phillipa Gill, *Google*  
Brighten Godfrey, *University of Illinois at Urbana–Champaign and VMware*  
Shyam Gollakota, *University of Washington*  
Ramesh Govindan, *University of Southern California*  
Chuanxiong Guo, *ByteDance*  
Andreas Haeberlen, *University of Pennsylvania*  
Kurtis Heimerl, *University of Washington*  
Wenjun Hu, *Yale University*  
Kyle Jamieson, *Princeton University*  
Junchen Jiang, *University of Chicago*  
Anuj Kalia, *Microsoft Research*  
Anurag Khandelwal, *Yale University*  
Ana Klimovic, *ETH Zurich*  
Dejan Kostic, *KTH Royal Institute of Technology*  
Franck Le, *IBM Research*  
Kate Lin, *National Chiao Tung University*  
Zaoxing Alan Liu, *Boston University*  
Jay Lorch, *Microsoft Research*  
Jonathan Mace, *Max Planck Institute for Software Systems (MPI-SWS)*

Harsha Madhyastha, *University of Michigan*  
Aurojit Panda, *New York University*  
Kyoongsoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*  
Chunyi Peng, *Purdue University*  
Ben Pfaff, *VMware Research*  
George Porter, *University of California, San Diego*  
Costin Raiciu, *University Politehnica of Bucharest and Correct Networks*  
Robert Ricci, *University of Utah*  
Michael Schapira, *The Hebrew University of Jerusalem*  
Stefan Schmid, *Technische Universität Berlin and University of Vienna*  
Brent Stephens, *University of Utah*  
Laurent Vanbever, *ETH Zurich*  
Shivaram Venkataraman, *University of Wisconsin–Madison*  
David Walker, *Princeton University*  
Jia Wang, *AT&T Labs*  
Michael Wei, *VMware Research*  
John Wilkes, *Google*  
Xiaowei Yang, *Duke University*  
Minlan Yu, *Harvard University*  
Ellen Zegura, *Georgia Institute of Technology*  
Ying Zhang, *Facebook*  
Yiying Zhang, *University of California, San Diego*  
Ben Zhao, *University of Chicago*  
Wenting Zheng, *Carnegie Mellon University*  
Lin Zhong, *Yale University*  
Danyang Zhuo, *Duke University*

### Steering Committee

Aditya Akella, *University of Wisconsin–Madison*  
Sujata Banerjee, *VMware Research*  
Ranjita Bhagwan, *Microsoft Research India*  
Casey Henderson, *USENIX Association*  
Jon Howell, *VMware Research*  
Arvind Krishnamurthy, *University of Washington*  
Jay Lorch, *Microsoft Research*  
James Mickens, *Harvard University*  
Jeff Mogul, *Google*  
George Porter, *University of California, San Diego*  
Timothy Roscoe, *ETH Zurich*  
Srinivasan Seshan, *Carnegie Mellon University*  
Renata Teixeira, *Netflix*  
Minlan Yu, *Harvard University*

## External Reviewers

Anubhavnidhi “Archie”  
Abhashkumar  
Rachit Agarwal  
Fawad Ahmad  
Lixiang Ao  
Rodrigo Bruno  
Ang Chen  
Italo Cunha  
Weiqi Feng  
Bryan Ford

Jiaqi Gao  
Rajrup Ghosh  
Junzhi Gong  
Arpit Gupta  
Indranil Gupta  
Dongsu Han  
Yitao Hu  
Ryan Huang  
Keon Jang  
Weifan Jiang

Swarun Kumar  
Chonlam Lao  
Minghao Li  
Devon Loehr  
Pooria Namyar  
Dave Oran  
Dan Pei  
Sivaram Ramanathan  
Christopher Rossbach  
Siddhartha Sen

Francis Yan  
Michelle X. Yeo  
Irene Zhang  
Mingyang Zhang  
Yang Zhou  
Noa Zilberman

Sriniv Seshan  
Srinath Setty  
Rob Sherwood  
Alex Snoeren  
Jiri Srba  
Srikanth Sundaresan  
Francois Taiani  
Matteo Varvello  
Yongqiang Xiong  
Zhiying Xu

## Message from the NSDI '22 Program Co-Chairs

Welcome to NSDI 2022!

We live in unprecedented times. We have been through waves of multiple covid variants, parents dealing with the uncertainty of school schedules, rapid scientific breakthroughs that resulted in effective vaccines, mass vaccination drives, and war in parts of the world that threatens dislodging the lives of millions of people. It is difficult to reason about the importance of our work in such turbulent times. But despite common as well as uniquely individual challenges, our community marches on, building on lessons we have learnt on operating during the pandemic.

NSDI '22 received a record number of submissions—396 papers in total: 104 in spring and 292 for the fall deadline. A total of 78 papers were accepted for an acceptance rate of 19.7%—the highest we have seen in a while. Papers were reviewed by a program committee of 65 experts from both academia and industry. One-shot revisions, first introduced to NSDI in 2019, have proved to be quite successful and we continue this practice.

We sincerely thank our reviewers who provided thoughtful feedback to our authors, including those who re-reviewed one-shot revisions from the prior NSDI edition (NSDI '21 Fall) and our expert external reviewers. We also want to thank our stand-in PC chairs (for chair-conflict papers) who helped out selflessly performing tasks that they had not necessarily signed up for when they agreed to be on our PC: Ellen Zegura, Mahesh Balakishnan, and Ben Y. Zhao. We thank George Porter and Harsha V. Madhyastha for going above and beyond the call of duty—and at short notice—to carefully read and help us select the best paper award winners this year. We are also very grateful to prior NSDI chairs Arvind Krishnamurthy, Jay Lorch, James Mickens, and Renata Teixeira for sharing their best practices with us. And we thank student volunteers Brian Singer and Milind Srivatsava for helping us run a smooth Fall PC meeting over Zoom. Finally, we also thank the paper authors; your submissions are what make NSDI such a great venue, and we hope that you will enjoy the conference program.

We are trying two new experiments this year. First, NSDI will be held as a hybrid event. We are excited that the program will be held in-person, but we recognize that there are many authors and attendees who will only be able to attend virtually as the threat of a new Covid variant looms large in many parts of the world. Second, for both safety (to avoid packing all attendees in a single room), as well as providing the large number of accepted papers with ample time to present their ideas, we are experimenting with a dual-track format. While changes to well-established ways of doing things make us a little anxious, we could not be more excited that USENIX is the organization shepherding us through these changes.

Which brings us to thanking one of the most important groups that has helped us: USENIX. We'd like to thank all of the USENIX staff who helped us to organize this year's conference right from the get go: from configuring the HotCRP server to dealing with camera-ready production of both papers and talks (and they had to do this twice for the spring and fall deadline), the USENIX staff provided invaluable advice and flawless execution. We are certain we will miss many names we ought to thank, and in some cases because we magically saw the result of your work but never got to know that you did it. Our heartfelt thanks to Casey Henderson (for patiently helping us on so many different dimensions), Olivia Vernetti, Camille Mulligan, Arnold Gatilao, Jasmine Murcia, Jessica Kim, Julia Hendrickson, Liz Markel, Sarah TerHune, and the rest of the USENIX team. You are a magical team, and we as a community are lucky to have such dedicated, caring, and supremely competent USENIX staff. We would be lost in the wilderness without you.

Stay safe and healthy, and enjoy NSDI in whichever format you choose to attend it!

Amar Phanishayee, *Microsoft Research*  
Vyas Sekar, *Carnegie Mellon University*  
NSDI '22 Program Co-Chairs

# 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)

April 4–6, 2022

Renton, WA, USA

## Monday, April 4

### Cluster Resource Management

<b>Efficient Scheduling Policies for Microsecond-Scale Tasks</b> .....	<b>1</b>
Sarah McClure and Amy Ousterhout, <i>UC Berkeley</i> ; Scott Shenker, <i>UC Berkeley, ICSI</i> ; Sylvia Ratnasamy, <i>UC Berkeley</i>	
<b>A Case for Task Sampling based Learning for Cluster Job Scheduling</b> .....	<b>19</b>
Akshay Jajoo, <i>Nokia Bell Labs</i> ; Y. Charlie Hu and Xiaojun Lin, <i>Purdue University</i> ; Nan Deng, <i>Google</i>	
<b>Starlight: Fast Container Provisioning on the Edge and over the WAN</b> .....	<b>35</b>
Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara, <i>University of Toronto</i>	

### Transport Layer - Part 1

<b>POWERTCP: Pushing the Performance Limits of Datacenter Networks</b> .....	<b>51</b>
Vamsi Addanki, <i>TU Berlin and University of Vienna</i> ; Oliver Michel, <i>Princeton University and University of Vienna</i> ; Stefan Schmid, <i>TU Berlin and University of Vienna</i>	
<b>RDMA is Turing complete, we just did not know it yet!</b> .....	<b>71</b>
Waleed Reda, <i>Université catholique de Louvain and KTH Royal Institute of Technology</i> ; Marco Canini, <i>KAUST</i> ; Dejan Kostić, <i>KTH Royal Institute of Technology</i> ; Simon Peter, <i>University of Washington</i>	
<b>FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism</b> .....	<b>87</b>
Rajath Shashidhara, <i>University of Washington</i> ; Tim Stamler, <i>UT Austin</i> ; Antoine Kaufmann, <i>MPI-SWS</i> ; Simon Peter, <i>University of Washington</i>	

### Video Streaming

<b>Swift: Adaptive Video Streaming with Layered Neural Codecs</b> .....	<b>103</b>
Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras, <i>Stony Brook University</i>	
<b>Ekyia: Continuous Learning of Video Analytics Models on Edge Compute Servers</b> .....	<b>119</b>
Romil Bhardwaj, <i>Microsoft and UC Berkeley</i> ; Zhengxu Xia, <i>University of Chicago</i> ; Ganesh Ananthanarayanan, <i>Microsoft</i> ; Junchen Jiang, <i>University of Chicago</i> ; Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, and Paramvir Bahl, <i>Microsoft</i> ; Ion Stoica, <i>UC Berkeley</i>	
<b>YuZu: Neural-Enhanced Volumetric Video Streaming</b> .....	<b>137</b>
Anlan Zhang and Chendong Wang, <i>University of Minnesota, Twin Cities</i> ; Bo Han, <i>George Mason University</i> ; Feng Qian, <i>University of Minnesota, Twin Cities</i>	

### Programmable Switches - Part 1

<b>NetVRM: Virtual Register Memory for Programmable Networks</b> .....	<b>155</b>
Hang Zhu, <i>Johns Hopkins University</i> ; Tao Wang, <i>New York University</i> ; Yi Hong, <i>Johns Hopkins University</i> ; Dan R. K. Ports, <i>Microsoft Research</i> ; Anirudh Sivaraman, <i>New York University</i> ; Xin Jin, <i>Peking University</i>	
<b>SwiSh: Distributed Shared State Abstractions for Programmable Switches</b> .....	<b>171</b>
Lior Zeno, <i>Technion</i> ; Dan R. K. Ports, Jacob Nelson, and Daehyeok Kim, <i>Microsoft Research</i> ; Shir Landau Feibish, <i>The Open University of Israel</i> ; Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein, <i>Technion</i>	
<b>Modular Switch Programming Under Resource Constraints</b> .....	<b>193</b>
Mary Hogan, <i>Princeton University</i> ; Shir Landau-Feibish, <i>The Open University of Israel</i> ; Mina Tahmasbi Arashloo, <i>Cornell University</i> ; Jennifer Rexford and David Walker, <i>Princeton University</i>	

## Security and Privacy

- Privid: Practical, Privacy-Preserving Video Analytics Queries** ..... 209  
Frank Cangialosi, *MIT CSAIL*; Neil Agarwal, *Princeton University*; Venkat Arun, *MIT CSAIL*; Junchen Jiang, *University of Chicago*; Srinivas Narayana and Anand Sarwate, *Rutgers University*; Ravi Netravali, *Princeton University*

- Spectrum: High-Bandwidth Anonymous Broadcast** ..... 229  
Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas, *MIT CSAIL*

- Donar: Anonymous VoIP over Tor** ..... 249  
Yérom-David Bromberg, Quentin Dufour, and Davide Frey, *Univ. Rennes - Inria - CNRS - IRISA*; Etienne Rivière, *UCLouvain*

## Network Troubleshooting and Debugging

- Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon** ..... 267  
Weitao Wang and Xinyu Crystal Wu, *Rice University*; Praveen Tammana, *Indian Institute of Technology Hyderabad*; Ang Chen and T. S. Eugene Ng, *Rice University*

- Collie: Finding Performance Anomalies in RDMA Subsystems** ..... 287  
Xinhao Kong, *Duke University and ByteDance Inc.*; Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, and Chuanxiong Guo, *ByteDance Inc.*; Danyang Zhuo, *Duke University*

- SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers** ..... 307  
Siva Kesava Reddy Kakarla, *University of California, Los Angeles*; Ryan Beckett, *Microsoft*; Todd Millstein, *University of California, Los Angeles, and Intentionet*; George Varghese, *University of California, Los Angeles*

## Operational Track - Part 1

- Decentralized cloud wide-area network traffic engineering with BLASTSHIELD** ..... 325  
Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj, *Microsoft*

- Detecting Ephemeral Optical Events with OpTel** ..... 339  
Congcong Miao and Minggang Chen, *Tencent*; Arpit Gupta, *UC Santa Barbara*; Zili Meng, Lianjin Ye, and Jingyu Xiao, *Tsinghua University*; Jie Chen, Zekun He, and Xulong Luo, *Tencent*; Jilong Wang, *Tsinghua University, BNRist, and Peng Cheng Laboratory*; Heng Yu, *Tsinghua University*

- Bluebird: High-performance SDN for Bare-metal Cloud Services** ..... 355  
Manikandan Arumugam, *Arista*; Deepak Bansal, *Microsoft*; Navdeep Bhatia, *Arista*; James Boerner, *Microsoft*; Simon Capper, *Arista*; Changhoon Kim, *Intel*; Sarah McClure, Neeraj Motwani, and Ranga Narasimhan, *Microsoft*; Urvish Panchal, *Arista*; Tommaso Pimpo, *Microsoft*; Ariff Premji, *Arista*; Pranjal Shrivastava and Rishabh Tewari, *Microsoft*

- Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling** ..... 371  
Yifan Li, *Tsinghua University and Alibaba Group*; Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu, *Alibaba Group*

## Wireless - Part 1

- Exploiting Digital Micro-Mirror Devices for Ambient Light Communication** ..... 387  
Talia Xu, Miguel Chávez Tapia, and Marco Zúñiga, *Technical University Delft*

- Whisper: IoT in the TV White Space Spectrum** ..... 401  
Tusher Chakraborty and Heping Shi, *Microsoft*; Zerina Kapetanovic, *University of Washington*; Bodhi Priyantha, *Microsoft*; Deepak Vasisht, *UIUC*; Binh Vu, Parag Pandit, Prasad Pillai, Yaswant Chabria, Andrew Nelson, Michael Daum, and Ranveer Chandra, *Microsoft*

- Learning to Communicate Effectively Between Battery-free Devices** ..... 419  
Kai Geissdoerfer and Marco Zimmerling, *TU Dresden*

- Saiyan: Design and Implementation of a Low-power Demodulator for LoRa Backscatter Systems** ..... 437  
Xiuzhen Guo, *Tsinghua University*; Longfei Shangguan, *University of Pittsburgh & Microsoft*; Yuan He, *Tsinghua University*; Nan Jing, *Yanshan University*; Jiacheng Zhang, Haotian Jiang, and Yunhao Liu, *Tsinghua University*

## Tuesday, April 5

### Reliable Distributed Systems

Graham: Synchronizing Clocks by Leveraging Local Clock Properties .....	453
Ali Najafi, <i>Meta</i> ; Michael Wei, <i>VMware Research</i>	

IA-CCF: Individual Accountability for Permissioned Ledgers .....	467
Alex Shamis and Peter Pietzuch, <i>Microsoft Research and Imperial College London</i> ; Burcu Canakci, <i>Cornell University</i> ; Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, and Antoine Delignat-Lavaud, <i>Microsoft Research</i> ; Matthew Kerner, <i>Microsoft Azure</i> ; Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, and Manuel Costa, <i>Microsoft Research</i> ; Mark Russinovich, <i>Microsoft Azure</i>	

DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks .....	493
Lei Yang, Seo Jin Park, and Mohammad Alizadeh, <i>MIT CSAIL</i> ; Sreeram Kannan, <i>University of Washington</i> ; David Tse, <i>Stanford University</i>	

### Raising the Bar for Programmable Hardware

Re-architecting Traffic Analysis with Neural Network Interface Cards .....	513
Giuseppe Siracusano, <i>NEC Laboratories Europe</i> ; Salvator Galea, <i>University of Cambridge</i> ; Davide Sanvito, <i>NEC Laboratories Europe</i> ; Mohammad Malekzadeh, <i>Imperial College London</i> ; Gianni Antichi, <i>Queen Mary University of London</i> ; Paolo Costa, <i>Microsoft Research</i> ; Hamed Haddadi, <i>Imperial College London</i> ; Roberto Bifulco, <i>NEC Laboratories Europe</i>	

Elixir: A High-performance and Low-cost Approach to Managing Hardware/Software Hybrid Flow Tables Considering Flow Burstiness .....	535
Yanshu Wang and Dan Li, <i>Tsinghua University</i> ; Yuanwei Lu, <i>Tencent</i> ; Jianping Wu, Hua Shao, and Yutian Wang, <i>Tsinghua University</i>	

Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing .....	551
Peixuan Gao and Anthony Dalleghio, <i>New York University</i> ; Yang Xu, <i>Fudan University</i> ; H. Jonathan Chao, <i>New York University</i>	

### Testing and Verification

Performance Interfaces for Network Functions .....	567
Rishabh Iyer, Katerina Argyraki, and George Canea, <i>EPFL</i>	

Automated Verification of Network Function Binaries .....	585
Solal Pirelli, <i>EPFL</i> ; Akvilė Valentukonytė, <i>Citrix Systems</i> ; Katerina Argyraki and George Canea, <i>EPFL</i>	

Differential Network Analysis .....	601
Peng Zhang, <i>Xi'an Jiaotong University</i> ; Aaron Gember-Jacobson, <i>Colgate University</i> ; Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li, <i>Xi'an Jiaotong University</i>	

KATRA: Realtime Verification for Multilayer Networks .....	617
Ryan Beckett, <i>Microsoft</i> ; Aarti Gupta, <i>Princeton University</i>	

### Programmable Switches - Part 2

Enabling In-situ Programmability in Network Data Plane: From Architecture to Language.....	635
Yong Feng and Zhikang Chen, <i>Tsinghua University</i> ; Haoyu Song, <i>Futurewei Technologies</i> ; Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu, <i>Tsinghua University</i>	

Runtime Programmable Switches.....	651
Jiarong Xing and Kuo-Feng Hsu, <i>Rice University</i> ; Matty Kadosh, Alan Lo, and Yonatan Piasetzky, <i>Nvidia</i> ; Arvind Krishnamurthy, <i>University of Washington</i> ; Ang Chen, <i>Rice University</i>	

IMap: Fast and Scalable In-Network Scanning with Programmable Switches .....	667
Guanyu Li, <i>Tsinghua University</i> ; Menghao Zhang, <i>Tsinghua University</i> ; Kuaishou Technology; Cheng Guo, Han Bao, and Mingwei Xu, <i>Tsinghua University</i> ; Hongxin Hu, <i>University at Buffalo, SUNY</i> ; Fenghua Li, <i>Tsinghua University</i>	

Unlocking the Power of Inline Floating-Point Operations on Programmable Switches .....	683
Yifan Yuan, <i>UIUC</i> ; Omar Alama, <i>KAUST</i> ; Jiawei Fei, <i>KAUST &amp; NUDT</i> ; Jacob Nelson and Dan R. K. Ports, <i>Microsoft Research</i> ; Amedeo Sazio, <i>Intel</i> ; Marco Canini, <i>KAUST</i> ; Nam Sung Kim, <i>UIUC</i>	

## Sketch-based Telemetry

**Dynamic Scheduling of Approximate Telemetry Queries** ..... 701  
Chris Misa, Walt O'Connor, Ramakrishnan Durairajan, and Reza Rejaie, *University of Oregon*; Walter Willinger, *NIKSUN, Inc.*

**HeteroSketch: Coordinating Network-wide Monitoring in Heterogeneous and Dynamic Networks** ..... 719  
Anup Agarwal, *Carnegie Mellon University*; Zaoxing Liu, *Boston University*; Srinivasan Seshan, *Carnegie Mellon University*

**SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches** ..... 743  
Hun Namkung, *Carnegie Mellon University*; Zaoxing Liu, *Boston University*; Daehyeok Kim, *Carnegie Mellon University and Microsoft*; Vyas Sekar and Peter Steenkiste, *Carnegie Mellon University*

## Transport Layer - Part 2

**An edge-queued datagram service for all datacenter traffic** ..... 761  
Vladimir Olteanu, *Correct Networks and University Politehnica of Bucharest*; Haggai Eran, *Technion and NVIDIA*; Dragos Dumitrescu, *Correct Networks and University Politehnica of Bucharest*; Adrian Popa and Cristi Baciu, *Correct Networks*; Mark Silberstein, *Technion*; Georgios Nikolaidis, *Intel*; Mark Handley, *UCL and Correct Networks*; Costin Raiciu, *Correct Networks and University Politehnica of Bucharest*

**Backpressure Flow Control** ..... 779  
Prateesh Goyal, *MIT CSAIL*; Preey Shah, *IIT Bombay*; Kevin Zhao, *University of Washington*; Georgios Nikolaidis, *Intel Barefoot Switch Division*; Mohammad Alizadeh, *MIT CSAIL*; Thomas E. Anderson, *University of Washington*

**Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets** ..... 807  
Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, and Alireza Farshin, *KTH Royal Institute of Technology*; Amir Roozbeh, *KTH Royal Institute of Technology and Ericsson Research*; Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić, *KTH Royal Institute of Technology*

## Troubleshooting

**Buffer-based End-to-end Request Event Monitoring in the Cloud** ..... 829  
Kaihui Gao, *Tsinghua University and Alibaba Group*; Chen Sun, *Alibaba Group*; Shuai Wang and Dan Li, *Tsinghua University*; Yu Zhou, Hongqiang Harry Liu, Lingjun Zhu, and Ming Zhang, *Alibaba Group*

**Characterizing Physical-Layer Transmission Errors in Cable Broadband Networks** ..... 845  
Jiyao Hu, Zhenyu Zhou, and Xiaowei Yang, *Duke University*

**How to diagnose nanosecond network latencies in rich end-host stacks** ..... 861  
Roni Haecki, *ETH Zurich*; Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, and Sujata Banerjee, *VMware*; Timothy Roscoe, *ETH Zurich*

## Wireless - Part 2

**CurvingLoRa to Boost LoRa Network Throughput via Concurrent Transmission** ..... 879  
Chenning Li, *Michigan State University*; Xiuzhen Guo, *Tsinghua University*; Longfei Shangguan, *University of Pittsburgh & Microsoft*; Zhichao Cao, *Michigan State University*; Kyle Jamieson, *Princeton University*

**PLatter: On the Feasibility of Building-scale Power Line Backscatter** ..... 897  
Junbo Zhang, *Carnegie Mellon University*; Elahe Soltanaghai, *University of Illinois at Urbana-Champaign*; Artur Balanuta, Reese Grimsley, Swaran Kumar, and Anthony Rowe, *Carnegie Mellon University*

**Passive DSSS: Empowering the Downlink Communication for Backscatter Systems** ..... 913  
Songfan Li, Hui Zheng, Chong Zhang, Yihang Song, Shen Yang, Minghua Chen, and Li Lu, *University of Electronic Science and Technology of China (UESTC)*; Mo Li, *Nanyang Technological University (NTU)*

## **Wednesday, April 6**

### **Operational Track - Part 2**

**Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models .....** 929  
Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, and Misha Smelyanskiy, *Facebook*; Murali Annavaram, *Facebook and USC*

**MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters .....** 945  
Qizhen Weng, *Hong Kong University of Science and Technology and Alibaba Group*; Wencong Xiao, *Alibaba Group*; Yinghao Yu, *Alibaba Group and Hong Kong University of Science and Technology*; Wei Wang, *Hong Kong University of Science and Technology*; Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding, *Alibaba Group*

**Evolvable Network Telemetry at Facebook .....** 961  
Yang Zhou, *Harvard University*; Ying Zhang, *Facebook*; Minlan Yu, *Harvard University*; Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong, *Facebook*

### **Edge IoT Applications**

**SwarmMap: Scaling Up Real-time Collaborative Visual SLAM at the Edge.....** 977  
Jingao Xu, Hao Cao, and Zheng Yang, *Tsinghua University*; Longfei Shangguan, *University of Pittsburgh & Microsoft*; Jialin Zhang, Xiaowu He, and Yunhao Liu, *Tsinghua University*

**In-Network Velocity Control of Industrial Robot Arms.....** 995  
Sándor Laki and Csaba Györgyi, *ELTE Eötvös Loránd University, Budapest, Hungary*; József Pető, *Budapest University of Technology and Economics, Budapest, Hungary*; Péter Vörös, *ELTE Eötvös Loránd University, Budapest, Hungary*; Géza Szabó, *Ericsson Research, Budapest, Hungary*

**Enabling IoT Self-Localization Using Ambient 5G Signals .....** 1011  
Suraj Jog, Junfeng Guan, and Sohrab Madani, *University of Illinois at Urbana Champaign*; Ruochen Lu, *University of Texas at Austin*; Songbin Gong, Deepak Vasishth, and Haitham Hassanieh, *University of Illinois at Urbana Champaign*

### **Cloud Scale Services**

**Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks.....** 1027  
Joshua Romero, *NVIDIA, Inc.*; Junqi Yin, Nouamane Laanait, Bing Xie, and M. Todd Young, *Oak Ridge National Laboratory*; Sean Treichler, *NVIDIA, Inc.*; Vitalii Starchenko and Albina Borisevich, *Oak Ridge National Laboratory*; Alex Sergeev, *Carbon Robotics*; Michael Matheson, *Oak Ridge National Laboratory*

**Cocktail: A Multidimensional Optimization for Model Serving in Cloud .....** 1041  
Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das, *The Pennsylvania State University*

**Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems.....** 1059  
Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia, *Stanford University*

**Orca: Server-assisted Multicast for Datacenter Networks.....** 1075  
Khaled Diab, Parham Yassini, and Mohamed Hefeeda, *Simon Fraser University*

### **ISPs and CDNs**

**Yeti: Stateless and Generalized Multicast Forwarding.....** 1093  
Khaled Diab and Mohamed Hefeeda, *Simon Fraser University*

**cISP: A Speed-of-Light Internet Service Provider .....** 1115  
Debopam Bhattacherjee, *ETH Zürich*; Waqar Aqeel, *Duke University*; Sangeetha Abdu Jyothi, *UC Irvine and VMware Research*; Ilker Nadi Bozkurt, *Duke University*; William Sentosa, *UIUC*; Muhammad Tirmazi, *Harvard University*; Anthony Aguirre, *UC Santa Cruz*; Balakrishnan Chandrasekaran, *VU Amsterdam*; P. Brighten Godfrey, *UIUC and VMware*; Gregory Laughlin, *Yale University*; Bruce Maggs, *Duke University and Emerald Technologies*; Ankit Singla, *ETH Zürich*

**Configanator: A Data-driven Approach to Improving CDN Performance.....** 1135  
Usama Naseer and Theophilus A. Benson, *Brown University*

<b>C2DN: How to Harness Erasure Codes at the Edge for Efficient Content Delivery .....</b>	<b>1159</b>
Juncheng Yang, <i>Carnegie Mellon University</i> ; Anirudh Sabnis, <i>University of Massachusetts, Amherst</i> ; Daniel S. Berger, <i>Microsoft Research and University of Washington</i> ; K. V. Rashmi, <i>Carnegie Mellon University</i> ; Ramesh K. Sitaraman, <i>University of Massachusetts, Amherst, and Akamai Technologies</i>	
<b>Cloud Scale Resource Management</b>	
<b>Optimizing Network Provisioning through Cooperation .....</b>	<b>1179</b>
Harsha Sharma, Parth Thakkar, Sagar Bharadwaj, Ranjita Bhagwan, Venkata N. Padmanabhan, Yogesh Bansal, Vijay Kumar, and Kathleen Voelbel, <i>Microsoft</i>	
<b>OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination .....</b>	<b>1195</b>
Liangcheng Yu, <i>University of Pennsylvania</i> ; John Sonchack, <i>Princeton University</i> ; Vincent Liu, <i>University of Pennsylvania</i>	
<b>CloudCluster: Unearthing the Functional Structure of a Cloud Service .....</b>	<b>1213</b>
Weiwei Pang, <i>University of Southern California</i> ; Sourav Panda, <i>University of California, Riverside</i> ; Jehangir Amjad and Christophe Diot, <i>Google Inc.</i> ; Ramesh Govindan, <i>University of Southern California</i>	
<b>Data Center Network Infrastructure</b>	
<b>Zeta: A Scalable and Robust East-West Communication Framework in Large-Scale Clouds .....</b>	<b>1231</b>
Qianyu Zhang, Gongming Zhao, and Hongli Xu, <i>University of Science and Technology of China</i> ; Zhuolong Yu, <i>Johns Hopkins University</i> ; Liguang Xie, <i>Futurewei Technologies</i> ; Yangming Zhao, <i>University of Science and Technology of China</i> ; Chunming Qiao, <i>SUNY at Buffalo</i> ; Ying Xiong, <i>Futurewei Technologies</i> ; Liusheng Huang, <i>University of Science and Technology of China</i>	
<b>Aquila: A unified, low-latency fabric for datacenter networks .....</b>	<b>1249</b>
Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat, <i>Google</i>	
<b>RDC: Energy-Efficient Data Center Network Congestion Relief with Topological Reconfigurability at the Edge .</b>	<b>1267</b>
Weitao Wang, <i>Rice University</i> ; Dingming Wu, <i>Bytedance Inc.</i> ; Sushovan Das, Afsaneh Rahbar, Ang Chen, and T. S. Eugene Ng, <i>Rice University</i>	
<b>Multitenancy</b>	
<b>Isolation Mechanisms for High-Speed Packet-Processing Pipelines.....</b>	<b>1289</b>
Tao Wang, <i>New York University</i> ; Xiangrui Yang, <i>National University of Defense Technology</i> ; Gianni Antichi, <i>Queen Mary University of London</i> ; Anirudh Sivaraman and Aurojit Panda, <i>New York University</i>	
<b>Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks .....</b>	<b>1307</b>
Yiwen Zhang, <i>University of Michigan</i> ; Yue Tan, <i>University of Michigan and Princeton University</i> ; Brent Stephens, <i>University of Illinois at Chicago</i> ; Mosharaf Chowdhury, <i>University of Michigan</i>	
<b>NetHint: White-Box Networking for Multi-Tenant Data Centers .....</b>	<b>1327</b>
Jingrong Chen, <i>Duke University</i> ; Hong Zhang, <i>University of California, Berkeley</i> ; Wei Zhang, <i>Duke University</i> ; Liang Luo, <i>University of Washington</i> ; Jeffrey Chase, <i>Duke University</i> ; Ion Stoica, <i>University of California, Berkeley</i> ; Danyang Zhuo, <i>Duke University</i>	
<b>Software Switching and Beyond</b>	
<b>Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing .....</b>	<b>1345</b>
Chaoliang Zeng, <i>Hong Kong University of Science and Technology</i> ; Layong Luo and Teng Zhang, <i>ByteDance</i> ; Zilong Wang, <i>Hong Kong University of Science and Technology</i> ; Luyang Li, <i>ICT/CAS</i> ; Wenchen Han, <i>Peking University</i> ; Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, and Feng Ning, <i>ByteDance</i> ; Kai Chen, <i>Hong Kong University of Science and Technology</i> ; Chuanxiong Guo, <i>ByteDance</i>	
<b>Scaling Open vSwitch with a Computational Cache .....</b>	<b>1359</b>
Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein, <i>Technion</i>	
<b>Backdraft: a Lossless Virtual Switch that Prevents the Slow Receiver Problem .....</b>	<b>1375</b>
Alireza Sanaee, <i>Queen Mary University of London</i> ; Farbod Shahinfar, <i>Sharif University of Technology</i> ; Gianni Antichi, <i>Queen Mary University of London</i> ; Brent E. Stephens, <i>University of Utah</i>	

# Efficient Scheduling Policies for Microsecond-Scale Tasks

Sarah McClure<sup>\*</sup>, Amy Ousterhout<sup>\*</sup>, Scott Shenker<sup>\*†</sup>, Sylvia Ratnasamy<sup>\*</sup>

<sup>\*</sup>UC Berkeley <sup>†</sup>ICSI

## Abstract

Datacenter operators today strive to support microsecond-latency applications while also using their limited CPU resources as efficiently as possible. To achieve this, several recent systems allow multiple applications to run on the same server, granting each a dedicated set of cores and reallocating cores across applications over time as load varies. Unfortunately, many of these systems do a poor job of navigating the tradeoff between latency and efficiency, sacrificing one or both, especially when handling tasks as short as 1  $\mu$ s.

While the implementations of these systems (threading libraries, network stacks, etc.) have been heavily optimized, the policy choices that they make have received less scrutiny. Most systems implement a single choice of policy for *allocating cores* across applications and for *load-balancing* tasks across cores within an application. In this paper, we use simulations to compare these different policy options and explore which yield the best combination of latency and efficiency. We conclude that work stealing performs best among load-balancing policies, multiple policies can perform well for core allocations, and, surprisingly, static core allocations often outperform reallocation with small tasks. We implement the best-performing policy choices by building on Caladan, an existing core-allocating system, and demonstrate that they can yield efficiency improvements of up to 13–22% without degrading (median or tail) latency.

## 1 Introduction

Modern datacenter applications often involve many short Remote Procedure Calls (RPCs) to other servers. These RPCs allow applications with large memory footprints to access memory on other servers [2, 49, 51, 62, 69], enable applications to leverage large amounts of compute over short timescales [6, 25, 46], and provide replication and consensus [58]. The service times of these tasks grow ever smaller, and today are often a single microsecond or less [10, 34].

Tasks with short service times are particularly vulnerable to latency inflation; even small overheads can increase the latency of a 1  $\mu$ s task by an order of magnitude [10]. This is problematic for today’s applications, which depend on low latency both at the median and at the tail of the distribution (e.g., 99% latency) [5, 19]. As a result, researchers have proposed many techniques to reduce the overheads of handling these short tasks. These systems improve software with low-latency network stacks and better load balancing (DPDK [1], ZygOS [66], Shinjuku [36], eRPC [38], etc.) or propose new hardware to deliver packets to cores more quickly (RPC-

Valet [18], NeBuLa [74], NanoPU [34], Cerebros [65]). They offer tail latencies of a few dozen microseconds with existing hardware [26, 38] or several microseconds with new hardware [34].

However, as Moore’s Law slows [23], datacenter operators are increasingly concerned not just with providing low latency but also with achieving high CPU efficiency [79]. To do so, they pack multiple applications on the same server so that background applications can use any CPU cycles not used by latency-sensitive applications, as their load varies over time [11, 35, 80]. Several recent research systems enable this deployment model by allocating a set of cores to each application and then reallocating cores across applications as load changes (e.g., IX [12], PerfISO [35], Arachne [67], Shenango [60], Caladan [26], and Fred [40]). These systems walk a delicate tightrope, attempting to make spare cycles available for batch applications without harming the latency or throughput of latency-sensitive applications. Thus researchers have heavily optimized these systems’ implementations, squeezing spare CPU cycles and extraneous cache misses out of their network stacks, threading libraries, and core-allocation mechanisms.

While there have been significant advances in these mechanisms, less effort has gone into studying the *policies* that these core-reallocating systems implement. Each system implements two main policies: (1) a policy for **load-balancing** tasks across cores within an application and (2) a policy for when to **reallocate cores** from one application to another. There are many possible choices for each policy: popular load-balancing policies include work stealing [14], work shedding, and steering tasks to less-loaded cores when they are first enqueued [55] while core-allocation policies may be based on queueing delay [12, 26, 60], the arrival of new tasks [40], or CPU utilization [35, 67]. And yet, each system typically implements a single choice of load-balancing and core-allocation policy, providing little clarity about how different policies compare.

Unfortunately, as we will show (§2), these policy choices can contribute to suboptimal performance, with existing systems sacrificing significant CPU efficiency in order to maintain low latency, especially with short tasks. The root of the problem is that as task durations shrink from 100  $\mu$ s to 1  $\mu$ s, the overheads of balancing tasks or reallocating cores (e.g., a 50 ns cache miss to probe state on a different core) become relatively more significant, and inefficient policies become much more costly. In this paper, we focus on these policies and ask: *what load-balancing and core-allocation policies*

*yield the best combination of latency (median and tail) and CPU efficiency for microsecond-scale tasks?*

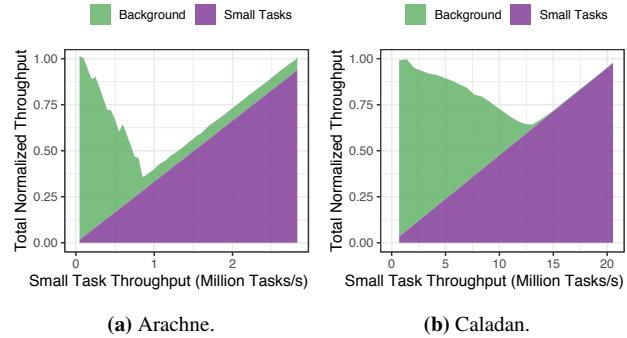
We focus on the *combination* of latency and efficiency because while ideally we would like to optimize both, there is an inherent tradeoff between the two. For example, allocating infinite cores could achieve optimal latency at the cost of terrible efficiency, while allocating a single core could achieve good (but perhaps not optimal) efficiency, but potentially high latency. The best one can hope for is to operate on the Pareto frontier of latency and efficiency; i.e., a point where it is not possible to improve one quantity without harming the other.

To compare policies fairly and independently from any specific implementation, we turn to simulations (§4). We use measurements of real systems to estimate the overheads of balancing tasks across cores within an application and of reallocating cores across applications. We then model simple versions of common load-balancing and core-allocation policies, and simulate them using our estimated overheads. We use these simulations to conduct an extensive factor analysis, teasing apart the impact of load-balancing policies and core-allocation policies on both latency and efficiency. From this analysis, we glean three key insights:

First, assuming commodity NIC hardware, *work stealing* is the load-balancing policy that yields the best latency and CPU efficiency and forms the Pareto frontier. We find that this conclusion is remarkably robust across different average service times, service time distributions, numbers of cores, latency metrics (e.g., median vs. 99%), whether cores are dynamically reallocated or statically partitioned, and how much overhead load-balancing a task entails.

Second, in contrast, our analysis of core-allocation policies shows that multiple policies can perform similarly well (though some policies perform significantly worse). We find that *revoking cores proactively, rather than waiting until they go idle to yield them to another application, makes it easier to achieve good efficiency with small tasks, especially with many cores*. We identify two policies (based on average queueing delay and average CPU utilization) that fit this criteria, perform well, and can be configured to make different tradeoffs along the Pareto frontier; two other policies used in current systems yielded worse latency, CPU efficiency, or both.

Third, even with the best core-allocation policies, if the average load is fixed, *with small tasks it is difficult to achieve better performance by reallocating cores than by allocating a fixed number of cores*. For our request patterns (modeled with exponentially-distributed inter-arrival times), reallocating cores in response to transient bursts does not improve latency (median or tail) relative to statically allocating the same average number of cores. Thus the main benefit of reallocating cores over short timescales with short tasks is the ability to quickly adapt to changes in *average* load. In contrast, when average task service times are longer—several microseconds or more—we find that reallocating cores does improve performance even with constant average load.



**Figure 1:** Total useful work done by two colocated applications—one background, the other handling small (about 1  $\mu$ s) memcached tasks—as we vary memcached’s load (for two existing systems).

From this factor analysis we conclude that barring technology changes (e.g., commercialization of recently proposed NIC hardware [18, 34, 65, 74]), for low latency and high CPU efficiency, work stealing is the best load-balancing policy, and our two new core-allocation policies based on average delay or average utilization (we refer to these policies as “delay range” and “utilization range”) perform best. We implement these policies in a real system by extending Caladan [26], a state-of-the-art system for reallocating cores which already supports work stealing. We demonstrate that when running memcached, a key-value store, delay range and utilization range can save up to 13–22% of cores relative to Shenango’s and Caladan’s core-allocation policies, without degrading median or tail latency (§6).

## 2 Motivation

To demonstrate the inefficiencies of existing systems when handling short tasks, we conduct an experiment in which we run two applications on a server: a latency-sensitive application that handles short tasks and a background application that consumes all extra CPU cycles. We use memcached [49], a key-value store with service times of about 1  $\mu$ s, as our latency-sensitive application. We vary the offered rate of memcached tasks and measure how much useful application-level work each application completes. We perform this experiment with two existing systems: Arachne [67] and Caladan [26].

Both systems yield latency improvements: Arachne’s 99% latency improves on that of Linux by hundreds of microseconds, while Caladan reduces this further, due partially to replacing Linux’s network stack with kernel bypass. However, in their efforts to provide low latency for the small tasks, these systems waste significant CPU resources. Figure 1 shows the total throughput achieved by each system, normalized by the maximum throughput the application can achieve when running alone on the configured set of cores (16 for Arachne and 32 for Caladan). Thus at the lowest and highest loads (where only one of the applications is running<sup>1</sup>), both systems are

<sup>1</sup> Arachne dedicates one core to each application, so its background throughput never reaches zero.

at their highest possible efficiency, achieving a total normalized throughput of 1.0. Ideally, the total throughput of both applications would remain at 1.0 as the small task load varies. However, at moderate loads, both systems suffer significant efficiency losses, wasting up to 64% or 36% of their cores, with Arachne and Caladan, respectively. This inefficiency is not exclusively bad; the excess cycles can be used to handle small tasks sooner, lowering latency.

From these results, it is clear that these systems are able to multiplex cores between applications, but they are extremely inefficient while doing so. When handling longer tasks (e.g., 10  $\mu$ s or 100  $\mu$ s), these systems become much more efficient. This begs the question: what is responsible for these efficiency losses with short tasks? These systems differ along many different dimensions: their core-allocation policies, their load-balancing policies, their threading libraries, and whether they use the Linux network stack (Arachne) or kernel-bypass (Caladan). The latter implementation aspects can contribute significantly, but they have been studied extensively by prior work. We focus instead on the policy aspects and seek to understand which load-balancing and core-allocation policies yield the best performance for small tasks.

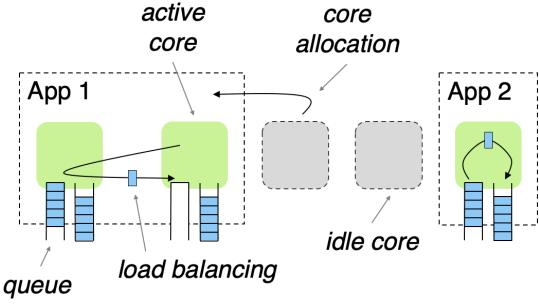
### 3 Design Space of Policies

If reallocating cores across applications and load balancing tasks between cores incurred no overhead (i.e., they could be done instantaneously), the optimal policies would be: (1) immediately grant an application a new core whenever a task arrives and yield the core when the task completes and (2) steer each newly arrived task to its newly granted core. With these policies, CPU usage would exactly match the time spent on tasks (100% efficient) and if an additional core was always available then tasks would never queue (zero added latency).

These idealized policies are sufficient with long task service times (e.g., 100  $\mu$ s or more), because the overheads of load balancing and core reallocation are relatively small (§4.3). However, with tasks as short as a single microsecond, load-balancing and core-allocation overheads become significant and we can no longer afford to perform both a core-allocation and a load-balancing action for every task that arrives; doing so wastes considerable CPU resources. For good performance with short tasks we must consider other policies. The key difference between distinct policies is when they choose to incur overheads (e.g., when a task arrives vs. when a queue builds up), and these choices determine their latency and CPU efficiency. Thus finding the best load-balancing and core-allocation policies amounts to asking the question: *given that load balancing and core allocation incur overheads, how should we spend those overheads most effectively?*

#### 3.1 Setting and Assumptions

While exploring different policies, we make several assumptions about our setting (illustrated in Figure 2). We assume that each server runs one or more *applications*, where each ap-



**Figure 2:** Applications use *load balancing* to balance tasks across cores and *core allocations* to adjust the number of cores available to each application.

plication is either a batch application that seeks high throughput and is latency-insensitive or a latency-sensitive application that handles short *tasks*. Each application is always allocated a specific number of cores; when an application yields a core, the core will be granted to another application if possible.

Tasks can either arrive from external sources (e.g., a packet arrives over the network or a storage operation completes) or be created by the local CPU (e.g., a thread spawns a new thread). We focus on settings with commodity NICs that spray packets randomly over available cores (e.g., with RSS [3]), though we also discuss how performance could change with recent proposals for new NIC hardware with advanced steering capabilities (§4.2.1). Unless specified otherwise, we assume that each core maintains its own queue(s) of tasks and that tasks are not intentionally re-ordered (cores handle them in FIFO order). We assume no preemption of running tasks and no *a priori* knowledge of how long each task will take to run.

#### 3.2 Policies

In this section, we summarize the main policies used for load balancing and core allocation today and describe when each incurs overheads; these are the policies we evaluate in our factor analysis (§4). The list is not exhaustive but rather an attempt to cover the main classes of existing policies as well as the theoretically optimal policies.

##### 3.2.1 Load-Balancing Policies

Load-balancing policies can perform load balancing either when a task arrives or once it has already been queued. The latter category can be further divided based on what triggers load balancing (either a lack of tasks for a core or a core with too many tasks). We begin by describing a theoretical optimum, and then describe four practical policies that fall into these categories. Note that these policies are not necessarily mutually exclusive.

**Single queue.** With no overheads, the theoretically ideal load-balancing policy places all tasks in a single shared queue. However, this approach limits throughput in practice due to contention for the single queue. Shinjuku [36] and RAMCloud [62] take this approach; Shinjuku can support only

System	Load-balancing Policy	Core-allocation Policy	
		Trigger for Adding a Core	Trigger for Revoking a Core
IX [12]	none	packet queueing delay	low CPU utilization
Arachne [67]	choice on enqueue with power-of-two choices [55]	number of runnable threads	low CPU utilization
Shenango [60], Caladan [26]	work stealing	max queueing delay of threads or packets	failure to work-steal
Fred [40]	steering on arrival or work-stealing once all cores are allocated	task arrives	task completes
Go [76]	work stealing	task arrives and no cores work stealing	failure to work-steal

**Table 1:** Load-balancing and core-allocation policies used by existing systems. Core-allocation policies are highlighted to indicate whether they rely on queueing, utilization, task arrival, or failure to find work.

about 5 million requests per second with a single queue.

**No load balancing.** Without load balancing, tasks are handled by the core they first arrive at, such as the core that spawns a thread or the core a packet or storage completion is steered to by hardware [12, 42, 64]. This approach incurs no load-balancing overheads.

**Enqueue choice.** Enqueue choice policies make a load-balancing decision about which core to assign a task to when the task is first created; tasks cannot be moved later. Existing systems commonly use “power of two choices” [55] to enqueue a task to the less-loaded of two randomly sampled cores [33, 67, 81]. When a task is first created, the creating core incurs overhead to sample queues on other cores (which can be done in parallel for small numbers of sampled cores) and to enqueue the task to the chosen core.<sup>2</sup>

**Work stealing.** When a core is idle, it searches for a core that has queued work, and then steals half the tasks from that core and moves them to its own queue [14]. This approach is used by the Go runtime [76], several multithreading platforms [9, 16, 17, 45, 47, 59, 68], and many research systems [26, 40, 44, 50, 60, 66, 81]. It incurs overhead to check other cores for queued work and move work to its local queue.

**Work shedding.** With work shedding, overloaded cores can shed load to other cores or request that other cores take some of their load. This has been considered by several theoretical papers [22, 73, 77] and for load-balancing systems in a variety of contexts [56, 78]. We consider a work-shedding policy in which a core that has had tasks queued for longer than a specified threshold selects a random core and indicates that it is overloaded. That core will then respond by stealing half of the overloaded core’s tasks; this is the primary source of overhead for this policy.

### 3.2.2 Core-Allocation Policies

All core-allocation policies incur overhead in the same way: by adding or revoking a core. Their overheads are primarily determined by how often they reallocate cores and consist of both the latency until a core is available after a reallocation decision is made and the CPU cycles that cannot be used productively while a core is being reallocated. The performance of each policy is determined by how effective the signals are that it uses to trigger core reallocations. Most policies make

<sup>2</sup>Note that “no load balancing” is a special case of enqueue choice in which there is only one choice and no overhead.

core-allocation decisions at fixed time intervals (e.g., every 5  $\mu$ s [60]), though some are triggered by other conditions.

We cannot easily model or compute an optimal core-allocation policy, i.e., one that achieves the optimal tail latency for a given CPU efficiency or vice versa. This is because finding the optimal tail latency for a given CPU usage bound or vice versa is NP-hard assuming a finite number of cores and non-constant service times; this can be shown by a reduction from the multiprocessor scheduling problem (see Appendix A.1). We now list the core-allocation policies we consider.

**Static.** With static core allocations, the number of cores allocated to each application cannot change over time, as in several research systems [42, 64, 66]. This incurs no overhead for core reallocations. However, each application must be provisioned with enough cores for peak load, wasting significant CPU resources as load varies over time, which is typical of datacenter workloads [11, 35].

**Per-task.** Systems such as Fred [40] with per-task core allocations grant a core to an application every time a task arrives. This incurs the overhead of a core allocation for each task, except when all cores are in use.<sup>3</sup>

**Queueing-based.** Policies based on queueing delay grant an application an additional core if the queueing—as measured by either the number or delay of threads, packets, or storage completions—exceeds a certain threshold. These policies vary in whether they trigger based on the maximum queueing across cores [26, 60] or use an average [12, 67].

**CPU utilization-based.** Utilization-based policies add or revoke cores based on the number of idle cores [35] or the average fraction of time cores spend working on tasks (as opposed to sitting idle or busy-spinning) [12, 67].

**Failure to find work.** In some systems, an application will yield a core when the core is unable to find any tasks to work on. This can happen when a core fails to find another core with queued work to steal from [26, 60, 76] or when it finishes its current task, with a per-task core-allocation policy [40].

### 3.3 Overheads

Both load balancing and core allocation entail overheads; in this section we discuss the magnitude of these overheads in typical systems today.

<sup>3</sup>Once all cores are allocated to an application, Fred places additional arriving tasks in per-core queues and cores use work stealing to find them.

**Load-balancing overheads.** Load-balancing overheads can be impacted by several factors: the CPU architecture (how long does it take to handle a cache miss? how many cache misses can be outstanding simultaneously?), the workload (how often is load-balancing state cached locally vs. modified on remote cores?), and speculative execution (how successfully can the CPU overlap cache misses with other instructions via speculative execution?). Despite these factors, we attempt to estimate the overheads of different load-balancing policies and in Section §4.2.1 we demonstrate that our conclusions about the relative performance of different policies are unlikely to change with different overheads.

Because load balancing requires communication between cores, its overhead arises primarily from cache misses while retrieving cache lines from the L2 cache of another core. Depending on the CPU microarchitecture, one such cache miss can cost between 30 ns (Intel Haswell) and 200 ns (Xeon Phi) [72]. A load-balancing operation moves state from one core to another; this typically entails about three cache misses to read a remote cache line, invalidate it so that it can be written in the local cache, and then a third cache miss when the remote core reads the modified cache line [67]. The overhead incurred by the core performing the load balancing will then be about two cache misses, or 60-400 ns.<sup>4</sup> For comparison, we measured that Caladan [26] takes about 120 ns on average to check via work stealing if another core has stealable work (in the form of queued packets, threads, or timers).

Note that a single core can typically have up to about 10 cache misses outstanding at once [24] (we confirmed through a microbenchmark [48] that this seems to be about 10-12 for our Intel Skylake servers). This enables small numbers of independent cache misses (such as those to sample the load on two different cores) to incur in parallel.

**Core-allocation overheads.** The latency for a core allocation to complete varies depending on the mechanism used to reallocate the core. At a bare minimum, reallocating a core requires an inter-processor interrupt (IPI) from the core that makes the reallocation decision to the core that will be reallocated to a different application; this takes about 1993 cycles or roughly 1  $\mu$ s [36]. Existing systems report slightly higher core-allocation latencies, varying from 2.2  $\mu$ s to reallocate an idle core or 7.4  $\mu$ s to reallocate a busy core in Shenango [61] to 29  $\mu$ s to reallocate a core in Arachne [67].

## 4 Factor Analysis

In this section, we perform a factor analysis to determine the relative performance of the load-balancing and core-allocation policies defined in §3.2. We cannot effectively compare different policies by comparing existing systems that implement them (e.g., Caladan vs. Arachne), because these systems differ in many aspects besides their policies (threading libraries, net-

work stacks, etc.). Even comparing different policies within a single implemented system can be challenging, because the optimal system design may vary depending on the policy. For example, systems may use different locking mechanisms to protect thread queues depending on whether only the local core can enqueue to them (as in work stealing) or if remote cores can also enqueue to them (as in enqueue choice). Thus, to decouple the behavior of the policies from the behavior of the systems that they are implemented in, we use simulations.

Our simulations rely on several parameters which define both the workload and assumptions about the possible underlying system. We find that our conclusions are quite robust to variations in these parameters, and therefore may be applicable to a wide variety of implementations and workloads. We have made the source code for our simulations available at <https://github.com/smccclure20/scheduling-policies-sim>.

### 4.1 Simulation Methodology

While our focus is on policy choices rather than implementation details, we do seek to model realistic overheads for cross-core communication and for allocating cores to applications. In order to fairly compare different policies, we use consistent values for each overhead, based on the overheads measured above (§3.3). We model the cross-core communication generally required for load balancing as taking 100 ns. We model the core-allocation overheads (both latency to allocate a core and wasted CPU cycles) as 5  $\mu$ s per core allocation. In §4.2.1, we will consider some different values for load-balancing overheads, though varying them by even 100% does not have a profound impact on our results. We discuss the implications of varying core-allocation overheads in §4.3.

Our overall model assumes that each core has a single local queue (i.e., no distinction between packet and thread queues) and that tasks arrive randomly at the queues of allocated cores. This is representative of a NIC randomly steering tasks to cores or of running threads randomly spawning an additional thread. Our simulator models each of the general policy approaches outlined in §3.2, with specific implementation choices made based on real system implementations whenever possible. We acknowledge that our model is a simplified view of these systems, but we found that the general trends of latency and efficiency are consistent between simulations and experiments, for the systems we evaluated (§6). Our simulator does not support preemption but could be extended to model systems which do [20, 36, 82]. We now describe the specific load-balancing and core-allocation policies that we simulate.

**Load-balancing policies.** We model no overheads for the idealized *single-queue* policy or for the *no load balancing* policy. For *enqueue choice*, when a task arrives, the core at which it arrives incurs the 100 ns overhead to move the task to its destination queue (the shortest queue from two randomly

<sup>4</sup>This is an approximation; the exact overhead will depend on application behavior.

sampled options).<sup>5</sup> When *work stealing* is enabled and a core does not have any work in its local queue, it begins iterating through the other cores, checking if there is available work to steal. Each check of a remote queue incurs the 100 ns overhead, as does the act of stealing any found tasks. With *work shedding*, each core checks if its queue’s current queueing delay is higher than the configured threshold after each task it finishes. If so, it selects a random core to notify or “flag.” The remote core will check for flags between each of its tasks, respond (if a flag is present) by stealing tasks from the overloaded queue so that the two queue lengths are balanced, and incur the 100 ns overhead.

**Core-allocation policies.** Our *per-task* policy (based on Fred [40]) immediately grants a new core to an application if one is available in the system whenever a new task arrives. The core at which the task is initially randomly placed pays a 100 ns overhead to place the task at the new core. When a core finishes a task, it checks if there are more queued tasks in the system than available cores and yields if there are not.

The remaining core-allocation policies make decisions at fixed time intervals. To model *Shenango* [60] and *Caladan* [26], at the end of every core-allocation interval, the simulation determines the maximum queueing delay across cores within an application. If it exceeds a specified threshold (typically the length of the interval itself), the simulation grants an additional core to that application. An application yields a core if the core attempts to work steal from every other core in the application and fails to find any tasks to steal. *Shenango* and *Caladan* have very similar policies; the main distinguishing factor in our model is the difference in their interval/threshold values (Table 2).

We also design and simulate two new core-allocation policies. First, we design a queueing-based policy called *delay range* which attempts to maintain a specified average queueing delay across all cores within an application. Every core-allocation interval (every 5  $\mu$ s), the simulation checks the average queueing delay. If it is below the specified lower bound, a core is revoked; if it is above the upper bound, a core is added. Similarly, with our *utilization range* policy, a core is added or removed whenever the average CPU utilization over the past interval (fraction of time spent handling tasks) falls outside the specified range.

There are three notable aspects of core-allocation systems that we do not model. First, some systems dedicate a scheduler core to making core-allocation decisions and initiating core allocations [26, 60, 67] while others have application cores perform these tasks in a distributed way [40, 76]. We do not model these distinctions and assume that all work for initiating core reallocations could be offloaded to a separate dedicated core. Second, we do not model the overheads incurred by applications measuring and exposing statistics to the dedicated core; in practice these overheads are small and

Parameter	Default Value
Work shedding delay threshold	2 $\mu$ s
Enqueue choices	2
Utilization range	75-95%
Delay range	0.5-1 $\mu$ s
Shenango max queueing threshold	5 $\mu$ s
Caladan max queueing threshold	10 $\mu$ s

**Table 2:** Canonical configuration parameters.

simply require application cores to write a small amount of state (e.g., timestamp when a task was queued) to shared memory. Third, we do not model the caching implications of reassigning a core from one application to another.

**Configuration.** Each policy has its own unique parameters. Unless stated otherwise, we use the default parameter values shown in Table 2. We chose these specific values based on the best overall performance seen for each policy, though we will discuss the implications of configurability throughout this section.

In all of our simulations, we use a canonical configuration of 32 cores, exponentially-distributed service times with an average of 1  $\mu$ s, Poisson arrivals, and an offered load that occupies 50% of the cores on average. Experiments below will vary many of these dimensions independently, but we will use this configuration by default. To contextualize the policy overheads described above, with the average task time set at 1  $\mu$ s, the overhead for load balancing is 10% of average task time while the overhead of core reallocation is 500%.

## 4.2 Load Balancing

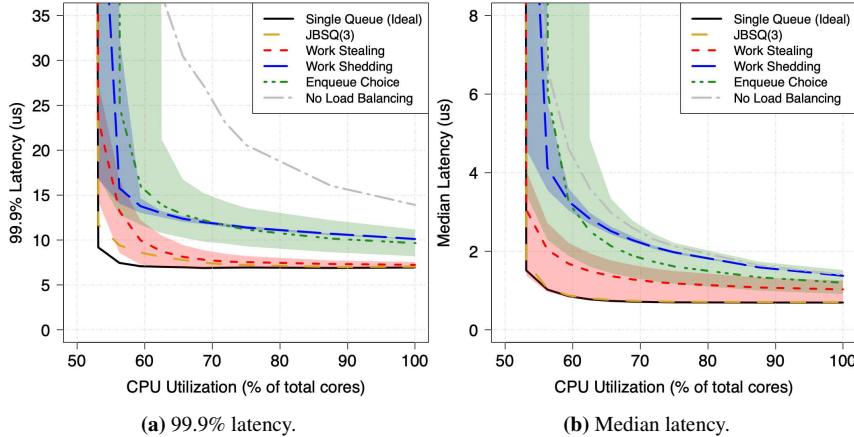
To understand how load-balancing policies impact performance, we first evaluate different load-balancing policies in a setting where cores are statically allocated (cores are never reallocated) (§4.2.1), and then evaluate whether core reallocations impact these findings (§4.2.2).

### 4.2.1 With Static Core Allocations

**Individual policies.** We first evaluate each load-balancing policy independent of any particular core-allocation policy by running each experiment with a fixed number of cores. This allows us to determine the relative performance of each approach when given the same number of total CPU cycles, since a given allocation policy will make different allocation decisions depending on the behavior of the specific load-balancing scheme, even under the same traffic. By decoupling the two, we can determine which end-to-end effects are due specifically to load-balancing policies.

Figure 3 shows the tail and median latencies (y-axis) of different load-balancing policies as we vary the number of statically-allocated cores (shown on the x-axis as a fraction of the total possible), while offering an average load of 50%. Each curve corresponds to a load-balancing policy with 100 ns overheads, while the shaded regions vary this from 0 ns to 200 ns. In general, approaches that operate lower and to the left in this graph are preferable. We will discuss the JBSQ

<sup>5</sup>We assume the options may be checked in parallel as explained in §3.3.



**Figure 3:** Performance of each load-balancing policy with different numbers of statically allocated cores. Shaded regions cover overheads 0-20% of average task time. The line in each region shows the canonical case of 10% load-balancing overheads (100 ns).

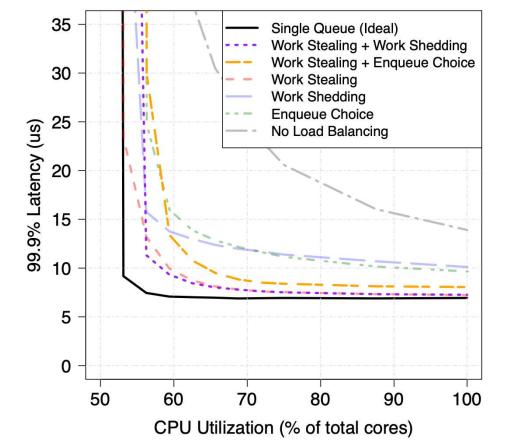
curve later.

**Finding 1:** *With static core allocations, work stealing achieves better latency (at the median and tail) for a given efficiency (number of allocated cores) than work shedding or enqueue choice.*

While all load-balancing policies yield significant improvements over no load balancing, work stealing consistently has significantly lower median and tail latency for the same number of statically-allocated cores than the other approaches; work stealing Pareto-dominates enqueue choice and work shedding. The relative performance between enqueue choice and work shedding is less consistent and varies depending on system and workload parameters such as the number of allocated cores, latency percentile, and service time distribution (Appendix A.2.2).

The enqueue choice curve is consistent with the well-known “power-of-two choices” result [55], showing that two choices of queues is much better than one (the “No Load Balancing” curve). This is particularly true when there is no overhead (as modeled in [55]) which is demonstrated by the lower bound of enqueue choice’s shaded region in Figure 3. Despite this, enqueue choice still performs worse than work stealing. Further measurements revealed that this is due to three main limitations: (1) per-task load-balancing overheads that cap the possible throughput and add latency to all tasks, (2) a limited number of queue choices, and (3) placement based on number of queued tasks rather than the sum of service times of queued tasks. Overall, (2) and (3) can result in periods of load imbalance in which tasks are queued and cores are idle, but there is no way for the idle cores to assist with those “stranded” tasks. Choosing by the sum of the service times in the queue [30, 31] or increasing the number of choices can improve tail latency, though these are not always practical, and reducing the overheads to 0 provided a bigger performance benefit than either of those changes individually.

The tail latency gap between work stealing and work shed-



**Figure 4:** Latency curves for combinations of work stealing with other load-balancing policies, with static core allocations.

ding can be explained by the steps necessary to move a task that ends up contributing to tail latency to the core that ultimately handles it. With work shedding, for a task at an overloaded core, time is spent waiting to cross the signalling threshold, waiting for the core to complete its current task and raise a flag, and waiting for the remote core to respond. In work stealing, tasks simply wait until a work-stealing core checks their queue; the latency of this depends primarily on the number of excess cores. With a work-shedding queueing threshold of 2  $\mu$ s we found that on average tasks that were shed spent 3.1-4.5  $\mu$ s queued on cores other than the one that ultimately handled the task, compared to 0.3-1.4  $\mu$ s with work stealing. Most tasks at the tail are stolen at least once, explaining the corresponding gap between the two in tail latency. Lowering the queueing threshold only yields marginal improvements, because at higher loads most cores will always have a flag pending. In addition, without preemption, tasks still incur delays from the other two steps described above.

We now investigate the robustness of these results to changes in overheads in case our overhead estimates are not representative of certain underlying hardware or better technology arises in the future. Note that this does not apply to single queue simulations or those with no load balancing as they have no overheads. By looking at the upper or lower bounds of the shaded regions in Figure 3, we see that for the same overhead, work stealing consistently outperforms the other approaches. Even if inter-core communication was free for enqueue choice and work shedding, work stealing with 200 ns overheads outperforms for most numbers of allocated cores. Further, work stealing consistently achieves the best performance even if we model the load-balancing overhead as 400 ns, the upper bound of our estimate from §3.3.

Work stealing’s superior performance is robust across different latency percentiles (median to 99.9%) (Figure 3), average service times (*e.g.*, 1, 10, 100  $\mu$ s) (Figure 5), numbers of cores (Figure 6), loads (Appendix A.2.1), and ser-

vice time distributions (exponential, constant, bimodal) (Appendix A.2.2). For all service time distributions evaluated, the ordering of the static curves remained the same as in exponential distributions shown. Though, when service times are constant, the specific choice of load-balancing policy has less overall impact on system performance.

**Combining policies.** Notably, these load-balancing approaches are not mutually exclusive. Since each policy takes effect at a different time in the handling of a task, work stealing can take advantage of extra cycles while work shedding addresses excessively loaded cores or enqueue choice proactively tries to balance queues. Accordingly, we simulated work stealing combined with each other approach with static core allocations.

**Finding 2:** *With static core allocations, adding shedding on top of work stealing provides some latency benefit (primarily at the tail) while adding enqueue choice to work stealing makes performance unchanged or worse.*

This is demonstrated in Figure 4. We see that adding enqueue choice to a system that already employs work stealing does not improve performance. There are two reasons for this, depending on what efficiency (x-axis) we are operating at: (1) with few cores available, enqueue choice adds significant overhead per-task which degrades throughput, and (2) with many cores available, there is little room for improvement between work stealing and single queue. When adding work shedding to work stealing, however, there are some benefits since the shedding mechanism can help balance out queues under high-load conditions when work stealing lacks the extra cycles to help, though the benefit is fairly limited to certain efficiencies as the overheads of flagging can become excessive when spare cycles are rare.

**Leveraging hardware.** Given these results, we ask two questions motivated by recent advances in hardware: (1) what if the NIC can perform more intelligent distribution than simple hashing? and (2) what impact would handling many cache misses in parallel have? (1) is motivated by recently proposed systems such as the NanoPU [34] which selects queues for incoming packets according to join bounded shortest queue (JBSQ) [43].<sup>6</sup> JBSQ is known to achieve good performance with tail latency improvements up to 10  $\mu$ s over work stealing, as shown in Figure 3. However, this boost requires new hardware to direct incoming traffic intelligently.

To address (2), we simulated scenarios where the underlying hardware could resolve several cache misses at once (as described in §3.3). Ultimately, this capability means that load-balancing policies may communicate with multiple cores for the price of one (e.g., check 10 cores for the presence of work in work stealing). However, we found that these modifica-

<sup>6</sup>JBSQ( $n$ ) queues up to  $n$  outstanding tasks at each core (including the task currently being handled) and maintains any surplus in a central queue [43]. We evaluate the case of 3 outstanding tasks, as in NanoPU, though we label this as JBSQ(3) in the terminology of [43] rather than JBSQ(2) as in NanoPU.

tions provided marginal benefits at best, even assuming that processing the results of parallel checks incurs no overhead.

In general, work stealing was consistently the best performing load-balancing policy when given the same number of cycles as other approaches even as overheads and workload parameters vary. Broadly, work stealing achieves high performance by avoiding per-task overheads and leveraging idle cores to avoid stranding tasks at overloaded cores. Ultimately, absent new hardware, work stealing is the best option for load-balancing approaches among those we evaluated. While work stealing’s superiority may seem unsurprising given its widespread use, we believe that we are the first to compare it against other policies and demonstrate its benefits when handling microsecond-scale tasks with realistic load-balancing overheads.

#### 4.2.2 With Dynamic Core Allocations

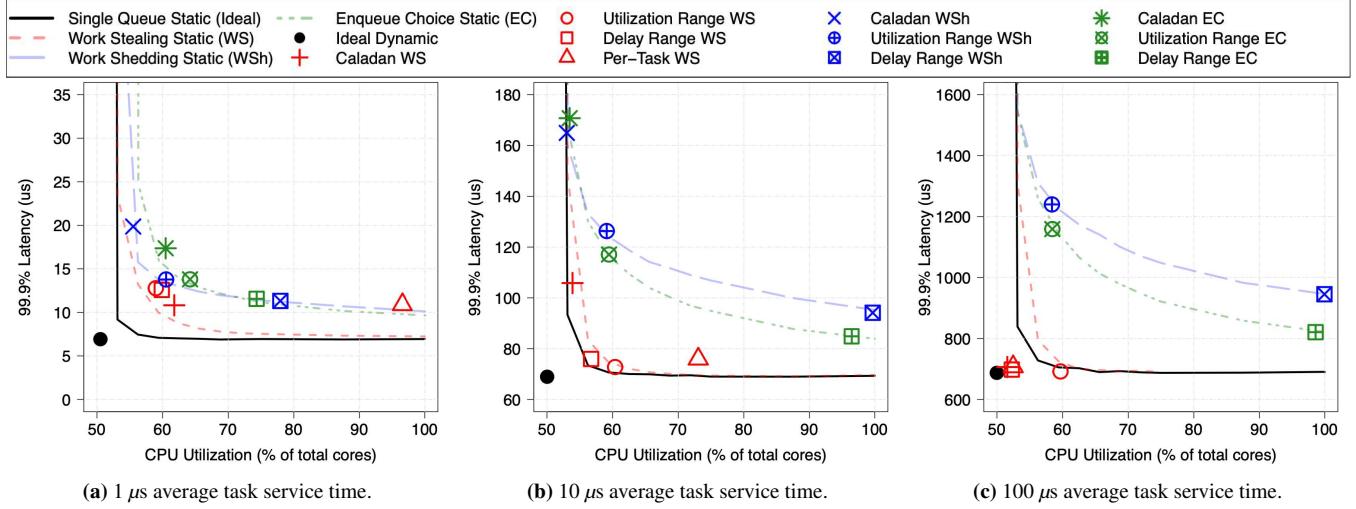
Next, we consider how load-balancing policies perform when cores can also be reallocated: does reallocating cores change the findings above? When the number of cores allocated to a given application varies over time, it becomes harder to compare approaches (combinations of load-balancing and core-allocation policies). Each combination represents a single point in the tradeoff space between latency and efficiency. If one combination has better latency but worse efficiency than another (i.e., neither is Pareto dominant), which is preferable? Some core-allocation policies are configurable and could be tuned to operate at the same efficiency to compare their latencies. However, not all approaches are tunable (e.g., per-task allocations), so this methodology cannot be used to compare all policies. Thus it is not always possible to say that one policy combination is definitively better than another.

We attempt to pair each load-balancing policy with each other core-allocation policy, but some pairings require modifications or are not reasonable. In Shenango/Caladan, cores park upon failing to find any work to steal. We modify this to work with other load-balancing policies by revoking cores after they spin for the time it would take to check all cores in traditional work stealing, assuming no additional work arrives in the meantime. Per-task allocations maintain the invariant that the number of active cores is equal to the minimum of the number of tasks present and the total number of cores. This is only reasonable with a work-conserving load-balancing policy, so we only evaluate per-task core allocations with the work-stealing load-balancing policy.

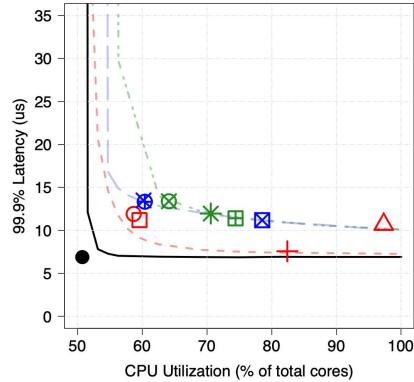
With this in mind, we simulated all coherent combinations of load-balancing and core-allocation policies to compare how they explore the available tradeoff space. The results across different average task durations are shown in Figure 5.

**Finding 3:** *When cores are dynamically reallocated, work stealing performs better than shedding or enqueue choice. This is robust against all factors mentioned in Finding 1.*

Figure 5a shows each combination of load-balancing and core-allocation policies with static-allocation curves for ref-



**Figure 5:** Combinations of each core-allocation policy with a load-balancing policy at 50% load with static-allocation curves for each load-balancing policy for reference. Each policy uses the canonical configurations from §4.1. Points for each combination of policies are the color of their load-balancing curve and have the shape type (squares, circles, stars, or triangles) of their core-allocation scheme.



**Figure 6:** Core-allocation and load-balancing policy combinations for 64 cores and 1  $\mu$ s tasks. Refer to the legend in Figure 5.

erence. Comparing each core-allocation policy (shape type) across load-balancing policies (colors), we see that work stealing always performs best in terms of proximity to the single queue curve. Note that this graph only shows one choice of parameters for each core-allocation policy, but some can be configured to make different latency vs. efficiency tradeoffs. We generally chose the configuration closest to the bottom-left of the graph, though we will discuss configurability broadly in §4.3.

While adding dynamic core allocations makes the individual performance of each load-balancing policy less clear, overall work stealing still consistently performs better than other load-balancing approaches (absent new hardware).

### 4.3 Core Allocation

In this section, we compare the performance of different core-allocation policies. Since core-allocation policies are designed to react to changes in load, their performance tends to be tightly coupled with the load-balancing policy employed.

Better load-balancing policies will more effectively use the available cycles, allowing the core-allocation policy to be more conservative in granting cores. Therefore, we evaluate each core-allocation policy across each load-balancing policy and seek to find patterns in the tradeoffs between efficiency and latency that each core-allocation policy makes.

We note that some existing systems use an additional dedicated core (such as Shenango’s IOKernel) to perform core allocations [26, 60, 67]. We do not count these cores as we are focusing on policy rather than the implementation of that policy. If we were to include these cores, all efficiency results for these policies would incur an additional 3% CPU utilization for a 32-core system.

We began by asking the question: does reallocating cores yield better performance than sticking with a constant number of cores? One might expect that even with constant average load, being able to react to bursts in load over small time scales would yield significant performance benefits. Surprisingly, we found that the answer to this question is often ‘no’.

**Finding 4:** *For short tasks, none of the core-allocation policies we tried achieved better latency (median or tail) for a given average efficiency than static core allocations (with the same load-balancing policy). However, this becomes possible with longer tasks.*

In Figure 5a, none of the core-allocation policies achieve better tail latency for the same efficiency as a static allocation (the points fall up and to the right of their corresponding static core-allocation curves). As shown in Figures 5b and 5c, when the average task service time is longer (e.g., 10  $\mu$ s or 100  $\mu$ s), some policy combinations (points) can achieve better performance than their static-allocation curves. With work stealing and 10  $\mu$ s service times, delay range, utilization range, and Caladan all beat the static curve for 99.9% tail latency, but

not for the median (omitted for space). This is true for  $100\ \mu s$  service times as well, with per-task also beating the work-stealing curve. As the average task duration increases, the relative importance of the core-allocation overhead decreases and allocating new cores for additional tasks becomes reasonably efficient. The only policy combinations which beat their respective curve include work stealing as the load-balancing policy. Work stealing leverages extra cycles to distribute load while enqueue choice and work shedding are limited in impact since newly added cores will spin idly, unable to handle tasks until a new task arrives or they are flagged by another core.

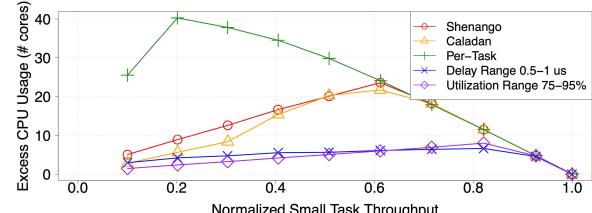
The only method we have found that can outperform the static curve with tasks as small as  $1\ \mu s$  requires the core-allocation system to be extremely reactive, making core-allocation decisions more frequently than  $5\ \mu s$  and giving the new cores to the application faster than in  $5\ \mu s$ . More frequent allocations are challenging in a real-world implementation because of the overheads of checking state and initiating core reallocations. For example, in Shenango, these actions take roughly  $2.1\ \mu s$  or  $3.4\ \mu s$  with 32 or 64 application cores, respectively [61]. Completing each core reallocation in less than a few microseconds is similarly challenging (§3.3).

Even though core allocations may not provide performance benefits with short tasks, one may employ a core-allocation policy to ensure that the application can adapt to changes in load. Average load in datacenters tends to vary over time [11], so allocating a static number of cores for a constant load would require provisioning for the peak load, wasting CPU cycles over time as load varies. Reacting more slowly to changes in load is also unlikely to perform well; prior work has shown that reactions at 50 ms timescales can cause significant tail latency spikes [60].

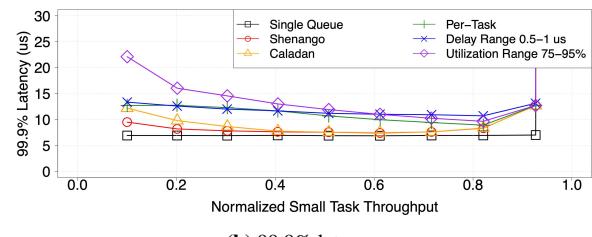
Assuming that achieving better performance than the static-allocation curves is unlikely for small tasks, we evaluate the different core-allocation policies in terms of the consistency of their performance and their ability to achieve high CPU efficiency. For some core-allocation policies, the placement relative to a static curve can vary significantly depending on the workload and load, making it difficult for an operator to configure the policy to achieve their goals (e.g., a specific tail latency or CPU efficiency target). By comparing the tradeoffs core-allocation policies make across workloads and loads, we find the policies that exhibit consistent performance.

**Finding 5:** Policies that explicitly optimize for an end-to-end user-visible metric (e.g., delay range and utilization range) have more consistent performance, as measured by those metrics, across different configurations.

For example, Figure 5 illustrates that for Caladan and per-task, the operating point changes with different service times. In contrast, utilization range and delay range specify a range on the x and y axes of the graphs, respectively, that the system should not leave. This generally forces the points to specific



(a) Efficiency measured in excess cores.



(b) 99.9% latency.

**Figure 7:** Performance of core-allocation policies paired with work stealing across loads for 64 cores. Efficiency is measured in the excess cores used compared to the single queue simulation.

regions of their static-allocation curves (when the curves cannot be crossed). For example, utilization range points achieve close to 60% CPU utilization across all service times in Figure 5.

Delay range and utilization range also have more predictable performance across different loads. In Figure 7, we illustrate how performance of different core-allocation policies varies with load (when paired with work stealing). We use 64 cores instead of 32 in order to sweep a wider range of loads. Figure 7a shows the efficiency measured as excess cores in comparison to the single queue ideal simulation (i.e., total number of cores used by a given policy minus those used in the ideal case) while Figure 7b shows the tail latency. Caladan, Shenango, and per-task have inconsistent efficiency and tail latency across loads, while delay range and utilization range each keep their respective end-to-end metric relatively constant. Overall, we found that policies such as delay range and utilization range have consistent performance across workloads and configurations, enabling the operator to directly tune the policy’s parameters to achieve a specific end-to-end performance objective.

Next, we consider whether each core-allocation policy can be configured to operate near the bend of each static-allocation curve, achieving high CPU efficiency while only minimally compromising in tail latency.

**Finding 6:** Yielding cores only when no work is found (when there is no queued work or work stealing fails) makes it challenging to achieve good efficiency with small tasks, especially with many cores.

The policies that yield cores only when no work is found (Caladan, Shenango, and per-task) cannot always achieve good CPU efficiency, especially with many cores. Here we focus on analyzing each core-allocation policy when paired

with work-stealing, as it performs best. Figure 5a illustrates that per-task achieves poor CPU efficiency with 32 cores, while Figure 6 shows that per-task and Caladan both achieve poor CPU efficiency with 64 cores, using more than 80% of CPU cores for a workload that only requires 50% of cores. In contrast, delay range and utilization range both operate near the bend of the static-allocation curves for 32 and 64 cores. Figure 7 illustrates that utilization range and delay range can save up to 15 cores for similar tail latency across loads.

The efficiency of the Shenango, Caladan, and per-task policies is limited because these policies are slow to yield excess cores. With per-task core allocations, before all cores are allocated the efficiency cannot reach higher than  $T/(T+R)$  where  $T$  is the average task time and  $R$  is the core-allocation overhead, because a core is allocated for every task. Similarly, in policies that yield cores only when work stealing fails (Shenango and Caladan), a significant amount of cycles can be wasted searching through all other cores to never find work or only to find it late in the search. As the number of cores increases, this effect gets worse. Neither Shenango/Caladan nor per-task can be configured to avoid these inefficiencies. Therefore, to achieve high efficiency across workloads and configurations, a core-allocation policy must revoke cores proactively, even when there is or may be some queued work.

We did assess other core-allocation policies such as maintaining a buffer of idle (or work-stealing) cores of a certain size (similar to PerfISO [35]) and enforcing this buffer at every allocation interval. However, this approach tended to be too noisy with short core-allocation intervals and performed significantly worse than other policies.

All together, we found that it is difficult to outperform static core allocations with small tasks, and if the average load is constant and known *a priori*, then statically allocating cores is the best option. However, when load is unknown or changes over time, dynamic allocation policies that proactively revoke cores perform best.

#### 4.4 Policy Takeaways

Overall, our factor analysis found that without new hardware, the best approach is to use work stealing as the load-balancing policy with delay range or utilization range for core allocations, depending on which end-to-end metric is more important to specify and stabilize. Both of these policies are able to operate close to the work-stealing static curve with short tasks or better than the curve with long tasks. Both are robust in the face of service time variability, different service time distributions, load changes, and changes in number of cores. Lastly, both are configurable, allowing the operator to choose whether they prefer CPU efficiency or tail latency (and to what extent). These approaches are intuitive; since core-allocation policies make a tradeoff between CPU efficiency and tail latency, using either parameter effectively as a signal for reallocating cores and controlling where to operate in the space of tradeoffs makes sense.

## 5 Implementation

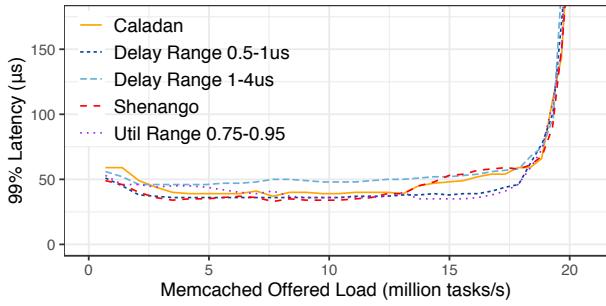
We implement our policies in a real system by extending Caladan [75]; our source code is available at <https://github.com/shenango/caladan-policies>. Like its predecessor Shenango [60], Caladan’s key components are its application runtime and its dedicated scheduler core, which implements the core-allocation policy. Caladan provides lightweight user-level threading, a high-performance network stack, and load balancing via work stealing. It also enables higher network throughput and its core-allocation mechanisms are more scalable compared to those of Shenango.

We implement both delay range and utilization range atop Caladan. This requires small modifications to both the runtime (50 LOC) and to the scheduler (125 LOC). The Caladan runtime already exposes information about the queueing delay of threads and packets to the scheduler core; we augment this with information about CPU utilization (time spent executing the application vs. in the runtime scheduler) as well. We also add the ability for application cores to yield voluntarily when notified by the scheduler core to do so. When application cores enter the runtime scheduler between tasks, they check if they should yield; for efficiency we do not preempt cores while they are handling tasks. In the scheduler core, we simply add logic for polling the utilization information exposed by applications, and use this or the delay information (depending on the current policy) to decide whether to add or revoke cores. When a core revocation is necessary, the scheduler revokes the core that currently has the least amount of queued work.

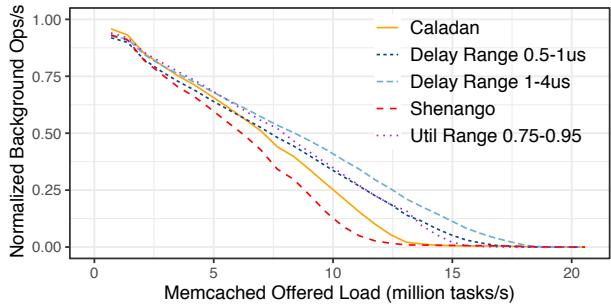
Measuring the CPU utilization of application cores over fine timescales is more challenging in practice than in simulation. This is because we do not interrupt running tasks to record CPU usage and only record how CPU time is spent whenever a task starts or finishes. Thus if a task runs for the entirety of a 5  $\mu$ s core-allocation interval, the scheduler core will observe 0 cycles spent in both the application and the runtime scheduler for that core. The scheduler core handles this by assuming that when application cores report no CPU usage for a core-allocation interval, their utilization is 100%, and it adds a core. In the case when an application has zero allocated CPU cores, CPU utilization is not a useful metric for deciding if an application needs more cores. Thus regardless of the core-allocation policy, the scheduler core always uses the arrival of packets to decide when to grant an application its first core, as in Shenango and Caladan.

## 6 Evaluation

The goal of our evaluation is to verify that the high-performing policies we identified above can actually yield performance improvements in practice for a real system. Unfortunately, varying the load-balancing policy within a single system would likely involve significant system changes, making a fair comparison difficult (§4). Thus we focus on evaluating the core-allocation policies. We start with a system (Caladan)



(a) Tail latency for memcached.



(b) Normalized throughput of the background application.

**Figure 8:** Performance of two applications under different core-allocation policies, when implemented atop Caladan. The x-axis varies the load of memcached.

that uses the best-performing load-balancing policy (work stealing) and evaluate its performance with different core-allocation policies. We evaluate four policies: Shenango [60], Caladan [26], delay range, and utilization range.

**Experimental setup.** We conduct experiments using two dual-socket servers with 28-core Intel Xeon Platinum 8176 CPUs operating at 2.1 GHz. Our server machine is equipped with a 40 Gbits/s Mellanox Connect X-5 Bluefield NIC (we do not use the SmartNIC features) and our client machine is equipped with an Intel E810C 100 Gbits/s NIC.<sup>7</sup> We enable hyperthreads and disable TurboBoost and frequency scaling. We use 32 hyperthreads on the second socket (to which our NICs are attached). We use Ubuntu 20.04 with kernel version 5.4.0.

**Applications.** We evaluate the different policies using *memcached* (v1.5.6) [49], a popular key-value store, as our latency-sensitive application. We use *loadgen*, Caladan’s open-loop load generator, to generate requests with Poisson arrivals over UDP [75]. Our workload consists of a mixture of read and write requests according to Facebook’s USR request distribution [8]; requests have service times of about 1  $\mu$ s. We run the *swaptions* workload from the PARSEC benchmark suite [13] as a background application and allow it to use all CPU cycles not used by memcached.

<sup>7</sup>We run Caladan in “queue steering mode” in which we reconfigure the mappings between NIC queues and cores when core allocations change [61] because our NICs do not support Caladan’s default “flow steering mode.”

## 6.1 Policy Comparisons

Our experimental results show that different policies yield different latency vs. CPU efficiency tradeoffs, but that delay range and utilization range generally outperform Shenango and Caladan, confirming the findings of our simulation-based factor analysis. Figure 8a shows the tail latency of memcached while Figure 8b shows the throughput achieved by the background application, both as we vary the load offered to memcached (x-axis). We show results for two different configurations of delay range to illustrate the impact of tuning the target range.

In Figure 8, utilization range and delay range ( $0.5\text{-}1 \mu\text{s}$ ) achieve similar tail latency for memcached as Caladan and Shenango, while achieving higher CPU efficiency for the background application. In addition, all of these policies yield similar median latency for memcached (not shown). Shenango is least efficient overall, and these two new policies achieve up to 22% more of the total possible throughput for the background application (7 hyperthreads worth) than Shenango. Compared to Caladan, these policies achieve up to 13% more throughput for the background application (4 hyperthreads worth). This is because with Shenango and Caladan’s policies, memcached spends much more time in the runtime scheduler, primarily work stealing (up to 26% and 21% of its time, respectively). In contrast, with the other policies, CPU time in the scheduler is much lower. For example, with utilization range, memcached spends less than 14% of its time in the scheduler at all except the lowest loads. By proactively revoking unused cores rather than waiting for work stealing to fail to find tasks to handle, delay range and utilization range can achieve higher CPU efficiency without degrading the performance for memcached.

Both the delay range and utilization range policies take as input a target range, and these ranges can be adjusted to make different tradeoffs between tail latency and CPU efficiency. As an example, Figure 8 shows two different ranges for delay range. Delay range  $1\text{-}4 \mu\text{s}$  achieves about 2 hyperthreads worth of additional throughput for the batch application compared to delay range  $0.5\text{-}1 \mu\text{s}$ , at the cost of  $10\text{-}15 \mu\text{s}$  of tail latency.

## 7 Related Work

**Load-balancing policies.** Load-balancing policies have been studied extensively, both theoretically and in the context of real systems. Several systems have adopted the ideal policy of maintaining a single shared queue [36, 62], though they experience throughput bottlenecks as a result. Others take the opposite approach and perform no load balancing in software, leaving it to the NIC [12, 64] or storage device [42] to randomly distribute work across cores; these approaches suffer from load imbalances.

Work stealing was originally proposed as a way of efficiently scheduling multithreaded computations across multi-

ple cores [14, 39, 71]. Variants of work stealing have been studied thoroughly [54, 57] and adopted in task-parallel platforms such as OpenMP [59], IntelTBB [68], Cilk [47], Habanero [9, 16], X10 [17], Java Fork/Join [45], and the Go runtime [76]. More recently, work stealing has been adopted in datacenter systems as a way to provide low tail latency [26, 44, 60, 66, 81]. Similarly, past work has analyzed the power-of-two choices load-balancing policy [55] as well as variants of it, such as those that consider known service times [31] or general service time distributions [15]. Arachne [67], SKQ [81], and many other systems [29, 33, 63, 82] leverage power-of-two or the more general power-of-k choices for load balancing. Others have studied work shedding approaches [73] and compared them to other policies [22, 77]. Finally, several recent proposals implement more advanced load-balancing policies such as JBSQ [43] in NIC hardware [18, 34, 70, 74].

Our findings are consistent with past comparisons of load-balancing policies. For example, we confirm that “work-first” load-balancing policies such as work stealing have better performance [21, 22, 27]. However, our analysis differs in two key ways. First, we are not aware of any prior work that compares load-balancing policies in the presence of realistic load-balancing overheads; prior work either assumes no overhead or analyzes a single system and its policy and overheads. Second, prior work evaluates metrics such as delay, throughput, and communication rate, but does not consider CPU efficiency. In contrast, we compare the tradeoffs that different policies make in terms of latency and efficiency, in the presence of load-balancing overheads.

**Core-allocation policies.** Existing systems adopt a variety of different policies for deciding when to reallocate cores, either across different applications or between cores available for applications and those designated for network processing or a file system. These approaches make decisions based on task arrivals [40], queueing delay [12, 20, 26, 52, 53, 60, 67], CPU utilization [12, 20, 35, 41, 67], or failure to find work [4, 7, 21, 27, 40, 76]. None of these systems compare different policies in the presence of the same overheads, so it is not possible to determine from these works which policies provide the best combination of latency and efficiency. Some past work points out that work-stealing cores can waste considerable CPU cycles, and proposes policies for yielding cores to mitigate this [4, 7, 21]. However, these policies target throughput and fairness for longer tasks (e.g., hundreds of microseconds or more); in contrast, our analysis focuses on which policies provide the best efficiency and latency for microsecond-scale tasks, and thus yields different conclusions.

**Implementing policies.** The systems Syrup [37] and ghOST [32] enable users to control scheduling policies in the kernel scheduler, network stack, and network card from code written in userspace. These systems are complementary to our work; they make it easier to express scheduling policies but do not specify which policies users should implement.

## 8 Conclusion

Numerous systems have been designed to support latency-sensitive datacenter applications while dynamically allocating cores to react to changes in load. However, these systems often come with a significant efficiency penalty with short tasks. In this paper, we systematically evaluated the effects of different policy choices on efficiency and latency to determine which load-balancing and core-allocating schemes achieve the best performance when considering realistic overheads. Work stealing is the definitive best policy option in today’s hardware while the core-allocation space is more complex. We designed and implemented two core-allocation policies which provide consistent and configurable performance on the Pareto frontier when paired with work stealing and demonstrated how they can improve efficiency without sacrificing latency.

## 9 Acknowledgments

We thank our shepherd Ana Klimovic, the anonymous reviewers, John Ousterhout, and the members of NetSys for their useful feedback. We thank Daniel Grier for assistance with the NP-hardness proof. This work was funded in part by NSF Grants 1817116 and 1704941, and by grants from Intel, VMware, Ericsson, Futurewei, and Cisco.

## References

- [1] Dpdk. <https://www.dpdk.org/>.
- [2] redis. <https://redis.io/>.
- [3] Rss. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>.
- [4] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–32, 2008.
- [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*, pages 63–74, 2010.
- [6] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.
- [7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.
- [8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. 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*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.
- [9] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736, 2009.
  - [10] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.
  - [11] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.
  - [12] 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. *ACM Transactions on Computer Systems (TOCS)*, 34(4):1–39, 2016.
  - [13] C. Bienia. *Benchmarking modern multiprocessors*. Princeton University, 2011.
  - [14] R. D. Blumofe and C. E. Leiserson. Scheduling multi-threaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
  - [15] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.
  - [16] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.
  - [17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.
  - [18] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of  $\mu$ s-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.
  - [19] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
  - [20] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 621–637, 2021.
  - [21] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. Bws: balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 365–378, 2012.
  - [22] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1):53–68, 1986.
  - [23] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.
  - [24] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi-and many-core memory systems through microbenchmarking. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.
  - [25] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramanian, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, 2017.
  - [26] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.
  - [27] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.
  - [28] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.
  - [29] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

- [30] T. Hellemans, T. Bodas, and B. Van Houdt. Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–35, 2019.
- [31] T. Hellemans and B. Van Houdt. On the power-of-d choices with least loaded server selection. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–22, 2018.
- [32] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 588–604, 2021.
- [33] J. Hwang, M. Vuppala, S. Peter, and R. Agarwal. Rearchitecting linux storage stack for  $\mu$ s latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.
- [34] S. Ibanez, A. Mallory, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256, 2021.
- [35] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.
- [36] K. Kaffles, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for  $\mu$ second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.
- [37] K. Kaffles, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 605–620, 2021.
- [38] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.
- [39] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM (JACM)*, 40(3):765–789, 1993.
- [40] M. Karsten and S. Barghi. User-level threading: Have your cake and eat it too. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–30, 2020.
- [41] A. Kaufmann, T. Stampler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.
- [42] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash  $\approx$  local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.
- [43] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.
- [44] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018.
- [45] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.
- [46] C. Lee and J. Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 149–154, 2019.
- [47] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.
- [48] D. Lemire. Code used on daniel lemire’s blog. <https://github.com/lemire/Code-used-on-Daniel-Lemire-s-blog/tree/master/2019/01/01>.
- [49] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.
- [50] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. 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*, pages 1–13, 2016.
- [51] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

- [52] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpacı-Dusseau, and R. H. Arpacı-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 819–835, 2021.
- [53] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.
- [54] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 212–221, 1998.
- [55] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.
- [56] M. Nandagopal, K. Gokulnath, and V. R. Uthariaraj. Sender initiated decentralized dynamic load balancing for multi cluster computational grid environment. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, pages 1–4. 2010.
- [57] D. Neill and A. Wierman. On the benefits of work stealing in shared-memory multiprocessors. *Department of Computer Science, Carnegie Mellon University, Tech. Rep*, 2009.
- [58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
- [59] OpenMP. Openmp application programming interface. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>, 2018.
- [60] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.
- [61] A. E. Ousterhout. *Achieving high CPU efficiency and low tail latency in datacenters*. PhD thesis, Massachusetts Institute of Technology, 2019.
- [62] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.
- [63] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.
- [64] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014.
- [65] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2021.
- [66] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.
- [67] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.
- [68] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 2007.
- [69] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.
- [70] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun. Elastic RSS: Co-scheduling packets and cores using programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.
- [71] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245, 1991.
- [72] H. Schweizer, M. Besta, and T. Hoefer. Evaluating the cost of atomic operations on modern architectures. <https://arxiv.org/pdf/2010.09852.pdf>.
- [73] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.
- [74] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The nebula rpc-optimized

- architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 199–212. IEEE, 2020.
- [75] The Caladan Authors. Caladan’s open-source release. <https://github.com/shenango/caladan>.
- [76] The Go Community. The go programming language. <https://golang.org>.
- [77] B. Van Houdt. Randomized work stealing versus sharing in large-scale systems with non-exponential job sizes. *IEEE/ACM Transactions on Networking*, 27(5):2137–2149, 2019.
- [78] R. V. Van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, 2001.
- [79] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.
- [80] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391, 2013.
- [81] S. Zhao, H. Gu, and A. J. Mashtizadeh. Skq: Event scheduling for optimizing tail latency in a traditional os kernel. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 759–772, 2021.
- [82] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, Nov. 2020.

## A Appendix

### A.1 Proof of NP-Hardness for Optimal Core Allocations

In this appendix, we prove that finding the optimal tail latency for a given CPU usage bound or vice versa is NP-hard, assuming a finite number of cores and non-constant service times. We show this using a reduction from the multiprocessor scheduling decision problem.

#### Multiprocessor Scheduling Problem [28]

*Input:* A non-zero number of cores  $c$ , a set of tasks  $T$  where each task  $t$  has a positive integer service time (or length)  $l(t)$ , and an overall deadline  $D$  for completing all tasks.

*Question:* Is there a schedule of the tasks  $T$  over the  $c$  cores that meets the overall deadline  $D$ ? Such a schedule assigns a start time to each task  $t$  such that there are never more than  $c$  tasks being handled simultaneously and for each task, its start time plus  $l(t)$  is at most  $D$ .

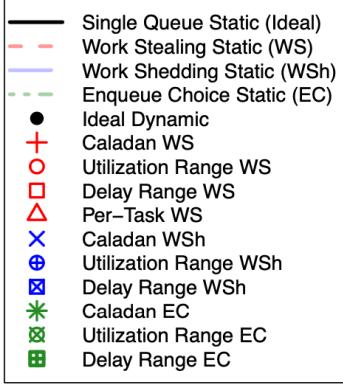
The Multiprocessor Scheduling Problem is NP-complete, assuming that all tasks do not have the same service time; with constant service times, this problem is trivial [28].

#### Optimal Core-Allocation Problem

*Input:* A non-zero number of cores  $c$  where each core can be either *on* or *off*, and transitioning from off to on requires a start-up time of  $S$ ; a set of tasks  $T$  where each task  $t$  has an arrival time  $a(t)$  and a positive integer service time  $l(t)$ ; the total “wasted” CPU time  $W$ , or time spent by cores while they are starting up or on but not handling a task; a tail latency percentile  $P < 1$  (e.g., 99.9<sup>th</sup> percentile); and a tail latency target  $L$ .

*Question:* Is there a schedule for the  $c$  cores and the tasks  $T$  such that tasks are only scheduled on cores that are *on*, the wasted CPU time is at most  $W$ , and the latency at percentile  $P$  is at most  $L$ ? A schedule for the cores assigns periods of *on* and *off* time to each, noting that it takes  $S$  time to transition from *off* to *on*. A schedule for the tasks assigns a start time to each task  $t$  such that the start time for  $t$  is at least  $a(t)$  and the number of tasks being handled simultaneously never exceeds the number of cores that are *on*. Finally, for  $P$  percent of the tasks, their start time plus  $l(t)$  is at most  $L$ .

We can reduce the multiprocessor scheduling problem to the optimal core allocation problem as follows. The number of cores in the core allocation problem matches that in the multiprocessor scheduling problem and we set  $L = D$ . We construct the set of tasks for the core allocation problem by replicating the tasks and their service times from the multiprocessor scheduling problem and setting them to all arrive at the beginning (i.e.,  $a(t) = 0$  for all  $t \in T$ ). In addition, we add additional dummy tasks with  $l(t) > D$  so that the tasks in the multiprocessor scheduling problem constitute  $P$  percent of the total tasks in the core allocation problem; because the dummy tasks cannot possibly meet the latency bound, the problem is only solvable by having all non-dummy tasks meet the latency bound. Finally we set the start-up time  $S$  to be zero and the



**Figure 9:** Performance of core-allocation and load-balancing policies from Figure 5 at additional loads.

wasted CPU time bound  $W$  to be high enough to be irrelevant (e.g.,  $W \geq c \cdot D$ ). We leave it as a simple exercise to show that there exists a polynomial time algorithm for such an instance of the optimal core allocation problem if and only if there is a polynomial time algorithm for the corresponding instance of the multiprocessor scheduling problem. In addition, the optimal core allocation problem is clearly in NP; thus it is NP-complete.

Because the optimal core allocation decision problem is NP-complete, the optimization problem of finding the optimal tail latency for a given efficiency bound or vice versa is NP-hard. This proof assumes that the service time distribution  $I(t)$  is not constant; the optimization problem with constant service times may also be NP-hard but this cannot be shown using the proof above.

## A.2 Extended Factor Analysis

In this appendix we include additional data omitted for space in the factor analysis.

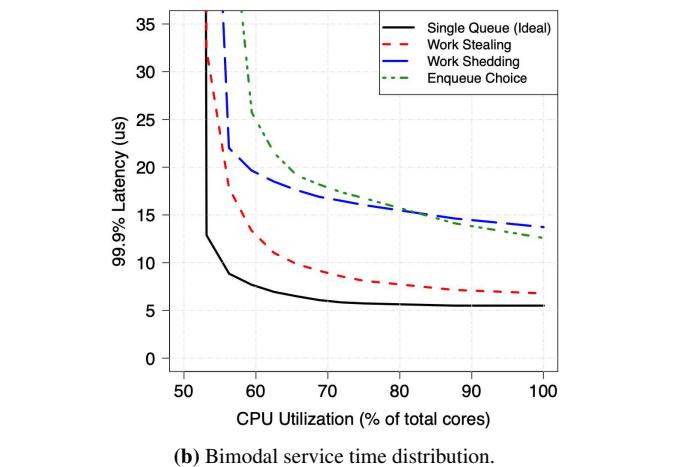
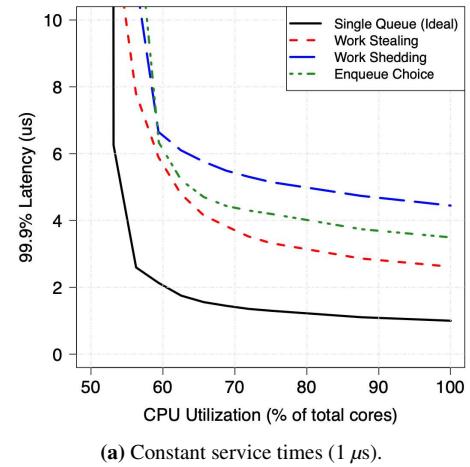
### A.2.1 Additional Loads for Static Curves

In Figure 5, we compared the performance of different load-balancing policies across different average service times to demonstrate that beating static allocations is more difficult with short tasks. The graphs look at both efficiency and latency simultaneously by keeping load constant. In Figure 9, we vary the offered load to 30% and 70%. To see a complete view of efficiency and latency (without static load-balancing curves for reference) across load, see Figure 7.

### A.2.2 Additional Service Time Distributions

We compared the load-balancing policies across different service time distributions. Specifically, we created static allocation performance curves for each load-balancing policy for both constant service times of  $1\ \mu s$  and a bimodal distribution with 500 ns service times for 90% of requests and  $5.5\ \mu s$  for the remaining 10% (average service time of  $1\ \mu s$ ). In Figure 10, we see that across these different service time distributions, work stealing consistently outperforms the other

load-balancing policies. Since load-balancing choices are less significant to end-to-end performance when service times are constant (Figure 10a), work stealing provides smaller benefits.



**Figure 10:** Performance of load-balancing policies with static core allocations for different service time distributions.

# A Case for Task Sampling based Learning for Cluster Job Scheduling

Akshay Jajoo\*

akshay.jajoo@nokia-bell-labs.com

Y. Charlie Hu

ychu@purdue.edu

Xiaojun Lin

linx@purdue.edu

Nan Deng

dengnan@google.com

## Abstract

The ability to accurately estimate job runtime properties allows a scheduler to effectively schedule jobs. State-of-the-art online cluster job schedulers use history-based learning, which uses past job execution information to estimate the runtime properties of newly arrived jobs. However, with fast-paced development in cluster technology (in both hardware and software) and changing user inputs, job runtime properties can change over time, which lead to inaccurate predictions.

In this paper, we explore the potential and limitation of real-time learning of job runtime properties, by proactively sampling and scheduling a small fraction of the tasks of each job. Such a task-sampling-based approach exploits the similarity among runtime properties of the tasks of the same job and is inherently immune to changing job behavior. Our analytical and experimental analysis of 3 production traces with different skew and job distribution shows that learning in space can be substantially more accurate. Our simulation and testbed evaluation on Azure of the two learning approaches anchored in a generic job scheduler using 3 production cluster job traces shows that despite its online overhead, learning in space reduces the average Job Completion Time (JCT) by  $1.28\times$ ,  $1.56\times$ , and  $1.32\times$  compared to the prior-art history-based predictor. Finally, we show how sampling-based learning can be extended to schedule DAG jobs and achieve similar speedups over the prior-art history-based predictor.

## 1 Introduction

In big-data compute clusters, jobs arrive online and compete to share the cluster resources. In order to best utilize the cluster and to ensure that jobs also meet their service level objectives, efficient scheduling is essential. However, as jobs arrive online, their runtime characteristics are not known *a priori*. Due to this lack of information, it is challenging for the cluster scheduler to determine the right job execution order that optimizes scheduling metrics such as maximal resource utilization or application service level objectives.

An effective way to tackle the challenges of cluster scheduling is to learn the runtime characteristics of pending jobs, which allows the scheduler to exploit offline scheduling algorithms that are known to be optimal, *e.g.*, Shortest Job First (SJF) for minimizing the average completion time. Indeed, there has been a large amount of work [27, 36, 43, 44, 47, 49,

52, 55] on learning job runtime characteristics to facilitate cluster job scheduling.

In essence, all of the previous learning algorithms learn job runtime characteristics from observing historical executions of the same jobs, which execute the same code but process different sets of data, or of similar jobs, which have matching features such as the same application name, the same job name, or the same user who submitted the job.

The effectiveness of the above *history-based* learning schemes critically rely on two conditions to hold true: (1) The jobs are recurring; (2) The performance of the same or similar jobs will remain consistent over time.

In practice, however, the two conditions often do not hold true. First, many previous work have acknowledged that not all jobs are recurrent. For example, in the traces used in Corral [43] and Jockey [30], only 40% of the jobs are recurrent, and Morpheus [44] shows that only 60% of the jobs are recurrent. Second, even the authors of history-based prediction schemes such as 3Sigma [47] and Morpheus [44] strongly argued why runtime properties of jobs, even with the same input, will not remain consistent and will keep evolving. The primary reason is due to updates in cluster hardware, application software, and user scripts to execute the cluster jobs. Third, our own analysis of three production cluster traces (§4) have also shown that historical job runtime characteristics have considerable variations.

In this paper, we explore an alternative approach to learning runtime properties of distributed jobs online to facilitate cluster job scheduling. The approach is motivated by the following key observations about distributed jobs running on shared clusters: (1) a job typically has a *spatial dimension*, *i.e.*, it typically consists of many tasks; and (2) the tasks (in the same phase) of a job typically execute the same code and process different chunks of similarly sized data [9, 16]. These observations suggest that if the scheduler first schedules a few sampled tasks of a job, known as pilot tasks, to run till finish, it can use the observed runtime properties of those tasks to accurately estimate those of the whole job. Effectively, such a *task-sampling-based* approach learns job properties in the spatial dimension. We denote the new learning scheme as SLEARN, for “learning in space”.

Intuitively, by using the execution of pilot tasks to predict the properties of other tasks, SLEARN avoids the primary drawback of history-based learning techniques, *i.e.*, relying on jobs to be recurring and job properties to remain stationary over time. However, learning in space introduces two new challenges: (1) its estimation accuracy can be affected

\*The work was done while the author was pursuing his Ph.D. at Purdue University.

by the variations of task runtime properties, *i.e.*, task skew; (2) delaying scheduling the remaining tasks of a job till the completion of sampled tasks may potentially hurt the job’s completion time.

In this paper, we perform a comprehensive comparative study of history-based learning (learning in time) and sampling-based learning (learning in space), to systematically answer the following questions: (1) *Can learning in space be more accurate than learning in time?* (2) *If so, can delaying scheduling the remaining tasks of a job till the completion of sampled tasks be more than compensated by the improved accuracy and result in improved job performance, e.g., completion time?*

We answer the first question via quantitative analysis, and trace and experimental analysis based on three production job traces, including two public cluster traces from Google released in 2011 and 2019 [8, 11] and a private trace from 2Sigma [1]. We answer the second question by designing a generic scheduler that schedules jobs based on job runtime estimates to optimize a given performance metric, *e.g.*, average job completion time (JCT), and then plug into the scheduler different prediction schemes, in particular, learning in time and learning in space, to compare their effectiveness.

We summarize the major findings and contributions of this paper as follows:

- Based on literature survey and analysis using three production cluster traces, we show that history is not a stable and accurate predictor for runtime characteristics of distributed jobs.
- We propose SLEARN, a novel learning approach that uses sampling in the spatial dimension of jobs to learn job runtime properties online. We also provide solutions to practical issues such as dealing with thin jobs (jobs with a few tasks only) and work conservation.
- Via quantitative, trace and experimental analysis, we demonstrate that SLEARN can predict job runtime properties with much higher accuracy than history-based schemes. For the 2Sigma, Google 2011, and Google 2019 cluster traces, the median prediction error are 18.98%, 13.68%, and 51.84% for SLEARN but 36.57%, 21.39%, and 71.56% for the state-of-the-art history-based 3Sigma, respectively.
- We show that learning job runtime properties by sampling job tasks, although delays scheduling the remaining tasks of a job, can be more than compensated by the improved accuracy, and as a result reduces the average JCT. In particular, our extensive simulations and testbed experiments using a prototype on a 150-node cluster in Microsoft Azure show that compared to the prior-art history-based predictor, SLEARN reduces the average JCT by  $1.28\times$ ,  $1.56\times$ , and  $1.32\times$  for the extracted 2Sigma, Google 2011 and Google 2019 traces, respectively.

- We show how the sampling-based learning can be extended to schedule DAG jobs. Using a DAG trace generated from the Google 2019 trace, we show a hybrid sampling-based and history-based scheme reduces the average JCT by  $1.25\times$  over a pure history-based scheme.

## 2 Background and Related Work

In this section, we provide a brief background on the cluster scheduling problem, review existing learning-based schedulers, and discuss their weaknesses.

### 2.1 Cluster Scheduling Problem

In both public and private clouds, clusters are typically shared among multiple users to execute diverse jobs. Such jobs typically arrive online and compete for shared resources. In order to best utilize the cluster and to ensure that jobs also meet their service level objectives (SLOs), efficient job scheduling is essential. Since jobs arrive online, their runtime characteristics are not known *a priori*. This lack of information makes it challenging for the scheduler to determine the right order for running the jobs that maximizes resource utilization and/or meets application SLOs. Additionally, jobs have different SLOs. For some meeting deadlines is important while for others faster completion or minimizing the use of networks is more important. Such a diverse set of objectives pose further challenges to effective job scheduling [19, 30, 31, 43, 44, 55, 56].

### 2.2 Job Model

We consider big-data compute clusters running data-parallel frameworks such as Hadoop [4], Hive [6], Dryad [37], Scope [22], and Spark [7] that run simple MapReduce jobs [28] or more complex DAG-structured jobs, where each job processes a large amount of data. Each job consists of one or multiple stages, such as map or reduce, and each stage partitions the data into manageable chunks and runs many parallel tasks, each for processing one data chunk.

### 2.3 Existing Learning-based Schedulers

An effective way to tackle the challenges of cluster scheduling is to learn runtime characteristics of pending jobs. As such cluster schedulers using various learning methods have been proposed [19, 21, 25, 36, 43–45, 47, 49, 50, 52]. In essence, all previous learning schemes are *history-based*, *i.e.*, they learn job characteristics by observations made from the past job executions.<sup>1</sup> In particular, existing learning approaches can be broadly categorized into the following groups, as summarized in Table 1.

**Learning offline models.** Corral’s prediction model is designed with the primary assumptions that most jobs are

---

<sup>1</sup>Some recent work use the characteristics of completed mini-batches as a proxy for the remaining mini-batches, to improve the scheduling of ML jobs [54]. However, such jobs are different in that the mini-batches in general experience significantly less (task-level) variations than what we studied in this paper.

Table 1: Summary of selected previous work that use history-based learning techniques.

Name	Property estimated	Estimation technique	Learning frequency
Corral [43]	Job runtime	Offline model (not updated)	On arrival
DCOSR [36]	Memory elasticity profile	Offline model (not updated)	Scheduler dependent
Jockey [30]	Job runtime	Offline simulator	Periodic
3Sigma [47]	Job runtime history dist.	Offline model	On arrival

recurring in nature, and the latency of each stage of a multi-stage job is proportional to the amount of data processed by it, which do not always hold true [43].

DCOSR [36] predicts the memory usage for data parallel compute jobs using an offline model built from a fixed number of profile runs that are specific to the framework and depend on the framework’s properties. Any software update in the existing frameworks, addition of new framework or hardware update will require an update in profile.

For analytics jobs that perform the same computation periodically on different sets of data, Tetris [32] takes measurements from past executions of a job to estimate the requirements for the current execution.

**Learning offline models with periodic updates.** Jockey [30] periodically characterizes job progress at runtime, which along with a job’s current resource allocation is used by an offline simulator to estimate the job’s completion time and update the job’s resource allocation. Jockey relies on job recurrences and cannot work with new jobs.

**Learning from similar jobs.** Instead of using execution history from the exact same jobs, JVUPredict [51] matches jobs on the basis of some common features such as application name, job name, the user who owns the job, and the resource requested by the job. 3Sigma [47] extends JVUPredict [51] by introducing a new idea on prediction: instead of using point metrics to predict runtimes, it uses full distributions of relevant runtime histories. However, since it is impractical to maintain precise distributions for each feature value, it resorts to approximating distributions, which compromises the benefits of having full distributions.

## 2.4 Learning from History: Assumptions and Reality

Predicting job runtime characteristics from history information relies on the following two conditions to hold, which we argue may not be applicable to modern day clusters.

**Condition 1: The jobs are recurring.** Many previous works have acknowledged that not all jobs are recurrent. For example, the traces used in Corral [43] and Jockey [30] show that only 40% of the jobs are recurrent and Morpheus [44] shows that 60% of the jobs are recurrent.

**Condition 2: The performance of the same or similar jobs will remain consistent over time.** Previous works [30, 43, 44, 47] that exploited history-based prediction have considered jobs in one of the following two categories. (1) *Recurring jobs*: A job is re-scheduled to run on newly arriving data; (2) *Similar jobs*: A job has not been seen before but has some attributes in common with some jobs executed in the past [47, 51]. Many of the history-based approaches only predict for recurring jobs [30, 43, 44], while some others [25, 45, 47, 51] work for both categories of jobs.

However, even the authors of history-based prediction schemes such as 3Sigma [47] and Morpheus [44] strongly argued why runtime properties of jobs, even with the same input, will keep evolving. The primary reason is that updates in cluster hardware, application software, and user scripts to execute the cluster jobs affect the job runtime characteristics. They found that in a large Microsoft production cluster, within a one-month period, applications corresponding to more than 50% of the recurring jobs were updated. The source code changed by at least 10% for applications corresponding to 15-20% of the jobs. Additionally, over a one-year period, the proportion of two different types of machines in the cluster changed from 80/20 to 55/45. For a same production Spark job, there is a 40% difference between the running time observed on the two types of machines [44].

For these reasons, although the state-of-the-art history-based system 3Sigma [47] uses sophisticated prediction techniques, the predicted running time for more than 23% of the jobs have at least 100% error, and for many the prediction is off by an order of magnitude.

## 3 SLEARN – Learning in Space

In this paper, we explore an alternative approach to learning job runtime properties online in order to facilitate cluster job scheduling. The approach is motivated by the following key observations about distributed jobs running in shared clusters: (1) a distributed job has a spatial dimension, *i.e.*, it typically consists of many tasks; (2) all the tasks in the same phase of a job typically execute the same code with the same settings [9, 12, 16], and differ in that they process different chunks of similarly sized data. Hence, it is likely that their runtime behavior will be statistically similar.

The above observations suggest that if the scheduler first schedules a few sampled tasks of a job to run till finish, it can use the observed runtime properties of those tasks to accurately estimate those of the whole job. In a modular design, such an online learning scheme can be decoupled from the cluster scheduler. In particular, upon a job arrival, the predictor first schedules sampled tasks of the job, called *pilot tasks*, till their completion, to learn the job runtime properties. The learned job properties are then fed into the cluster job scheduler, which can employ different scheduling policies to meet respective SLOs. Effectively, the new scheme learns job properties in the spatial dimension, *i.e., learning in*

Table 2: Comparison of learning in time and learning in space of job runtime properties.

	Applicability	Adaptiveness	Accuracy	Runtime overhead
Time	Recurring jobs	No/Yes	Depends	No
Space	New/Recurring jobs	Yes	Depends	Yes

space. We denote the new learning scheme as SLEARN.

Table 2 summarizes the pros and cons of the two learning approaches along four dimensions: (1) **Applicability**: As discussed in §2.3, most history-based predictors cannot be used for the jobs of a new category or for categories for which the jobs are rarely executed. In contrast, learning in space has no such limitation; it can be applied to any new job. (2) **Adaptiveness to change**: Further, history-based predictors assume job runtime properties persist over time, which often does not hold, as discussed in §2.4. (3) **Accuracy**: The accuracy of the two approaches are directly affected by how they learn, *i.e.*, in space versus in time. The accuracy of history-based approaches is affected by how stable the job runtime properties persist over time, while that of sampling-based approach is affected by the variation of the task runtime properties, *i.e.*, the extent of task skew. (4) **Runtime overhead**: The history-based approach has an inherent advantage of having very low to zero runtime overhead. It performs offline analysis of historical data to generate a prediction model. In contrast, sampling-based predictors do not have offline cost, but need to first run a few pilot tasks till completion before scheduling the remaining tasks. This may potentially delay the execution of non-sampled tasks.

The above qualitative comparison of the two learning approaches raises the following two questions: (1) *Can learning in space be more accurate than learning in time?* (2) *If so, can delaying scheduling the remaining tasks of a job till the completion of sampled tasks be more than compensated by the improved accuracy, so that the overall job performance, e.g., completion time, is improved?* We answer the first question via analytical, trace and experimental analysis in §4 and the second question via a case study of cluster job scheduling using the two types of predictors in §5.

## 4 Accuracy Analysis

In this section, we perform an in-depth study of the prediction accuracy of the two learning approaches: *learning in time* (history-based learning) and *learning in space* (task-sampling-based learning). Both approaches can potentially be used to learn different job properties for different optimization objectives. In this paper, we focus on job completion time because it is an important metric that has been intensively studied in recent work [23, 24, 29, 33, 35, 36, 43, 47].

### 4.1 Analytical Comparison

We first present a theoretical analysis of the prediction accuracies of the two approaches. We caution that here we use a highly-stylized model (e.g., two jobs and normal task-length

distributions), which does not capture the possible complexity in real clusters, such as heavy parallelism across servers and highly-skewed task-length distributions. Nonetheless, it reveals important insights that help us understand in which regimes history-based schemes or sampling-based schemes will perform better. Consider a simple case of two jobs  $j_1$  and  $j_2$ , where each job has  $n$  tasks. The size of each task of  $j_1$  is known. Without loss of generality, let us assume that the task size of  $j_1$  is 1. Thus, the total size of  $j_1$  is  $n$ . The size of a task of  $j_2$  is however unknown. Let  $x$  denote the average task size of  $j_2$ , and this its total size is  $nx$ . Clearly, if we knew  $x$  precisely, then we should have scheduled  $j_1$  first if  $x > 1$  and  $j_2$  first if  $x \leq 1$ . However, suppose that we only know the following: (1) (Prior distribution:)  $x$  follows a normal distribution with mean  $\mu$  and variance  $\sigma_o^2$ ; (2) Given  $x$ , the size of a random task of the job follows a normal distribution with mean  $x$  and variance  $\sigma_1^2$ . Intuitively,  $\sigma_o^2$  captures the variation of mean task-lengths *across* many *i.i.d.* copies of job  $j_2$ , *i.e.*, job-wise variation, while  $\sigma_1^2$  captures the variation of task-lengths *within* a single run of job  $j_2$ , *i.e.*, task-wise variation. We note that the parameters  $\sigma_o^2$  and  $\sigma_1^2$  are *not* used by the predictors below.

Now, consider two options for estimating the mean task-length  $x$ : (1) A history-based approach (§4.1.1) and (2) a sampling-based approach where we sample  $m$  tasks from  $j_2$  (§4.1.2).

#### 4.1.1 History-based Schemes

Since no samples of job  $j_2$  are used, the best predictor for its mean task length is  $\mu$ . In other words, the scheduling decision will be based on  $\mu$  only. The difference between the true mean task length,  $x$ , and  $\mu$  is simply captured by the job-wise variance  $\sigma_o^2$ .

#### 4.1.2 Sampling-based Schemes

Suppose that we sample  $m$  tasks from  $j_2$ . Collect the sampled task lengths into a vector:

$$\vec{y} = (y_1, y_2, \dots, y_m).$$

Then, based on our probabilistic model, we have

$$P(y_i|x) = \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(y_i-x)^2}{2\sigma_1^2}}, \quad P(\vec{y}|x) = \prod_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(y_i-x)^2}{2\sigma_1^2}}$$

We are interested in an estimator of  $x$  given  $\vec{y}$ . We have

$$\begin{aligned} P(x|\vec{y}) &= \frac{P(\vec{y}|x) \cdot P(x)}{P(\vec{y})} = \frac{P(\vec{y}|x) \cdot P(x)}{\int_x P(\vec{y}|x) \cdot P(x) dx} \\ &= \frac{1}{\sqrt{2\pi}} \left[ \frac{m}{\sigma_1^2} + \frac{1}{\sigma_o^2} \right]^{\frac{1}{2}} \cdot e^{-\left( \frac{m}{2\sigma_1^2} + \frac{1}{2\sigma_o^2} \right) \left( x - \frac{\sum_{i=1}^m \frac{1}{\sigma_1^2} y_i + \frac{1}{\sigma_o^2} \mu}{\frac{m}{\sigma_1^2} + \frac{1}{\sigma_o^2}} \right)}, \end{aligned}$$

where the last step follows from standard results on the posterior distribution with Gaussian priors (see, *e.g.*, [18]). In other words, conditioned on  $\vec{y}$ ,  $x$  also follows a normal distribution

$$\text{with mean} = \frac{\sum_{i=1}^m \frac{1}{\sigma_1^2} y_i + \frac{1}{\sigma_o^2} \mu}{\frac{m}{\sigma_1^2} + \frac{1}{\sigma_o^2}} \text{ and variance} = \frac{1}{\frac{m}{\sigma_1^2} + \frac{1}{\sigma_o^2}}.$$

Table 3: Summary of trace properties.

Trace	Arrival time	Resource requested	Resource usage	Indiv. task duration
2Sigma	Yes	Yes	No	Yes
Google 2011	Yes	Yes	Yes	Yes
Google 2019	Yes	Yes	Yes	Yes

Note that this represents the estimator quality using the information of both job-wise variations and task-wise variations. If the estimator is not informed of the job-wise variations, we can take  $\sigma_o^2 \rightarrow +\infty$ , and the conditional distribution of  $x$  given  $\bar{y}$  becomes normal with mean  $\frac{1}{m} \sum_{i=1}^m y_i$  and variance  $\frac{\sigma_1^2}{m}$ .

From here we can draw the following conclusions. First, whether history-based schemes or sampling-based schemes have better prediction accuracy for an unknown job depends on the relationship between job-wise variations  $\sigma_o^2$  and the task-wise variation  $\sigma_1^2$ . If the job-wise variation is large but the task-wise variation is small, i.e.,  $\sigma_o^2 >> \frac{\sigma_1^2}{m}$ , then sampling-based schemes will have better prediction accuracy. Conversely, if the job-wise variation is small but the task-wise variation is large, i.e.,  $\sigma_o^2 << \frac{\sigma_1^2}{m}$ , then history-based schemes will have better prediction accuracy. Second, while the accuracy of history-based schemes is fixed at  $\sigma_o^2$ , the accuracy of sampling-based schemes improves as  $m$  increases. Thus, when we can afford the overhead of more samples, the sampling-based schemes become favorable. Our results from experimental data below will further confirm these intuitions.

## 4.2 Trace-based Variability Analysis

Our theoretical analysis in §4.1 provides insights on how the prediction accuracies of the two approaches depend on the variation of job run times across time and space. To understand how such variations fare against each other in practice, we next measure the actual variations in three production cluster traces. Table 3 summarizes the information available in the traces that are used in our analysis.

**Traces.** Our first trace is provided by 2Sigma [1]. The cluster uses an internal proprietary job scheduler running on top of a Mesos cluster manager [2]. This trace was collected over a period of 7 months, from January to July 2016, and from 441 machines and contains approximately 0.4 million jobs [17].

We also include two publicly available traces from Google released in May 2011 and May 2019 [8, 11], collected from 1 and 8 Borg [53] cells over periods of 29 and 31 days, respectively. The machines in the clusters are highly heterogeneous, belonging to at least three different platforms that use different micro-architectures and/or memory technologies [20]. Further, according to [9], the machines in the same platform can have substantially different clock rates, memory speed, and core counts. Since the original Google 2019 trace has data from 8 different cells located in 8 different locations,

and given that we already have two other traces from the US, we chose the batch tier of Cluster G in the Google 2019 trace, which is located in Singapore [12], as our third trace to diversify our trace collection.

We calculate the variations in task runtimes for each job across time and across space as follows.

**Variation across time.** To measure the variation in mean task runtime for a job across the history, we follow the following prediction mechanism defined in 3Sigma [47] to find similar jobs.

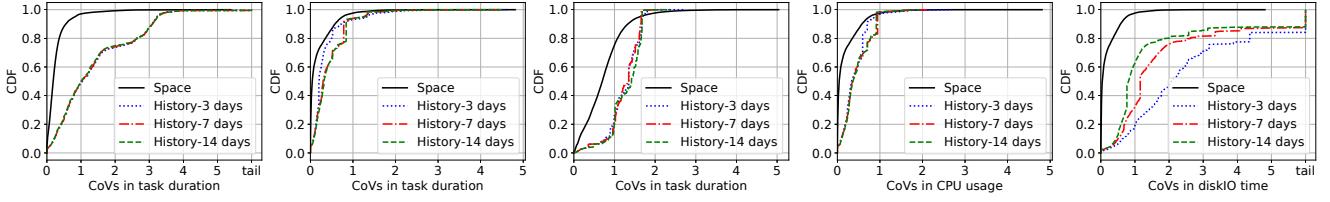
As discussed in §2.3, 3Sigma [47] uses multiple features to identify a job and predicts its runtime using the feature that gives the least prediction error in the past. We include all six features used in 3Sigma: application name, job name, user name (the owner of the job), job submission time (day and hour), and resources requested (cpu and memory) by the job.

For each feature, we define the set of similar jobs as all the jobs executed in the history window (defined below) that had the same feature value. Next, we calculate the average task runtime of each job in the set. Then, we calculate the *Coefficient of Variation* (CoV) of the average task runtimes across all the jobs in the set. We repeat the above process for all the features. We then compare the CoV values thus calculated and pick the minimum CoV. Effectively, the above procedure selects the least possible variation across history.

**Varying the history length in prediction across time.** 3Sigma used the entire history for prediction. Intuitively, the length of the history affects the trade-off between the number of similar jobs and the staleness of the history information. For this reason, we optimized 3Sigma by finding and using the history length that gives the least variation. Specifically, we define the length of history based on a window size  $w$ , i.e., the number of past consecutive days. In our analysis below, we vary  $w$  among 3, 7, and 14 for the three traces.

**Variation across space.** To measure the extent of variation across space, we look at the CoV ( $\text{CoV} = \frac{\sigma}{\mu}$ ) in the task runtimes within a job. As shown in §4.1, the variance in the task runtime predicted from sampling is  $\frac{\sigma_1^2}{m}$ , where  $\sigma_1^2$  is the variance in the runtimes across all the tasks within the job and  $m$  is the number of tasks sampled. Thus, we first estimate  $\sigma_1^2$  from all tasks within the job. We then report the CoV of our task runtime prediction after sampling  $m$  tasks as  $\frac{\sigma_1/\sqrt{m}}{\mu}$ . Our complete scheduler design in §5.1 uses an adaptive sampling algorithm which mostly uses 3% for the three traces. Thus, for measuring the extent of variation across space here, we assume a 3% sampling ratio and plot  $\frac{\sigma_1}{(\sqrt{0.03 \times \text{numberOfTasksInJob}}) \times \mu}$ .

**Variability comparison.** For consistency, all analysis results here are for the same, shortest trace period that can be used for sliding-window-history based analysis, e.g., the last 15 days under the 14-day window for the 29-day Google 2011 trace. (The analysis then varies the length of the sliding window in history-based learning.)



(a) Task runtime – 2Sigma (b) Task runtime – Google 11 (c) Task runtime – Google 19 (d) CPU usage – Google 11 (e) Disk IO time – Google 11  
 Figure 1: CDF of CoV of runtime properties across space and across time with varying history windows, using the 2Sigma, Google 2011 and Google 2019 traces. Single-task jobs are excluded from the analysis across space.

Table 4: CoV in task runtime across time and across space for the the 2Sigma, Google 2011, and Google 2019 traces.

Trace	CoV over Time		CoV over Space	
	P50	P90	P50	P90
2Sigma	1.00	3.10	0.18	0.55
Google 2011	0.20	0.73	0.04	0.58
Google 2019	1.35	1.67	0.70	1.33

Fig. 1(a)–Fig. 1(c) show the CDFs of CoVs in task duration measured across space and across history for multiple history window sizes for the three traces. We see that in general using a shorter sliding window reduces the prediction error of 3Sigma, and the CoVs across tasks are moderately lower than the CoVs across history for the Google 2011 trace but significantly lower for 2Sigma and Google 2019 traces. For example, for the 2Sigma trace, the CoV across history is higher than the CoV across tasks for 85.40% of the jobs (not seen in Fig. 1(a) as jobs are ordered differently in different CDFs) and for more than 30% of the jobs, the CoV across history is at least 12.10× higher than the CoV across tasks.

Table 4 summarizes the results, where the CoVs across time correspond to the best history window size, *i.e.*, 3 days for both Google traces and 14 days for the 2Sigma trace. As shown in the table, the P50 (P90) CoV across history are 1.00 (3.10) for the 2Sigma trace, 0.20 (0.73) for the Google 2011 trace, and 1.35 (1.67) for the Google 2019 trace. In contrast, the P50 (P90) CoV value across the task duration of the same set of jobs is much lower, 0.18 (0.55) for the 2Sigma trace, 0.04 (0.58) for the Google 2011 trace, and 0.70 (1.33) for the Google 2019 trace.

Fig. 1(d) and Fig. 1(e) further show the CDF of CoVs for CPU usage and Disk IO time for the Google 2011 trace (such resource usage is not available in the 2Sigma trace). The figures show that the variation in the values of these properties when sampled across space is also considerably lower compared to the variation observed over time.

### 4.3 Experimental Prediction Error Analysis

Recall from our analysis in §4.1 that lower task-wise variation than job-wise variation (§4.2) will translate into better prediction accuracy of sampling-based schemes over history-based schemes. While our analysis in §4.1 assumes normal distribution, we believe that a similar conclusion will hold

in more general settings. To validate this, we next implement a sampling-based predictor SLEARN, and experimentally compare it against a state-of-the-art history-based predictor 3Sigma [47] in estimating the job runtimes directly on production job traces.

**Workload characteristics.** Since the three production traces described in §4.2 are too large, as in 3Sigma [47], we extracted smaller traces for experiments using the procedure described below.

Since the history-based predictor 3Sigma needs a history trace, we followed the same process as in [47] to extract the training trace for 3Sigma and the execution trace for all predictors, in three steps. (1) We divided each original trace in chronological order in two halves. (2) We compressed 2Sigma jobs to 150 tasks or fewer, by applying a compression ratio of original cluster size/150. Since the Google traces do not have many wide jobs yet the original clusters are very wide, with 12.5K machines, we dropped jobs with more than 150 tasks<sup>2</sup>. (3) We next selected the execution trace following the process below from the second half; these became 2STrace, GTrace11 and GTrace19, respectively. (4) We then selected jobs from the first half of each original trace that are feature-clustered with those jobs in the execution trace to form the "history" trace for 3Sigma.

We extracted the execution trace from each of the above-mentioned second halves by randomly selecting 1250 jobs with equal probability. Then, for each extracted trace, we adjust the arrival time of the jobs so that the average cluster load matches that in the original trace [8, 11, 17]. Table 5 summarizes the workload per window of the extracted traces, where a window is defined as a 1000-second interval sliding by 100 seconds at a time, and the load per window is the total runtime of all the jobs arrived in that window, normalized by the total number of CPUs in the cluster times the window length, *i.e.*, 1000s. We see that for all three traces, the average system load is close to 1, though the load fluctuates over time, which is preserved by the random uniform job extraction.

**Prediction mechanisms and experimental setups.** We implement the 3Sigma predictor following its description

<sup>2</sup>This is to avoid potential bias towards SLEARN. A job with more than 150 tasks will have to be scheduled in more than one phase, which will be in favor of SLEARN by diminishing the sampling overhead.

Table 5: Statistics for system load per 1000s sliding window.

Trace	Average	P50	P90
2STrace	1.05	0.13	2.47
GTrace11	1.01	0.29	1.49
GTrace19	1.04	0.09	0.91

in [47]. After learning the job runtime distribution (§4.2), it uses a utility function of the estimated job runtime associated with every job to derive its estimated runtime from the distribution, by integrating the utility function over the entire runtime distribution. Since our goal is to minimize the average JCT, we used a utility function that is inversely proportional to the square of runtime. We kept all the default settings we learned from the authors of 3Sigma [47].

As in §4.2, SLEARN samples  $\max(1, 0.03 \cdot S)$  tasks per job, where  $S$  is the number of tasks in the job. We only show the results for wide jobs (with 3 or more tasks) as in the complete SLEARN design (§5.1.1), only wide jobs go through the sampling phase.

**Results.** Fig. 2 shows the CDF of percentage error in the predicted job runtimes for the three traces. We see that SLEARN has much better prediction accuracy than 3Sigma. For 2STrace, GTrace11, and GTrace19, the P50 prediction error are 18.30%, 9.15%, 21.39% for SLEARN but 36.57%, 21.39%, 71.56% for 3Sigma, respectively, and the P90 prediction error are 58.66%, 49.95%, 92.25% for SLEARN but 475.78%, 294.52%, 1927.51% for 3Sigma, respectively.

## 5 Integrating Sampling-based Learning with Job Scheduling: A Case Study

In this section, we answer the second key question about the sampling-based learning: Can delaying scheduling the remaining tasks till completing the sampled tasks be compensated by the improved prediction accuracy? We answer it through extensive simulation and testbed experiments.

Our approach is to design a generic scheduler, denoted as GS, that schedules jobs based on job runtime estimates to optimize a given performance metric, average job completion time (JCT). We then plug into GS different prediction schemes to compare their end-to-end performance.

### 5.1 Scheduler and Predictor Design

#### 5.1.1 Generic Scheduler GS

GS replaces the scheduling component of a cluster manager like YARN [5]. The key scheduling objective of GS is to minimize the average JCT. Additionally, GS aims to avoid starvation.

The scheduling task in GS is divided into two phases, (1) job runtime estimation, and (2) efficient and starvation-free scheduling of jobs whose runtimes have been estimated. We focus here on the scheduling mechanism and discuss the different job runtime estimators in the following sections.

**Inter-job scheduling.** Shortest job first (SJF) is known to be optimal in minimizing the average JCT when job execution depends on a single resource. Previous work has shown that scheduling distributed jobs even with prior knowledge is NP-hard (e.g., [24]), and an effective online heuristic is to order the distributed jobs based on each job’s total size [23, 39–41]. In GS we use a similar heuristic; the jobs are ordered based on their total estimated runtime, i.e.,  $\text{mean task runtime} \times \text{number of tasks}$ .

**Starvation avoidance.** SJF is known to cause starvation to long jobs. Hence, in GS we adopt a well-known multi-level priority queue structure to avoid job starvation [23, 26, 38, 46, 48]. Once GS receives the runtime estimates of a job, it assigns the job to a priority queue based on its runtime. Within a queue, we use FIFO to schedule jobs. Across the queues, we use weighted sharing of resources, where a priority queue receives a resource share according to its priority.

In particular, GS uses  $N$  queues,  $Q_0$  to  $Q_{N-1}$ , with each queue having a lower queue threshold  $Q_q^{lo}$  and a higher threshold  $Q_q^{hi}$  for job runtimes. We set  $Q_0^{lo} = 0$ ,  $Q_{N-1}^{hi} = \infty$ ,  $Q_{q+1}^{lo} = Q_q^{hi}$ . A queue with a lower index has a higher priority. GS uses exponentially growing queue thresholds, i.e.,  $Q_{q+1}^{hi} = E \cdot Q_q^{hi}$ . To avoid any bias, we use the multiple priority queue structure with the same configuration when comparing different job runtime estimators.

**Basic scheduling operation.** GS keeps track of resources being used by each priority queue. It offers the next available resource to a queue such that the weighted sharing of resources among the queues for starvation avoidance is maintained. Resources offered to a queue are always offered to the job at the head of the queue.

#### 5.1.2 SLEARN

To seamlessly integrate SLEARN with GS, we need to use one of the priority queues for scheduling sampled tasks. We denote it as the sampling queue.

**Fast sampling.** One design challenge is how to determine the priority for the sampling queue w.r.t. the other priority queues. On one hand, sampled tasks should be given high priority so that the job runtime estimation can finish quickly. On the other hand, the jobs whose runtimes have already been estimated should not be further delayed by learning new jobs. To balance the two factors, we use the second highest priority in GS as the sampling queue.

**Handling thin jobs.** Recall that in SLEARN, when a new job arrives, SLEARN only schedules its pilot tasks, and delays other tasks until the pilot tasks finish and the job runtime is estimated. Such a design choice can inadvertently lead to higher JCTs for thin jobs, e.g., a two-task job would experience serialization of its two tasks. To avoid JCT degradations for thin jobs, we place a job directly in the highest priority queue if its width is under a threshold `thinLimit`.

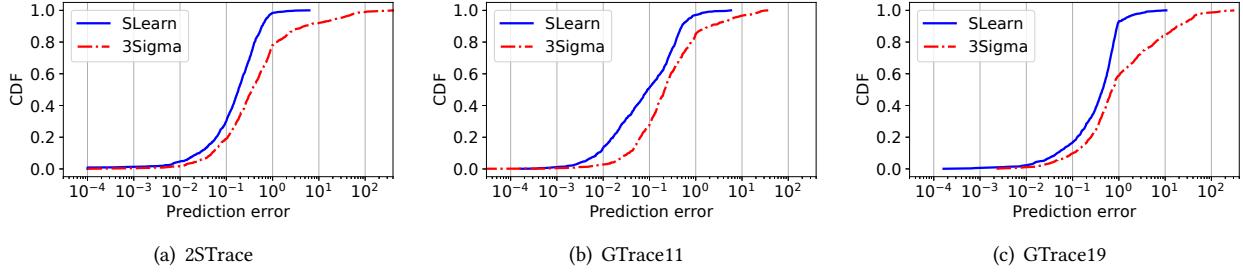


Figure 2: Job runtime prediction accuracy.

**Basic operations.** Upon the arrival of a new job, the cluster manager asynchronously communicates the job’s information to GS, which relays the information to SLEARN. If the number of tasks in the job is under `thinLimit`, SLEARN assigns it to the highest priority queue; otherwise, the job is assigned to the sampling queue, where a subset of its tasks (*pilot tasks*) will be scheduled to run. Once a job’s runtime is estimated from sampling, it is placed in the priority queue corresponding to its runtime estimate where the rest of its tasks will be scheduled.

**How many and which pilot tasks to schedule?** When a new job arrives, SLEARN first needs to determine the number of pilot tasks. Sampling more tasks can give higher estimation accuracy, but also consumes more resources early on, which can potentially delay other jobs, if the job turns out to be a long job and should have been scheduled to run later under SJF. Further, we found the best sampling ratio appears to vary across difference traces. To balance the trade-off, we use an adaptive algorithm to dynamically determine the sampling ratio, as shown in Figure 3. The basic idea of the algorithm is to suggest a sampling ratio that has resulted in the lowest job completion time normalized by the job runtime based on the recent past. To achieve this, for every value in a defined range of possible sampling ratios (between 1% and 5%), it maintains a running score (`srScoreMap`), which is the average normalized JCT of  $T$  recently finished jobs that used the corresponding sampling ratio. In practice we found a  $T$  value of 100 works reasonably well. During system start-up, it tries sampling ratios of 2%, 3%, and 4% for the first  $3T$  jobs (Line 2–7). It further tries sampling ratios of 1% and 5% if going down from 3% to 2% or going up from 3% to 4% reduces the normalized JCT. Afterwards, for each new job, it uses the sampling ratio that has the lowest running score. Finally, upon each job completion, the score map is updated (Line 16–24).

Once the sampling ratio is chosen, SLEARN selects pilot tasks for a job randomly.

**How to estimate from sampled tasks?** Several methods such as bootstrapping, statistical mean or median can be used to predict job properties from sampled tasks. In GS, we use empirical mean to predict the mean task runtime.

**Work conservation.** When the system load is low, some

```

1: procedure GETCURRENTSAMPLINGPERCENTAGE(Job j)
2:   if j in First  $T$  jobs then
3:     return 3
4:   else if j in Second  $T$  jobs then
5:     return 2
6:   else if j in Third  $T$  jobs then
7:     return 4
8:   minScore = getMinValue(srScoreMap)
9:   if minScore.SR == 2 then
10:    if 1.1*minScore.value < srScoreMap[3].value then
11:      return 1
12:   if minScore.SR == 4 then
13:     if srScoreMap[3].value > 1.1*minScore.value then
14:       return 5
15:   return minScore.SR
16: procedure UPDATESCOREONJOBCOMPLETION(Job j)
17:   sr = j.sr                                 $\triangleright$  Get j’s sampling ratio.
18:   normalizedJCT = j.jct                       $\triangleright$  Get j’s normalized JCT.
19:   UpdateScoresMap(sr, normalizedJCT)
20: procedure UPDATESCOREMAPS(sr, normalizedJCT)
21:   if Len(jobWiseSrScoresMap[sr])> $T$  then
22:     Drop first element of jobWiseSrScoresMap[sr]
23:   jobWiseSrScoresMap[sr].append(normalizedJCT)
24:   srScoreMap[sr].value = mean(jobWiseSrScoresMap[sr])

```

Figure 3: Adaptive sampling algorithm in SLEARN.

machines may be idle while the non-sampling tasks are waiting for the sampling tasks to finish. In such cases, SLEARN schedules non-sampling tasks of jobs to run on otherwise idle machines. In work conservation, the jobs are scheduled in the FIFO order of their arrival.

### 5.1.3 Baseline Predictors and Policies

We compare SLEARN’s effectiveness against four different baseline predictors and two policies: (1) **3Sigma**: as discussed in §4.3. (2) **3SigmaTL**: same as 3Sigma but handles thin jobs in the same way as SLEARN; they are directly placed in the highest priority queue. This is to isolate the effect of thin job handling. (3) **POINT-EST**: same as 3Sigma, with the only difference being that, instead of integrating a utility function over the entire runtime history, it predicts a point estimate (median in our case) from the history. (4) **LAS**: The Least Attained Service [48] policy approximates SJF online

without explicitly learning job sizes, and is most recently implemented in the Kairos [29] scheduler. LAS uses multiple priority queues and the priority is inversely proportional to the service attained so far, *i.e.*, the total execution time so far. We use the sum of all the task execution time to be consistent with all the other schemes. (5) FIFO: The FIFO policy in YARN simply prioritizes jobs in the order of their arrival. Since FIFO is a starvation free policy, there is no need for multiple priority queues. (6) ORACLE: ORACLE is an ideal predictor that always predicts with 100% accuracy.

## 5.2 Experimental Results

We evaluated SLEARN’s performance against the six baseline schemes discussed above by plugging them in GS and execute the 3 traces (2STrace, GTrace11, and GTrace19) using large scale simulations and on a 150-node testbed cluster in Azure (§5.2.6).

### 5.2.1 Experimental Setup

**Cluster setup.** We implemented GS, SLEARN and baseline estimators with 11 KLOC of Java and python2. We used an open source java patch for Gridmix [15] and open source java implementation of NumericHistogram [13] for Hadoop. We used some parts from DSS, an open source job scheduling simulator [10], in simulation experiments.

We implemented a proxy scheduler wrapper that plugs into the resource manager of YARN [5] and conducted real cluster experiments on a 150-node cluster in MS Azure [14].

Following the methodology in recent work on cluster job scheduling [25, 47, 51], we implemented a synthetic generator based on the Gridmix implementation to replay jobs that follow the arrival time and task runtime from the input trace. The Yarn master runs on a standard DS15 v2 server with 20-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 140GB memory, and the slaves run on D2v2 with the same processor with 2-core and 7GB memory.

**Parameters.** The default parameters for priority queues in GS in the experiments are: starting queue threshold ( $Q_0^{hi}$ ) is  $10^6$  ms, exponential threshold growth factor ( $E$ ) is 10, number of queues ( $N$ ) is set to 10, and the weights for time sharing assigned to individual priority queues decrease exponentially by a factor of 10. Previous work (*e.g.*, [23]) and our own evaluation have shown that the scheduling results are fairly insensitive to these configuration parameters. We omit their sensitivity study here due to page limit. SLEARN chooses the number of pilot tasks for wide jobs using the adaptive algorithm described in §5.1.2 and the threshold for thin jobs is set to 3. We evaluate the effectiveness of adaptive sampling in §5.2.2 and the sensitivity to thinLimit in §5.2.8.

**Performance metrics.** We measure three performance metrics in the evaluation: JCT speedup, defined as the ratio of a JCT under a baseline scheme over under SLEARN, the job runtime estimation accuracy, and job waiting time.

Table 6: Performance improvement of SLEARN over 3Sigma under adaptive sampling and fixed-ratio sampling.

	Fraction of tasks chosen as pilot tasks						
	1%	2%	3%	4%	5%	10%	Adap.
2STrace							
P50 pred. error (%)	19.4	19.0	19.0	18.7	18.4	16.9	19.0
Avg. JCT speedup (×)	1.24	1.23	1.27	1.26	1.27	1.28	1.28
P50 speedup (×)	0.93	0.92	0.93	0.92	0.93	0.91	0.92
GTrace11							
P50 pred. error (%)	14.4	14.0	13.6	13.1	12.7	9.09	13.7
Avg. JCT speedup (×)	1.52	1.55	1.54	1.56	1.58	1.51	1.56
P50 speedup (×)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GTrace19							
P50 pred. error (%)	55.7	53.8	47.1	46.5	42.1	36.1	51.8
Avg. JCT speedup (×)	1.31	1.31	1.31	1.32	1.28	1.24	1.32
P50 speedup (×)	1.07	1.07	1.05	1.05	1.01	1.00	1.07

**Workload.** We used the same training data for history-based estimators and the test traces (2STrace, GTrace11 and GTrace19) as described in §4.3.

### 5.2.2 Effectiveness of Adaptive Sampling

In this experiment, we evaluate the effectiveness of our adaptive algorithm for task sampling. Fig. 4 shows how the sampling ratio selected by the adaptive algorithm for each job varies between 1% and 5% over the duration of the three traces. We further compare average JCT speedup and P50 speedup under the adaptive algorithm with those under a fixed sampling ratio, ranging between 1% and 10%. Table 6 shows that the adaptive sampling algorithm leads to the best speedups for 2STrace and GTrace19 and is about only 1% worse than the best for GTrace11. Interestingly, we observe that no single sampling ratio works the best for all traces. Nonetheless, the adaptive algorithm always chooses one that is the best or closest to the best in terms of JCT speedup. More importantly, we see that the adaptive algorithm does not always use the sampling ratio with the best prediction accuracy, which shows that it effectively balances the tradeoff between prediction accuracy and sampling overhead.

### 5.2.3 Prediction Accuracy

SLEARN achieves more accurate estimation of job runtime over 3Sigma – the details were already discussed in §4.3.

### 5.2.4 Average JCT Improvement

We now compare the JCT speedups achieved using SLEARN over using the five baseline schemes defined in §5.1.3.

Fig. 5(a) shows the results for 2STrace. We make the following observations. (1) Compared to ORACLE, SLEARN achieves an average and P50 speedups of  $0.79\times$  and  $0.73\times$ , respectively. This is because SLEARN has some estimation error; it places 10.91% of wide jobs in the wrong queues, 3.54% in lower queues and 7.37% in higher queues. (2) SLEARN improves the average JCT over 3Sigma by  $1.28\times$ . This significant improvement of SLEARN comes from much higher prediction accuracy compared to 3Sigma (Fig. 2). (3) The

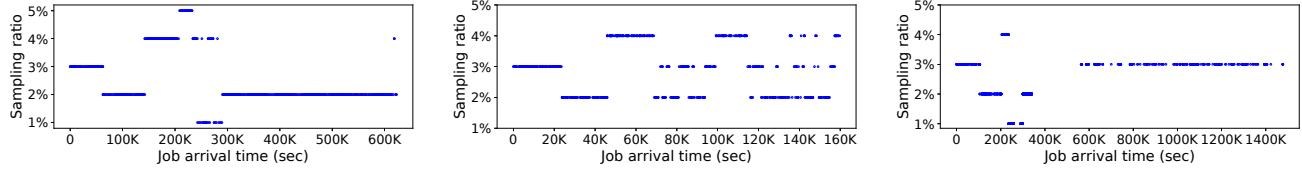


Figure 4: Sampling ratios selected by the adaptive sampling algorithm. The duration of initial 3T jobs appear varying due to uneven arrival times.

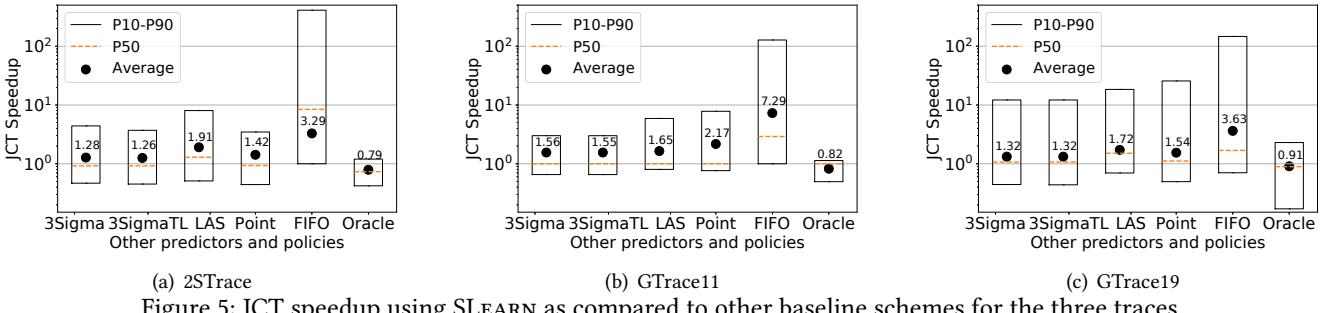


Table 7: Percentage of the wide jobs that had correct queue assignment.

Prediction Technique	SLEARN	3Sigma
2STrace	89.09%	73.84%
GTrace11	86.45%	76.20%
GTrace19	73.96%	58.07%

improvement of SLEARN over 3SigmaTL,  $1.26\times$ , is similar to that over 3Sigma, confirming thin job handling only played a small role in the performance difference of the two schemes. To illustrate SLEARN’s high prediction accuracy, we show in Table 7 the fraction of wide jobs that were placed in correct queues by SLEARN and 3Sigma. We observe that SLEARN consistently assigns more wide jobs to correct queues than 3Sigma for all three traces. (4) Compared to POINT-EST, SLEARN improves the average JCT by  $1.42\times$ . Again, this is because SLEARN estimates runtimes with higher accuracy. (5) Compared to LAS, SLEARN achieves an average JCT speedup of  $1.91\times$  and P50 speedup of  $1.29\times$ . This is because LAS pays a heavy penalty in identifying the correct queues of jobs by moving them across the queues incrementally. (6) Lastly, compared with FIFO, SLEARN achieves an average JCT speedup of  $3.29\times$  and P50 speedup of  $8.45\times$ .

Fig. 5(b) shows the results for GTrace11. Scheduling under SLEARN again outperforms all other schemes. In particular, using SLEARN improves the average JCT by  $1.56\times$  compared to using 3Sigma,  $1.55\times$  compared to using 3SigmaTL,  $2.17\times$  compared to using Point-Est, and  $1.65\times$  compared to using the LAS policy. Fig. 5(c) shows that scheduling under SLEARN outperforms all other schemes for GTrace19 too. In particular, using SLEARN improves the average JCT by  $1.32\times$ ,  $1.32\times$ ,  $1.54\times$ , and  $1.72\times$  compared to using 3Sigma, 3SigmaTL, Point-Est and the LAS policy, respectively.

In summary, our results above show that SLEARN’s higher estimation accuracy outweighs its runtime overhead from sampling, and as a result achieves much lower average job completion time than history-based predictors and the LAS policy for the three production workloads.

### 5.2.5 Impact of Sampling on Job Waiting Time

To gain insight into why sampling pilot tasks first under SLEARN does not hurt the overall average JCT, we next compare the *normalized waiting time* of jobs, calculated as the average waiting time of its tasks under the respective scheme, divided by the mean task length of the job.

Fig. 6 shows the CDF of the normalized job waiting time under SLEARN and 3Sigma. We see that the CDF curves can be divided into three segments. (1) The first segment, where both SLearn and 3Sigma have normalized waiting time (NWT) less than 0.04, covers 36.58% of the jobs, and 35.57% of the jobs are common. The jobs have almost identical NWT, much lower than 1 under both schemes. This happens because during low system load periods, e.g., lower than 1, the scheduler will schedule all the tasks to run under both scheme; under SLEARN it schedules non-sampled tasks of jobs to run before their sampled tasks complete due to work conservation. (2) The second segment, where both schemes have NWT between 0.04 and 1.90, covers 30.51% of the jobs, and 20.38% of the jobs are common. Out of these 20.38%, 29.81% have lower NWT under SLEARN and 70.19% have lower NWT under 3Sigma. This happens because when the system load is moderate, the jobs experience longer waiting time under SLEARN than under 3Sigma because of sampling delay. (3) The third segment, where both schemes have NWT above 1.90, cover 32.91% of the jobs, and 24.68% of jobs are common. Out of these 24.68%, 83.08% have lower waiting time under SLEARN and 16.92% under 3Sigma. This happens

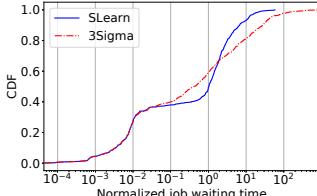


Figure 6: CDF of waiting times for wide jobs in GTrace11.

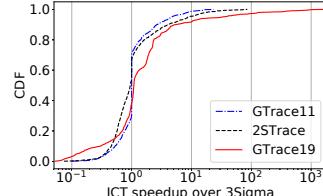


Figure 7: [Testbed] CDF of JCT speedup: SLEARN vs 3Sigma.

Table 8: Breakdown of jobs based on total duration and width (number of tasks) for 2STrace. Shown in brackets are a bin’s fraction of all the jobs in the trace in terms of job count and total job runtime.

	width < 3 (thin)	width $\geq 3$ (wide)
size < $10^3$ s (sm)	bin-1 (4.55%, 0.01%)	bin-2 (28.73%, 0.06%)
size $\geq 10^3$ s (lg)	bin-3 (14.29%, 5.41%)	bin-4 (52.43%, 94.52%)

because when the system load is relatively high, although jobs incur the sampling delay under SLEARN, they also experience queuing delay under 3Sigma, and the more accurate prediction of SLEARN allows them to be scheduled following Shortest Job First more closely than under 3Sigma.

A detailed analysis of how the system load of the trace affects the relative job performance under the two predictors can be found in the Appendix in [42].

### 5.2.6 Testbed Experiments

We next perform end-to-end evaluation of SLEARN and 3Sigma on our 150-node Azure cluster. Fig. 7 shows the CDF of JCT speedups using SLEARN over 3Sigma using 2STrace, GTrace11 and GTrace19. SLEARN’s performance on the testbed is similar to that observed in the simulation. In particular, SLEARN achieves average JCT speedups of 1.33 $\times$ , 1.46 $\times$ , and 1.25 $\times$  over 3Sigma for the 2STrace, GTrace11, and GTrace19 traces, respectively.

### 5.2.7 Binning Analysis

To gain insight into how different jobs are affected by SLEARN over 3Sigma, we divide the jobs into four bins in Table 8 for 2STrace and show the JCT speedups for each bin in Fig. 8. The results for the other two traces are similar and are omitted due to page limit.

We make the following observations. (1) SLEARN improves the JCT for 82.46% of the jobs in Bin-1 and the average JCT speedup for the bin is 10.54 $\times$ . This happens because the jobs in this bin are thin and hence SLEARN assigns them high priorities, which is also the right thing to do since these jobs are also small. (2) For bin-2, SLEARN achieves an average JCT speedup of 1.86 $\times$  from better prediction accuracy of SLEARN. The speedups are lower than for Bin-1 as the jobs have to undergo sampling. However, Bin-1 and Bin-2 make up only 0.01% and 0.06% of the total job runtime and thus have little impact on the overall JCT. (3) Bin-3, which has

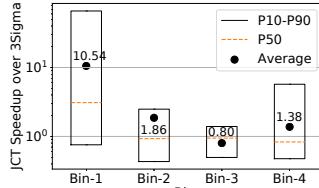


Figure 8: Performance breakdown down into the bins in Table 8. Figure 9: JCT speedup using SLEARN-DAG over baselines for GTrace19-DAG.

Table 9: Sensitivity analysis for thinLimit. Table shows average JCT speedup over 3Sigma.

thinLimit	2	3	4	5	6
2STrace	1.23x	1.28x	1.14x	0.97x	0.84x
GTrace11	1.54x	1.56x	1.55x	1.54x	1.53x
GTrace19	1.33x	1.32x	1.32x	1.30x	1.29x

14.29% of the jobs and accounts for 5.41% of the total job size, has a slowdown of 20.00%. The main reason is that SLEARN treats thin jobs in the FIFO order, whereas 3Sigma schedules them based on predicted sizes. (4) Bin-4, which accounts for a majority of the job and total job size, has an average speedup of 1.38 $\times$ , which contributes to the overall speedup of 1.28 $\times$ . The job speedups come from more accurate job runtime estimation of SLEARN over 3Sigma. Finally, we note that while for the 2Sigma trace, the majority of thin jobs are large, for the Google 2011 (Google 2019) trace, only 1.90% (1.60%) of the total number of jobs are thin and large and they make up only 0.5% (0.5%) of the total job runtime..

### 5.2.8 Sensitivity to Thin Job Bypass

Finally, we evaluate SLEARN’s sensitivity to thinLimit. Table 9 shows that for GTrace11 and GTrace19, the average JCT speedup barely varies with thinLimit, but for 2STrace, there is a big dip when increasing thinLimit to 4 or 5. This is because a significant number of jobs in 2STrace have width 4, which causes the number of thin jobs to increase from 18.84% to 58.50% when increasing thinLimit from 4 to 5.

## 6 Scheduling for DAG Jobs

In earlier sections, we have focused on the benefits of sampling-based prediction. On the other hand, we envision that there are situations where it would be beneficial to combine sampling-based and history-based predictions. Below, we present our preliminary work applying such a hybrid strategy for scheduling DAG jobs. We will discuss several other use cases of a hybrid strategy in §7. Note that for multi-phase DAG jobs, simply applying sampling-based prediction to each phase in turn cannot estimate the whole DAG runtime ahead of time. Instead, our hybrid design below aims to learn the runtime properties and optimize the performance of a multi-phase DAG job *as a whole* (e.g., [30, 33]).

**Hybrid learning for DAGs (SLEARN-DAG).** The key idea

of SLEARN-DAG is to adjust history-based prediction of the runtime of DAG jobs using sampling-based learning of its first stage. Upon arrival of a new DAG job, we estimate the runtime of its first stage using sampling-based prediction as described in §5.1.2, denoted as  $d_s$ . We also estimate the duration of this stage using history-base 3Sigma, denoted as  $d_h$ , and compute the adjustment ratio of  $\frac{d_s}{d_h}$ . For each of the remaining stages of the DAG, we predict their runtime using 3Sigma and then multiply it with the adjustment ratio. In a nutshell, this hybrid design reduces the error of history-based prediction due to staleness of the learning data, while avoiding the delay of sampling across all other stages.

**History-based learning for DAGs (3SIGMA-DAG).** This is a straight-forward extension of 3Sigma. Upon arrival of a DAG job, it predicts independently the runtime for each stage using the 3Sigma and sums up the estimated runtime of all stages as the estimated runtime of the entire DAG.

We similarly extended other baselines described in §5.1.3 for DAG job.

**Experimental setup.** We evaluated SLEARN-DAG against 3SIGMA-DAG by replaying cluster trace in simulation experiments based on GS (§5.1.1). We kept the simulation setup and parameters the same as used in the other experiments. In particular, a DAG is placed in the corresponding priority queue based on its estimated total runtime.

**DAG Traces.** The only publically available DAG trace we could find is a trace from Alibaba [3], which could not be used as it does not contain features required for history-based prediction using 3Sigma. Instead, we followed the ideas in previous work, e.g., Branch Scheduling [34], to generate a synthetic DAG trace of about 900 jobs using the Google 2019 trace [11], denoted as GTrace19-DAG. The number of stages in DAGs in the GTrace19-DAG was randomly choosen to be between 2-5 and each stage is a complete job from the Google 2019 trace. The jobs that are part of the same DAG have the same *jobname* and the same *username*.

**Results.** The results in Fig. 9 show that SLEARN-DAG achieves significant speedup over other designs. The speedup is  $1.26\times$  over 3SIGMA-DAG,  $2.15\times$  over LAS-DAG, and  $1.74\times$  over POINT-EST-DAG. Looking deeper, we find that our sampling-based prediction still yields higher prediction accuracy: the P50 prediction error is 33.90% for SLEARN-DAG, compared to 47.21% for 3SIGMA-DAG. On the other hand, for DAG jobs the relative overhead of sampling (e.g, the delay) is lower since only the first stage is sampled. Together, they produce speedup comparable to earlier sections.

## 7 Discussions and Future Work

**Combining history and sampling.** In addition to improving the scheduling of DAG jobs (§6), we discuss several additional motivations for combining history- and sampling-based learning. (1) For workloads with both recurring and

first-time jobs, sampling-based learning can be used to estimate properties for first-time jobs, while history-based learning can be used for recurring jobs. (2) When the workload has both thin and wide jobs, history-based learning can be used for estimating the runtime for thin jobs, while sampling-based learning is used for wide jobs. (3) History-based learning can be used to establish a prior distribution, and sampling-based approach can be used to refine the posterior distribution. Such a combination is potentially more accurate than using either approach alone. For example, knowing the prior distribution of task lengths can help to develop better max task-length predictors, which can be useful for jobs with deadlines. (4) Though not seen in the production traces used in our study, in cases when task-wise variation and job-wise variation fluctuate, adaptively switching between the two prediction schemes may also help. (5) When the cluster is heterogeneous, an error adjustment using history, similar to what we did in §6, can be applied.

**Dynamic adjustment of ThinLimit.** ThinLimit is a subjective threshold. It helps in segregating jobs for which waiting time due to sampling overshadows the improvement in prediction accuracy. The optimal choice of this limit will depend on the cluster load at the moment and hence can be adaptively chosen like the sampling percentage (Fig. 3 on page ).

**Heterogeneous clusters.** Extending sampling-based learning to heterogeneous clusters requires adjusting the task sampling process. One idea is to schedule pilot tasks on homogeneous servers and then scale their runtime to different types of servers using the ratio of machine speeds.

## 8 Conclusions

In this paper, we performed a comparative study of task-sampling-based prediction and history-based prediction commonly used in the current cluster job schedulers. Our study answers two key questions: (1) Via quantitative, trace and experimental analysis, we showed that the task-sampling-based approach can predict job runtime properties with much higher accuracy than history-based schemes. (2) Via extensive simulations and testbed experiments of a generic cluster job scheduler, we showed that although sampling-based learning delays non-sampled tasks till completion of sampled tasks, such delay can be more than compensated by the improved accuracy over the prior-art history-based predictor, and as a result reduces the average JCT by  $1.28\times$ ,  $1.56\times$ , and  $1.32\times$  for three production cluster traces. These results suggest task-sampling-based prediction offers a promising alternative to the history-based prediction in facilitating cluster job scheduling.

**Acknowledgement** We thank our shepherd Sangeetha Abdu Jyothi and the anonymous reviewers for their helpful comments. This work was supported in part by NSF grant 2113893.

## References

- [1] 2sigma hedge fund. [www.twosigma.com](http://www.twosigma.com).
- [2] 2sigma's proprietary job scheduler. <https://www.twosigma.com/insights/article/cook-a-fair-preemptive-resource-scheduler-for-compute-clusters/>.
- [3] Alibaba cluster trace. <https://github.com/alibaba/clusterdata>.
- [4] Apache hadoop. <http://hadoop.apache.org>.
- [5] Apache hadoop yarn. <https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>.
- [6] Apache hive. <http://hive.apache.org>.
- [7] Apache spark. <http://spark.apache.org>.
- [8] Cluster trace from google - 2011. [https://github.com/google/cluster-data/blob/master/ClusterData2011\\_2.md](https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md).
- [9] A document released by google containing schema and details of the cluster trace released by google. [https://drive.google.com/open?id=0B5g07T\\_gRDg9Z0lsSTEtTWtpOW8](https://drive.google.com/open?id=0B5g07T_gRDg9Z0lsSTEtTWtpOW8).
- [10] Dss scheduler. <https://github.com/epfl-labos/DSS>.
- [11] Google cluster-usage traces, retrieved 21st july 2020. <https://research.google/tools/datasets/google-cluster-workload-traces-2019/>.
- [12] Google cluster-usage traces, retrieved 21st july 2020. <https://drive.google.com/file/d/10r6cnJ5cJ89fpWCgj7j4LtlBqYN9Ri9/view>.
- [13] Hadoop patch for numeric histogram. <https://issues.apache.org/jira/browse/YARN-2672>.
- [14] Microsoft azure. <http://azure.microsoft.com>.
- [15] A patch for gridmix. <https://issues.apache.org/jira/browse/YARN-2672>.
- [16] Personal communication with a 2sigma engineer regarding properties of the 2sigma trace used.
- [17] A private trace collected by 2sigma engineers from their clusters. [www.twosigma.com](http://www.twosigma.com).
- [18] Results on the posteriro distribution with gaussian priors. <https://people.eecs.berkeley.edu/~jordan/courses/260-spring10/lectures/lecture5.pdf>.
- [19] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, Philadelphia, PA, 2014. USENIX Association.
- [20] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan De-Bardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, 2018. USENIX Association.
- [21] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.
- [22] Ronnie Chaiken, Bob Jenkins, Per-AAke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008. <http://dx.doi.org/10.14778/1454159.1454166>.
- [23] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM ’15*, pages 393–406, New York, NY, USA, 2015. ACM.
- [24] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM ’14*, pages 443–454, New York, NY, USA, 2014. ACM.
- [25] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC ’18*, pages 121–134, New York, NY, USA, 2018. ACM.
- [26] Edward G Coffman and Leonard Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM (JACM)*, 15(4):549–576, 1968.
- [27] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you’re late don’t blame us! In *Proceedings of the ACM Symposium on Cloud Computing, SOCC ’14*, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

- [28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
- [29] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 135–148, New York, NY, USA, 2018. ACM.
- [30] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.
- [31] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.
- [32] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.
- [33] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, 2016. USENIX Association.
- [34] Zhiyao Hu, Dongsheng Li, Yiming Zhang, Deke Guo, and Ziyang Li. Branch scheduling: Dag-aware scheduling for speeding up data-parallel jobs. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [35] Zhe Huang, Bharath Balasubramanian, Michael Wang, Tian Lan, Mung Chiang, and Danny HK Tsang. Need for speed: Cora scheduler for optimizing completion-times in the cloud. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 891–899. IEEE, 2015.
- [36] Calin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 97–109, Santa Clara, CA, 2017. USENIX Association.
- [37] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.
- [38] Akshay Jajoo, Rohan Gandhi, and Y. Charlie Hu. Graviton: Twisting space and time to speed-up coflows. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.
- [39] Akshay Jajoo, Rohan Gandhi, Y. Charlie Hu, and Cheng-Kok Koh. Saath: Speeding up coflows by exploiting the spatial dimension. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 439–450, New York, NY, USA, 2017. ACM.
- [40] Akshay Jajoo, Y. Charlie Hu, and Xiaojun Lin. Your coflow has many flows: Sampling them for fun and speed. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 833–848, Renton, WA, 2019. USENIX Association.
- [41] Akshay Jajoo, Y. Charlie Hu, and Xiaojun Lin. A case for flow sampling based learning for coflow scheduling, 2021. <http://arxiv.org/abs/2108.11255>.
- [42] Akshay Jajoo, Y. Charlie Hu, Xiaojun Lin, and Nan Deng. A case for task sampling based learning for cluster job scheduling, 2021. <http://arxiv.org/abs/2108.10464>.
- [43] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.
- [44] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayananamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, 2016. USENIX Association.

- [45] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4):1 – 12, 2004.
- [46] Misja Nuyens and Adam Wierman. The foreground–background queue: a survey. *Performance evaluation*, 65(3-4):286–307, 2008.
- [47] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys ’18, pages 2:1–2:17, New York, NY, USA, 2018. ACM.
- [48] Idris A. Rai, Guillaume Urvoy-Keller, and Ernst W. Biersack. Analysis of las scheduling for job size distributions with high variance. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’03, pages 218–228, New York, NY, USA, 2003. ACM.
- [49] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC ’16, pages 415–427, New York, NY, USA, 2016. ACM.
- [50] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times using historical information. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 122–142, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [51] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A. Kozuch, and Gregory R. Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. In *Technical Report CMU-PDL-16-104*. Carnegie Mellon University, 2016.
- [52] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys ’16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.
- [53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.
- [54] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.
- [55] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, WENCHI Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 481–494, Boston, MA, 2018. USENIX Association.
- [56] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys ’10, pages 265–278, New York, NY, USA, 2010. ACM.



# Starlight: Fast Container Provisioning on the Edge and over the WAN

Jun Lin Chen  
*University of Toronto*

Daniyal Liaqat  
*University of Toronto*

Moshe Gabel  
*University of Toronto*

Eyal de Lara  
*University of Toronto*

## Abstract

Containers, originally designed for cloud environments, are increasingly popular for provisioning workers outside the cloud, for example in mobile and edge computing. These settings, however, bring new challenges: high latency links, limited bandwidth, and resource-constrained workers. The result is longer provisioning times when deploying new workers or updating existing ones, much of it due to network traffic.

Our analysis shows that current piecemeal approaches to reducing provisioning time are not always sufficient, and can even make things worse as round-trip times grow. Rather, we find that the very same layer-based structure that makes containers easy to develop and use also makes it more difficult to optimize deployment. Addressing this issue thus requires rethinking the container deployment pipeline as a whole.

Based on our findings, we present Starlight: an accelerator for container provisioning. Starlight decouples provisioning from development by redesigning the container deployment protocol, filesystem, and image storage format. Our evaluation using 21 popular containers shows that, on average, Starlight deploys and starts containers  $3.0\times$  faster than the current state-of-the-art implementation while incurring no runtime overhead and little (5%) storage overhead. Finally, it is backwards compatible with existing workers and uses standard container registries.

## 1 Introduction

Docker and other container engines are a popular approach for software provisioning due to their low overhead, standardization, and ease of use [3, 41, 53, 60]. They provide isolation and standardized packaging for application files, and are supported by a large suite of standard tools [16, 18, 21, 23, 24]. Unlike VMs, containers are lightweight and easy to update: even lightweight VMs [1, 38] require re-building and re-deploying the entire image. Container images, on the other hand, are built as a stack of layers; updating a component can be as simple as rebuilding its layer rather than the entire image [15].

Similarly, we can extend a container by adding layers to the top of its stack. Deploying is also straightforward: fetch compressed layers from a *registry* server such as Docker Hub, decompress them, mount using a layered filesystem [34], and start the container process. The stack-of-layers structure thus makes containers easy to develop and maintain, and fits well into modern development workflows [5].

Though originally designed to be used inside a cloud datacenter [20], containers are becoming increasingly popular in edge computing, mobile, and multi-cloud settings [11, 22, 25, 47, 56, 61].<sup>1</sup> Placing workers outside the cloud and closer to the user brings many advantages such as lower latency, bandwidth and power reduction, and privacy [52, 59]. Containers can be used to provision network functions at mobile base stations [13], Function-as-a-Service (FaaS) runtimes on local datacenters [43], local replicas in distributed stores [42], or components of distributed applications [61].

However, as systems grow larger and more complex, fast container provisioning is increasingly important. For example, Container-as-a-Service (CaaS) and FaaS providers must be able to provision workers quickly [3, 41, 60]. Another common case is rolling software updates, where we must update software across many thousands of workers [6, 50]. Edge computing brings its own set of challenges: high latency upstream links, bandwidth limits, resource-constrained local datacenters and workers, and user mobility. Pulling container images from a registry in the cloud to an edge worker takes a long time over wide-area links [25]. Another issue is user mobility, which causes frequent reconfigurations [57], making worker provisioning a common operation. Finally, limited resources in edge datacenters means that placing a local registry or cache at every edge can be expensive [25].

While there is work on improving container provisioning time, many are designed for the cloud [25, 60, 63], and are ill-suited for edge computing scenarios. For example, FaaS-Net [60] uses a tree of workers to deploy containers in parallel, which is infeasible when latency is large and bandwidth

<sup>1</sup>The distinctions between these settings are not relevant for this work, hence we will refer to all of these using the umbrella term “edge computing”.

is limited. Another popular approach is on-demand download [28,37,58], where we start containers early and download files on demand. These scale poorly with even moderate latency, even though many containerized applications do not necessarily require all mounted files immediately.

**Our Contributions** We identify three barriers to fast container provisioning. First, the layer-based structure that makes containers so convenient also prevents effectively applying common optimizations such as eliminating redundancy and downloading files on-demand. Second, the pull-based design of current approaches, where workers request what they need, becomes detrimental as latency grows. Finally, current approaches do not explicitly address the common scenario of software updates. We argue that faster provisioning requires a holistic approach to container deployment.

Motivated by these insights, we present **Starlight: an accelerator for provisioning container-based applications** that decouples the mechanism of container provisioning from container development. Starlight maintains the convenient stack-of-layers structure of container images, but uses a different representation when deploying them over the network. The development and operational pipelines remain unchanged: users can use existing containers, tools, and registries. In designing Starlight, we revisit every aspect of the container deployment mechanism:

- A redesigned **worker-cloud deployment protocol** sends all file metadata first, allowing containers to start before file contents are available. It uses a push-based approach to avoid costly round-trip requests: workers can specify what they already have in store, so we send only the files they need in the order they would be needed.
- On the worker side, we use a **new filesystem** to mount files as soon as metadata is available, allowing our **custom snapshotter plugin** to start containers quickly while downloading file contents in the background. When a container opens a file whose contents are pending, we block until the contents are available.
- Workers connect to a new **proxy** component in the cloud which implements the new protocol. The proxy optimizes the list and order of files on-demand, across multiple layers and containers. This reduces duplication and makes updates faster. The proxy works transparently with existing infrastructure: compressed layers are stored in a standard registry, and legacy workers can connect to that registry as normal.
- A seekable **compressed layer format** allows the proxy to send individual compressed files to the worker without having to decompress stored layers first. This format has low overhead (average of 4.2%) and is backwards compatible with existing workers and registries, so there is no need to store container images in two formats.

We use 21 popular container images to evaluate Starlight across a range of network latencies, bandwidths, and scenarios. Our results show that Starlight substantially outper-

forms other approaches across all latencies, with  $3.0\times$  faster provisioning than a state-of-the-art baseline [21], and  $1.9\times$  faster on average than the next best approach [58]. Starlight also improves provisioning inside the cloud; for example it can deploy updates 35% faster than prior work [58]. In fact, Starlight containers often start faster than the time it would take to merely download an optimized container image. Finally, Starlight has little-to-no runtime overhead: its worker performance matches the standard state-of-the-art approach.

Starlight is currently available as an open source project at <https://github.com/mc256/starlight>.

## 2 Background

A *container* is a process that is isolated from the host system using techniques such as cgroups and namespaces [35]. A container is structured as a stack of *layers*, where each layer contains a part of the filesystem tree for the containerized application. Layers are mounted by the container process using a filesystem such as OverlayFS [34] that presents the containerized application with a *merged view*: files in upper layers replace those in lower layers, making it easy to update container contents using copy-on-write from lower layers. Most layers are read-only; writes go to a top read-write layer using copy-on-write as needed. A *container image* is the set of files and associated metadata that represent the container at rest (i.e., when it is not running). Concretely, container images are comprised of container configuration metadata and a sequence of *compressed layers*: compressed files that store the files in the layer and their associated metadata.

Containers are easy to develop, maintain, and deploy, due to their layer-based structure and standardized tooling. For example, developers can build new containerized applications by adding layers on top of an existing container image; packaging application updates is similarly straightforward. This also makes security updates for underlying components fast and automatic: applying an update simply requires updating the base layer. The repository of container images (the *registry* server) thus resembles a tree where individual images are split off from a common point.

Containers also make software provisioning easy using a three-phase process managed by a *container engine* on the worker such as containerd [21] or Docker [18]: (i) **pull** the requested container image from the registry and decompress its layers, (ii) **create** a container instance by preparing an initial snapshot of its filesystem state, and (iii) **start** the container instance, which involves mounting the snapshot filesystem and starting the container process using a standard *runtime*.

### 2.1 Edge Computing

Edge computing, defined broadly in this work, is the idea of placing computing resources outside the cloud, closer to the data or end users [52,62]: near the network edge (e.g., local

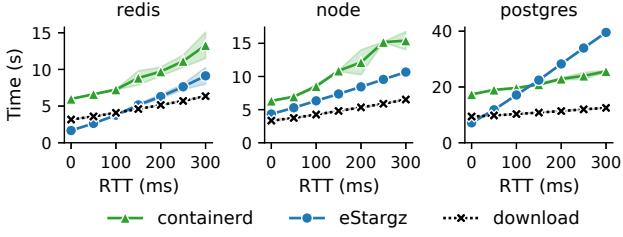


Figure 1: Mean container provisioning time across a range of latencies; shaded area show standard deviation across 5 runs.

datacenter, base station), user devices (e.g., mobile phone), or even in low Earth orbit [14]. We include in this definition settings such as mobile computing, content delivery networks (CDNs), Internet of Things (IoT), wide-area networks, and multi-cloud deployments.

Edge computing provides many benefits. For example, the short distance to users and data means faster and more consistent response times. And, since we no longer need to send all data to the cloud for processing, it improves privacy and reduces bandwidth usage. Other benefits include robustness to network failures and emission reduction [51, 52, 54].

Computing on edge workers has its drawbacks, however. First, they have much higher round-trip times (RTT) to the cloud. Recent work has found RTT ranging from 10ms to 400ms [10, 59] on terrestrial internet, and medians of 45–724ms on satellite-based internet [40]. Cross-datacenter latencies are also high, with one cloud provider reporting RTT between 2 to 400ms [45]. Bandwidth is also limited, with inter-datacenter bandwidths of 30–250Mbps [48]. Second, unlike cloud datacenters that offer virtually endless compute and storage, edge data centers are typically resource-constrained [54]. This encourages aggressive repurposing of workers, which makes fast provisioning even more important. For example, maintaining a pool of “hot” workers for elasticity is common in the cloud FaaS infrastructures, but is more expensive on the edge [43]. Lastly, edge and mobile applications are more affected by user mobility than cloud applications: as users move the nearest edge datacenter changes, which entails more frequent reconfiguration [57], i.e., provisioning.

### 3 Motivation

To explore the effect of latency on containers, we use containerd [21] to provision three popular containers over a 100Mbps connection with variable latency (see §5.1 for technical details). Figure 1 shows provisioning time, defined as the time it takes for the containerized application to download, decompress, start, and be ready. For comparison, we also show the download time for a file of equivalent size (dashed lines). We observe that containerd time increases substantially as RTT grows, and can even triple when RTT is

300ms. Moreover, in all cases provisioning time increases at a faster rate than would be expected simply due to extra network latency, which can be seen by comparing provisioning time to download time.

We also compare to eStargz [58], a recent approach that accelerates provisioning by starting the container before its layers have finished downloading and retrieving individual files on-demand. Prior work has found that many files are not used during container startup [28], indeed our three example containers access less than 1% of their files during startup (comprising 1–39% of data). Rather than wait until all files are available, eStargz starts the container quickly and download files on-demand [27, 28, 65]. It goes further by optimizing the order of files in each compressed layer such that the “hot” files needed early in container startup are placed first, thus avoiding redundant requests. Workers first fetch the hot part of each layer, start the container, and continue fetching the remaining files in the background or lazily on-demand.

As Figure 1 shows, when latency is small eStargz can accelerate provisioning. However, as RTT grows eStargz scales worse than the baseline and can even become slower than baseline containerd, as demonstrated for postgres with RTT of 150ms or above.

In the rest of this section, we analyze what makes optimizing provisioning difficult. We find that the root cause for slow provisioning time is the overall design of the provisioning pipeline: it is pull-based, designed around the stack-of-layers abstraction container images, and does not explicitly consider container updates. We show below that this design hinders optimization effort – both on the edge and in the cloud.

### 3.1 Pull-based Protocol

The protocol used to deploy containers to workers is pull-based: workers simply download the compressed layers they need from the registry using HTTP requests. This straightforward design avoids redundant pulls of layers that the worker already has, and works well inside datacenters. However, outside the cloud this can cause queueing delays, since registry implementations limits the number of concurrent connections per client to 2 or 3 [17]. Most containers have more layers [28], so the resulting cumulative delay adds up as RTT grows. Increasing the maximum number of concurrent connections could overwhelm the registry and may be impractical for resource-constrained workers.

On-demand downloading further exacerbates queuing by making even more HTTP requests to the registry. eStargz [58] uses a filesystem file access trace to determine the file order in compressed layers. In practice, however, the file access order of container workloads is not entirely deterministic due to multi-threading and runtime configuration. Container startup is thus slowed as multiple HTTP request due to out-of-order file accesses queue in the registry and delay one another.<sup>2</sup>

<sup>2</sup>Interestingly, excessive round-trips and queuing delays were also ob-

## 3.2 Layered-based Structure

Container images are structured as a stack of independent layers: each layer is stored separately, and contains its own metadata (e.g., list of files). While convenient for development, we argue that this makes optimizing provisioning more difficult: first, the information on container contents is distributed across multiple layers; second, because layers are the wrong granularity for provisioning protocols; and third, layer reuse does not capture updates well.

**Distributed Metadata** The first issue is that file metadata, including the list of files in the container contents, is not sent separately as part of the container image. Rather, each compressed layer includes its own list of files, and their metadata is intermingled with file contents. Yet, we cannot start a container early since list of files in a container is unknown until all layers are retrieved.

Consider again eStargz: since standard container images lack file metadata, eStargz stores a table of contents (ToC) at the end of every layer. Unfortunately, neither the size of compressed layers nor the exact beginning of the ToC is encoded in the image metadata. This in turn means at least two and perhaps three HTTP requests per layer: one to determine the size of its compressed image file, another to retrieve the layer’s ToC from an estimated position before the end, and potentially a third if the ToC is larger than expected.

Fixing this is not trivial, since container images are standardized; careless changes would make development harder. For example, adding a table of contents to container image metadata requires changing the standard and updating a huge number of existing tools used by developers [18, 21, 23, 46].

**Layer vs. File Granularity** Second, and perhaps counter-intuitively, the layer-based structure makes deployment slower due to cross-layer (and cross-container) redundancy. Containers evolve one layer at a time by extending other images with new layers. To update a file, we first copy it from the original read-only layer to the top read-write layer. Changing file metadata (e.g., ownership) also requires copying since layers cannot refer to each other. In both cases the original file remains in the previous layer, with no indication that this has happened. This cross-layer data duplication cannot be captured explicitly since file metadata is stored in the layers, and cannot be exploited by compression since layers are compressed independently.

Table 1 illustrates the cost of such redundancy for our sample containers by comparing the required download size using the baseline layer-based approach, to the size of an optimized “delta” update that only includes changed files and removes duplicates across layers.<sup>3</sup> The inflation in update sizes ranges from 1.23 $\times$  (redis) to a whopping 10.54 $\times$  (node). Indeed,

served in mobile web browsers that use HTTP/2 [36]. The underlying causes, however, are quite different (handshaking and packet losses, respectively). Determining whether the mitigation approaches in QUIC are applicable for container provisioning (or vice versa) is beyond the scope of this work.

Container	From → To	Baseline	Delta
redis	6.2.1 → 6.2.2	9.6	7.8
node (alpine)	16-3.11 → 16-3.12	39.0	3.7
postgres	13.1 → 13.2	109.5	24.9

Table 1: Package size (MB) of standard and optimized update.

recent analysis of Docker Hub [64] found that 90% of layers are only referenced by a single image, but over 99.4% of files had duplicates. Exploiting this cross-layer duplication during provisioning is difficult since file metadata is distributed across multiple layer.

While there has been prior work that proposes deduplicating the registry [55, 63], this does not reduce provisioning time since (by design) the downloaded container images and provisioning protocol remain the same. Rather, such work focus on saving registry space.

**Limited Layer-reuse** Ideally, an updated container image would share common layers with its previous version, so deploying updates requires only fetching and decompressing the new layers. Unfortunately, even a minor change to a single layer low in the stack causes cascading effect where all layers above it must be updated, even though their contents are mostly identical [15]. On such example is updating a worker from `postgres:13.1` to `postgres:13.2`. These two container images share no layers since an update to the `debian:buster-20210208-slim` image forced an update to all downstream layers. Provisioning this update requires downloading and decompressing the entire image, even though the total size of changed files is much smaller. Our analysis of 21 popular containers (Table 2) suggests that layer reuse only captures 3% of duplication, on average.

## 3.3 No Explicit Update Support

Provisioning a worker is not a rare operation. Rather, over the lifetime of a worker, we will deploy containers many times and for different reasons: initial provisioning, software updates, security patches, and so on. This even more common on edge workers due to user mobility and limited resources at edge datacenters (§2.1). This not only results in frequent provisioning, but also means that worker contents is highly diverse: as workers get updated and repurposed, the version of the container image available in local storage varies from worker to worker. As discussed above, such updates are an opportunity for optimization since many of the files have not, in fact changed (§3.2).

However, the current design of the provisioning pipeline does not allow users to express update operations explicitly.

<sup>3</sup>Flattening container images down to a single merged layer this way would mitigate many of the issues we discuss. However, would also eliminate the advantages of containers in the first place (§2), and would require an optimized image for every potential update path [47, 55].

Under the current approach, updates are treated as any other deployment: the worker simply pulls the needed layers from the registry. Depending on the update, this may or may not result in faster provisioning. While in theory we could prepare optimized provisioning packages in advance, the diversity in worker contents makes this approach impractical. A better approach is to compile the provisioning package dynamically, on-demand, by taking into account what is already available at the worker when selecting which files to include. Doing so, however, requires a capacity to express worker updates, which the current provisioning protocol does not support.

## 4 Starlight

Starlight is designed to accelerate container provisioning by considering the deployment pipeline holistically. In designing Starlight, we set out to achieve several goals. First, accelerate deployment on both low and high-latency links, and scale gracefully as latency grows. Second, preserve the advantages of containers for application developers. For the same reason, Starlight should be easy to adopt incrementally, without causing interference or requiring abrupt changes to working systems – Starlight should be backwards-compatible with non-Starlight workers, work with existing infrastructure, and have low overhead. Finally, Starlight should better support the common scenario of container updates.

### 4.1 Design Considerations

Starlight’s design is driven by four key principles, informed by our analysis of container provisioning (§3): (1) start containers early, (2) send workers only what they need, (3) use a push-based design to avoid costly round-trips, and (4) prioritize worker performance over cloud effort. These lead to the following design decisions:

- The provisioning protocol should not resemble the stack-of-layer structure of container images. Instead, it should be pushed-based, and operate at file rather than layer granularity. The list and order of files should be jointly optimized across multiple layers and containers.
- The provisioning protocol should cleanly separate file metadata from contents, and send the metadata first. This allows Starlight to start containers early by mounting a “mock” filesystem while downloading contents in the background.
- The provisioning protocol should let workers explicitly request updates and specify what is available to them locally.
- Avoid changing registry by placing a *proxy* located near it, which lets us to change provisioning protocol without affecting existing workers.
- The proxy should create provisioning packages on-demand based on what the worker already has available. This makes updating workers more efficient, avoids inflating the registry with packages for every conceivable update, and places

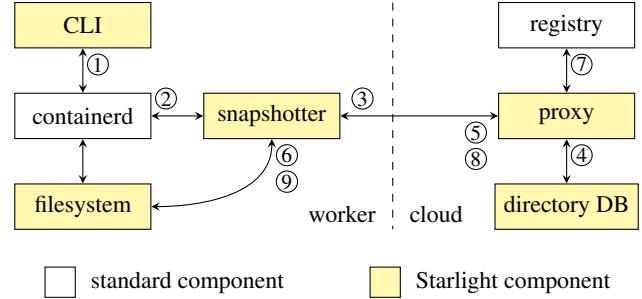


Figure 2: Starlight architecture.

the computational burden on the cloud where it is cheaper (§2.1). Supporting this requires storing a table of contents and file metadata for every container.

- Storing compressed layers using a seekable backwards-compatible format allows both Starlight and legacy workers to use the same compressed layer files and standard registries, and avoids inflating the registry size.
- Use standard container images to support the large ecosystem of existing tools for building, storing, and serving containers [16, 18, 21, 23, 24, 30, 46].

### 4.2 Overview

Figure 2 shows Starlight’s architecture, which is comprised of a *proxy* and a *directory database* (§4.4) in the cloud next to a standard registry; and a *snapshotter* plugin (§4.5) on the worker. The proxy and the snapshotter plugin communicate using the Delta Bundle Protocol (§4.3). The snapshotter plugin manages the lifecycle of the *filesystem* (§4.6) for the container instance. We also include a command line tool.

We first describe Starlight’s operation at a high level, and how it maps to the three-step PULL-CREATE-START process.

Once the user issues a worker PULL command to deploy a container ①, the command is received by the standard *containerd* daemon. *containerd* then forwards the command to the Starlight *snapshotter* daemon ②, and waits for confirmation that the requested images have been found. The Starlight *snapshotter* opens an HTTPS connection to the Starlight *proxy* and sends the list of requested containers as well as the list of relevant containers that already exist on the worker ③. The proxy queries the *directory database* ④ for the list of files in the various layers of the requested container image, as well as the image already available on the worker. The proxy will then begin computing the *delta bundle* that includes the set of distinct compressed file contents that the worker does not already have, specifically organized to speed up deployment; In the background, the proxy issues a series of HTTPS requests to the registry ⑦ to retrieve the compressed contents of files needed for delta bundle. Once the contents of the delta bundle has been computed, the proxy creates a *Starlight manifest* (SLM) – the list of file metadata,

container manifests, and other required metadata – and sends it to the snapshotter ⑤, which notifies `containerd` that the PULL phase has finished successfully.

We can then use the SLM execute the CREATE phase: configuring the container and the Starlight filesystem ⑥.

Starlight then proceeds to the START phase: it mounts then launch container instances. Note that even though the container instances have launched, at this point the worker does not have the contents of many files (or perhaps all, for new deployments). Such files are mounted based on metadata only; when opened, the Starlight filesystem will block until file contents have arrived from the proxy. The proxy streams file contents back to the snapshotter ⑧, which in turn updates the filesystem to unblocks the access of the file ⑨. To optimize provisioning times, the proxy sends files in the order they will (likely) be needed; this order is determined when building the delta bundle, and relies on preprocessed information stored in the directory database.

### 4.3 Delta Bundle Protocol

Starlight uses a novel Delta Bundle Protocol to send container images to the worker. This single-request HTTP-based protocol is designed to reduce unnecessary transfers, avoid round-trips, and prioritise information needed to start the container. Much of Starlight’s design is informed by the need to support the Delta Bundle Protocol.

A provisioning request includes the name and version of the requested the container image as part of the request URL, and optionally the old version in the worker’s local storage. The response consists of two parts: a header and a body.

**The Header** The header contains of all the information needed to start container instances of the requested images. It is comprised of (1) a Starlight Manifest (SLM), (2) a table of all layers digests from both the existing and requested container images, and (3) other data required by the implementation such as protocol version and authentication..

An SLM includes a standard Open Container Initiative (OCI) container image manifest file [24, 39], an OCI configuration file for the instance, a list of indices into the table of layer digests, and the filesystem table of content (ToC).

The ToC presents a merged (flattened) view of the requested container’s filesystem (§2), providing sufficient information for the worker to mount the container’s filesystem using StarlightFS without waiting for the response body. Each entry in the ToC includes the file name and path, type (e.g. regular file, link, or directory), attributes (e.g. ownership and timestamps), and an SHA256 hash of the file content. Additionally, every entry includes an index to the shared layer digests table in the delta bundle header, which enables reusing file contents on the worker’s local storage. Finally, each entry also has an offset field which points to the file’s *payload* – compressed file content – in the body of the delta bundle.

**Using the SLM** The name, metadata, offset, and index into the digest list allow workers to reconstruct the requested container’s filesystem. For new or updated files – those that the worker does not already have in its local storage – the offset points to the payload. This allows multiple file entries to reuse the same payload in the body of delta bundle, reducing the transfer volume. Alternatively, if a file’s metadata has changed (e.g., ownership), the payload already exists on the worker. The ToC entry thus contains the new metadata, an empty payload offset, and an index pointing to the original layer in the list of digests.

**The Delta Bundle Body** The body is a sequence of payloads (compressed file contents) for new or updated files, sent in the order which they are likely to be accessed. Since the header allows multiple file to reference the same payload – all payloads in the body are unique.

### 4.4 Proxy and Directory Database

Despite the name, the Starlight proxy is not merely a proxy server or a simple bridge. It is in charge of optimizing and building the delta bundle sent to the workers, as well as collecting and analyzing filesystem traces used in this optimization.

**The Directory Database** The directory database stores the table of contents and file metadata for each container image in the registry, as well as additional information used by the proxy to compute and optimize the delta bundle.

Whenever a new container image is uploaded to the registry (triggered manually or by hooks), the proxy captures file metadata from all layers, generates the ToC for the merged view of the image, and then save the ToC, container manifest, and image configuration file to the directory database.

The ToC in the directory database is the same as the ToC included in the SLM with additional fields that facilitate building the delta bundle body. First, it records the source compressed layer file, payload offset, and size for each file to help retrieve it from the registry. Second, it includes two extra columns, rank sum and hit count, used when sorting payloads; we discuss these below.

**Trace Collection** To sort payloads in the order that the worker is likely to access, Starlight collects filesystem traces from the worker to analyse the file usage. Trace collection is identical to running a container until it reports it is ready. When initiated by the user, the worker starts the container image locally using a special mode of the Starlight filesystem (§4.6) that collects file accesses. The worker then uploads the trace to the proxy, which ranks all files in the trace according to their access order. Finally, for each file in a container image, the proxy increases its hit count by one and adds its rank to the rank sum column. The average rank of a file can be computed from its rank sum and count.

Since file access can be non-deterministic, our design supports multiple collection runs. Collecting one trace usually

takes up to 2 minutes per run, depending on the container. By default, we collect 10 traces for each container.

Note that while prior work [37, 58] stores file order information per layer inside the compressed layers, Starlight associates this information with a container image and stores it outside the registry. This provides several benefits. First, it is possible to update file usage without rebuilding the compressed layers. Second, it allows for the likelihood that a file in a layer used by different containers to be accessed differently during startup. Third, new container image can reuse the traces from a previous version – solving the cold start problem. Finally, it allows for future development such as adjusting payload orders online based on provisioning feedback.

**Provisioning Process** A provisioning request from a worker contains the names and tags of two images: the image requested for deployment, denoted by  $R$ , and the old version of the image in its local storage (assuming there is one) denoted by  $A$ . To build the delta bundle, the proxy first retrieves metadata from the directory database for both container images  $R$  and  $A$ . It then issues a series of asynchronous requests to the registry to retrieve the compressed layers for  $R$ . These will be used to construct the body of the delta bundle.<sup>4</sup> It then proceeds to prepare an optimized delta bundle header and send it to the worker. Once all requested layers to arrive from the registry, the proxy sends the delta bundle body: for each payload in the delta bundle body as determined by the header, we copy compressed file content directly from the compressed layer and send them to the worker.

**Optimizer** The optimizer is responsible for selecting which compressed file content (payload) should be included in the body of the delta bundle and in which order, and then building the delta bundle header. Crucially, the optimizer does not require retrieving the compressed layers; the directory database contains all necessary information to build the delta bundle header. The optimization proceeds in several phases:

- **Merge:** load the merged (flattened) ToC for  $R$  and  $A$  from the directory database, denote them  $T_R$  and  $T_A$ .
- **Difference:** Compute the set difference  $T' = T_R \setminus T_A$ : for every file  $f$  in  $T_R$ , we look for a corresponding entry  $f'$  in  $T_A$  with the same hash and name. If we find one, we update the source layer index for  $f$  in  $T_R$  to the corresponding one in the old entry in  $T_A$  update its source layer index to the corresponding layer of  $f'$ . This step takes  $O(|T_R| + |T_A|)$  time and  $O(1)$  space.
- **Consolidate:** Consolidate files in  $T'$  with the same payload. Assuming the chance of hash collision is low, this step takes  $O(|T_R|)$  time and  $O(|T_R|)$  space.

<sup>4</sup>Our current implementation retrieves entire compressed layers. This does not substantially affects provisioning time since the registry and the proxy are located in the same cloud datacenter. Nevertheless, we stress that Starlight’s directory database and the seekable image format support retrieving only the contents of compressed files, by issuing HTTP range requests to the registry when building the delta bundle body. We are planning to implement this optimization in the immediate future.

- **Select:** Remove from  $T'$  files already available on the worker (whose source layer is in  $T_A$ ).
- **Sort:** Sort the payloads in order of increasing average rank,  $O(|T'| \log |T'|)$ . If different files point to the same payload due to previous steps, use the lowest rank.

**Compressed Layer Format** The current format used to store compressed layers is the tar gzip format: a sequence of concatenated files with interleaved headers for metadata (e.g., timestamps, ownership), compressed as one data stream. This format is non-seekable: extracting a specific file requires decompressing the entire compressed layer until we reach the file, which takes time.

eStargz [58] uses an alternative seekable compressed layer format that compresses files individually (or 4KB chunks of larger a file) and appends an index at the end of the compressed layer into the offsets of compressed files and chunks. To maintain backwards compatibility, each file includes tar headers and footers, so the tarball data stream remains unchanged. The result is an increase in the size of compressed layers due to the index at the end and the additional tar headers and footers. Furthermore, compression is less effective since file are compressed separately.

Our proposed format follows similar ideas, with three differences. First, we do not include an index at the end of the compressed layer, and instead use the directory database to store the table of contents. This not only reduces the overhead of our compressed layer format, but allows the proxy to build the delta bundle while fetching compressed layers in the background. Second, we do not need to split files into 4kb chunks since we retrieve files wholly, which simplifies our provisioning protocol and reduces the size of the ToC. Finally, since we do not need the metadata in tar headers and footers during provisioning, we do not include them as part of the compressed stream of file contents, which further reduces payload size. The overhead of Starlight’s format is only 4.4% for containers in Table 2 comparable to eStargz (4.7%).

## 4.5 Snapshotter Plugin

The `containerd` snapshotter daemon manages the life cycle of a container filesystem: from downloading images to keeping track of changes in the container’s mounted file system. We take advantage of the snapshotter plugin-based design [7] to write a snapshotter plugin to support Starlight provisioning. Figure 3 shows an overview of the Starlight snapshotter plugin, which includes two components: the *downloader* and *metadata manager*. The snapshotter also maintains the instances of the user space component of StarlightFS – one for every mounted container instance.

**Delta Bundle Downloader** The downloader is responsible for downloading the delta bundle from the proxy and decompressing the payloads to designated locations (if an image has been completely downloaded, it is not started). Once the downloader receives the delta bundle header, it saves the

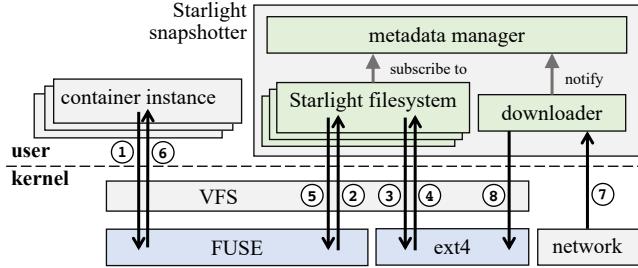


Figure 3: The flow of filesystem requests in Starlight.

SLM to the local storage and creates a metadata manager using the SLM. At this point, we have enough information to mount and start the container, so the snapshotter notifies the containerd daemon that the PULL phase has finished. In the background the downloader keeps receiving the payloads: it decompresses the payloads to its designated location according to the source layer information in the received SLM and the share layer digests in the delta bundle. Once the decompression of the payload has finished, it notifies the metadata manager. When a payload belongs to multiple files, the downloader creates hard links to avoid writing to the underlying filesystem multiple times.

**Metadata Manager** Multiple container instances can start from the same container image. The metadata manager therefore acts as a centralize place for managing file’s availability and its metadata. It maintains file metadata of all the files in a container image and manages files’ actual location in the host file system. Most importantly, it manages the availability of file contents, and notifies StarlightFS once a file payload has decompressed. Once the worker has downloaded the entire image, we store its SLM locally so that future container instances launch from the local storage. When removing an old container image, the metadata manager removes any hard link references (if any) and copies the file to a new location if it is used by a newer version of this image.

## 4.6 The Starlight Filesystem (StarlightFS)

The customized filesystem serves two goals. First, we need to start containers early using only their SLM. Second, we want to reuse file contents across layers and images. As OverlayFS and other filesystems do not support both of these features, we use FUSE [33] to implement StarlightFS.

**Structure** StarlightFS relies on the underlying host filesystem (e.g., ext4), similar to a typical OverlayFS and FUSE-OverlayFS. Like OverlayFS, StarlightFS provides a container with a merged view that combines multiple directories in the underlying filesystem that represent multiple read-only layers and a single read-write layer.

Starlight maintains a filesystem tree in memory, created from the merged view ToC in SLM. Each file (or directory) node keeps track of the actual location of the file contents –

whether it is in the read-only layer, in the read-write layer, or pending payload. As with the ToC, some nodes might reference read-only layers from the previous version of the container image (§4.3). Nodes of pending files will be notified by the metadata manager when the payload is available (in our implementation, by subscribing to a Go signal channel in the corresponding file entry of the metadata manager). The user space portion of StarlightFS is located in the snapshotter process to allow such low-overhead communication.

Note that StarlightFS does not maintain any file system state on its own, nor does it have any on-disk structures. Metadata for files in read-only layers is stored in the ToC. State for files in the read-write layer (i.e., mutable state) is stored in the underlying host filesystem, with changes forwarded to it immediately. For example, if a file is deleted by the container, we write a whiteout entry to the read-write layer, similarly to OverlayFS [34]. In case of a crash, error or remount, the tree and all other state are rebuilt using the saved SLM and underlying filesystem.

**Operation** When starting a container instance, the snapshotter creates a filesystem instance which builds a filesystem tree from the metadata manager’s ToC for this container image.

Figure 3 shows the flow of operations in StarlightFS. When a container instance performs a file operation, it is forwarded to StarlightFS via FUSE ①②. In the best scenario, the content of the file is already in the local filesystem (e.g., ext4 in Figure 3). Starlight uses the file path provided by the file node to opens the underlying file ③④ and then return the file handle back to the container instance ⑤⑥. In case the file contents are still pending, but the operation only involves reading metadata (e.g. GETATTR), StarlightFS returns the metadata immediately using the information in the file node.

When an operation on a pending file involves setting metadata (e.g. SETATTR) or accessing file content (e.g. OPEN, FSYNC), StarlightFS blocks the operation until the file is ready by subscribing to a Go signal channel associated with the file’s ToC entry in the metadata manager. Once the downloader has extracted the file payload ⑦⑧, it notifies the corresponding entry in the metadata manager, which closes the channel associated with the file’s ToC entry. This releases any filesystem tree nodes that are waiting for the payload, while newly created instances will not be able to subscribe to a closed message channel. StarlightFS can then load the file from the local storage and update the file metadata if necessary ③④, then return the file to the container instance ⑤⑥. If this requires changing the file metadata or content, this file will be copied from the read-only layer to the read-write layer. All subsequent requests will be forwarded to the read-write layer.

## 5 Evaluation

We use 21 popular container images to evaluate Starlight’s performance in both controlled and real-world networks. Our

main metric is *provisioning time*, defined as the time from the initial command to deploy a container on a worker, to the time the containerized application reports it is ready (as with HelloBench [28], this is determined by monitoring the application’s `stdout`). To show the benefit of Starlight, we define two types of provisioning: a *fresh* deployment means the container worker does not have any prior images in its local storage, while an *update* means deploying the next available version of a container to a worker that already has the previous version deployed.

## 5.1 Experimental Setup

We use AWS EC2 to run our experiments. Container workers use `m5a.large` instances (AMD EPYC 7000 at 2.5GHz) with 2 cores (vCPUs) and 8GB RAM. The registry server runs Docker Registry 2.0<sup>5</sup> v2.7.1 on a `c5.xlarge` instance (Intel Xeon at 3.4GHz) with 4 vCPUs and 8GB RAM. The Starlight proxy and the metadata database run on a second `c5.xlarge` instance. All the machines run Ubuntu 20.04.3 LTS. We use Linux’s Traffic Control tool [2, 29] to control round-trip time and bandwidth between the worker and the other machines. Bandwidth is limited to 100Mbps unless otherwise specified. For cloud experiments, bandwidth and latency are not limited (RTT is  $\sim 0.15$ ms). Each experiment is run 5 times.

**Benchmark Approaches** We compare Starlight to two state-of-the-art approaches: the `containerd` baseline [21] v1.5.0 and eStargz [8, 58] v0.6.3.<sup>6</sup> The **baseline** implementation first downloads and decompresses all new compressed layers before launching the container. **eStargz** presorts the files in each compressed layer according their expected order of use, and uses on-demand “lazy” download during deployment to handle for unexpected accesses: when a running container opens a file whose contents are not yet available, eStargz pauses the container and requests the file from the registry. We also plot two reference times: **warm startup** time denotes the container startup time once its image has already been downloaded and decompressed to local storage; **wget** time denotes the time to compute and download the Starlight delta bundle over the network, serving as a lower bound on provisioning time when not starting containers early.

**Containers** We evaluate Starlight on a variety of popular containers from Docker Hub [30]. Since many of the containers in the original HelloBench container suite [28] are outdated and can no longer be deployed using modern tools, we instead take several of its most popular containers, finally, we add several container images used in edge computing applications. The full list of containers is available in Appendix A.1.

<sup>5</sup>This is the official Docker registry server [30, 49].

<sup>6</sup>We do not compare to Slacker [28] as its source is not public and since eStargz is explicitly designed to supersede it in performance and features. Similarly, our preliminary experiments showed eStargz offers similar or superior performance to DADI [37].

## 5.2 Provisioning Time

Figure 4 shows the average normalized provisioning time for all the containers in Table 2 across a range of round-trip times (RTT) and network bandwidths. We normalize the provisioning time of each container to the time it takes to deploy a fresh worker using the baseline approach over a 100Mbps network with 0.15ms RTT. We also show the 95% confidence intervals to help establish statistical significance [12].

Our first immediate observation is that Starlight is the fastest provisioning approach for all latencies, bandwidths, and scenarios we study, except when provisioning fresh workers in the cloud, where Starlight provides similar performance to eStargz. It is significantly faster than both the state-of-the-art baseline approach and eStargz. Overall, Starlight provisioning is  $3.0\times$  faster on average than the baseline, and  $1.9\times$  faster than eStargz. Surprisingly, Starlight also frequently outperforms wget. In other words, Starlight early start design and effective scheduling of file payloads allows it to provision a fresh worker faster than the time it takes to merely download an optimized package. Conversely, eStargz, which also starts containers early, is on average slower than wget except when bandwidth is 54Mbps and RTT is low. Neither early start nor building optimized container images is sufficient in isolation; Starlight effectiveness is the result of its holistic design.

**Effect of Latency** When RTT is very low (i.e., inside a single datacenter), Both Starlight and eStargz are significantly faster than the baseline. However eStargz scales poorly when RTT grows due to its pull-based design that requests “out-of-order” files on-demand (§3). As latency grows, delays due to these requests add up: eStargz’s provisioning time at RTT=300ms grows by  $3.7\times$  when going from RTT of 0ms to 300ms on a 500Mbps network. In comparison, the baseline provision time only doubles. For high bandwidth, high latency networks (e.g., satellite links) eStargz performance is close to the baseline approach, especially for updates.

Starlight, on the other hand, is far less sensitive to latency than the other approaches: its provisioning time grows at a slower rate than the baseline, eStargz, and wget. Starlight scales well not because its prediction of file access order is perfect (it is not), but rather due to its push-based design. Unlike eStargz, Starlight avoids flooding the registry with HTTP requests when containers open files “out of order”, and instead waits for the file to arrive.

**Deploying Updates** Since updates are a common operation (§3.3), we also consider the provisioning time for updating containers on existing workers.

Starlight is very successful in optimizing updates: provisioning updates using Starlight (bottom row of Figure 4) is on average  $1.7\times$  faster than an equivalent fresh deployment (top row) using Starlight, and  $2.5\times$  faster than baseline fresh deployment.<sup>7</sup> The other approaches only show modest

<sup>7</sup>Harmonic mean of speedups across all bandwidths and latencies.

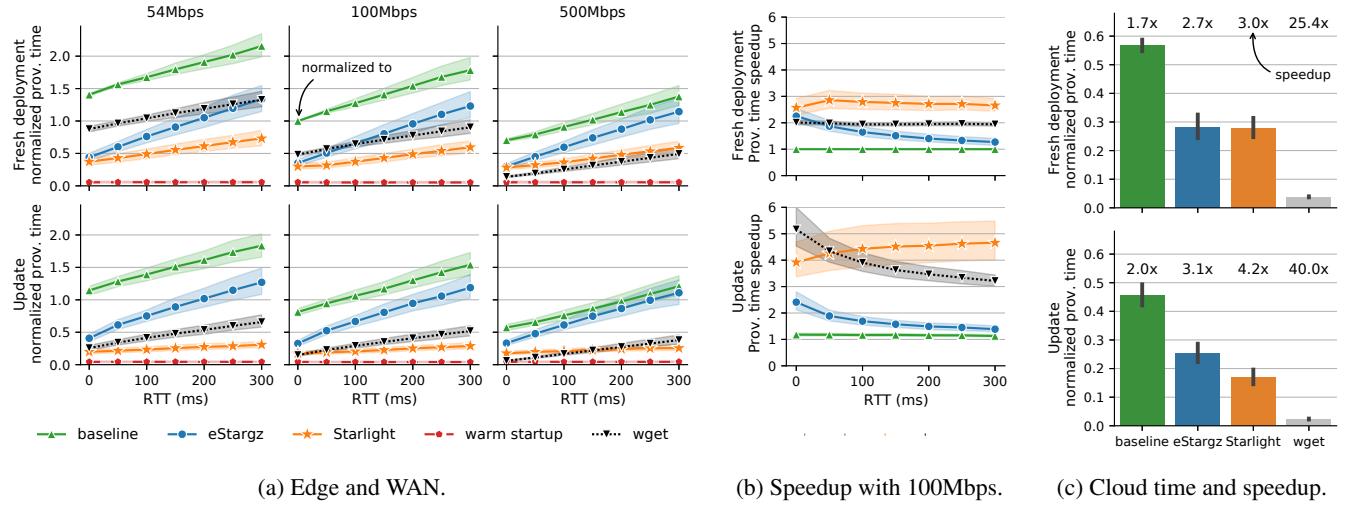


Figure 4: (a) Normalized provisioning time for different methods, round-trip times, and network bandwidth, aggregated across containers in Table 2. Solid line shows the geometric mean [19]; shaded areas show 95% confidence intervals. Top row shows fresh deployment, bottom row shows updates. Time is normalized to fresh deployment of the same container using the baseline approach with RTT of 0ms and a 100Mbps connection. (b) Speedups over baseline with 100Mbps (solid line shows harmonic mean). (c) Provisioning time and speedup in the cloud (RTT approximately  $\sim 0.15$ ms, no bandwidth restriction).

improvement when provisioning updates: average update provisioning time for baseline and eStargz are close to those of fresh deployment. Additionally, we observe that Starlight update provisioning scales much better than the two other approaches as RTT grows. Finally, Starlight’s transfer volume is smaller: the size of a median Starlight update is 30% that of a fresh update using the size of a baseline fresh deployment, while for eStargz and the baseline updates are 99% (figure omitted for space).

As we discuss in §3, layer reuse is low in real-world containers, and even the on-demand “lazy” approach of eStargz must still fetch file metadata from all layers. Conversely, Starlight optimizes updates at a finer file-level granularity, and also stores all file metadata at the beginning of the delta bundle. The result is that Starlight is much better able to exploit redundancy in updates, significantly outperforming the benchmark approaches.

**Effect of Bandwidth** Can increasing bandwidth help mitigate slow provisioning time? We find that higher bandwidth does not provide a corresponding improvement in provisioning time at higher RTT, even for the baseline approach at 0.15ms. This is not surprising: container provisioning is not purely bandwidth-bound task, since we must also decompress and start containers.

**Very low bandwidth** We repeated our experiments with a 5Mbps network. At such low bandwidth, transmission time overwhelms the effect of latency: normalized provisioning time for fresh deployments is 9–10.5 $\times$  higher (compared to 100Mbps network with 0.15ms RTT) for baseline and wget, while eStargz and Starlight reduce it to 2.5–4 $\times$ . For up-

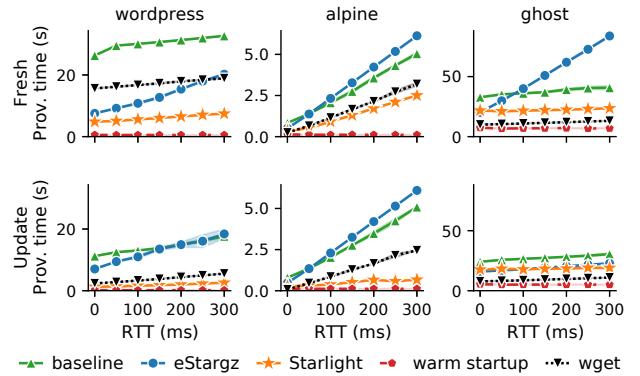


Figure 5: Provisioning times versus round-trip latency for selected containers. Shaded areas show standard deviation.

dates, the baseline is 8–9 $\times$ , wget and eStargz are 2.5–3 $\times$ , and Starlight the fastest at 1 $\times$  across the range of RTT values.

Interestingly, the network is so slow that Flink class loader times out when opening one of the class files when provisioning with eStargz and Starlight. This is the only case we have found of timeout due to on-demand downloading. Indeed, such timeouts are very rare in practice since most software does not timeout on read-only `open()` calls, and software that does must handle timeouts correctly to function with NFS mounts and other distributed filesystems. Nevertheless, we could mitigate such issues by automatically or manually sorting these files earlier in the delta bundle. Starlight’s on-demand optimizer makes this straightforward.

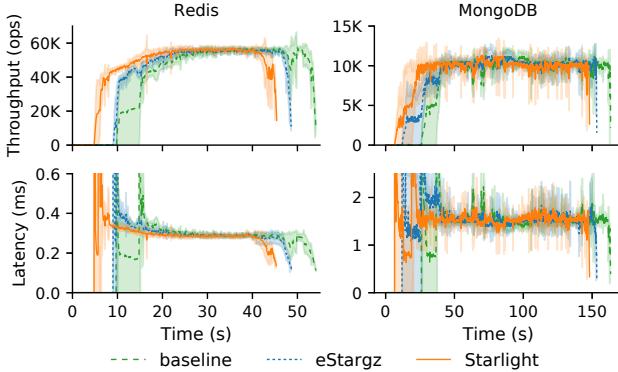


Figure 6: Redis (left) and MongoDB (right) performance during provisioning. Shaded areas show standard deviation.

**Individual Analysis** Figure 5 shows provisioning time of selected containers across a range of latencies at 100Mbps.

We find that eStargz is bottlenecked by queuing delays caused by on-demand file downloads, and can be slower than even the baseline for RTT over 50ms. Starlight outperforms both except for ghost at 0ms, which is the worst case for Starlight: a 84K file container whose delta bundle takes 3 seconds to build. See Appendix A.2 for in-depth analysis.

### 5.3 Performance

We measure application, worker, and proxy performance. Unless otherwise noted, proxy-worker RTT is set to 150ms.

**Application Performance** Ideally, containers deployed using Starlight would exhibit similar application performance as those deployed using the baseline approach, especially during provisioning when Starlight is decompressing files.

To confirm this, we measure application performance for two databases: Redis (in-memory) and MongoDB (disk-based). We run YCSB [9] Workload A (50% read/write ratio) on a separate m5a.1 large instance as the client while we perform a fresh deployment the containerized application, and measure the throughput and read latency of database operations. We repeat each experiment 5 times; each run consists of 2 million database operations, long enough sufficient to finish provisioning and for application performance to stabilize.

Figure 6 depicts throughput and latency over time for both applications. With Starlight, the worker starts handling requests and finishes processing workload earlier than with the other two methods. Additionally, it reaches the same maximum throughput and minimum query latency.

In summary, Starlight workers exhibit no performance overhead compared to the baseline approach and eStargz, and moreover the time gained by early provisioning directly translates to finishing jobs faster.

**Worker CPU Usage and Memory** We measured the total CPU time used by the snapshotter and containerd daemons

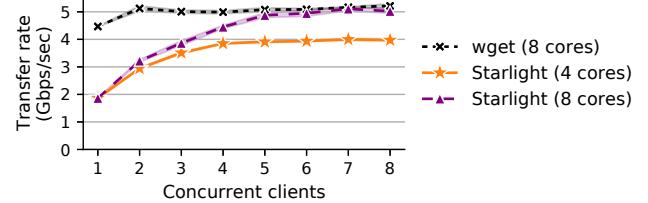


Figure 7: Scalability of Starlight proxy as the achieved transfer rate for different number of concurrent workers. Network bandwidth is capped at 5Gbps, and RTT is  $\sim 0.15$ ms.

during provisioning of containers in Table 2. CPU usage is largely determined by image size, up to 40 seconds of CPU time for the largest image. Median CPU time was 12 seconds for the baseline, 15.1 seconds for eStargz, and 9.8 seconds for Starlight, since it is more effective in removing cross-layer duplicate files. This is consistent with our finding that containerized application exhibit no performance overhead.

Starlight memory usage, measured as total maximum resident set size of the snapshotter and containerd daemons, is linear in the number of files (140MB plus 9.5KB per file,  $R^2=0.784$ ) since it maintains file metadata (§4.5 and §4.6). Memory use for both Starlight and eStargz is similar, ranges from under 200MB for most containers to 1GB for ghost – a massive container image with over 84K files. A recent analysis [64] finds that the median container image has 1,090 files, while 70% of images have less than 20,000 files – approximately 330MB for Starlight.

**Optimization Time** Optimizing the delta bundle is by far the most computationally intense operation for the proxy. We find we can compute delta bundles for images of up to 30K files in under one second (figure omitted for space), which includes most of Table 2; the sole exception is ghost at 84K files, which takes three seconds. Similarly, 80% of the container images in Docker Hub [64] have fewer than 30K files, and could therefore be processed within one second. Finally, the time to build delta bundle could be eliminated completely for common deployments by placing a cache in front of the Starlight proxy; we do not do so in any of our experiments.

**Scalability** We use Apache Benchmark to measure the achievable transfer rate of the Starlight proxy as we increase the number of concurrent clients repeatedly requesting the Redis delta bundle (36.8MB). This is equivalent to the common setup where hundreds of simultaneous Starlight worker requests are load-balanced across multiple replicas of the proxy, and the goal is to saturate the bandwidth – if the proxy is network bound, we are serving as many clients as the network supports. For this experiment, we run with no artificial bandwidth or latency limits. For reference, we request an image of equivalent size from an nginx webserver. Figure 7 shows a Starlight proxy running on a 4-core instance is able to saturate about 80% of the link bandwidth before becoming

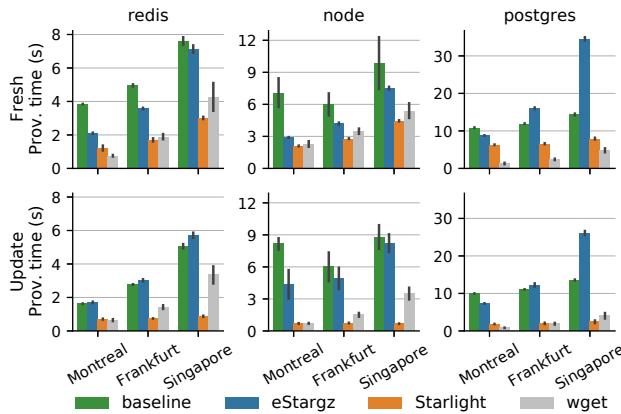


Figure 8: Provisioning time when moving the worker between different datacenters. Errors bars show standard deviation. The registry is located in North Virginia.

bottlenecked due to the need to optimize the delta bundle. Once we switch to 8 cores, it becomes network bound.

## 5.4 Geo-Distributed WAN Experiment

Thus far we have evaluated Starlight using controlled experiments in a single AWS datacenter. Here, we evaluate Starlight performance in a multi-cloud (wide area network) setup running in multiple datacenters over the real network. We place the registry in us-east-1 region (N. Virginia) and move the worker to increasingly distant locations: ca-central-1 (average RTT to registry 14ms, bandwidth 4.24Gbps), eu-central-1 (89ms, 2.79Gbps), and ap-southeast-1 (209ms, 1.15Gbps).

Figure 8 shows the provisioning time for fresh and update deployment. Results generally match our previous observations: Starlight substantially outperforms the baseline and eStargz, and in many cases is faster than a simple wget of the delta bundle. eStargz is sensitive to increased latency, in some cases becoming slower than the baseline approach. Finally, Starlight support for container updates is much more effective than the other approaches, and can reduce provisioning time to a fraction of the other approaches.

## 6 Related Work

There are several streams of work on container provisioning.

**On-Demand Download** Slacker [28] starts containers early and uses NFS to load files on-demand without requiring the entire container image. CRFS [27] follows a similar idea, but uses a seekable tar gzip format with more efficient compression, allowing it to work with standard registries. DADI [37] also uses on-demand fetching but operates at the block level, which requires a customized image format and registry. eStargz [58] uses collected filesystem traces to identify files

needed during provisioning and prefetch them first, before switching to on-demand downloading. Starlight also sorts files based on collected traces, but its push-based design scales better with higher latency. Moreover, Starlight’s protocol is file-based rather than layer-based as prior approaches.

**Peer-to-peer** Some approaches use workers to help provision other workers, Wharf [65] and Shifter [26] propose client-side image sharing: workers act as caches, serving locally stored images to other workers. FID [32], CoMiCon [44], and Kraken [31] are P2P docker registries that help reduce registry load by utilizing the bandwidth of workers in the datacenter. Similarly, FaaSNet [60] uses a tree of workers to accelerate provisioning inside datacenters for scaling Function-as-a-Service workloads inside a datacenter. These approaches tend to focus on single datacenter setting with the goal of reducing registry load. They may not be applicable outside the datacenter or where bandwidth and other worker resources are limited. Conversely, Starlight is focused on accelerating provisioning without increasing worker load.

**Registry optimizations** Fu et al. [25] and Anwar et al. [4] propose smart caching and prefetching image layers from the back-end object store to the registry using the production workload, in order to do large scale software provisioning. Starlight is orthogonal to, and compatible with, these works since it does not require changing the registry.

## 7 Conclusion

Containers have evolved in a single datacenter environment, but are increasingly used in geo-distributed settings such as edge, mobile, and multi-cloud environments. We revisit several of the design decisions behind containers, and show that while they are convenient for developers, they slow down provisioning. Starlight redesigns the provisioning pipeline to support faster container deployment, while maintaining the layer-based structure that makes containers easy to develop and maintain. Empirical evaluation using a large set of popular containers shows Starlight provisioning times are significantly smaller than existing approaches, while incurring no performance overhead. Moreover, Starlight is backwards compatible and makes use of existing registries. Starlight is available as an open-source project at: <https://github.com/mc256/starlight>.

Starlight’s design opens several avenues for improvement. For example, since the delta bundle is optimized on-demand, we can improve it and even tailor it to specific scenarios by collecting traces online during deployment, or by training an ML model to predict which files will be needed first. Another improvement is support for repurposing workers: by modifying the optimizer and extending the delta bundle design, we could optimize switching between arbitrary sets of containers.

## References

- [1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.
- [2] Werner Almesberger. Linux traffic control - implementation overview. Technical report, EPFL ICA, 1998.
- [3] Amazon. Amazon Elastic Container Service (Amazon ECS). <https://aws.amazon.com/ecs/>.
- [4] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nan-nan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, February 2018. USENIX Association.
- [5] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.
- [6] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *16th Systems Administration Conference (LISA 02)*, Philadelphia, PA, November 2002. USENIX Association.
- [7] Containerd. Snapshots design. <https://github.com/containerd/containerd/blob/main/design/snapshots.md>.
- [8] Containerd. Stargz snapshotter. <https://github.com/containerd/stargz-snapshotter>.
- [9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC ’10*, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [10] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. *Surrounded by the Clouds: A Comprehensive Cloud Reachability Study*, page 295–304. Association for Computing Machinery, New York, NY, USA, 2021.
- [11] Breno Costa, Joao Bachiega, Leonardo Rebouças de Carvalho, and Aleteia P. F. Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Comput. Surv.*, 55(2), January 2022.
- [12] Geoff Cumming, Fiona Fidler, and David L. Vaux. Error bars in experimental biology . *Journal of Cell Biology*, 177(1):7–11, 04 2007.
- [13] Richard Cziva and Dimitrios P. Pezaros. Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.
- [14] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20*, page 939–954, New York, NY, USA, 2020. Association for Computing Machinery.
- [15] Docker. Best practices for writing dockerfiles. [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/).
- [16] Docker. Docker compose. <https://github.com/docker/compose>.
- [17] Docker. Docker documentation. <https://docs.docker.com/engine/reference/commandline/dockerd/>.
- [18] Docker. Empowering app development for developers | docker. <https://www.docker.com/>.
- [19] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.
- [20] The Linux Foundation. Cloud native computing foundation. <https://cncf.io>.
- [21] The Linux Foundation. containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. <https://containerd.io/>.
- [22] The Linux Foundation. K3s: Lightweight kubernetes. <https://k3s.io>.
- [23] The Linux Foundation. Kubernetes. <https://kubernetes.io/>.
- [24] The Linux Foundation. Open container initiative. <https://opencontainers.org/>.

- [25] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.
- [26] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for HPC. In *Journal of physics: Conference series*, volume 898, page 082021. IOP Publishing, 2017.
- [27] Google. CRFS: Container registry filesystem. <https://github.com/google/crfs>.
- [28] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpacı-Dusseau, and Remzi H. Arpacı-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association.
- [29] Stephen Hemminger. Network emulation with NetEm. In *Linux Conf Australia*, pages 18–23, 2005.
- [30] Docker Inc. Docker Hub: Container image library | app containerization. <https://registry.hub.docker.com/>.
- [31] Uber Inc. Kraken - p2p docker registry capable of distributing tbs of data in seconds. <https://github.com/uber/kraken>.
- [32] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. Fid: A faster image distribution system for docker platform. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self\* Systems (FAS\*W)*, pages 191–198, 2017.
- [33] The kernel development community. Fuse the linux kernel documentation. <https://www.kernel.org/doc/html/latest/filesystems/fuse.html>.
- [34] kernel.org. Overlay filesystem – the linux kernel documentation. <https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html>.
- [35] Petros Koutoupis. Everything you need to know about Linux containers, part i: Linux control groups and process isolation. *Linux Journal*, 2018, 2018.
- [36] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’17*, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.
- [37] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: Block-level image service for agile and elastic application deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 727–740. USENIX Association, July 2020.
- [38] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP ’17*, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.
- [39] Scott McCarty. A practical introduction to container terminology, 2018. <https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction>.
- [40] Isla Mcketta. How Starlink’s satellite internet stacks up against HughesNet and Viasat around the globe, 2021. <https://www.speedtest.net/insights/blog/starlink-hughesnet-viasat-performance-q2-2021/>.
- [41] Microsoft. Azure container instances. <https://azure.microsoft.com/en-us/services/container-instances/>.
- [42] Seyed Hossein Mortazavi, Mohammad Salehe, Moshe Gabel, and Eyal de Lara. Feather: Hierarchical querying for the edge. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 271–284, 2020.
- [43] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing, SEC ’17*, New York, NY, USA, 2017. Association for Computing Machinery.
- [44] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A co-operative management system for docker container images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 116–126, 2017.
- [45] Mahesh Nayak, Kumud Dwivedi, and Cheryl McGuire. Azure network round-trip latency statistics, 2021. <https://docs.microsoft.com/en-us/azure/networking/azure-network-latency>.

- [46] The Containers Organization. Buildah: a tool that facilitates building open container initiative (oci) container images. <https://buildah.io/>.
- [47] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. Toward lighter containers for the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.
- [48] Valerio Persico, Alessio Botta, Pietro Marchetta, Antonio Montieri, and Antonio Pescap. On the performance of the wide-area networks interconnecting public-cloud datacenters around the globe. *Comput. Netw.*, 112(C):67–83, January 2017.
- [49] CNCF Distribution Project. Distribution - the toolkit to pack, ship, store, and deliver container content. <https://github.com/distribution/distribution>.
- [50] Prashanth Rajivan, Efrat Aharonov-Majar, and Cleotilde Gonzalez. Update now or later? effects of experience, cost, and risk preference on update decisions. *Journal of Cybersecurity*, 6(1):tyaa002, 2020.
- [51] Brian Ramprasad, Alexandre da Silva Veith, Moshe Gabel, and Eyal de Lara. Sustainable computing on the edge: A system dynamics perspective. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile ’21, page 64–70, New York, NY, USA, 2021. Association for Computing Machinery.
- [52] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.
- [53] J. Shah and D. Dubaria. Building modern clouds: Using Docker, Kubernetes & Google Cloud Platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019.
- [54] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
- [55] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.
- [56] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association.
- [57] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortezaei, Moshe Gabel, and Eyal de Lara. Reconfigurable streaming for the mobile edge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile ’19, page 153–158, New York, NY, USA, 2019. Association for Computing Machinery.
- [58] Kohei Tokunaga. Startup containers in lightning speed with lazy image distribution on containerd, Apr 2020.
- [59] B. Varghese, E. De Lara, A. Ding, C. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, and P. Willis. Revisiting the arguments for edge computing research. *IEEE Internet Computing*, (01):1–1, jun 5555.
- [60] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaS-Net: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.
- [61] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.
- [62] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.
- [63] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt. Slimmer: Weight loss secrets for Docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519, 2019.
- [64] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali R. Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, Sep. 2019.
- [65] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’18, page 174–185, New York, NY, USA, 2018. Association for Computing Machinery.

## A Appendix

### A.1 Container Images Used in Evaluation

Table 2 below lists containers and version tags used in our experiments; combined, they have over 15 billion downloads in Docker Hub.

Category	Images
Linux	alpine:3.13.4 ubuntu:focal-20210401
Web	memcached:1.6.8 nginx:1.19.10 httpd:2.4.43
Data	mysql:8.0.23 mariadb:10.5.8 redis:6.2.1 mongo:4.0.23 postgres:13.1 rabbitmq:3.8.13
Services	registry:2.7.0 wordpress:php7.3-fpm ghost:3.42.5-alpine
Dev	node:16-alpine3.11 openjdk:11.0.11-9-jdk golang:1.16.2 python:3.9.3
Edge	flink:1.12.3-scala_2.11-java8 cassandra:3.11.9 eclipse-mosquitto:2.0.9-openssl

Table 2: Container images used in our evaluation.

### A.2 Analysis of Selected Containers

Figure 5 shows provisioning time of selected containers across a range of latencies at 100Mbps.

When updating wordpress, the baseline approach is able to reuse 4 out of 18 layers, making it faster in update. eStargz, though faster than the baseline approach in fresh deployments, does not benefit much from this layer reuse since it is bottlenecked by on-demand file downloads. Starlight, on the other hand, is much faster than either approach, reducing update provisioning time by approximately 8×.

For alpine eStargz is slower than the baseline when RTT is above 50ms. This is because the alpine image is small and its file access pattern is not entirely deterministic. Provisioning time is thus dominated by queuing delays due layer downloads and on-demand file downloads. Starlight also suffers somewhat from out-of-order file accesses, but is still able to deploy the container quickly, and is even faster than wget.

Finally, we discuss ghost – the worst case for Starlight. Starlight’s provisioning time with low RTT is 10% higher

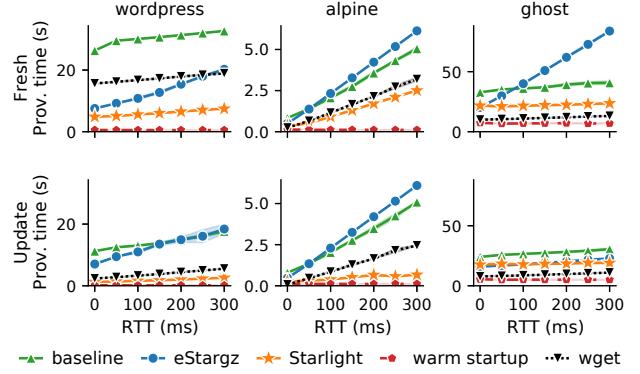


Figure 5: Provisioning times versus round-trip latency for selected containers. Shaded areas show standard deviation. (figure repeated from page 10)

than eStargz’s – the only container where this happens. Building a delta bundle takes 3 seconds for this 84K file container. eStargz provisioning time grows quickly with latency, however, and Starlight outperforms it when RTT is above 50ms.

# POWERTCP: Pushing the Performance Limits of Datacenter Networks\*

Vamsi Addanki

*TU Berlin*

*University of Vienna*

Oliver Michel

*Princeton University*

*University of Vienna*

Stefan Schmid

*TU Berlin*

*University of Vienna*

## Abstract

Increasingly stringent throughput and latency requirements in datacenter networks demand fast and accurate congestion control. We observe that the reaction time and accuracy of existing datacenter congestion control schemes are inherently limited. They either rely only on explicit feedback about the network state (e.g., queue lengths in DCTCP) or only on variations of state (e.g., RTT gradient in TIMELY). To overcome these limitations, we propose a novel congestion control algorithm, POWERTCP, which achieves much more fine-grained congestion control by adapting to the bandwidth-window product (henceforth called power). POWERTCP leverages in-band network telemetry to react to changes in the network instantaneously without loss of throughput and while keeping queues short. Due to its fast reaction time, our algorithm is particularly well-suited for dynamic network environments and bursty traffic patterns. We show analytically and empirically that POWERTCP can significantly outperform the state-of-the-art in both traditional datacenter topologies and emerging reconfigurable datacenters where frequent bandwidth changes make congestion control challenging. In traditional datacenter networks, POWERTCP reduces tail flow completion times of short flows by 80% compared to DCQCN and TIMELY, and by 33% compared to HPCC even at 60% network load. In reconfigurable datacenters, POWERTCP achieves 85% circuit utilization without incurring additional latency and cuts tail latency by at least 2x compared to existing approaches.

## 1 Introduction

The performance of more and more cloud-based applications critically depends on the underlying network, requiring datacenter networks (DCNs) to provide extremely low latency and high bandwidth. For example, in distributed machine learning applications that periodically require large data transfers, the network is increasingly becoming a bottleneck [36]. Similarly, stringent performance requirements are introduced by today’s trend of resource disaggregation in datacenters where fast access to remote resources (e.g., GPUs or memory) is pivotal

for the overall system performance [36]. Building systems with strict performance requirements is especially challenging under bursty traffic patterns as they are commonly observed in datacenter networks [12, 16, 47, 53, 55].

These requirements introduce the need for fast and accurate network resource management algorithms that optimally utilize the available bandwidth while minimizing packet latencies and flow completion times. Congestion control (CC) plays an important role in this context being “a key enabler (or limiter) of system performance in the datacenter” [34]. In fact, fast reacting congestion control is not only essential to efficiently adapt to bursty traffic [29, 48], but is also becoming increasingly important in the context of emerging reconfigurable datacenter networks (RDCNs) [13, 14, 20, 33, 38, 39, 50]. In these networks, a congestion control algorithm must be able to quickly ramp up its sending rate when high-bandwidth circuits become available [43].

Traditional congestion control in datacenters revolves around a bottleneck link model: the control action is related to the state i.e., queue length at the bottleneck link. A common goal is to efficiently control queue buildup while achieving high throughput. Existing algorithms can be broadly classified into two types based on the feedback that they react to. In the following, we will use an analogy to electrical circuits<sup>1</sup> to describe these two types. The first category of algorithms react to the absolute network state, such as the queue length or the RTT: a function of network “effort” or **voltage** defined as the sum of the bandwidth-delay product and in-network queuing. The second category of algorithms rather react to variations, such as the change of RTT. Since these changes are related to the network “flow”, we say that these approaches depend on the **current** defined as the total transmission rate. We tabulate our analogy and corresponding network quantities in Table 1. According to this classification, we call congestion control protocols such as CUBIC [21], DCTCP [7], or Vegas [15] **voltage-based CC** algorithms as

<sup>1</sup>This analogy is inspired from S. Keshav’s lecture series based on mathematical foundations of computer networking [31]. We emphasize that our power analogy is meant for the networking context considered in this paper and it should not be applied to other domains of science.

\*Research was conducted at the University of Vienna during 2020-21.

Quantity	Analogy
Total transmission rate (network flow)	Current ( $\lambda$ )
BDP + buffered bytes (network effort)	Voltage (v)
$\text{Current} \times \text{Voltage}$	Power ( $\Gamma$ )

Table 1: Analogy between metrics in networks and in electrical circuits. Note that the network here is the “pipe” seen by a flow and not the whole network.

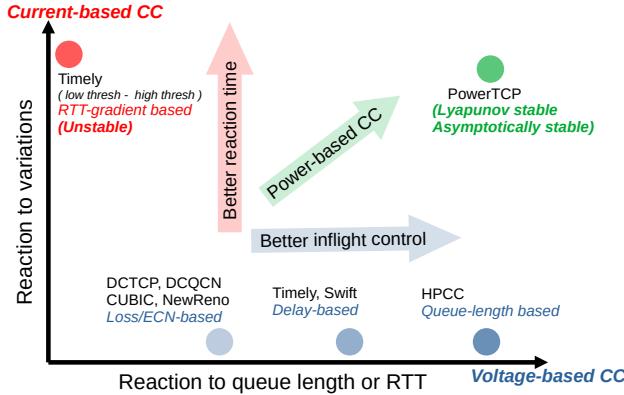


Figure 1: Existing congestion control algorithms are fundamentally limited to a single dimension in their window (or rate) update decisions and are unable to distinguish between two scenarios across multiple dimensions.

they react to absolute properties such as the bottleneck queue length, delay, Explicit Congestion Notification (ECN), or loss. Recent proposals such as TIMELY [41] are **current-based CC** algorithms as they react to the variations, such as the RTT-gradient. In conclusion, we find that existing congestion control algorithms are fundamentally limited to one of the two dimensions (voltage or current) in the way they update the congestion window.

We argue that the input to a congestion control algorithm should rather be a function of the two-dimensional state of the network (i.e., both voltage and current) to allow for more informed and accurate reaction, improving performance and stability. In our work, we show that there exists an accurate relationship between the optimal adjustment of the congestion window, the network voltage and the network current. We analytically show that the optimal window adjustment depends on the product of network voltage and network current. We call this product **network power**: current  $\times$  voltage, a function of both queue lengths and queue dynamics.

Figure 1 illustrates our classification. Existing protocols depend on a single dimension, voltage or current. This can result in imprecise congestion control as the protocol is unable to distinguish between fundamentally different scenarios, and, as a result, either reacts too slowly or overreacts, both impeding performance. Accounting for both voltage and current, i.e., power, balances accurate inflight control and fast reaction, effectively providing the best of both worlds.

In this paper we present POWERTCP, a novel *power*-based congestion control algorithm that accurately captures both *voltage* and *current* dimensions for every control action using measurements taken within the network and propagated through in-band network telemetry (INT). POWERTCP is able to utilize available bandwidth within one or two RTTs while being stable, maintaining low queue lengths, and resolving congestion rapidly. Furthermore, we show that POWERTCP is Lyapunov-stable, as well as asymptotically stable and has a convergence time as low as five update intervals (Appendix A). This makes POWERTCP highly suitable for today’s datacenter networks and dynamic network environments such as in reconfigurable datacenters.

POWERTCP leverages in-network measurements at programmable switches to accurately obtain the bottleneck link state. Our switch component is lightweight and the required INT header fields are standard in the literature [36]. We also discuss an approximation of POWERTCP for use with non-programmable, legacy switches.

To evaluate POWERTCP, we focus on a deployment scenario in the context of RDMA networks where the CC algorithm is implemented on a NIC. Our results from large-scale simulations show that POWERTCP reduces the 99.9-percentile short flow completion times by 80% compared to DCQCN [56] and by 33% compared to the state-of-the-art low-latency protocol HPCC [36]. We show that POWERTCP maintains near-zero queue lengths without affecting throughput or incurring long flow completion times even at 80% load. As a case study, we explore the benefits of POWERTCP in reconfigurable datacenter networks where it achieves 80 – 85% circuit utilization and reduces tail latency by at least 2 $\times$  compared to the state-of-the-art [43]. Finally, as a proof-of-concept, we implemented POWERTCP in the Linux kernel and the telemetry component on an Intel Tofino programmable line-rate switch using P4 [18].

In summary, our key contributions in this paper are:

- We reveal the shortcomings of existing congestion control approaches which either only react to the current state or the dynamics of the network, and introduce the notion of *power* to account for both.
- POWERTCP, a power-based approach to congestion control at the end-host which reacts faster to changes in the network such as an arrival of burst, fluctuations in available bandwidth etc.,
- An evaluation of the benefits of POWERTCP in traditional DCNs and RDCNs.
- As a contribution to the research community and to facilitate future work, all our artefacts have been made publicly available at:  
<https://powertcp.self-adjusting.net>.

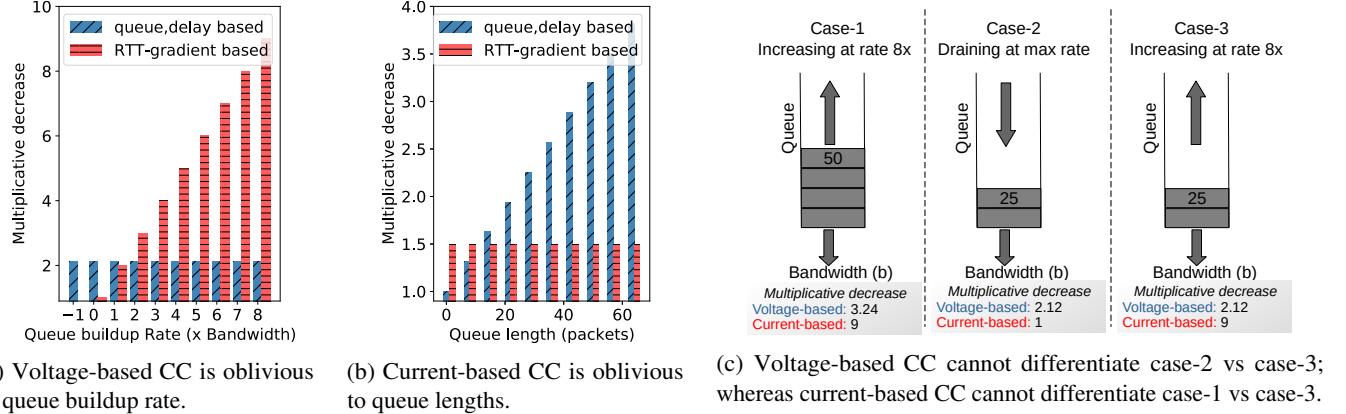


Figure 2: Existing CC schemes, classified as voltage and current-based, are orthogonal in their response to queue length and queue buildup rate.

## 2 Motivation

We first provide a more detailed motivation of our work by highlighting the benefits and drawbacks of existing congestion control approaches. In the following, **voltage-based CC** refers to the class of end-host congestion control algorithms that react to the state of the network in absolute values related to the bandwidth-delay product, such as bottleneck queue length, delay, loss, or ECN; **current-based CC** refers to the class of algorithms that react to changes in the state, such as the RTT-gradient. Voltage-based CC algorithms are likely to exhibit better stability but are fundamentally limited in their reaction time. Current-based CC algorithms detect congestion faster but ensuring stability may be more challenging. Indeed, **TIMELY** [41], a current-based CC, deployed at Google datacenters, turned out to be unstable [57] and evolved to **SWIFT** [34], a voltage-based CC.

Orthogonal to our approach, receiver-driven transport protocols [22, 26, 42] have been proposed which show significant performance improvements. A receiver-driven transport approach relies on the assumption that datacenter networks are well-provisioned and claims that congestion control is unnecessary; for example “NDP performs no congestion control whatsoever in a Clos topology” [22]. The key difference is that receiver-driven approaches take feedback from the ToR downlink at the receiver which can only identify congestion at the last hop, whereas sender-based approaches rely on a variety of feedback signals to identify congestion anywhere along the path. In this paper, we focus on the sender-based congestion control approach which can in principle handle congestion anywhere along the round-trip path between a sender and a receiver, even in oversubscribed datacenters.

To take a leap forward and design fine-grained datacenter congestion control algorithms, we present an analytical approach and study the fundamental problems faced by existing algorithms. We first formally express the desirable properties of a datacenter congestion control law (§2.1) and then analytically identify the drawbacks of existing control laws

(§2.2). Finally, we discuss the lessons learned and formulate our design goals (§2.3).

### 2.1 Desirable Control Law Properties

Among various desired properties of datacenter congestion control, high throughput and low tail latency are most important [7, 36, 41] with fairness and stability being essential as well [54, 57]. Achieving these properties simultaneously can be challenging. For example, to realize high throughput, we may aim to keep the queue length at the bottleneck link large; however, this may increase latency. Thus, an ideal CC algorithm must be capable of maintaining near-zero queue lengths, achieving both high throughput and low latency. It must further minimize throughput loss and latency penalty caused by perturbations, such as bursty traffic.

In order to formalize our requirements, we consider a single-bottleneck link model widely used in the literature [24, 40, 54, 57]. Specifically, we assume that all senders use the same protocol, transmit long flows<sup>2</sup> sharing a common bottleneck link with bandwidth  $b$ , and have a base round trip time  $\tau$  (excluding queuing delays). In this model, equilibrium is a state reached when the window size and bottleneck queue length stabilize. We now formally express the desired equilibrium state that captures our performance requirements in terms of the sum of window sizes of all flows (aggregate window size)  $w(t)$ , bandwidth delay product  $b \cdot \tau$ , and bottleneck queue length  $q(t)$ :

$$0 < q(t) < \varepsilon \quad (1)$$

$$b \cdot \tau \leq w(t) < b \cdot \tau + \varepsilon$$

$$\dot{q}(t) = 0; \dot{w}(t) = 0$$

where  $\varepsilon$  is a positive integer. First, this captures the requirement for high throughput i.e., when  $w(t) > b \cdot \tau$  and  $q(t) > 0$ , the number of inflight bytes are greater than the bandwidth-delay product (BDP) and the queue length is greater than zero.

<sup>2</sup>Note that, although most DC flows are short flows, most DC traffic volume (bytes) is from long flows [7, 9].

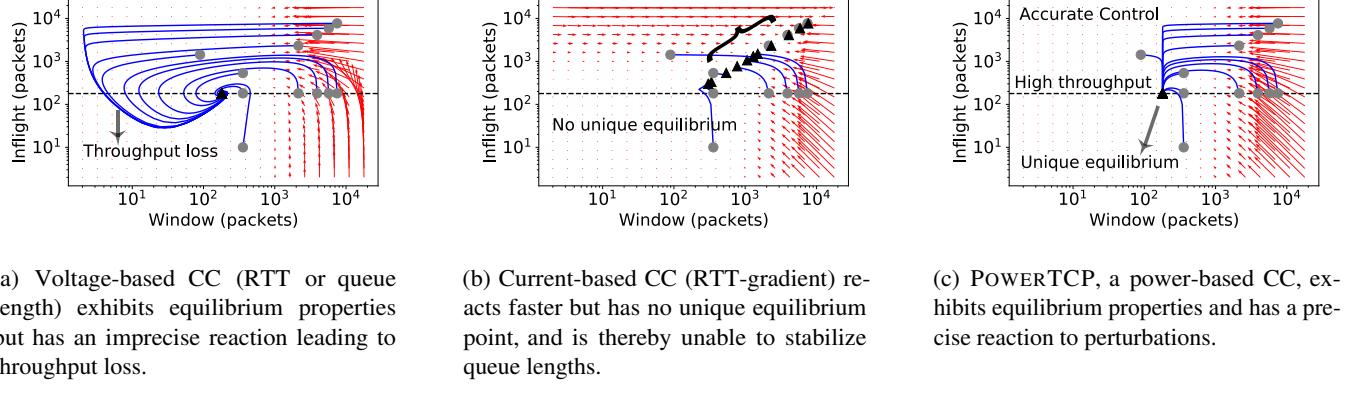


Figure 3: Phase plots showing the trajectories of existing schemes and our approach POWER TCP from different initial states (circles) to equilibrium (triangles). At each point on the plane, arrows show the direction in which the system moves. An example is depicted with bottleneck link bandwidth 100Gbps and a base RTT of 20μs. BDP is shown by a horizontal dotted line and any trajectory going below this line indicates throughput loss.

Second, from  $w(t) < b \cdot \tau + \epsilon$  and  $q(t) < \epsilon$ , the queue length is at most  $\epsilon$ , thereby achieving low latency. Finally, for the system to stabilize, we need that  $\dot{q}(t) = 0$  and  $\dot{w}(t) = 0$ .

As simple as these requirements are, it is challenging to control the aggregate window size  $w(t)$  while CC operates per flow. In addition to the equilibrium state requirement, we need fast response to perturbations. The response must minimize the distance from the equilibrium i.e., minimize the latency or throughput penalty caused by a perturbation (e.g., incast or changes in available bandwidth).

In this work, we ask two fundamental questions:

**(Q1) Equilibrium point:** Do existing algorithms satisfy the equilibrium state in Eq. 1 for the aggregate window size? In addition to the equilibrium behavior, we are also interested in the reaction to a perturbation.

**(Q2) Response to perturbation:** What is the trajectory followed after a perturbation, i.e., the dynamics of the bottleneck queue as well as the TCP window sizes, from an initial point to the equilibrium point?

## 2.2 Drawbacks of Existing Control Laws

We now aim to analytically answer our questions above and shed light on the inefficiencies of existing protocols, both voltage-based and current-based. We begin by simplifying the congestion avoidance model of existing CC approaches we are interested in, specifically delay, queue length, and RTT-gradient based CC approaches as follows:

$$w_i(t + \delta t) = \gamma \cdot \left( w_i(t) \cdot \frac{e}{f(t)} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (2)$$

Here  $w_i$  is the window of a flow  $i$ ,  $\beta$  is the additive increase term,  $e$  is the equilibrium point that the algorithm is expected to reach,  $f(t)$  is the measured feedback and  $\gamma$  is the exponential moving average parameter. A queue length-based CC [36] sets the desired equilibrium point  $e$  as  $b \cdot \tau$  (BDP) and the feedback  $f(t)$  as the sum of bottleneck queue length and BDP i.e.,

voltage ( $v$ ). A delay-based CC [34] sets  $e$  to  $\tau$  (base RTT) and the feedback  $f(t)$  as RTT which is the sum of queuing delay and base RTT i.e.,  $\frac{\text{voltage}}{\text{bandwidth}}$  ( $\frac{v}{b}$ ). Similarly, the RTT-gradient approach [41] sets  $e$  to 1 and the feedback  $f(t)$  as one plus RTT-gradient i.e.,  $\frac{\text{current}}{\text{bandwidth}}$  ( $\frac{\lambda}{b}$ ). In Appendix B, we further justify how Eq. 2 captures existing control laws<sup>3</sup>. Note that our simplified model does not capture loss/ECN-based CC algorithms; however, there exists rich literature on the analysis of loss/ECN-based CC algorithms [24, 37] including DCTCP [7, 8]. We now use Euler's first order approximation to obtain the window dynamics as follows:

$$\dot{w}_i(t) = \frac{\gamma}{\delta t} \cdot \left( w_i(t) \cdot \frac{e}{f(t)} - w_i(t) + \beta \right) \quad (3)$$

Each flow  $i$  has a sending rate  $\lambda_i$  and hence the bottleneck queue experiences an aggregate arrival rate of  $\lambda$ . In our analogy,  $\lambda$  is the network current. We additionally use the traditional model of queue length dynamics which is independent of the control law [24, 40]:

$$\dot{q}(t) = \begin{cases} \lambda(t - t^f) - \mu(t) & q(t) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

where  $\lambda(t) = \frac{w(t)}{\delta t}$ . An equilibrium point is the window size  $w_e$  and queue length  $q_e$  that satisfies  $\dot{w}(t) = 0$  and  $\dot{q}(t) = 0$ .

We are now ready to answer the questions raised.

**Equilibrium point:** It is well-known from literature that loss/ECN-based schemes operate by maintaining a standing queue [8, 24, 27]. For example, TCP NewReno flows fill the queue to maximum (say  $q_{max}$ ) and then react by reducing windows by half. Consequently, the bottleneck queue-length oscillates between  $q_{max}$  and  $q_{max} - b \cdot \tau$  or zero if  $q_{max} < b \cdot \tau$ . DCTCP flows oscillate around the marking threshold  $K > \frac{b \cdot \tau}{7}$

<sup>3</sup>TIMELY, for example, is rate-based while our simplification is window-based. However, window and rate are interchangeable for update calculations.

which depends on BDP [7]. This does not satisfy our stringent requirement in Eq. 1. While ECN-based schemes reduce the amount of standing queue required, we still consider the standing queue which is proportional to bandwidth to be unacceptable given the increasing gap between bandwidth vs switch buffers.

It can be shown that there exists a unique equilibrium point for queue length and delay approaches (voltage-based CC) defined by Eq. 2. However, current-based CC and, in particular, RTT-gradient approaches do not have a unique equilibrium point suggesting a lack of control over queue lengths. Intuitively, RTT-gradient approaches quickly adapt the sending rate to stabilize the RTT-gradient ( $\dot{\theta} = \frac{\dot{q}}{b}$ ) which in turn only stabilizes the queue length gradient  $\dot{q}(t)$  but fails to control the absolute value of the queue length. It has indeed been shown that TIMELY, a current-based CC does not have a unique equilibrium [57].

Figure 3 visualizes the system behavior according to the window dynamics in Eq. 3 and the queue dynamics in Eq. 4. In Figure 3a we can see that voltage-based CC eventually reaches a unique equilibrium point. In contrast, in Figure 3b we see that current-based CC reaches different final points for different initial points, indicating that there exists no unique equilibrium point thereby violating the desired equilibrium state properties (Eq. 1). To give more context on this observation, in Figure 2 we show the reactions of different schemes for observed queue lengths and queue buildup rate. In Figure 2b, we can see that current-based CC has the same reaction for different queue lengths but exhibits a proportional reaction to queue buildup rate (Figure 2a); consequently, current-based CC cannot stabilize at a unique equilibrium point. Due to space constraints, we move the detailed proof of equilibrium points to Appendix B.

**Takeaway.** While voltage-based CC can in principle meet the desired equilibrium state requirements in Eq. 1, current-based CC cannot.

**Response to perturbation:** We observe an orthogonal behavior in the responses of voltage-based CC and current-based CC. In Figure 2b we show that voltage-based CC has a proportional reaction to increased queue lengths but a current-based CC approach has the same response for any queue length. Further in Figure 2a we observe that current-based CC has a proportional reaction to the rate at which queue is building up but a voltage-based CC has the same reaction for any rate of queue build up. This orthogonality in existing schemes often results in scenarios with either insufficient reaction or overreaction. To underline our observation, we use the system of differential equations (Eq. 3 and Eq. 4) to observe the trajectories taken by different control laws after a perturbation. We show the trajectories in Figure 3. Specifically, Figure 3a shows that voltage-based CC (queue length or delay based) eventually reaches a unique equilibrium point but overreacts in the response and losing throughput (window  $<$  BDP and  $q(t) = 0$ ) almost for every initial point. In Figure 3b we observe that current-based CC (RTT-gradient) reaches different

end points for different initial states and consequently does not have a single equilibrium point. However, we see that the initial response is faster with current-based CC due to their use of RTT-gradient which is arguably a superior signal to detect congestion onset even at low queue lengths.

**Takeaway.** Current-based CC is superior in terms of fast reaction but lacks equilibrium state properties while voltage-based CC eventually reaches a unique equilibrium but overreacts in its response for almost any initial state resulting in long trajectories from initial state to equilibrium state.

### 2.3 Lessons Learned and Design Goals

From our analysis we derive two key observations. First, both voltage and current-based CC have individual benefits. Particularly, voltage-based CC is desirable for the stringent equilibrium properties we require and current-based CC is desirable for fast reaction. Second, both voltage and current-based CC have drawbacks. On one hand, voltage-based CC is oblivious to congestion onset at low queue lengths and on the other hand current-based CC is oblivious to the absolute value of queue lengths. Moreover, voltage-based CC overreacts when the queue drains essentially losing throughput immediately after.

Based on these observations, our goal is to design a control law that systematically combines both voltage and current for every window update action. Specifically our aim is to design a congestion control algorithm with (i) equilibrium properties from Eq. 1 exhibited by voltage-based CC and (ii) fast response to perturbation exhibited by current-based CC. The challenges are to avoid inheriting the drawbacks of both types of CC, stability and fairness. However in order to design such a control law we face the following challenges:

- Finding an accurate relationship between window, voltage and current. ▷ Property 1
- Ensuring stability, convergence and fairness. ▷ Theorem 1, 2, 3

## 3 Power-Based Congestion Control

Reflecting on our observations in §2, we seek to design a congestion control algorithm that systematically reacts to both the absolute value of the bottleneck queue length and its rate of change. Our aim is to address today’s datacenter performance requirements in terms of high throughput, low latency, and fast reaction to bursts and bandwidth fluctuations.

### 3.1 The Notion of Power

To address the challenges faced by prior datacenter congestion control algorithms and to optimize along both dimensions, we introduce the notion of *power* associated with the network pipe. Following the bottleneck link model from literature [24, 40], from Eq. 4 we observe that the window size is indeed related to the product of network voltage and network current which we call *power* (Table 1). This corresponds to the product of (i) total sending rate  $\lambda$  (current) and (ii) the

sum of BDP plus the accumulated bytes  $q$  at the bottleneck link (voltage), formally expressed in Eq. 5.

$$\underbrace{\Gamma(t)}_{\text{power}} = \underbrace{(q(t) + b \cdot \tau)}_{\text{voltage}} \cdot \underbrace{\lambda(t - t^f)}_{\text{current}} \quad (5)$$

Notice that the unit of power is  $\frac{\text{bit}^2}{\text{second}}$ . We will show the useful properties of power specifically under congestion. Using Eq. 4, we can rewrite Eq. 5 in terms of queue length gradient  $\dot{q}$  and the transmission rate  $\mu$  as,

$$\Gamma(t) = (q(t) + b \cdot \tau) \cdot (\dot{q}(t) + \mu(t)) \quad (6)$$

We now derive a useful property of power using Eq. 6 and Eq. 4 showing an accurate relationship of power and window.

**Property 1** (Relationship of Power and Congestion Window). *Power is the bandwidth-window product*

$$\Gamma(t) = b \cdot w(t - t^f)$$

Note that the property is over the aggregate window size i.e., the sum of window sizes of all flows sharing the common bottleneck. We emphasize that our notion of power is intended for the networking context and cannot be applied to other domains of science. In the following, we outline the benefits of considering the notion of power and how Property 1 can be useful in the context of congestion control.

### 3.2 Benefits of Power-Based CC

A power-based control law can exploit Property 1 to precisely update per flow window sizes. Accurately controlling aggregate window size is a key challenge for an end-host congestion control algorithm. A power-based CC overcomes this challenge by gaining precise knowledge about the aggregate window size from measured power. First, using power enables the window update action to account for the bottleneck queue lengths as well as the queue build-up rate. As a result, a power-based CC can rapidly detect congestion onset even at very low queue lengths. At the same time, a power-based CC also reacts to the absolute value of queue lengths, effectively dampening perturbations. Second, calculating power at the end-host requires no extra measurement and feedback mechanisms compared to INT based schemes such as HPCC [36].

### 3.3 The POWERTCP Algorithm

Driven by our observations, we carefully designed our control law based on power, capturing a systematic reaction to voltage (related to bottleneck queue length), as well as to current (related to variations in the bottleneck queue length).

**Control law:** POWERTCP is a window-based congestion control algorithm and updates its window size upon receipt of an acknowledgment. For a flow  $i$ , every window update is based on (i) current window size  $w_i(t)$ , (ii) additive increase  $\beta$ , (iii) window size at the time of transmission of the acknowledged segment  $w_i(t - \theta(t))$ , and (iv) power measured from

the feedback information. We refer the reader to Table 2 for the general notations being used. Formally, POWERTCP's control law can be expressed as

$$w_i(t) \leftarrow \gamma \cdot \left( w_i(t - \theta(t)) \cdot \frac{e}{f(t)} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (7)$$

$$e = b^2 \cdot \tau; \quad f(t) = \Gamma(t - \theta(t) + t^f)$$

where  $\gamma \in (0, 1]$  and  $\beta$  are parameters to the control law. The base round trip time  $\tau$  must be configured at compile time. If baseRTT is not precisely known, an alternative is to keep track of minimum observed RTT. We first describe how power  $\Gamma$  is computed and then present the pseudocode of POWERTCP in Algorithm 1.

**Feedback:** POWERTCP's control law is based on power. Note that power (Eq. 5) is only related to variables at the bottleneck link. In order to measure power, we leverage in-band network telemetry. Specifically, the workings of INT and the header fields required are the same as in HPCC (Figure. 4 in [36]). When a TCP sender sends out a packet  $P$  into the network, it additionally inserts an INT header  $INT$  into the packet. Each switch along the path then pushes metadata containing the egress queue length ( $glen$ ), timestamp ( $ts$ ), so far transmitted bytes ( $txBytes$ ), and bandwidth ( $b$ ). All values correspond to the time when the packet is scheduled for transmission. At the receiver, the received packet  $P INT 1,2,...r$  is read and the INT information is copied to the acknowledgment  $ACK$  packet  $A INT 1,2,...t$ . The sender then receives an  $ACK$  with an INT header and metadata inserted by all the switches along the path from sender to receiver and back to sender  $A INT 1,2,...r INT ...n$ . Here, the INT header and meta-data pushed by switches along the path serve as feedback and as an input to the CC algorithm.

**Accounting for the old window sizes:** POWERTCP's control law (Eq. 7) uses the past window size in addition to the current window size to compute the new window size. POWERTCP accounts for old window size by remembering current window size once per RTT.

**Algorithm:** Putting it all together, we now present the workflow of POWERTCP in Algorithm 1. Upon the receipt of a new acknowledgment (line 2), POWERTCP: (i) retrieves the old  $cwnd$  (line 3), (ii) computes the normalized power (line 19) i.e.,  $\frac{f(t)}{e}$  in Eq. 7, (iii) updates  $cwnd$  (line 5), (iv) sets the pacing rate (line 6), and (v) remembers the INT header metadata and updates the old  $cwnd$  once per RTT based on the  $ack$  sequence number (line 7).

Specifically, power is calculated in the function call to **NORMPOWER**. First, the gradient of queue lengths is obtained from the difference in queue lengths and difference in timestamps corresponding to an egress port (line 12). Then the transmission rate of the egress port is calculated from the difference in  $txBytes$  and timestamps (line 13). Current is calculated by adding the queue gradient and transmission rate (line 14). Then, the sum of BDP and the queue length gives

voltage (line 16). Finally, power is calculated by multiplying current and voltage (line 17). We calculate the base power (line 18) and obtain the normalized power (line 19). The normalized power is calculated for each egress port along the path and the maximum value is smoothed and used as an input to the control law.

Finally, the congestion window is updated in the function call to `UPDATEWINDOW` (line 26) where  $\gamma$  is the exponential moving average parameter and  $\beta$  is the additive increase parameter, both being parameters to the control law (Eq. 7)

---

**Algorithm 1:** POWERTCP

---

```

1 /* ack contains an INT header with
   sequence of per-hop egress port
   meta-data accessed as ack.H[i] */
```

**Input :** `ack` and `prevInt`  
**Output :** `cwnd`, `rate`

**procedure** `NEWACK(ack)`:

- 3    `cwndold` = `GETCWND(ack.seq)`
- 4    `normPower` = `NORMPOWER(ack)`
- 5    `UPDATEWINDOW(normPower, cwndold)`
- 6    `rate` =  $\frac{cwnd}{\tau}$
- 7    `prevInt` = `ack.H`; `UPDATEOLD(cwnd, ack.seq)`

**function** `NORMPOWER(ack)`:

- 9     $\Gamma_{norm} = 0$
- 10    **for** each egress port  $i$  on the path **do**
- 11       $dt = ack.H[i].ts - prevInt[i].ts$
- 12       $\dot{q} = \frac{ack.H[i].qlen - prevInt[i].qlen}{dt}$                $\triangleright \frac{dq}{dt}$
- 13       $\mu = \frac{ack.H[i].txBytes - prevInt[i].txBytes}{dt}$                $\triangleright txRate$
- 14       $\lambda = \dot{q} + \mu$                $\triangleright \lambda$  : Current
- 15       $BDP = ack.H[i].b \times \tau$
- 16       $v = ack.H[i].qlen + BDP$                $\triangleright v$  : Voltage
- 17       $\Gamma' = \lambda \times v$                $\triangleright \Gamma'$  : Power
- 18       $e = (ack.H[i].b)^2 \times \tau$
- 19       $\Gamma'_{norm} = \frac{\Gamma'}{e}$                $\triangleright \Gamma'_{norm}$  :Normalized power
- 20      **if**  $\Gamma' > \Gamma_{norm}$  **then**
- 21         $\Gamma_{norm} = \Gamma'$ ;  $\Delta t = dt$
- 22      **end if**
- 23    **end for**
- 24     $\Gamma_{smooth} = \frac{\Gamma_{smooth} \cdot (\tau - \Delta t) + \Gamma_{norm} \cdot \Delta t}{\tau}$                $\triangleright$  Smoothing
- 25    **return**  $\Gamma_{smooth}$

**function** `UPDATEWINDOW(power, ack)`:

- 27     $cwnd = \gamma \times \left( \frac{cwnd_{old}}{normPower} + \beta \right) + (1 - \gamma) \times cwnd$
- 28               $\triangleright \gamma$ : EWMA parameter
- 29               $\triangleright \beta$ : Additive Increase
- 30    **return** `cwnd`

---

**Parameters:** POWERTCP has only two parameters, that is the EWMA parameter  $\gamma$  and the additive increase parameter  $\beta$ .  $\gamma$  dictates the balance in reaction time and sensitivity to noise. We recommend  $\gamma = 0.9$  based on our parameter sweep over wide range of scenarios including traffic patterns that induce

rapid fluctuations in the bottleneck queue lengths. Reflecting the intuition for additive increase in prior work [36], we set  $\beta = \frac{HostBw \times \tau}{N}$  where  $N$  is the expected number of flows sharing host NIC,  $HostBw$  is the NIC bandwidth at the host and  $\tau$  is the base-RTT. This is to avoid queuing at the local interface or, in other words, to avoid making the host NIC a bottleneck, assuming a maximum of  $N$  flows share the host NIC bandwidth. Finally, all flows transmit at line rate in the first RTT and use  $cwnd_{init} = HostBw \times \tau$ . By transmitting at line rate, a new flow is able to discover the bottleneck link state and reduce its  $cwnd$  accordingly without getting throttled due to the presence of existing flows.

### 3.4 Properties of POWERTCP

POWERTCP comes with strong theoretical guarantees. We show that POWERTCP’s control law achieves asymptotic stability with a unique equilibrium point that satisfies our desired equilibrium state properties (Eq. 1). POWERTCP also guarantees rapid convergence to equilibrium and achieves fairness at the same time. In the following we outline POWERTCP’s properties and defer the proofs to Appendix A.

**Theorem 1** (Stability). *POWERTCP’s control law is Lyapunov-stable as well as asymptotically stable with a unique equilibrium point.*

**Theorem 2** (Convergence). *After a perturbation, POWERTCP’s control law exponentially converges to equilibrium with a time constant  $\frac{\delta t}{\gamma}$  where  $\delta t$  is the window update interval.*

**Theorem 3** (Fairness). *POWERTCP is  $\beta_i$  weighted proportionally fair, where  $\beta_i$  is the additive increase used by a flow  $i$ .*

Theorem 1 and Theorem 2 state the key properties of POWERTCP. First, the convergence with time constant of  $\frac{\delta t}{\gamma}$  shows the fast reaction to perturbations. Second, the system being asymptotically stable at low queue lengths satisfies our stringent equilibrium property discussed in §2. Indeed, **power** and Property 1 play a key role in the proof of Theorem 1 and Theorem 2 (Appendix A) revealing its importance in congestion control. In Figure 3c, we see the trajectories of POWERTCP from different initial states to a unique equilibrium without violating throughput and latency requirements, showing the accurate control enabled by power-based congestion control.

### 3.5 θ-POWERTCP: Standalone Version

POWERTCP’s control law requires in-network queue length information which can be obtained by using techniques such as INT. In order to widen its applicability, POWERTCP can still be deployed in datacenters with legacy, non-programmable switches through accurate RTT measurement capabilities at the end-host. In this case, we rearrange term  $\frac{e}{f}$  in Eq. 7 as follows,

$$\frac{e}{f} = \frac{b^2 \cdot \tau}{\Gamma} = \frac{b^2 \cdot \tau}{(\dot{q} + b) \cdot (q + b \cdot \tau)} = \frac{\tau}{\left(\frac{\dot{q}}{b} + 1\right) \cdot \left(\frac{q}{b} + \tau\right)}$$

finally, using the fact that  $\frac{q}{b} + \tau = \theta$  (RTT) and  $\frac{q}{b} = \dot{\theta}$  (RTT gradient), we reduce  $\frac{e}{f}$  to,

$$\frac{e}{f} = \frac{\tau}{(\dot{\theta} + 1) \cdot (\theta)} \quad (8)$$

where  $\dot{\theta}$  is the RTT-gradient and  $\theta$  is RTT. Using Eq. 8 in Eq. 7 allows for deployment even when INT is not supported by switches in the datacenter. Due to space constraints we moved the algorithm to Appendix D, presenting  $\theta$ -POWERTCP in Algorithm 2. This algorithm demonstrates how POWERTCP’s control law can be mimicked by using a delay signal without the need for switch support. However, as we will show later in our evaluation, there are drawbacks in using RTT instead of queue lengths. First, notice how queue lengths are changed to RTT, where we assume bottleneck *txRate* ( $\mu$ ) as bandwidth ( $b$ ). The implication is that, when using *txRate* which is essentially obtained from INT, the control law knows the exact transmission rate and rapidly fills the available bandwidth. But, when using RTT, the control law assumes the bottleneck is at maximum transmission rate and does not react by multiplicative increase and rather relies on slow additive increase to fill the available bandwidth. Secondly, in multi-bottleneck scenarios, the control law precisely reacts to the most bottlenecked link when using INT but reacts to the sum of queuing delays when using RTT. Nevertheless, under congestion, both POWERTCP and  $\theta$ -POWERTCP have the same properties in a single-bottleneck scenario.

### 3.6 Deploying POWERTCP

Modern programmable switches are able to export user-defined header fields and device metrics [18, 32]. These metrics can be embedded into data packets, a mechanism commonly referred to as in-band network telemetry (INT). POWERTCP leverages INT to obtain fine-grained, per-packet feedback about queue occupancies, traffic counters, and link configurations within the network. For deployment with legacy networking equipment, we have proposed  $\theta$ -POWERTCP which only requires accurate timestamps to measure the RTT.

We imagine POWERTCP and  $\theta$ -POWERTCP to be deployed on low-latency kernel-bypass stacks such as SNAP [11] or using NIC offload. Yet, in this work, instead of implementing our algorithms for these platforms, we show how POWERTCP and  $\theta$ -POWERTCP can readily be deployed by merely changing the control logic of existing congestion control algorithms. In particular, we compare our work to HPCC [36] which is based on INT feedback and SWIFT [34] which is based on delay feedback.

POWERTCP requires the same switch support and header format as HPCC, as well as packet pacing support from the NIC. Additionally, it does not maintain additional state compared to HPCC but requires one extra parameter  $\gamma$ , the moving average parameter for window updates. Similar to SWIFT and TIMELY,  $\theta$ -POWERTCP requires accurate packet timestamps from the NIC but it does not require any switch support. The simpler logic of  $\theta$ -POWERTCP (compared to POW-

ERTCP) only reacts once per RTT and reduces the number of congestion control function calls.

The core contribution of this paper is the design of a novel control law and we do not explore implementation challenges further at this point since POWERTCP does not add additional complexity compared to existing algorithms. Still, to confirm the practical feasibility of our approach, we implemented POWERTCP as a Linux kernel congestion control module. We also implemented the INT component as a proof of concept for the Intel Tofino switch ASIC [18].

The switch implementation is written in P4 and uses a direct counter associated with the egress port to maintain the so far transmitted bytes and appends this metric together with the current queue occupancy upon dequeue from the traffic manager to each segment. We leverage a custom TCP option type to encode this data and append 64 bit per-hop headers to a 32 bit base header. The implementation uses less than one out of 12 stages of the Tofino’s ingress pipeline (where the headers are prepared and appended) and less than one out of 12 stages in the egress pipeline (where the measurements are taken and inserted). The processing logic runs at line rate of 3.2 Tbit per second.

## 4 Evaluation

We evaluate the performance of POWERTCP and  $\theta$ -POWERTCP and compare against existing CC algorithms. Our evaluation aims at answering four main questions.

**(Q1)** How well does POWERTCP react to congestion?

We find that POWERTCP outperforms the state-of-the-art congestion control algorithms, reducing tail buffer occupancy and consequently tail latency under congestion by 30% when compared to HPCC and at least by 60% compared to TIMELY and DCQCN.

**(Q2)** Does POWERTCP introduce a tradeoff between throughput and latency?

Our evaluation shows that POWERTCP does not trade throughput for latency and that POWERTCP rapidly converges to near-zero queue lengths without losing throughput.

**(Q3)** How much can we benefit under realistic workloads?

We show that POWERTCP improves 99th-percentile flow completion times for short flows (< 10KB) by 33% compared to HPCC, by 99% compared to HOMA and by 74% compared to TIMELY and DCQCN even at moderate network loads. At the same time, we find that POWERTCP does not penalize long flows (> 1MB). In fact, we find that  $\theta$ -POWERTCP performs equally well for short flows compared to POWERTCP but performs similarly to TIMELY for medium and long flows.

**(Q4)** How does POWERTCP perform under high load and bursty traffic patterns?

Our evaluation shows that the benefits of POWERTCP are further enhanced under high loads and that POWERTCP remains stable even under bursty traffic.

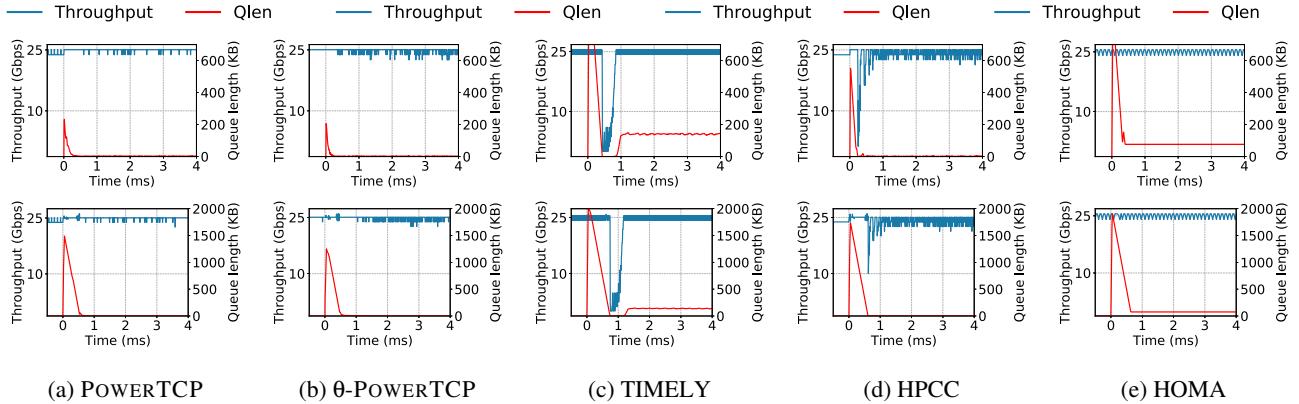


Figure 4: State-of-the-art congestion control algorithms vs POWERTCP in response to an incast. For each algorithm, we show the corresponding reaction to 10 : 1 incast in the top row and to 255 : 1 incast in the bottom row.

## 4.1 Setup

Our evaluation is based on network simulator NS3 [4].

**Topology:** We consider a datacenter network based on a Fat-Tree topology [5] with 2 core switches and 256 servers organized into four pods. Each pod consists of two ToR switches and two aggregation switches. The capacity of all the switch-to-switch links are 100Gbps and server-to-switch links are all 25Gbps leading to 4 : 1 oversubscription similar to prior work [49]. The links connecting to core switches have a propagation delay of  $5\mu s$  and all the remaining links have a propagation delay of  $1\mu s$ . We set up a shared memory architecture on all the switches and enable the Dynamic Thresholds algorithm [17] for buffer management across all the ports, commonly enabled in datacenter switches [1,2]. Finally we set the buffer sizes in our topology proportional to the bandwidth-buffer ratio of Intel Tofino switches [18].

**Traffic mix:** We generate traffic using the web search [7] flow size distribution to evaluate our algorithm using realistic workloads. We evaluate an average load (on the ToR uplinks) in the range of 20% – 95%. We also use a synthetic workload similar to prior work [6] to generate incast traffic. Specifically, the synthetic workload represents a distributed file system where each server requests a file from a set of servers chosen uniformly at random from a different rack. All the servers which receive the request respond at the same time by transmitting the requested part of the file. As a result, each file request creates an incast scenario. We evaluate across different request rates and request sizes.

**Comparisons and metrics:** We evaluate POWERTCP with and without switch support and compare to HPCC [36], DC-QCN [56], and TIMELY [41] representing sender-based control law approaches similar to POWERTCP and HOMA [42] representing receiver-driver transport. We report flow completion times and switch buffer occupancy metrics.

**Configuration:** We set  $\gamma = 0.9$  for POWERTCP and  $\theta$ -POWERTCP. Both HPCC and POWERTCP are configured with base-RTT ( $\tau$ ) set to the maximum RTT in our topology and  $HostBw$  is set to the server NIC bandwidth. The

product of base-RTT and  $HostBw$  is configured as RTTBytes for HOMA and the over-commitment level is set to 1 where HOMA performed best across different overcommitment levels in our setup. We report our results for all overcommitment levels (1-6) in Appendix C. We set the parameters for DCQCN following the suggestion in [36] which is based on experience and TIMELY parameters are set according to [41].

## 4.2 Results

**POWERTCP reacts rapidly yet accurately to congestion:** We evaluate POWERTCP’s reaction to congestion in two scenarios: (i) 10 : 1 small-scale incast and (ii) 255 : 1 large-scale incast. Figure 4 shows the aggregate throughput and the buffer occupancy at the bottleneck link for POWERTCP, TIMELY, HPCC and HOMA. First, at time  $t = 0$ , we launch ten flows simultaneously towards the receiver of a long flow leading to a **10:1** incast. We show in Figure 4a and Figure 4b that POWERTCP quickly mitigates the incast and reaches near zero queue lengths without losing throughput. In Figure 4d we see that HPCC indeed reacts quickly to get back to near-zero queue lengths. On one hand, however, HPCC does not react enough during the congestion onset and reaches higher buffer occupancy  $\approx 2x$  compared to POWERTCP and on the other hand loses throughput after mitigating the incast as opposed to POWERTCP’s stable throughput. TIMELY as shown in Figure 4c does not control the queue-lengths either and loses throughput after reacting to the incast. While HOMA sustains throughput, we observe from Figure 4e that HOMA does not accurately control bottleneck queue-lengths. Second, at time  $t = 0$ , in addition to the 10 : 1 incast, the 256<sup>th</sup> server sends a query request (§4.1) to all the other 255 servers which then respond at the same time, creating a **255:1** incast. From Figure 4a and Figure 4b (bottom row), we observe similar benefits from both POWERTCP and  $\theta$ -POWERTCP even at large-scale incast: both react quickly and converge to near-zero queue-lengths without losing throughput. In contrast, from Figure 4c and Figure 4d we see that TIMELY and HPCC lose throughput immediately after re-

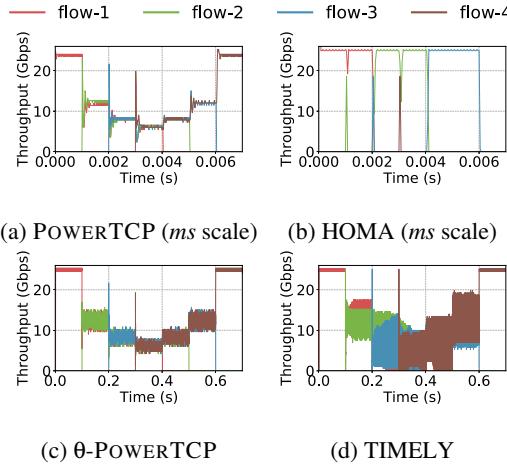


Figure 5: Fairness and stability

acting to the increased queue length. From Figure 4e we observe that HOMA reaches approximately 500KB higher queue-length compared to POWERTCP and cannot converge to near-zero queue-lengths quickly.

**POWERTCP is stable and achieves fairness:** POWERTCP not only reacts rapidly to reduce queue lengths but also features excellent stability. Figure 5 shows how bandwidth is shared by multiple flows as they arrive and leave. We see that POWERTCP stabilizes to a fair share of bandwidth quickly, both when flows arrive and leave, confirming POWERTCP’s fast reaction to congestion as well as the available bandwidth.

Figure 4a showing convergence and Figure 5a showing fairness and stability confirm the theoretical guarantees of POWERTCP. Hereafter, all our results are based on the setup described above, §4.1, using realistic workloads.

**POWERTCP significantly improves short flows FCTs:** In Figure 6 we show the 99.9-percentile flow completion times using POWERTCP and state-of-the-art datacenter congestion control algorithms. At 20% network load (Figure 6a), POWERTCP and  $\theta$ -POWERTCP improve 99.9-percentile flow completion times for short flows ( $< 10KB$ ) by 9% compared to HPCC and by 80% compared to TIMELY, DCQCN and HOMA. Even at moderate load of 60% (Figure 6b), short flows significantly benefit from POWERTCP as well as  $\theta$ -POWERTCP. Specifically, POWERTCP improves 99.9 percentile flow completion times for short flows by 33% compared to HPCC, by 99% compared to HOMA and by 74% compared to TIMELY and DCQCN.  $\theta$ -POWERTCP provides even greater benefits to short flows showing an improvement of 36% compared to HPCC and 82% compared to TIMELY and DCQCN. Indeed, web search workload being buffer-intensive, our results confirm the observations made in §2. TIMELY being a current-based CC, does not explicitly control queuing latency, while HPCC, a voltage-based CC, does not react as fast as POWERTCP to mitigate congestion resulting in higher flow completion times. Surprisingly, HOMA

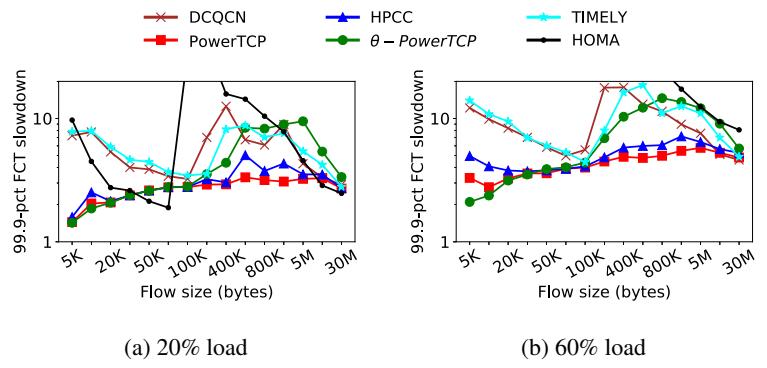


Figure 6: 99.9 percentile flow completion times with websearch workload  
(a) even at low network load, POWERTCP outperforms existing algorithms and (b) as the load increases the benefits of POWERTCP are enhanced. However, only short flows benefit from  $\theta$ -POWERTCP.

performs the worst, showing an order-of-magnitude higher FCTs for short flows at high loads as shown in Figure 6b.

We also evaluate across various loads in the range 20% – 95% and show the 99.9-percentile flow completion times for short flows in Figure 7a. In particular, we see that the benefits of POWERTCP and  $\theta$ -POWERTCP are further enhanced as the network load increases. POWERTCP (and  $\theta$ -POWERTCP) improve the flow completion times of short flows by 36% (and 55%) compared to HPCC. Short flows particularly benefit from POWERTCP due its accurate control of buffer occupancies close to zero. In Figure 7g we show the CDF of buffer occupancies at 80% load. POWERTCP consistently maintains lower buffer occupancy and cuts the tail buffer occupancy by 50% compared to HPCC.

**Medium sized flows also benefit from POWERTCP:** We find that POWERTCP not only improves short flow performance but also improves the 99.9-percentile flow completion times for medium sized flows ( $100KB – 1M$ ). In Figure 6 we see that POWERTCP consistently achieves better flow completion times for medium sized flows. Specifically, at 20% network load (Figure 6a), POWERTCP improves 99.9-percentile flow completion times for medium flows by 33% compared to HPCC, by 76% compared to HOMA and by 62% (and 50%) compared to TIMELY (and DCQCN). In Figure 6b, we observe similar benefits even at 60% load.

We notice from Figure 6a and Figure 6b that the performance of  $\theta$ -POWERTCP deteriorates sharply for medium sized flows.  $\theta$ -POWERTCP uses RTT for window update calculations. While RTT can be a good congestion signal, it does not signal under-utilization as opposed to INT that explicitly notifies the exact utilization. As a result, medium flows with  $\theta$ -POWERTCP experience 60% worse performance on average compared to POWERTCP and HPCC. We also observe similar performance for TIMELY that uses RTT as a congestion signal. Although delay is simple and effective for short flows performance even at the tail, our results show that delay

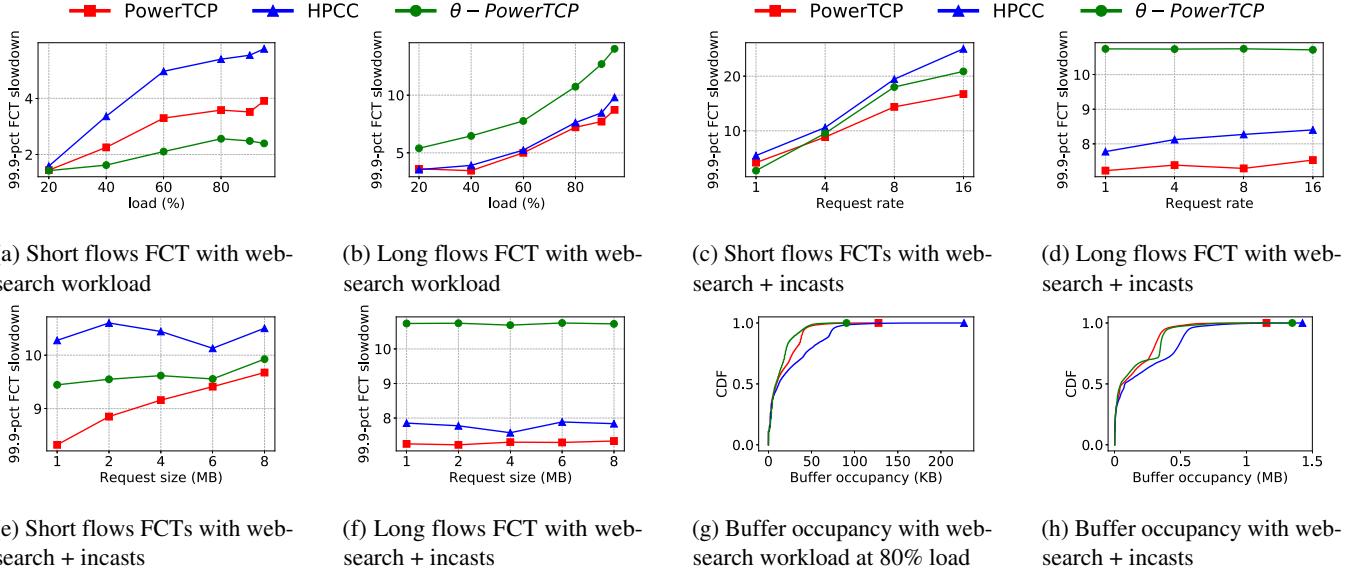


Figure 7: A detailed comparison of POWERTCP,  $\theta$ -POWERTCP and the state-of-the-art showing the benefits of POWERTCP and the trade-offs of  $\theta$ -POWERTCP. Particularly POWERTCP outperforms the state-of-the-art across a range of network loads even under bursty traffic. However,  $\theta$ -POWERTCP performs well for short flows but long flows cannot benefit from  $\theta$ -POWERTCP.

as a congestion signal is not ideal if not worse for medium sized flows.

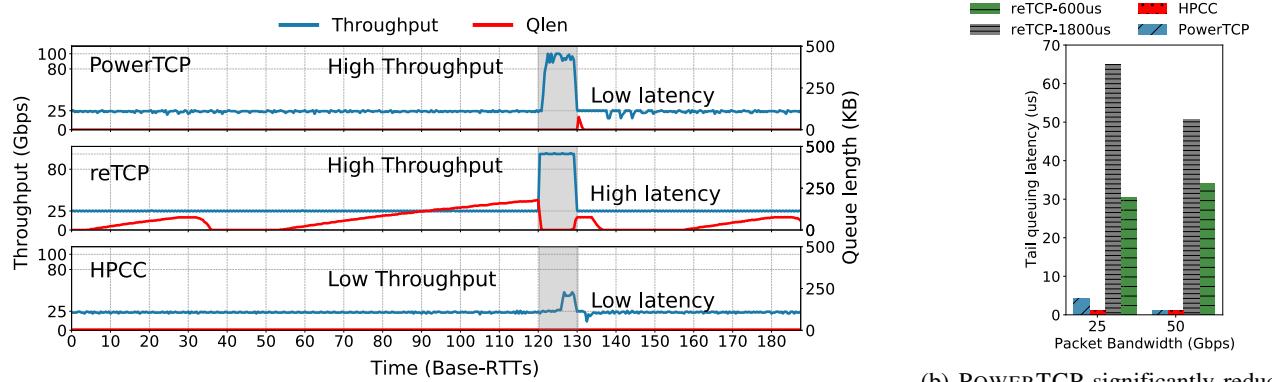
**POWERTCP does not penalize long flows:** Fast reaction to available bandwidth makes POWERTCP ideal for best performance across all flow sizes. We observe from Figure 6 that POWERTCP achieves flow completion times comparable to existing algorithms, indicating that POWERTCP does not trade throughput for low latency. Further, in Figure 7b we show the 99.9-percentile flow completion times for long flows across various loads. At low load, POWERTCP performs similar to HPCC and performs 9% better compared to HPCC at 90% network loads. However, we see that  $\theta$ -POWERTCP is consistently 35% worse on average across various loads compared to POWERTCP and HPCC.

**POWERTCP outperforms under bursty traffic:** We generate incast-like traffic described in §4.1 in addition to the web search workload at 80% load. In Figure 7c and Figure 7d we show the 99.9-percentile flow completion times for short and long flows across different request rates for a request size of 2MB. Note that by varying request rates, we are essentially varying the frequency of incasts. We observe that even under bursty traffic, POWERTCP improves 99.9-percentile flow completion times on average for short flows by 24% and for long flows by 10% compared to HPCC. Further POWERTCP outperforms at high request rates showing 33% improvement over HPCC for short flows. On the other hand,  $\theta$ -POWERTCP improves flows completion times for short flows but performs worse across all request rates compared to HPCC.

We further vary the request size at a request rate of four per second. Note that by varying the request size, we also vary the duration of congestion. In Figure 7e and Figure 7f,

we show the 99.9-percentile flow completion times for short and long flows. Specifically, in Figure 7e we observe that flow completion times with POWERTCP gradually increase with request size. POWERTCP, compared to HPCC, improves flow completion times of short flows by 20% at 1MB request size and improves by 7% at 8MB request size. At the same time, POWERTCP does not sacrifice long flows performance under bursty traffic. POWERTCP improves flow completion times for long flows by 5% on average compared to HPCC.  $\theta$ -POWERTCP's performance similar to previous experiments is on average 30% worse for long flows but 9% better for short flows compared to HPCC. We show the CDF of buffer occupancies under bursty traffic with 2MB request size and 16 per second request rate. Both POWERTCP and  $\theta$ -POWERTCP reduce the 99 percentile buffer by 31% compared to HPCC.

We note that HOMA’s performance in our evaluation is not in line with the results presented in [42]. Recent work [26] reports similar performance issues with HOMA. We suspect two possible reasons: (i) HOMA’s accuracy in controlling congestion is specifically limited in our network setup with an oversubscribed Fat-Tree topology where congestion at the ToR uplinks is a possibility which cannot be controlled by a receiver-driven approach such as HOMA. (ii) As pointed out by [26], HOMA’s original evaluation considered practically infinite buffers at the switches whereas switches in our setup are limited in buffer and use Dynamic Thresholds to share buffer. Further, even at 20% load, asymmetric RTTs in a Fat-Tree topology (consequently RTTBytes) across ToR pairs contributes to HOMA’s inaccuracy in controlling congestion.



(a) POWERTCP reacts rapidly to the available bandwidth achieving good circuit utilization.

(b) POWERTCP significantly reduces the tail latency

Figure 8: The benefits of POWERTCP in reconfigurable datacenter networks showing its ability to achieve good circuit utilization while significantly reducing the tail latency compared to reTCP.

## 5 Case Study: Reconfigurable DCNs

Given POWERTCP’s rapid reaction to congestion and available bandwidth, we believe that POWERTCP is well suited for emerging reconfigurable datacenter networks (RDCN) [44]. We now examine POWERTCP’s applicability in this context through a case study. Congestion control in RDCNs is especially challenging as the available bandwidth rapidly fluctuates due to changing circuits. In this section, we evaluate the performance of POWERTCP and compare against the state-of-the-art reTCP [43] and HPCC using packet-level simulations in NS3. We implement both POWERTCP and HPCC in the transport layer and limit their window updates to once per RTT for a fair comparison with reTCP. POWERTCP and HPCC flows initialize the TCP header with the unused option number 36. Switches are configured to append INT metadata to TCP options. It should be noted that TCP options are limited to 40 bytes. As a result, our implementation can only support at most four hops round-trip path length.

We evaluate in a topology with 25 ToR switches with 10 servers each and a single optical circuit switch connected to all the ToR switches. ToR switches are also connected to a separate packet switched network with 25Gbps links. The optical switch internally connects each input port to an output port and cycles across 24 matchings in a permutation schedule where the switch stays in a specific matching for 225 $\mu$ s (one day) and takes 20 $\mu$ s to reconfigure to the next matching (one night). In this setting, each pair of ToR switches has direct connectivity through the circuit switch once over a length of 24 matchings (one week). We use single-hop routing in the circuit network and a maximum base RTT is 24 $\mu$ s. Note that circuit-on time (i.e., one day) is approximately 10 RTTs. The links between servers and ToR switches are 25Gbps and circuit links are 100Gbps. We configure the ToR switches to forward packets exclusively on the circuit network when available. Switches are further equipped with per-destination virtual output queues (VOQs). Our setup is in line with prior work [43]. We set reTCP’s prebuffering to 1800 $\mu$ s based

on the suggestions in [43] and set to 600 $\mu$ s based on our parameter sweep for the minimum required prebuffering in our topology. We compare against both versions.

In Figure 8a, we show the time series of throughput and VOQ length for a pair of ToR switches. Specifically, the gray-shaded area in Figure 8a highlights the availability of high bandwidth through the circuit-switched network. On one hand, reTCP instantly fills the available bandwidth but incurs high latency due to prebuffering before the circuit is available. On the other hand, HPCC maintains low queue lengths but does not fill the available bandwidth. In contrast, POWERTCP fills the available bandwidth within one RTT and maintains near-zero queue lengths and thereby achieves both high throughput and low latency. We show the tail queuing latency incurred by reTCP, HPCC and POWERTCP in Figure 8b. We observe that POWERTCP improves the tail queuing latency at least by 5 $\times$  compared to reTCP. Our case study reveals that fine-grained congestion control algorithms such as POWERTCP can alleviate the circuit utilization problem in RDCNs without trading latency for throughput.

## 6 Related Work

Dealing with congestion has been an active research topic for decades with a wide spectrum of approaches, including buffer management [3, 10, 17] and scheduling [9, 25, 45, 46]. In the following, we will focus on the most closely related works on end-host congestion control.

Approaches such as [7, 51, 56] (e.g., DCTCP, D<sup>2</sup>TCP) rely on ECN as the congestion signal and react proportionally. Such algorithms require the bottleneck queue to grow up to a certain threshold, which results in queuing delays. ECN-based schemes remain oblivious to congestion onset and intensity. Protocols such as TIMELY [41], SWIFT [34], CDG [23], DX [35] rely on RTT measurements for window update calculations. TIMELY and CDG partly react to congestion based on delay gradients, remaining oblivious to absolute queue lengths. TIMELY, for instance, uses a threshold to fall back

to proportional reaction to delay instead of delay gradient. SWIFT, a successor of TIMELY, only reacts proportionally to delay. As a result, SWIFT cannot detect congestion onset and intensity unless the distance from target delay significantly increases. In contrast,  $\theta$ -POWERTCP also being a delay-based congestion control algorithm updates the window sizes using the notion of power. As a result,  $\theta$ -POWERTCP accurately detects congestion onset even at near-zero queue lengths.

XCP [30], D<sup>3</sup> [52], RCP [19] rely on explicit network feedback based on rate calculations within the network. However, the rate calculations are based on heuristics and require parameter tuning to adjust for different goals such as fairness and utilization. HPCC [36] introduces a novel use of in-band network telemetry and significantly improves the fidelity of feedback. Our work builds on the same INT capabilities to accurately measure the bottleneck link state. However, as we show analytically and empirically, HPCC’s control law then adjusts rate and window size solely based on observed queue lengths and lacks control accuracy compared to POWERTCP. Our proposal POWERTCP uses the same feedback signal but uses the notion of power to update window sizes leading to significantly more fine-grained and accurate reactions.

Receiver-driven transport protocols such as NDP [22], HOMA [42], and Aeolus [26] have received much attention lately. Such approaches are conceptually different from classic transmission control at the sender. Importantly, receiver-driven transport approaches make assumptions on the uniformity in datacenter topologies and oversubscription [22]. POWERTCP is a sender-based classic CC approach that uses our novel notion of power and achieves fine-grained control over queuing delays without sacrificing throughput.

## 7 Conclusion

We presented POWERTCP, a novel fine-grained congestion control algorithm. By reacting to both the current state of the network as well as its trend (i.e., power), POWERTCP improves throughput, reduces latency, and keeps queues within the network short. We proved that POWERTCP has a set of desirable properties, such as fast convergence and stability allowing it to significantly improve flow completion times compared to the state-of-the-art. Its fast reaction makes POWERTCP attractive for many dynamic network environments including emerging reconfigurable datacenters which served us as a case study in this paper. In our future work, we plan to explore more such use cases.

## Acknowledgments

We would like to thank our shepherd, Michael Schapira, as well as the anonymous NSDI reviewers for their useful feedback. This work is part of a project that has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme, consolidator project Self-Adjusting Networks (AdjustNet), grant agreement No. 864228, Horizon 2020, 2020-2025.



## References

- [1] Broadcom. 12.8 tb/s strataxgs tomahawk 3 ethernet switch series. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series>.
- [2] Broadcom. 2020. 25.6 tb/s strataxgs tomahawk 4 ethernet switch series. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series>.
- [3] Cisco nexus 9000 series switches. <https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-738488.html>.
- [4] Ns3 network simulator. <https://www.nsnam.org/>.
- [5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, page 63–74, 2008.
- [6] Mohammad Alizadeh and Tom Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st annual symposium on high-performance interconnects*, pages 71–74. IEEE, 2013.
- [7] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
- [8] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):73–84, 2011.

- [9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, page 435–446, 2013.
- [10] Maria Apostolaki, Laurent Vanbever, and Manya Ghobadi. Fab: Toward flow-aware buffer sharing on programmable switches. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1–6, 2019.
- [11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the ACM SIGCOMM 2016 Conference*, page 29–43, 2016.
- [12] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. In *Proc. ACM SIGMETRICS*, 2020.
- [13] Chen Avin and Stefan Schmid. Renets: Statically-optimal demand-aware networks. In *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.
- [14] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 782–797, 2020.
- [15] Lawrence S Brakmo, Sean W O’Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.
- [16] 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*, pages 73–82, 2009.
- [17] Abhijit K Choudhury and Ellen L Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions On Networking*, 6(2):130–140, 1998.
- [18] Intel Corporation. Intel Tofino, 2020. Retrieved Dec. 29, 2020 from <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html>.
- [19] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.
- [20] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 216–229, 2016.
- [21] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.
- [22] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, page 29–42, 2017.
- [23] David A Hayes and Grenville Armitage. Revisiting tcp congestion control using delay gradients. In *International Conference on Research in Networking*, pages 328–341. Springer, 2011.
- [24] Christopher V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. A control theoretic analysis of red. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1510–1519. IEEE, 2001.
- [25] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.
- [26] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 422–434, 2020.
- [27] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols, SIGCOMM ’88*, page 314–329, 1988.
- [28] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.

- [29] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202--208, 2009.
- [30] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89--102, 2002.
- [31] Srinivasan Keshav. *Mathematical foundations of computer networking*. Addison-Wesley, 2012.
- [32] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM '15 Demos*, 2015.
- [33] Janardhan Kulkarni, Stefan Schmid, and Paweł Schmidt. Scheduling opportunistic links in two-tiered reconfigurable datacenters. In *33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2021.
- [34] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 514–528, 2020.
- [35] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 403--415, Santa Clara, CA, July 2015. USENIX Association.
- [36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, pages 44--58, 2019.
- [37] S.H. Low, F. Paganini, and J.C. Doyle. Internet congestion control. *IEEE Control Systems Magazine*, 22(1):28--43, 2002.
- [38] William M Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1--18, 2020.
- [39] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 267--280, 2017.
- [40] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 151--160, 2000.
- [41] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blehm, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, page 537–550, 2015.
- [42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, page 221–235, 2018.
- [43] Matthew K Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651--666, 2020.
- [44] Matthew Nance Hall, Klaus-Tycho Foerster, Stefan Schmid, and Ramakrishnan Durairajan. A survey of reconfigurable optical networks. *Optical Switching and Networking*, 41:100621, 2021.
- [45] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 421--435, Boston, MA, March 2017. USENIX Association.
- [46] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the ACM SIGCOMM 2014 conference*, pages 307--318, 2014.
- [47] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G Andersen, Gregory R Ganger, Garth A Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST*, volume 8, pages 1--14, 2008.

- [48] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network’s (datacenter) network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.
- [49] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 735–749, 2020.
- [50] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking (ToN)*, 2016.
- [51] Balajee Vamanan, Jahangir Hasan, and T.N. Vijayku-  
mar. Deadline-aware datacenter tcp (d2tcp). In *Pro-  
ceedings of the ACM SIGCOMM 2012 Conference*, page  
115–126, 2012.
- [52] Christo Wilson, Hitesh Ballani, Thomas Karagiannis,  
and Ant Rowtron. Better never than late: Meeting dead-  
lines in datacenter networks. In *Proceedings of the ACM  
SIGCOMM 2011 Conference*, page 50–61, 2011.
- [53] Jackson Woodruff, Andrew W Moore, and Noa Zilber-  
man. Measuring burstiness in data center applications.  
In *Proceedings of the 2019 Workshop on Buffer Sizing*,  
2019.
- [54] Doron Zarchy, Radhika Mittal, Michael Schapira, and  
Scott Shenker. Axiomatizing congestion control. *Pro-  
ceedings of the ACM on Measurement and Analysis of  
Computing Systems*, 3(2):1–33, 2019.
- [55] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind  
Krishnamurthy. High-resolution measurement of data  
center microbursts. In *Proceedings of the 2017 Internet  
Measurement Conference*, pages 78–85, 2017.
- [56] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong  
Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Pad-  
hye, Shachar Raindel, Mohamad Haj Yahia, and Ming  
Zhang. Congestion control for large-scale rdma deploy-  
ments. *ACM SIGCOMM Computer Communication  
Review*, 45(4):523–536, 2015.
- [57] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra  
Padhye. Ecn or delay: Lessons learnt from analysis of  
dcqcn and timely. In *Proceedings of the 12th Interna-  
tional on Conference on emerging Networking EXperi-  
ments and Technologies*, pages 313–327, 2016.

## A Analysis

Our analysis is based on a single bottleneck link model widely used in the literature [24, 40, 54, 57]. Specifically, we assume that all senders use the same protocol, transmit long flows sharing a common bottleneck link with bandwidth  $b$ , and have a base round trip time  $\tau$  (excluding queuing delays). We denote at time  $t$  queue length as  $q(t)$ , aggregate window size as  $w(t)$ , window size of a sender  $i$  as  $w_i(t)$ , forward propagation delay between sender and bottleneck queue as  $t^f$ , the round-trip time as  $\theta(t)$  and a base round-trip time as  $\tau$ . Here  $w(t) = \sum_i w_i(t)$ .

We additionally use the traditional model of queue length dynamics which is independent of the control law [24, 40]

$$\dot{q}(t) = \frac{w(t - t^f)}{\theta(t)} - b \quad (9)$$

where  $\theta(t)$  is given by,

$$\theta(t) = \frac{q(t)}{b} + \tau \quad (10)$$

Power at time  $t$  denoted by  $\Gamma(t)$  as defined in §3.1 is expressed as,

$$\Gamma(t) = \underbrace{(q(t) + b \cdot \tau)}_{\text{voltage}} \cdot \underbrace{(\dot{q}(t) + \mu(t))}_{\text{current}} \quad (11)$$

POWERTCP’s control law at a source  $i$  is given by,

$$w_i(t + \delta t) = \gamma \cdot \left( \frac{w_i(t - \theta(t)) \cdot e}{f(t)} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (12)$$

where  $e$  and  $f(t)$  are given by,

$$e = b^2 \cdot \tau$$

$$f(t) = \Gamma(t - \theta(t) + t^f)$$

and  $\beta$  is the additive increase term and  $\gamma \in (0, 1]$  serves as the weight given for new updates using EWMA. Both  $\beta$  and  $\gamma$  are parameters to the control law.

Using the properties of power (Property 1), the aggregate window size at time  $t - \theta(t)$  can be expressed in terms of power as,

$$w(t - \theta(t)) = \frac{\Gamma(t - \theta(t) + t^f)}{b} = \frac{f(t)}{b} \quad (13)$$

Suppose an *ack* arrives at time  $t$  acknowledging a segment, time  $t - \theta(t)$  corresponds to the time when the acknowledged segment was transmitted.

**Theorem 1** (Stability). POWERTCP’s control law is Lyapunov-stable as well as asymptotically stable with a unique equilibrium point.

Notation	Description
$b$	bottleneck bandwidth
$q$	bottleneck queue length
$\tau$	base RTT
$t^f$	sender to bottleneck delay
$\theta$	round trip time RTT
$w_i$	window size of a flow $i$
$w$	aggregate window size (of all flows)
$\gamma$	EWMA parameter
$\beta$	additive increase
$e$	desired equilibrium point
$f$	feedback
$\lambda_i$	sending rate of a flow $i$
$\lambda$	Current: aggregate sending rate
$v$	Voltage
$\Gamma$	Power

Table 2: Key notations used in this paper. Additionally for any variable say  $x$ ,  $\dot{x}$  denotes its derivative with respect to time i.e.,  $\frac{dx}{dt}$ .

*Proof.* First, we rewrite Eq. 12 as follows to obtain the aggregate window  $w$ ,

$$\sum_i w_i(t + \delta t) = \sum_i \gamma \cdot \left( \frac{w_i(t - \theta(t)) \cdot e}{f(t)} + \hat{\beta} \right) + \sum_i (1 - \gamma) \cdot w_i(t)$$

$$\text{let } \hat{\beta} = \sum_i \beta$$

$$w(t + \delta t) = \gamma \cdot \left( \frac{w(t - \theta(t)) \cdot e}{f(t)} + \hat{\beta} \right) + (1 - \gamma) \cdot w(t)$$

by rearranging the terms in the above equation we obtain,

$$w(t + \delta t) - w(t) = \gamma \cdot \left( -w(t) + \frac{w(t - \theta(t)) \cdot e}{f(t)} + \hat{\beta} \right)$$

dividing by  $\delta t$  on both sides in the above equation and using Euler's first-order approximation, we derive the window dynamics for POWERTCP as follows,

$$\dot{w}(t) = \gamma_r \cdot \left( -w(t) + \frac{w(t - \theta(t)) \cdot e}{f(t)} + \hat{\beta} \right) \quad (14)$$

where  $\gamma_r = \frac{\gamma}{\delta t}$ . Using Eq. 13 and substituting  $e = b^2 \cdot \tau$ , Eq. 14 reduces to,

$$\dot{w}(t) = \gamma_r \cdot \left( -w(t) + b \cdot \tau + \hat{\beta} \right) \quad (15)$$

In the system defined by Eq. 9 and Eq. 14, when the window and the queue length stabilize i.e.,  $\dot{w}(t) = 0$  and  $\dot{q}(t) = 0$ , it is easy to observe that there exists a unique equilibrium point  $(w_e, q_e) = (b \cdot \tau + \hat{\beta}, \hat{\beta})$ . We now apply a change of variable from  $t$  to  $t - t^f$  in Eq. 15 and linearize Eq. 15 and Eq. 9 around  $(w_e, q_e)$ ,

$$\delta \dot{w}(t - t^f) = -\gamma_r \cdot \delta w(t - t^f) \quad (16)$$

$$\delta \dot{q}(t) = -\frac{\delta q(t)}{\tau} + \frac{\delta w(t - t^f)}{\tau} \quad (17)$$

We now convert the above differential equations to matrix form,

$$\begin{bmatrix} \delta \dot{q}(t) \\ \delta \dot{w}(t) \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau} & \frac{1}{\tau} \\ 0 & -\gamma_r \end{bmatrix} \times \begin{bmatrix} \delta q(t) \\ \delta w(t) \end{bmatrix}$$

It is then easy to observe that the eigenvalues of the system are  $-\frac{1}{\tau}$  and  $-\gamma_r$ . Since  $\tau$  (base RTT) and  $\gamma_r = \frac{\gamma}{\delta t}$  are both positive, we see that both the eigenvalues are negative. This proves that the system is both lyapunav stable and asymptotically stable.  $\square$

**Theorem 2** (Convergence). *After a perturbation, POWERTCP's control law exponentially converges to equilibrium with a time constant  $\frac{\delta t}{\gamma}$  where  $\delta t$  is the window update interval.*  
*Proof.* A perturbation at time  $t = 0$  causes the window to shift from  $w_e = c \cdot \tau + \hat{\beta}$  to say  $w_{init}$ . We solve the differential equation in Eq. 15 and obtain the following equation,

$$w(t) = w_e + \underbrace{(w_{init} - w_e) \cdot e^{-\gamma_r t}}_{\text{exponential decay}} \quad (18)$$

From Eq. 18 we can see that, for any error  $e = w_e - w_{init}$  caused by a perturbation,  $e$  exponentially decays with a time constant  $\frac{1}{\gamma_r} = \frac{\delta t}{\gamma}$ . Hence for  $e$  to decay 99.3%, it takes  $\frac{5 \cdot \delta t}{\gamma}$  time.  $\square$

**Theorem 3** (Fairness). *POWERTCP is  $\beta_i$  weighted proportionally fair, where  $\beta_i$  is the additive increase used by a flow  $i$ .*

*Proof.* Recall that POWERTCP's control law for each flow  $i$  is defined as,

$$w_i(t + \delta t) = \gamma \cdot \left( \frac{w_i(t - \theta(t)) \cdot e}{f(t)} + \beta_i \right) + (1 - \gamma) \cdot w_i(t)$$

From the proof of Theorem 1, we know that the equilibrium point for aggregate window size and queue length is  $(w_e, q_e) = (b \cdot \tau + \hat{\beta}, \hat{\beta})$ . Using this equilibrium we can also obtain the equilibrium value for  $f(t)$  as,

$$f_e = (\hat{\beta} + b \cdot \tau) \cdot b$$

We can then show that  $w_i$  has an equilibrium point.

$$(w_i)_e = \frac{\hat{\beta} + b \cdot \tau}{\hat{\beta}} \cdot \beta_i$$

We use the argument that window sizes and rates are synonymous especially that POWERTCP uses pacing with rate  $r_i = \frac{w_i}{\tau}$ . We can then easily observe that the rate allocation is approximately max-min fair if  $\beta_i$  are small enough but  $\beta_i$  proportionally fair in general.  $\square$

## B Justifying the Simplified Model

We considered a simplified control law model to study existing control laws in §2. Here we justify how the simplified model approximately captures the existing control laws. Our simplified model for congestion window update at time  $t + \delta t$  is defined in Eq. 19 as a function of current congestion window size, a target  $e$ , the feedback  $f(t)$ , an additive increase  $\beta$  and an exponential moving average parameter  $\gamma$ .

$$w_i(t + \delta t) = \gamma \cdot \underbrace{\left( w_i(t) \cdot \frac{e}{f(t)} + \beta \right)}_{\text{update}} + (1 - \gamma) \cdot w_i(t) \quad (19)$$

where  $e$  and  $f(t)$  are given by,

$$e = \begin{cases} b \cdot \tau & \text{queue-length based CC} \\ \tau & \text{delay-based CC} \\ 1 & \text{RTT-gradient based CC} \end{cases} \quad (20)$$

$$f(t) = \begin{cases} q(t - \theta(t) + t^f) + b \cdot \tau & \text{queue-length based CC} \\ \frac{q(t - \theta(t) + t^f)}{b} + \tau & \text{delay-based CC} \\ \frac{q(t - \theta(t) + t^f)}{b} + 1 & \text{RTT-gradient based CC} \end{cases} \quad (21)$$

We first use Euler's first order approximation and obtain the aggregate window ( $\sum w$ ) dynamics for the simplified model,

$$\dot{w}(t) = \frac{\gamma}{\delta t} \cdot \left( w(t) \cdot \frac{e}{f(t)} - w(t) + \beta \right) \quad (22)$$

In order for the system to stabilize, we require  $q(t) = 0$  and  $\dot{w}(t) = 0$ . Using Eq. 9 and Eq. 22 and applying equilibrium conditions and assuming that  $f(t)$  stabilizes,

$$q_e = w_e - b \cdot \tau \quad (23)$$

$$w_e = \frac{\hat{\beta}}{1 - \frac{e}{f}} \quad (24)$$

Recall that  $\hat{\beta} = \sum \beta_i$ , the sum of additive increase terms of all flows sharing a bottleneck. To show whether there exists a unique equilibrium point, it remains to show whether Eq. 23 and Eq. 24 have a unique solution for  $w_e$  and  $q_e$ . We now show how the simplified model captures existing control laws and show the equilibrium properties.

**Queue length or inflight-based control law:** Substituting  $e = b \cdot \tau$  and  $f(t) = q(t - \theta(t) + t^f) + b \cdot \tau$ , we express the simplified queue length based control law as,

$$w_i(t + \delta t) = \gamma \cdot \left( \frac{w_i(t) \cdot b \cdot \tau}{q(t - \theta(t) + t^f) + b \cdot \tau} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (25)$$

notice that the update is an MIMD based on inflight bytes. Eq. 25 captures control laws based on inflight bytes; for example HPCC [36].

A system defined by queue length based control law (Eq. 25) and the queue length dynamics (Eq. 9, there exists a unique equilibrium point. It can be observed that Eq. 24 for queue length based control law gives  $w_e = b \cdot \tau + \hat{\beta}$  and  $q_e = \hat{\beta}$ .

**Delay-based control law:** Substituting  $e = \tau$  and  $f(t) = \frac{q(t - \theta(t) + t^f)}{b} + \tau$ , we express the simplified delay-based control law as,

$$w_i(t + \delta t) = \gamma \cdot \left( \frac{w_i(t) \cdot \tau}{\frac{q(t - \theta(t) + t^f)}{b} + \tau} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (26)$$

where the window update is an MIMD based on RTT. Eq. 26 captures control laws based on RTT; for example FAST [28].

Similar to queue-length based CC, a system defined by delay-based control law (Eq. 26 and the queue length dynamics (Eq. 9, there exists a unique equilibrium point. It can be observed that Eq. 24 for delay-based control law gives  $w_e = b \cdot \tau + \hat{\beta}$  and  $q_e = \hat{\beta}$ .

**RTT-gradient based control law:** Substituting  $e = 1$  and  $f(t) = \frac{q(t - \theta(t) + t^f)}{b} + 1$ , we express the simplified RTT-gradient based control law as,

$$w_i(t + \delta t) = \gamma \cdot \left( \frac{w_i(t) \cdot 1}{\frac{q(t - \theta(t) + t^f)}{b} + 1} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (27)$$

where the window update is an MIMD based on RTT-gradient. Eq. 27 by rearranging the terms, captures control laws based on RTT-gradient such as TIMELY [41].

In contrast to queue-length and delay-based CC, RTT-gradient based CC has no unique equilibrium point since  $f(t) = \frac{q(t - \theta(t) + t^f)}{b} + 1$  stabilizes when  $\dot{q} = 0$ . However only  $\dot{q} = 0$  leads to window dynamics Eq. 27 also to stabilize ( $\dot{w} = 0$ ) at any queue lengths. As a result under RTT-gradient control law, Eq. 23 and Eq. 24 do not have a unique solution and consequently we can state that RTT-gradient based CC has no unique equilibrium point.

## C HOMA's Overcommitment

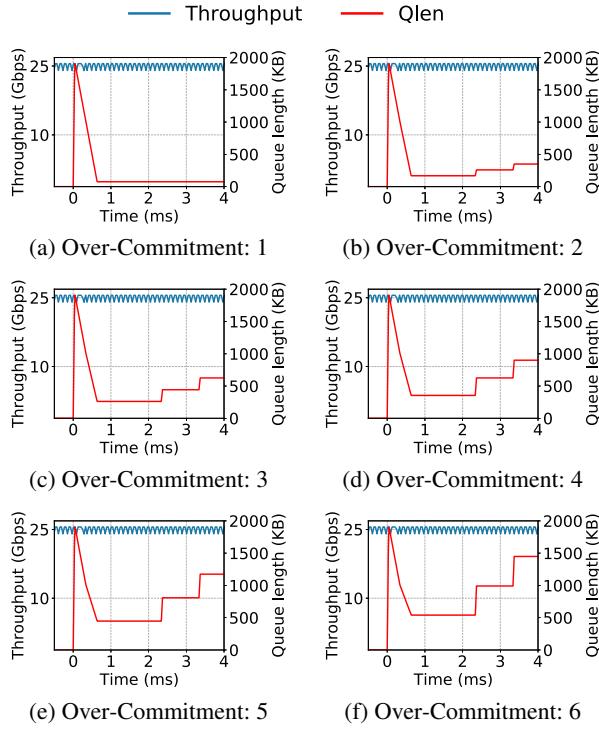


Figure 9: HOMA's reaction to 255 : 1 incast at different over-commitment levels.

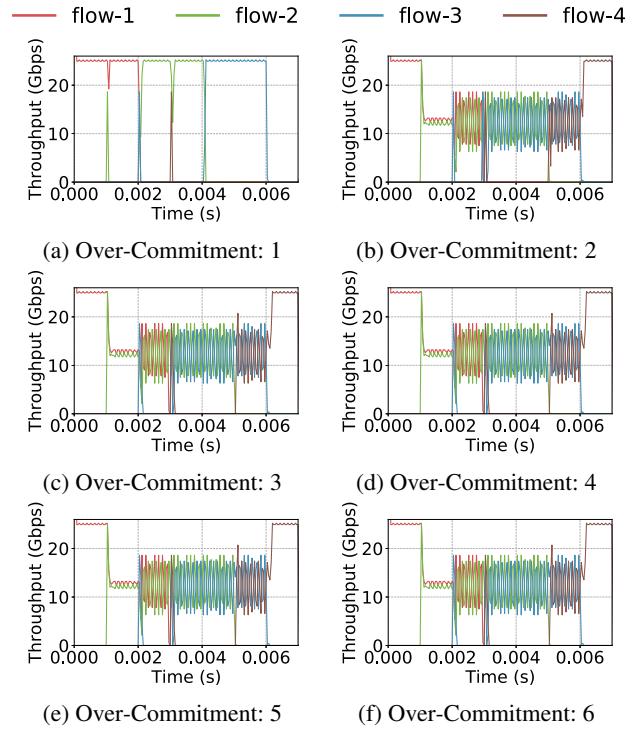


Figure 11: HOMA's fairness at different over-commitment levels.

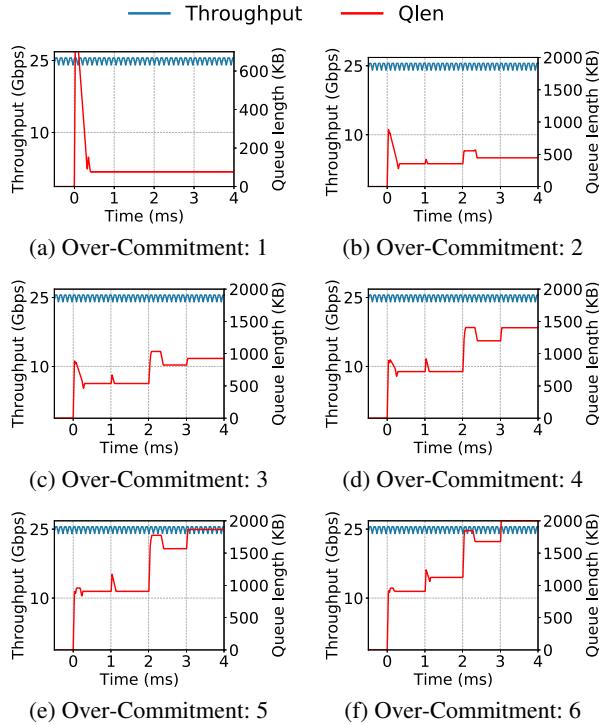


Figure 10: HOMA's reaction to 10 : 1 incast at different over-commitment levels.

## D θ-POWERTCP

We present θ-POWERTCP: standalone version of POWERTCP which does not require switch support and only requires accurate packet timestamp support at the end-host.

**Algorithm 2:** θ-POWERTCP (w/o switch support)

---

```

1 /*  $t_c$  is the timestamp upon ack arrival */
2 Input : ack
3 Output: cwnd, rate
4 procedure NEWACK(ack):
5   cwndold = GETCWND(ack.seq)
6   normPower = NORMPOWER(ack)
7   UPDATEWINDOW(normPower, cwndold)
8   rate =  $\frac{cwnd}{\tau}$ 
9   prevRTT = RTT
10  tcprev = tc
11  UPDATEOLD(cwnd, ack.seq)
12  function NORMPOWER(ack):
13    dt = tc - tcprev
14     $\dot{\theta} = \frac{RTT - prevRTT}{dt}$   $\triangleright \frac{dRTT}{dt}$ 
15     $\Gamma_{norm} = \frac{(\dot{\theta}+1) \times RTT}{\tau}$   $\triangleright \Gamma_{norm}$  :Normalized power
16     $\Gamma_{smooth} = \frac{\Gamma_{smooth} \cdot (\tau - \Delta t) + \Gamma_{norm} \cdot \Delta t}{\tau}$ 
17    return  $\Gamma_{smooth}$ 
18  function UPDATEWINDOW(power, ack):
19    if ack.seq < lastUpdated then  $\triangleright$  per RTT
20      | return cwnd
21    end if
22    cwnd =  $\gamma \times \left( \frac{cwnd_{old}}{normPower} + \beta \right) + (1 - \gamma) \times cwnd$ 
23     $\triangleright \gamma$ : EWMA parameter
24     $\triangleright \beta$ : Additive Increase
25    lastUpdated = snd_nxt
26  return cwnd

```

---

# RDMA is Turing complete, we just did not know it yet!

Waleed Reda

*Université catholique de Louvain  
KTH Royal Institute of Technology*

Marco Canini

*KAUST*

Dejan Kostić

*KTH Royal Institute of Technology*

Simon Peter

*University of Washington*

## Abstract

It is becoming increasingly popular for distributed systems to exploit offload to reduce load on the CPU. Remote Direct Memory Access (RDMA) offload, in particular, has become popular. However, RDMA still requires CPU intervention for complex offloads that go beyond simple remote memory access. As such, the offload potential is limited and RDMA-based systems usually have to work around such limitations.

We present RedN, a principled, practical approach to implementing complex RDMA offloads, without requiring any hardware modifications. Using *self-modifying* RDMA chains, we lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. We explore what is possible in terms of offload complexity and performance with a commodity RDMA NIC. We show how to integrate these RDMA chains into applications, such as the Memcached key-value store, allowing us to offload complex tasks such as key lookups. RedN can reduce the latency of key-value get operations by up to  $2.6\times$  compared to state-of-the-art KV designs that use one-sided RDMA primitives (e.g., FaRM-KV), as well as traditional RPC-over-RDMA approaches. Moreover, compared to these baselines, RedN provides performance isolation and, in the presence of contention, can reduce latency by up to  $35\times$  while providing applications with failure resiliency to OS and process crashes.

## 1 Introduction

As server CPU cycles become an increasingly scarce resource, offload is gaining in popularity [23, 28, 30–32, 36]. System operators wish to reserve CPU cycles for application execution, while common, oft-repeated operations may be offloaded. NIC offloads, in particular, have the benefit that they reside in the network data path and NICs can carry out operations on in-flight data with low latency [31].

For this reason, remote direct memory access (RDMA) [15] has become ubiquitous [20]. Mellanox ConnectX NICs [4] have pioneered ubiquitous RDMA support and Intel has added RDMA support to their 800 series of Ethernet network adapters [7]. RDMA focuses on the offload of simple message

passing (via SEND/RECV verbs) and remote memory access (via READ/WRITE verbs) [15]. Both primitives are widely used in networked applications and their offload is extremely useful. However, RDMA is not designed for more complex offloads that are also common in networked applications. For example, remote data structure traversal and hash table access are not normally deemed realizable with RDMA [39]. This led to many RDMA-based systems requiring multiple network round-trips or to reintroduce involvement of the server’s CPU to execute such requests [18, 22, 26, 27, 35, 37, 41].

To support complex offloads, the networking community has developed a number of SmartNIC architectures [2, 3, 11, 14, 17]. SmartNICs incorporate more powerful compute capabilities via CPUs or FPGAs. They can execute arbitrary programs on the NIC, including complex offloads. However, these SmartNICs are not ubiquitous and their smaller volume implies a higher cost. SmartNICs can cost up to  $5.7\times$  more than commodity RDMA NICs (RNICs) at the same link speed (§2.1). Due to their custom architecture, they are also a management burden to the system operator, who has to support SmartNICs apart from the rest of the fleet.

We ask whether we can avoid this tradeoff and attempt to use the ubiquitous RNICs to realize complex offloads. To do so, we have to solve a number of challenges. First, we have to answer if and how we can use the RNIC interface, which consists only of simple data movement verbs (READ, WRITE, SEND, RECV, etc.) and no conditionals or loops, to realize complex offloads. Our solution has to be general so that offload developers can use it to build complex *RDMA programs* that can perform a wide range of functionality. Second, we have to ensure that our solution is efficient and that we understand the performance and performance variability properties of using RNICs for complex offloads. Finally, we have to answer how complex RNIC offloads integrate with existing applications.

In this paper, we show that RDMA is *Turing complete*, making it possible to use RNICs to implement complex offloads. To do so, we implement conditional branching via *self-modifying* RDMA verbs. Clever use of the existing compare

and-swap (CAS) verb enables us to dynamically modify the RNIC execution path by editing subsequent verbs in an RDMA program, using the CAS operands as a predicate. Just like self-modifying code executing on CPUs, self-modifying verbs require careful control of the execution path to avoid consistency issues due to RNIC verb prefetching. To do so, we rely on the WAIT and ENABLE RDMA verbs [28, 34] that provide execution dependencies. WAIT allows us to halt execution of new verbs until past verbs have completed, providing strict ordering among RDMA verbs. By controlling verb prefetching, ENABLE enforces consistency for verbs modified by preceding verbs. ENABLE also allows us to create loops by re-triggering earlier, already-executed verbs in an RDMA work queue—allowing the NIC to operate autonomously without CPU intervention.

Based on these primitives, we present RedN, a principled, practical approach to implementing complex RNIC offloads. Using self-modifying RDMA programs, we develop a number of building blocks that lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. Using these abstractions, we explore what is possible in terms of offload complexity and performance with just a commodity RNIC. We show how to integrate complex RNIC offloads, developed with RedN principles, into existing networked applications. RedN affords offload developers a practical way to implement complex NIC offloads on commodity RNICs, without the burden of acquiring and maintaining SmartNICs. Our code is available at: <https://redn.io>.

We make the following contributions:

- We present RedN, a principled, practical approach to offloading arbitrary computation to RDMA NICs. RedN leverages RDMA ordering and compare-and-swap primitives to build conditionals and loops. We show that these primitives are sufficient to make RDMA Turing complete.
- Using RedN, we present and evaluate the implementation of various offloads that are useful in common server computing scenarios. In particular, we implement hash table lookup with Hopscotch hashing and linked list traversal.
- We evaluate the complexity and performance of offload in a number of use cases with the Memcached key-value store. In particular, we evaluate offload of common key-value get operations, as well as performance isolation and failure resiliency benefits. We demonstrate that RNIC offload with RedN can realize all of these benefits. It can reduce average latency of get operations by up to  $2.6\times$  compared to state-of-the-art one-sided RDMA key-value stores (e.g., FaRM-KV [22]), as well as traditional two-sided RPC-over-RDMA implementations. Moreover, RedN provides superior performance isolation, improving latency by up to  $35\times$  under contention, while also providing higher availability under host-side failures.

## 2 Background

RDMA was conceived for high-performance computing (HPC) clusters, but it has grown out of this niche [20]. It

is becoming ever-more popular due to the growth in network bandwidth, with stagnating growth in CPU performance, making CPU cycles an increasingly scarce resource that is best reserved to running application code. With RNICs now considered commodity, it is opportunistic to explore the use-cases where their hardware can yield benefits. These efforts, however, have been limited by the RDMA API, which constrains the expression of many complex offloads. Consequently, the networking community has built SmartNICs using FPGAs and CPUs to investigate new complex offloads.

### 2.1 SmartNICs

To enable complex network offloads, SmartNICs have been developed [1, 2, 10, 11]. SmartNICs include dedicated computing units or FPGAs, memory, and several dedicated accelerators, such as cryptography engines. For example, Mellanox BlueField [11] has  $8\times$  ARMv8 cores with 16GB of memory and  $2\times$  25GbE ports. These SmartNICs are capable of running full-fledged operating systems, but also ship with lightweight runtime systems that can provide kernel-bypass access to the NIC’s IO engines.

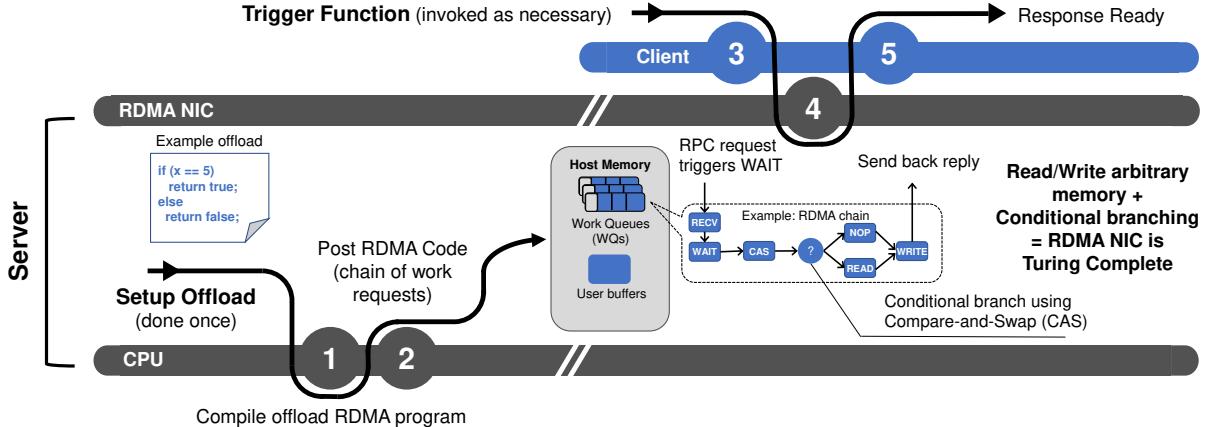
**Related work on SmartNIC offload.** SmartNICs have been used to offload complex tasks from server CPUs. For example, StRoM [39] uses an FPGA NIC to implement RDMA verbs and creates generic kernels (or building blocks) that perform various functions, such as traversing linked lists. KV-Direct [30] uses an FPGA NIC to accelerate key-value accesses. iPipe [31] and Floem [36] are programming frameworks that simplify complex offload development for primarily CPU-based SmartNICs. E3 [32] transparently offloads microservices to SmartNICs.

**The cost of SmartNICs.** While SmartNICs provide the capabilities for complex offloads, they come at a cost. For example, a dual-port 25GbE BlueField SmartNIC at \$2,340 costs  $5.7\times$  more than the same-speed ConnectX-5 RNIC at \$410 (cf. [13]). Another cost is the additional management required for SmartNICs. SmartNICs are a special piece of complex equipment that system administrators need to understand and maintain. SmartNIC operating systems and runtimes can crash, have security flaws, and need to be kept up-to-date with the latest vendor patches. This is an additional maintenance burden on operators that is not incurred by RNICs.

### 2.2 RDMA NICs

The processing power of RDMA NICs (RNICs) has doubled with each subsequent generation. This allows RNICs to cope with higher packet rates and more complex, hard-coded offloads (e.g., reduction operations, encryption, erasure coding).

We measure the verb processing bandwidth of several generations of Mellanox ConnectX NICs, using the Mellanox `ib_write_bw` benchmark. This benchmark performs 64B RDMA writes and, as such, it is not network bandwidth limited due to the small RDMA write size. We find that the verb processing bandwidth doubles with each generation, as we can



**Figure 1: RDMA NICs can implement complex offloads if we allow conditional branches to be expressed. Conditional branching can be implemented by using CAS verbs to modify subsequent verbs in the chain, without any hardware modification.**

see in Table 1. This is primarily due to a doubling in processing units (PUs) in each generation.<sup>1</sup> As a result, ConnectX-6 NICs can execute up to 110 million RDMA verbs per second using a single NIC port. This increased hardware performance further motivates the need for exploiting the computational power of these devices.

**Related work on RDMA offload.** RDMA has been employed in many different contexts, including accelerating key-value stores and filesystems [19, 22, 26, 35, 44], consensus [18, 27, 37, 41], distributed locking [45], and even nuanced use-cases such as efficient access in distributed tree-based indexing structures [46]. These systems operate within the confines of RDMA’s intended use as a *data movement* offload (via remote memory access and message passing). When complex functionality is required, these systems involve multiple RDMA round-trips and/or rely on host CPUs to carry out the complex operations.

Within the storage context, Hyperloop [28] demonstrated that pushing the RNIC offload capabilities is possible. Hyperloop combines RDMA verbs to implement complex storage operations, such as chain replication, without CPU involvement. However, it does not provide a blueprint for offloading arbitrary processing and cannot offload functionality that uses any type of conditional logic (e.g., walking a remote data structure). Moreover, the Hyperloop protocol is likely incompatible with next-generation RNICs, as its implementation relies on changing work request ownership—a feature that is deprecated for ConnectX-4 and newer cards.

Unlike this body of previous work, we aim to unlock the general-purpose processing power of RNICs and provide an

unprecedented level of programmability for complex offloads, by using novel combinations of existing RDMA verbs (§3).

### 3 The RedN Computational Framework

To achieve our aforementioned goals, we develop a framework that enables complex offloads, called RedN. RedN’s key idea is to combine widely available capabilities of RNICs to enable self-modifying RDMA programs. These programs—chains of RDMA operations—are capable of executing dynamic control flows with conditionals and loops. Fig. 1 illustrates the usage of RedN. The setup phase involves (1) preparing/compiling the RDMA code required for the service and (2) posting the output chain(s) of RDMA WRs to the RNIC. Clients can then use the offload by invoking a trigger (3) that causes the server’s RNIC to (4) execute the posted RDMA program, which returns a response (5) to the client upon completion.

To further understand this proposed framework, we first look into the execution models offered by RNICs, and the ordering guarantees they provide for RDMA verbs. We then look into the expressivity of traditional RDMA verbs and explore parallels with CPU instruction sets. We use these insights to describe strategies for expressing complex logic using traditional RDMA verbs, *without requiring any hardware modifications*.

#### 3.1 RDMA execution model

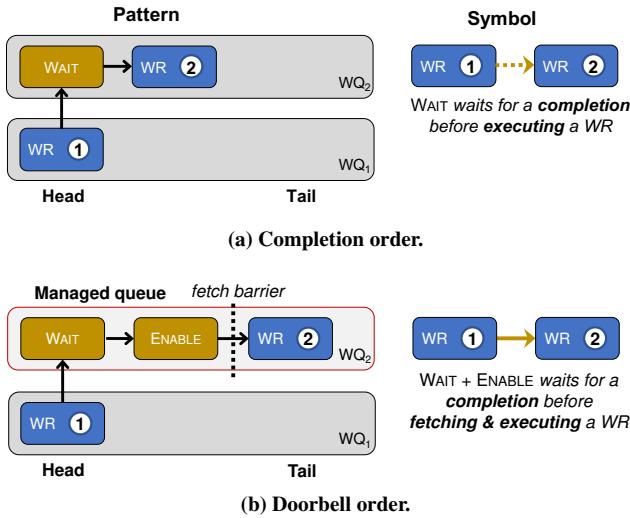
The RDMA interface specifies a number of data movement verbs (READ, WRITE, SEND, RECV, etc.) that are *posted* as *work requests* (WRs) by offload developers into *work queues* (WQs) in host memory. The RNIC starts execution of a sequence of WRs in a WQ once the offload developer triggers a *doorbell*—a special register in RNIC memory that informs the RNIC that a WQ has been updated and should be executed.

**Work request ordering.** Ordering rules for RDMA WRs distinguish between write WRs and non-write WRs that return a value. Within each category of operations, RDMA guarantees in-order execution of WRs within a single WQ. In particular, write WRs (i.e., SEND, WRITE, WRITEIMM) are totally or-

<sup>1</sup>Discussions with Mellanox affirmed our findings.

RNIC	PUs	Throughput
ConnectX-3 (2014)	2	15M verbs/s
ConnectX-5 (2016)	8	63M verbs/s
ConnectX-6 (2017)	16	112M verbs/s

**Table 1: Number of Processing Units (PUs) and performance of various ConnectX generations.**



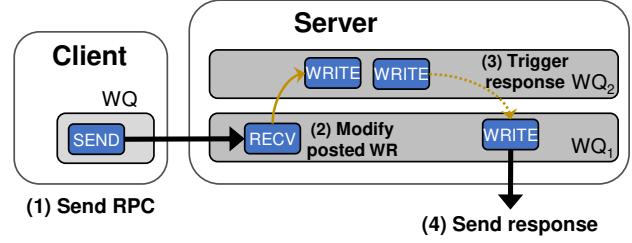
**Figure 2: Work request ordering modes that guarantee a total order of operations 2a and, a more restrictive “doorbell” order 2b, where operations are fetched by the NIC one-by-one. The symbols on the right will be used as notation for these WR chains in the examples of §3.**

dered with regard to each other, but writes may be reordered before prior non-write WRs.

We call the default RDMA ordering mode *work queue (WQ) ordering*. Sophisticated offload logic often requires stronger ordering constraints, which we construct with the help of two RDMA verbs. Fig. 2 shows two stricter ordering modes that we introduce and how to achieve them.

The WAIT verb stops WR execution until the completion of a specified WR from another WQ or the preceding WR in the same WQ. We call this *completion ordering* (Fig. 2a). It achieves total ordering of WRs along the execution chain (which potentially involves multiple WQs). It can be used to enforce data consistency, similar to data memory barriers in CPU instruction sets—to wait for data to be available before executing the WRs operating on the data. Moreover, WAIT allows developers to *pre-post* chains of RDMA verbs to the RNIC, without immediately executing them.

In all the aforementioned ordering modes, the RNIC is free to prefetch into its cache the WRs within a WQ. Thus, the execution outcome reflects the WRs at the time they were fetched, which can be incoherent with the versions that reside in host memory in case these were later modified. To avoid this issue, the RNIC allows placing a WQ into *managed* mode, in which WR prefetch is disabled. The ENABLE verb is then used to explicitly start the prefetching of WRs. This allows for existing WRs to be modified within the WQ, as long as this is done before completion of the posted ENABLE—similar to an instruction barrier. We achieve a full (data and instruction) barrier, by using WAIT and ENABLE in sequence. We call this *doorbell ordering* (Fig. 2b). Doorbell ordering allows developers to modify WR chains in-place. In particular, it allows for *data-dependent, self-modifying* WRs.



**Figure 3: Clients can trigger posted operations. Thick solid lines represent (meta)data movements.**

Thus, we have shown that we can control WR fetch and execution via special verbs, which we will exploit in the next section to develop full-fledged RDMA programs. These verbs are widely available in commodity RNICs (e.g., Mellanox terms them cross-channel communication [34]).

### 3.2 Dynamic RDMA Programs

While a static sequence of RDMA WRs is already a rudimentary RDMA program, complex offloads require *data-dependent execution*, where the logic of the offload is dependent on input arguments. To realize data-dependent execution, we construct *self-modifying RDMA code*.

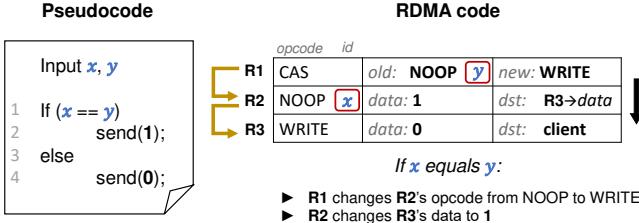
**Self-modifying RDMA code.** Doorbell ordering enables a restricted form of self-modifying code, capable of data-dependent execution. To illustrate this concept, we use the example of a server host that offloads an RPC handler to its RNIC as shown in Fig. 3. The RPC response depends on the argument set by the client and thus the RDMA offload is data-dependent. The server posts the RDMA program that consists of a set of WRs spanning two WQs. The client invokes the offload by issuing a SEND operation. At the RNIC, the SEND triggers the posted RECV operation. Observe that RECV specifies where the SEND data is placed. We configure RECV to inject the received data into the posted WR chain in WQ2 to modify its attributes. We achieve this by leveraging doorbell ordering, to ensure that posted WRs are not prefetched by the RNIC and can be altered by preceding WRs.

This is an instance of self-modifying code. As such, clients can pass arguments to the offloaded RPC handler and the RNIC will dynamically alter the executed code accordingly. However, this by itself is not sufficient to provide a Turing complete offload framework.

**Turing completeness of RDMA.** Turing completeness implies that a system of data-manipulation rules, such as RDMA, are computationally universal. For RDMA to be Turing complete, we need to satisfy two requirements [25]:

- T1:** Ability to read/write arbitrary amounts of memory.
- T2:** Conditional branching (e.g. if/else statements).

T1 can be satisfied for limited amounts of memory with regular RDMA verbs, whereas T2 has not been demonstrated with RDMA NICs. However, to truly be capable of accessing an *arbitrary* amount of memory, we need a way of realizing loops. Loops open up a range of sophisticated use-cases and



**Figure 4: Simple if example and equivalent RDMA code.** Conditional execution relies on self-modifying code using CAS to enable/disable WRs based on the operand values.

lower the number of constraints that programmers have to consider for offloads. To highlight their importance, we add them as a third requirement, necessary to fulfill the first:  
**T3:** The ability to execute code repeatedly (loops).

In the next sub-sections, we show how dynamic execution can be used to satisfy all the aforementioned requirements. A proof sketch of Turing completeness is given in Appendix A.

### 3.3 Conditionals

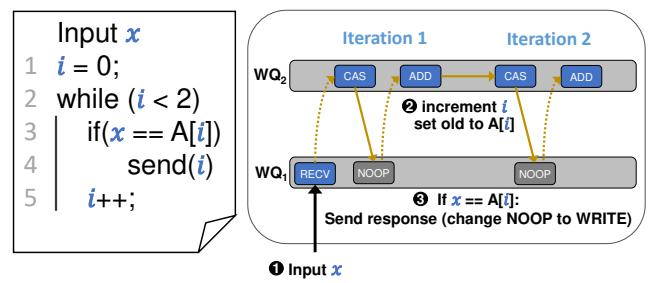
Conditional execution—choosing what computation to perform based on a runtime condition—is typically realized using conditional branches, which are not readily available in RDMA. To this end, we introduce a novel approach that uses self-modifying CAS verbs. The main insight is that this verb can be used to check a condition (i.e., equality of  $x$  and  $y$ ) and then perform a swap to modify the attributes of a WR. We describe how this is done in Fig. 4. We insert a CAS that compares the 64-bit value at the address of R2's *opcode* attribute (initially NOOP) with its *old* parameter (also initially NOOP). We then set the *id* field of R2 to  $x$ . This field can be manipulated freely without changing the behavior of the WR, allowing us to use it to store  $x$ . Operand  $y$  is stored in the corresponding position in the *old* field of R1. This means that if  $x$  and  $y$  are equal, the CAS operation will succeed and the value in R1's *new* field—which we set to WRITE—will replace R2's opcode. Hence, in the case  $x = y$ , R2 will change from a NOOP into a WRITE operation. This WRITE is set to modify the *data* value of the return operation (R3) to 1. If  $x$  and  $y$  are not equal, the default value 0 is returned.

Now that we have established the utility of this technique for basic conditionals, we next look into how to can be used to support loop constructs.

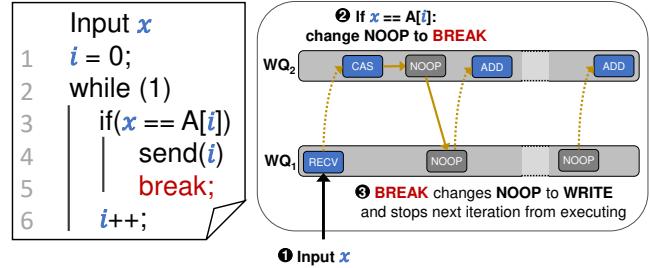
### 3.4 Loops

To support loop constructs efficiently, we require (1) conditional branching to test the loop condition and break if necessary, and (2) WR re-execution, to repeat the loop body. We develop each, in turn, below.

Consider the while loop example in Figure 5. This offload searches for  $x$  in an array  $A$  and sends the corresponding index. The loop is static because  $A$  has finite size (in this case, size = 2), known a priori. To simplify presentation, consider the case  $A[i] = i, \forall i$ . Without this simplification, the example would include an additional WRITE to fetch the value at  $A[i]$ .



**Figure 5: while loop using CAS.** Loop is unrolled since loop size is fixed and set to 2.



**Figure 6: while loop with breaks realized using CAS.** To implement breaks, we use CAS to change a NOOP WR to an RDMA WRITE, which then stops subsequent iterations from executing.

The loop body uses a CAS verb to implement the if condition (line 3), followed by an ADD verb to increment  $i$  (line 6). Given that the loop size is known a priori (size = 2), RedN can unroll the while loop in advance and post the WRs for all iterations. As such, there is no need to check the condition at line 2. For each iteration, if the CAS succeeds, the NOOP verb in  $WQ_1$  will be changed to WRITE—which will send the response back to the client. However, it is clear that, regardless of the comparison result, all subsequent iterations will be executed. This is inefficient since, if the send (line 4) occurs before the loop is finished, a number of WRs will be wastefully executed by the NIC. This is impractical for larger loop sizes or if the number of iterations is not known a priori.

**Unbounded loops and termination.** Figure 6 modifies the previous example to make it such that the loop is unbounded. For efficiency, we add a break that exits the loop if the element is found. The role of break is to prevent additional iterations from being executed. We use an additional NOOP that is formatted such that, once transformed into a WRITE by the CAS operation, it prevents the execution of subsequent iterations in the loop. This is done by modifying the last WR in the loop such that it does not trigger a completion event. The next iteration in the loop, which WAITS on such an event (via completion ordering), will therefore not be executed. Moreover, the WRITE will also modify the opcode of the WR used to send back the response from NOOP to WRITE.

As such, break allows efficient and unbounded loop execution. However, it still remains necessary for the CPU to post WRs to continue the loop after all its WRs are executed. This consumes CPU cycles and can even increase latency if the CPU is unable to keep up with the speed of WR execution.

RedN Constructs		Number of WRs	Operand limit [bits]
	if	1C + 1A + 3E	48
while	Unrolled	1C + 1A + 3E	
	Recycled	3C + 2A + 4E	

**Table 2: Breakdown of the overhead of our constructs with different offload strategies.** *C* refers to copy verbs, *A* refers to atomic RDMA verbs, and *E* refers to WAIT/ENABLE verbs. while loops that use WQ recycling incur 2 additional READS, 1 ADD, and 1 ENABLE WR.

**Unbounded loops via WQ recycling.** To allow the NIC to recycle WRs without CPU intervention, we make use of a novel technique that we call *WQ recycling*. RNICs iterate over WQs, which are circular buffers, and execute the WRs therein. By design, each WR is meant to execute only once. However, there is no fundamental reason why WRs cannot be reused since the RNIC does not actually erase them from the WQ. To enable recycling of a WR chain, we insert a WAIT and ENABLE sequence at the tail of the WQ. This instructs the RNIC to wrap around the tail and re-execute the WR chain for as many times as needed.

It is important to note that WQ recycling is not a panacea. To allow the tail of the WQ to wrap around, all posted WAIT and ENABLE WRs in the loop need to have their *wqe\_count* attribute updated. This attribute is used to determine the index of the WR that these ordering verbs affect. In ConnectX NICs, these indices are maintained internally by the RNIC and their values are monotonically increasing (instead of resetting after the WQ wraps around). As such, the *wqe\_count* values need to be incremented to match. This incurs overhead (as seen in Table 2) and requires an additional ADD operation in combination with other verbs. As such, loop unrolling, where each iteration is manually posted by the CPU, is overall less taxing on the RNIC. However, WQ recycling avoids CPU intervention, allowing the offload to remain available even amid host software failures (as we will see later in §5.6).

### 3.5 Putting it all together

With conditional branching, we can dynamically alter the control flow of any function on an RNIC. Loops allow us to traverse arbitrary data structures. Together, we have transformed an RNIC into a general processing unit. In this section, we discuss the usability aspects from overhead, security, programmability, and expressiveness perspectives.

**Building blocks.** We abstract and parameterize the RDMA chains required for conditional branching and looping into if and while constructs. The overhead in terms of RDMA WR chains of our constructs is shown in Table 2. We can see a breakdown of the minimum number of operations required for each. Inequality predicates, such as  $<$  or  $>$ , can also be supported by combining equality checks with MAX or MIN, as seen later in Table 3. However, their availability is vendor-specific and currently only supported by ConnectX NICs.

**Operand limits.** RedN’s limit is based on the supported size for the CAS verb, which is 64 bits. The operand is provided

as a 48-bit value, encoded in its *id* and other neighboring fields (which can also be freely modified without affecting execution). The remaining bits are used for modifying the opcode of the WR depending on the result of the comparison. We note that our advertised limits only signify what is possible with the number of operations we allocate for our constructs. For instance, despite the 48-bit operand limit for our constructs, we can chain together multiple CAS operations to handle different segments of a larger operand (we do not rely on the atomicity property of CAS). As such, there is no fundamental limitation, only a performance penalty.

**Offload setup.** To offload an RDMA program, clients first create an RDMA connection to the target server and send an RPC to initiate the offload. We envision that the server already has the offload code; however, other ways of deploying the offload are possible. Upon receiving a connection request, the server creates one or more managed local WQs to post the offloaded code. Next, it registers two main types of memory regions for RDMA access: (a) a code region, and (b) a data region. The code region is the set of remote RDMA WQs created on the server, which are unique to each client and need to be accessible via RDMA to allow self-modifying code. Code regions are protected by memory keys—special tokens required for RDMA access—upon registration (at connection time), prohibiting unauthorized access. The data region holds any data elements used by the offload (e.g., a hash table). Data regions can be shared or private, depending on the use-case.

**Security.** RedN does not solve security challenges in existing RDMA or Infiniband implementations [40]. However, RedN can help RDMA systems become more secure. For such systems, *one-sided* RDMA operations (e.g., RDMA READ and WRITE) are frequently used [22, 28, 33, 35, 42, 43] as they avoid CPU overheads at the responder. However, doing so requires clients to have direct read and/or write memory access. This can compromise security if clients are buggy and/or malicious. To give an example, FaRM allows clients to write messages directly to shared RPC buffers. This requires clients to behave correctly, as they could otherwise overwrite or modify other clients’ RPCs. RedN allows applications to use *two-sided* RDMA operations (e.g., SEND and RECV), which do not require direct memory access, while still fully bypassing server CPUs. As we demonstrate in our use-cases in §5, SEND operations can be used to trigger offload programs without any CPU involvement.

**Isolation.** Given that RedN implements dynamic loops, clients can abuse such constructs to consume more than their fair share of resources. Luckily, popular RNICs, like ConnectX, provide WQ rate-limiters [6] for performance isolation. As such, even if clients trigger non-terminating offload code, they still have to adhere to their assigned rates. Moreover, offloaded code can be configured by the servers to be auditable through completion events, created automatically after a WR is executed. These events can be monitored and servers can terminate connections to clients running misbehaving code.

**Parallelism.** RDMA WR fetch and execution latencies are more costly compared to CPU instructions, as WRs are fetched/executed via PCIe (microseconds vs. nanoseconds). As such, to hide WR latencies, it is important to parallelize logically unrelated operations. Like threads of execution in a CPU, each WQ is allocated a single RNIC PU to ensure in-order execution without inter-PU synchronization. As such, we carefully tune our offloaded code to allow unrelated verbs to execute on independent queues to be able to parallelize execution as much as possible. The benefits of parallelism are evaluated in §5.2.

## 4 Implementation

Our offload framework is implemented in C with  $\sim 2,300$  lines of code—this includes our use cases ( $\sim 1400$ ), and convenience wrappers for RDMA verbs (*libverbs*) API ( $\sim 900$ ).

Our approach does not require modifying any RDMA libraries or drivers. RedN uses low-level functions provided by Mellanox’s ConnectX driver (*libmlx5*) to expose in-memory WQ buffers and register them to the RNIC, allowing WRs to be manipulated via RDMA verbs. We configure the ConnectX-5 firmware to allow the WR *id* field to be manipulated freely, which is required for conditional operations as well as WR recycling. This is done by modifying specific configuration registers on the NIC [12].

RedN is compatible with any ConnectX NICs that support WAIT and ENABLE (e.g., ConnectX-3 and later models).

## 5 Evaluation

We start by characterizing the underlying RNIC performance (§5.1) to understand how it affects our implemented programming constructs. Then, in our evaluation against state-of-the-art RNIC and SmartNIC offloads, we show that RedN:

1. Speeds up remote data structure traversals, such as hash tables (§5.2) and linked lists (§5.3) compared to vanilla RDMA offload;
2. Accelerates (§5.4) and provides performance isolation (§5.5) for the Memcached key-value store;
3. Provides improved availability for applications (§5.6)—allowing them to run in spite of OS & process crashes;
4. Exposes programming constructs generic enough to enable a wide-variety of use-cases (§5.2–§5.6);

**Testbed.** Our experimental testbed consists of  $3 \times$  dual-socket Haswell servers running at 3.2 GHz, with a total of 16 cores, 128 GB of DRAM, and 100 Gbps dual-port Mellanox ConnectX-5 Infiniband RNICs. All nodes are running Ubuntu 18.04 with Linux Kernel version 4.15 and are connected via back-to-back Infiniband links.

**NIC setup.** For all of our experiments, we use reliable connection (RC) RDMA transport, which supports the RDMA synchronization features we use. All WQs that enforce doorbell order are initialized with a special “managed” flag to disable the driver from issuing doorbells after a WR is posted. The WQ size is set to match that of the offloaded program.

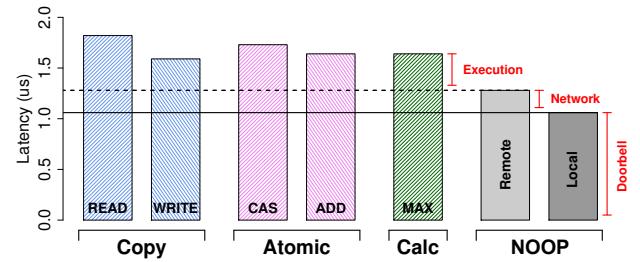


Figure 7: Latencies of different RDMA verbs. The solid line marks the latency of ringing the doorbell via MMIO. The difference between dashed and solid lines estimates network latency.

### 5.1 Microbenchmarks

We run microbenchmarks to break down RNIC verb execution latency, understand the overheads of our different ordering modes, and determine the processing bandwidth of different RDMA verbs and of our constructs.

#### 5.1.1 RDMA Latency

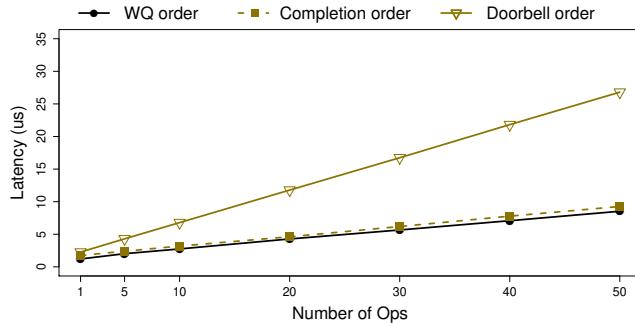
We break down the performance of RDMA verbs, configured to perform 64B IO, by measuring their average latencies after executing them 100K times. All verbs are executed remotely, unless otherwise stated. As seen in Fig. 7, WRITE has a latency of  $1.6 \mu\text{s}$ . It uses posted PCIe transactions, which are one-way. Comparatively, non-posted verbs such as READ or atomics such as fetch-and-add (ADD) and compare-and-swap (CAS) need to wait for a PCIe completion and take  $\sim 1.8 \mu\text{s}$ .<sup>2</sup> Overall, the execution time difference is small among verbs, even for more advanced, vendor-specific *Calc* verbs that perform logical and arithmetic computations (e.g., MAX).

To break down the different latency components for RDMA verb execution, we first estimate the latency of issuing a doorbell and copying the WR to the RNIC. This can be done by measuring the execution time of a NOOP WR. This time can be subtracted from the latencies of other WRs to give an estimate of their execution time once the WR is available in the RNIC’s cache. We also quantify the network cost by executing remote and local loopback NOOP WRs (shown on the right-hand side) and measuring the difference—roughly  $0.25 \mu\text{s}$  for our back-to-back connected nodes. Overall, these results show low verb execution latency, justifying building more sophisticated functions atop. We next measure the implications of ordering for offloads.

#### 5.1.2 Ordering Overheads

We show the latency of executing chains of RDMA verbs using different ordering modes. All the posted WRs within a chain are NOOP, to simplify isolating the performance impact of ordering. We start by measuring the latency of executing a chain of verbs posted to the same queue but absent any constraints (WQ order), and compare it to the ordering modes

<sup>2</sup>Older-generation NICs (e.g., ConnectX-4) use a proprietary concurrency control mechanism to implement atomics, resulting in higher latencies than later generations that rely on PCIe atomic transactions.



**Figure 8: Execution latency of RDMA verbs posted using different ordering modes. More restrictive modes such as Doorbell order add non-negligible overheads as it requires the NIC to fetch WRs sequentially.**

that we introduced in Fig. 2—completion order and doorbell order. WQ order only mandates in-order updates to memory, which allows for increased concurrency. Operations that are not modifying the same memory address can execute concurrently and the RNIC is free to prefetch multiple WRs with a single DMA<sup>3</sup>. We can see in Fig. 8 that the latency of a single NOOP is 1.21 μs and the overhead of adding subsequent verbs is roughly 0.17 μs per verb. The first verb is slower since it requires an initial doorbell to tell the NIC that there is outstanding work. For completion ordering, less concurrency is possible since WRs await the completions of their predecessors, and the overhead of increases slightly to 0.19 μs per additional WR. For doorbell order, no latency-hiding is possible, as the NIC has to fetch WRs from memory one-by-one, which results in an overhead of 0.54 μs per additional WR. These results signify that, doorbell ordering should be used conservatively, as there is more than 0.5 μs latency increase for every instance of its use, compared to more relaxed ordering modes.

### 5.1.3 RDMA Verb Throughput

We show the throughput of the common RDMA verbs in Table 3 for a single ConnectX-5 port. ConnectX cards assign compute resources on a per port basis. For ConnectX-5, each port has 8 PUs. Atomic verbs, such as CAS, offer a comparatively limited throughput ( $8 \times$  lower than regular verbs) due to memory synchronization across PCIe.

In addition, we measure the performance of RedN’s if and while constructs. Using 48-bit operands, a ConnectX-5 NIC can execute 700K if constructs per second. This is due to the need for CAS to ensure doorbell ordering between CAS and the subsequent WR it modifies. This causes the throughput to be bound by NIC processing limits. Unrolled while loops require the same number of verbs per iteration as an if statement and their throughput is identical. while loops with WQ recycling have reduced performance due to having to execute more WRs per iteration.

<sup>3</sup>The number of operations fetched by the RNIC can change dynamically. The Prefetch mechanism in ConnectX RNICs is proprietary.

Operation		Throughput (M ops/s)	Support
Atomic	CAS	8.4	Native
	ADD		
Copy	READ	65	Mellanox
	WRITE		
Calc	MAX	63	Mellanox
Constructs	if	0.7	RedN
	while	Unrolled	
		Recycled	

**Table 3: Throughput of common RDMA verbs and RedN’s constructs on a single port of a ConnectX-5. if and unrolled while have identical performance. while loops with WQ recycling require additional WRs and therefore have a lower throughput.**

## 5.2 Offload: Hash Lookup

After evaluating the overheads of RedN’s ordering modes and constructs, we next look into the performance of RedN for offloading remote access to popular data structures. We first look into hash tables, given their prominent use in key-value stores for indexing stored objects. To perform a simple *get* operation, clients first have to lookup the desired key-value entry in the hash table. The entry can either have the value directly inlined or a pointer to its memory address. The value is then fetched and returned back to the client. Hopscotch hashing is a popular hashing scheme that resolves collisions by using  $H$  hashes for each entry and storing them in 1 out of  $H$  buckets. Each bucket has a *neighborhood* that can probabilistically hold a given key. A lookup might require searching more than one bucket before the matching key-value entry is found. To support dynamic value sizes, we assume the value is not inlined in the bucket and is instead referenced via a pointer.

For distributed key-value stores built with RDMA, *get* operations are usually implemented in one of two ways:

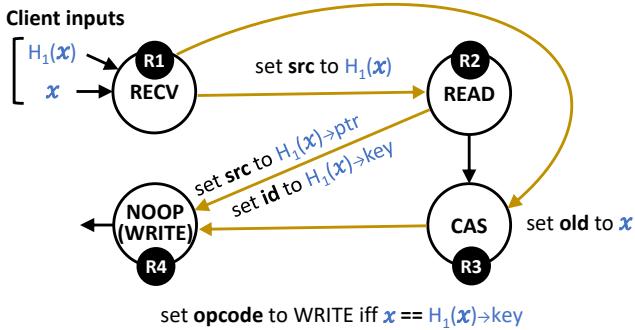
**One-sided** approaches first retrieve the key’s location using a one-sided RDMA READ operation and then issue a second READ to fetch the value. These approaches typically require two network round-trips at a minimum. This greatly increases latency but does not require involvement of the server’s CPU. Many systems utilize this approach to implement lookups, including FaRM [22] and Pilaf [35].

**Two-sided** approaches require the client to send a request using an RDMA SEND or WRITE. The server intercepts the request, locates the value and then returns it using one of the aforementioned verbs. This widely used [19, 26] approach follows traditional RPC implementations and avoids the need for several roundtrips. However, this comes at the cost of server CPU cycles.

### 5.2.1 RedN’s Approach

To offload key-value *get* operations, we leverage the offload schemes introduced in §3.3 and §3.4.

Fig. 9 describes the RDMA operations involved for a single-hash lookup. To *get* a value corresponding to a key, the client first computes the hashes for its key. For this use-case, we set the number of hashes to two, which is common in practice [24]. The client then performs a SEND with the value of

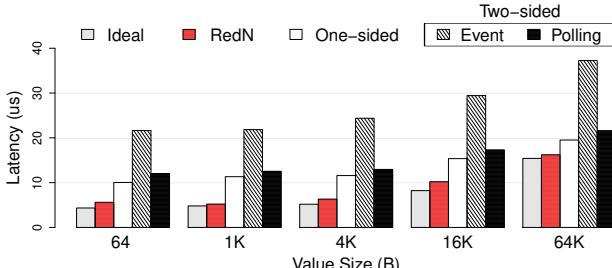


**Figure 9: Hash lookup RDMA program.** Black arrows indicate order of execution of WRs in their WQs. Brown arrows indicate self-modifying code dependencies and require doorbell ordering.  $x$  is the requested key and  $H_1(x)$  is its first hash. The acronym  $src$  indicates the “source address” field of WRs.  $old$  indicates the “expected value” at the target address of the CAS operation. The  $id$  field is used for storing conditional operands.

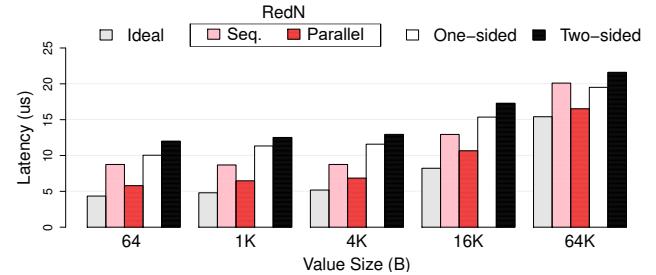
the key  $x$  and address of the first bucket  $H_1(x)$ , which are then captured via a RECV WR posted on the server. The RECV WR (R1) inserts  $x$  into the  $old$  field of the CAS WR (R3) and the bucket address  $H_1(x)$  into the READ WR (R2). The READ WR retrieves the bucket and sets the source address ( $src$ ) of the response WR (R4) to the address of the value ( $ptr$ ). It also inserts the bucket’s key into the  $id$  field to prepare it for the conditional check. Finally, CAS (R3) checks whether the expected value  $old$ , which is set to key  $x$ , matches the  $id$  field in (R4), which is set to the bucket’s  $key$ . If equal, (R4)’s opcode is changed from NOOP to WRITE, which then returns the value from the bucket. Given that each key may be stored in multiple buckets (two in our setup), these lookups may be performed sequentially or in parallel, depending on the offload configuration.

### 5.2.2 Results

We evaluate our approach against both one-sided and two-sided implementations of key-value *get* operations. We use FaRM’s approach [22] to perform one-sided lookups. FaRM uses Hopscotch hashing to locate the key using approximately two RDMA READs — one for fetching the buckets in a neighborhood that hold the key-value pairs and another for reading the actual value. The neighborhood size is set to 6 by default, implying a  $6\times$  overhead for RDMA metadata operations. For



**Figure 10: Average latency of hash lookups.** *Ideal* shows the latency of a single network round-trip READ.



**Figure 11: Average latency of hash lookups during collisions.** *Ideal* shows the latency of a single network round-trip READ.

two-sided lookups, our RPC to the host involves a client-initiated RDMA SEND to transmit the *get* request, and an RDMA WRITE initiated by the server to return the value after performing the lookup.

**Latency.** Fig. 10 shows a latency comparison of KV *get* operations of RedN against one-sided and two-sided baselines. We evaluate two distinct variations of two-sided. The *event-based* approach blocks for a completion event to avoid wasting CPU cycles, whereas the *polling-based* approach dedicates one CPU core for polling the completion queue. We use 48-bit keys and vary the value size. The value size is given on the x-axis. In this scenario, we assume no hash collisions and that all keys are found in the first bucket. RedN is able to outperform all baselines — fetching a 64 KB key-value pair in 16.22  $\mu$ s, which is within 5% of a single network round-trip READ (Ideal). RedN is able to deliver close-to-ideal performance because it bypasses the server’s CPU *and* fetches the value in a single network RTT. Compared to RedN, one-sided operations incur up to  $2\times$  higher latencies, as they require two RTTs to fetch a value. Two-sided implementations do not incur any extra RTT; however, they require server CPU intervention. The polling-based variant consumes an entire CPU core but provides competitive latencies. Event-based approaches block for completion events to avoid wasting CPU cycles and incur much higher latencies as a consequence. RedN is able to outperform polling-based and event-based approaches by up to 2 and  $3.8\times$ , respectively. Given the much higher latencies of event-based approaches, for the remainder of this evaluation, we will only focus on polling-based approaches and simply refer to them hereafter as *two-sided*.

Fig. 11 shows the latency in the presence of hash collisions. In this case, we assume a worst case scenario, where the key-value pair is always found in the second bucket. In this scenario, we introduce two offload variants for RedN — RedN-Seq & RedN-Parallel. The former performs bucket lookups sequentially within a single WQ. The latter parallelizes bucket lookups by performing the lookups across two different WQs to allow execution on different NIC PUs. We can see that RedN-Parallel maintains similar latencies to lookups with no hash collisions (i.e., RedN in Fig. 10), since bucket lookups are almost completely parallelized. It is worth noting that parallelism in this case does not cause unnecessary data movement, since the value is only returned when the corresponding

Hash lookup	IO Size			
	$\leq 1\text{ KB}$		64 KB	
Port config.	Single	Dual	Single	Dual
Rate (ops/s)	500K	1M	180K	190K
Bottleneck	NIC PU	IB bw	PCIe bw	

Table 4: NIC throughput of hash lookups and its bottlenecks.

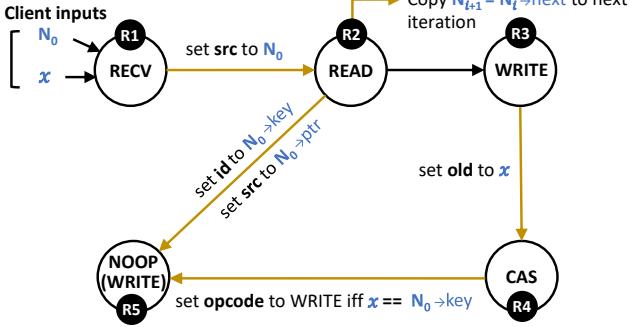


Figure 12: Linked list RDMA program.

key is found. For the other bucket, the WRITE operation (R4 in Fig. 9) is a NOOP. RedN-Seq, on the other hand, incurs at least 3  $\mu\text{s}$  of extra latency as it needs to search the buckets one-by-one. As such, whenever possible, operations with no dependencies should be executed in parallel. The trade-off is having to allocate extra WQs for each level of parallelism.

**Throughput.** We describe our throughput in Table 4. At lower IO, RedN is bottlenecked by the NIC’s processing capacity due to the use of doorbell ordering—reaching 500K ops/s on a single port (1M ops/s with dual ports). At 64 KB, RedN reaches the single-port IB bandwidth limit (~ 92 Gbps). Dual-port configs are limited by ConnectX-5’s 16  $\times$  PCIe 3.0 lanes.

**SmartNIC comparison.** We compare our performance for hashtable *gets* against StRoM [39], a programmable FPGA-based SmartNIC. Since we do not have access to a programmable FPGA, we extract the results from [39] for comparison, and report them in Table 5. RedN uses the same experimental settings as before. Our hashtable configuration is functionally identical to StRoM’s and our client and server nodes are also connected via back-to-back links. We can see that RedN provides lower lookup latencies than StRoM. StRoM uses a Xilinx Virtex 7 FPGA, which runs at 156.25 MHz, and incurs at least two PCIe roundtrips to retrieve the key and value. Our evaluation shows that RedN can provide latency that is in-line with more expensive SmartNICs.

IO Size	System	Median	99 <sup>th</sup> ile
64 B	RedN	5.7 $\mu\text{s}$	6.9 $\mu\text{s}$
	StRoM	~7 $\mu\text{s}$	~7 $\mu\text{s}$
4 KB	RedN	6.7 $\mu\text{s}$	8.4 $\mu\text{s}$
	StRoM	~12 $\mu\text{s}$	~13 $\mu\text{s}$

Table 5: Latency comparison of hash *gets*. Results for StRoM obtained from [39].

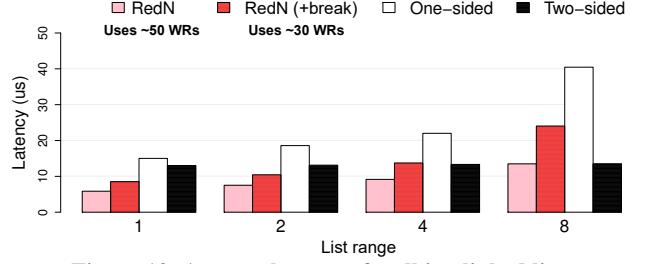


Figure 13: Average latency of walking linked lists.

### 5.3 Offload: List Traversal

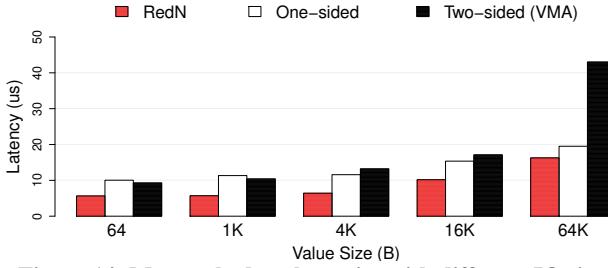
Next, we explore another data structure also popularly used in storage systems. We focus on linked lists that store key-value pairs, and evaluate the overhead of traversing them remotely using our offloads. Similar to the previous use-case, we focus on one-sided approaches, as used by FaRM and Pilaf [22, 35].

Linked list processing can be decomposed into a *while* loop for traversing the list and an *if* condition for finding and returning the key. We describe the implementation of our offload in Fig. 12. The client provides the key  $x$  and address of the first node in the list  $N_0$ . A READ operation (R2) is then performed to read the contents of the first node and update the values for the return operation (R5). We also use a WRITE operation (R3) to prepare the CAS operation (R4) by inserting key  $x$  in its *old* field. As an optimization, this WRITE can be removed and, instead,  $x$  can be inserted directly by the RECV operation. This, however, will need to be done for every CAS to be executed and, as such, this approach is limited to smaller list sizes, since RECVs can only perform 16 scatters.

For this use-case, we introduce two offload variations. The first, referred to simply as RedN, uses the implementation in Fig. 12. The second uses an additional *break* statement between R4 and R5 to exit the loop in order to avoid executing any additional operations.

#### 5.3.1 Results

Fig. 13 shows the latency of one-sided and two-sided variants against RedN at various linked list ranges — where range represents the highest list element that the key can be randomly placed in. The size of the list itself is set to a constant value of 8. We setup the linked list to use key and value sizes of 48 bits and 64 bytes, respectively, and perform 100k list traversals for each system. The requested key is chosen at random for each RPC. In the variant labelled “RedN”, we do not use *breaks* and assume that all 8 elements of the list need to be searched. RedN outperforms all baselines for all list ranges until 8 — providing up to a 2× improvement. RedN (+break) executes a break statement with each iteration and performs worse than RedN due to the extra overhead of checking the condition of the *break*. However, using a break statement increases the offload’s overall efficiency since no unneeded iterations are executed after the key is found — using an average of 30 WRs across all experiments. Without breaks, RedN will need to execute all subsequent iterations even after the key-value



**Figure 14: Memcached *get* latencies with different IO sizes.**

pair is found/returned and it uses more than 65% more WRs. As such, while RedN is able to provide better latencies, using a break statement is more sensible for longer lists.

#### 5.4 Use Case: Accelerating Memcached

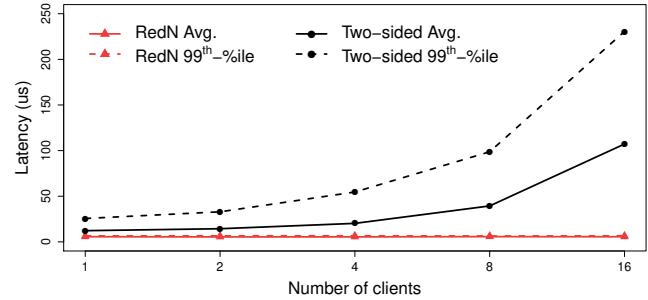
Based on our earlier experience offloading remote data structure traversals, we set out to see: 1) how effective our aforementioned techniques are in a real system, and 2) what are the challenges in deploying it in such settings. Memcached is a key-value store that is often used as a caching service for large-scale storage services. We use a version of Memcached that employs cuckoo hashing [24]. Since Memcached does not natively support RDMA, we modify it with  $\sim 700$  LoC to integrate RDMA capabilities, allowing the RNIC to register the hash table and storage object memory areas. We also modify the buckets, so that the addresses to the values are stored in big endian — to match the format used by the WR attributes. We then use RedN to offload Memcached’s *get* requests to allow them to be serviced directly by the RNIC without CPU involvement. We compare our results to various configurations of Memcached.

To benchmark Memcached, we use the Memtier benchmark, configure it to use UDP (to reduce TCP overheads for the baselines), and issue 1 million *get* operations using different key-value sizes. To create a competitive baseline for two-sided approaches, we use Mellanox’s VMA [9]—a kernel-bypass userspace TCP/IP stack that boosts the performance of sockets-based applications by intercepting their socket calls and using kernel-bypass to send/receive data. We configure VMA in polling-mode to optimize for latency. In addition, we also implement a one-sided approach, similar to the one introduced in section 5.2.

Fig. 14 shows the latency of *gets*. As we can see, RedN’s offload for hash *gets* is up to  $1.7\times$  faster than one-sided and  $2.6\times$  faster than two-sided. Despite the latter being configured in polling-mode, VMA incurs extra overhead since it relies on a network stack to process packets. In addition, to adhere to the sockets API, VMA has to memcpy data from send and receive buffers, further inflating latencies—which is why it performs comparatively worse at higher value sizes.

#### 5.5 Use Case: Performance Isolation

One of the benefits of exposing the latent turing power of RNICs is to enforce isolation among applications. CPU con-



**Figure 15: Memcached *get* latencies under hardware contention with varying numbers of writer-clients.**

tention in multi-tenant and cloud settings can lead to arbitrary context switches, which can, in turn, inflate average and tail latencies. We explore such a scenario by sending background traffic to Memcached using one or more writer (clients). These writers generate *set* RPCs in a closed loop to load the Memcached service. At the same time, we use a single reader client to generate *get* operations. To stress CPU resources while minimizing lock contention, each reader/writer is assigned a distinct set of 10K keys, which they use to generate their queries. The keys within each set are accessed by the clients sequentially.

We can see in Fig. 15 that, as we increase the # of writers, both the average and 99<sup>th</sup> percentile latencies for two-sided increase dramatically. For RedN, CPU contention has no impact on the performance of the RNIC and both the average and 99<sup>th</sup> percentiles sit below 7  $\mu$ s. At 16 writers, RedN’s 99<sup>th</sup> percentile latency is  $35\times$  lower than the baseline.

This indicates that RNIC offloads can also have other useful effects. Service providers may opt to offload high priority traffic for more predictable performance or allocate server resources to tenants to reduce contention.

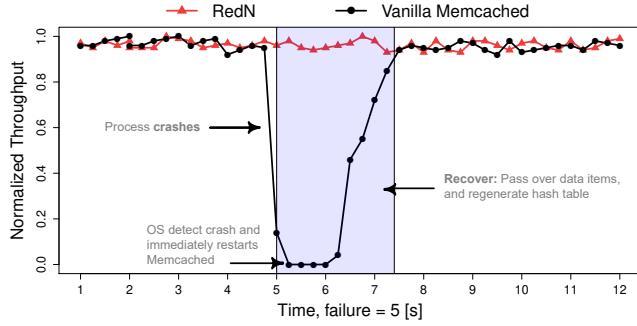
#### 5.6 Use Case: Failure Resiliency

We now consider server failures and how failure is affected by RNICs. Table 6 shows failure rates of server software and hardware components. NICs are much less likely to fail than software components—NIC annualized failure rate (AFR) is an order of magnitude lower. Even more importantly, NICs are partially decoupled from their hosts and can still access memory (or NVM) in the presence of an OS failure. This means that RNICs are capable of offloading key system functionality that can allow servers to continue operating despite OS failures (albeit in a degraded state). To put this to the test, we conduct a fail-over experiment to explore how RedN can enhance a service’s failure resiliency.

**Process crashes.** We look into how we can allow an RNIC to continue serving RPCs after a Memcached instance crashes. We find that this is not simple in practice. RNICs access many resources in application memory (e.g., queues, doorbell records, etc.) that are required for functionality. If the process hosting these resources crashes, the memory belonging to

Component	AFR	MTTF	Reliability
OS	41.9%	20,906	99%
DRAM	39.5%	22,177	99%
NIC	1.00%	876,000	99.99%
NVM	< 1.00%	2 million	99.99%

**Table 6:** Failure rates of different server components [8, 37]. AFR means annualized failure rate, whereas MTTF stands for mean time to failure and is expressed in hours. RNICs can still access memory even in the presence of an OS failure.



**Figure 16:** RedN can survive process crashes and continue serving RPCs via the RNIC without interruption.

these components will be automatically freed by the operating system resulting in termination of the RDMA program. To counteract this, we use [38] forks to create an empty hull parent for hosting RDMA resources and then allow Memcached to run as a child process. Linux systems do not free the resources of a crashed child until the parent also terminates. As such, keeping the RDMA resources tied to an empty process allows us to continue operating in spite of application failures. We run an experiment (timeline shown in Fig. 16) where we send *get* queries to a single instance of Memcached and then simply kill Memcached during the run. The OS detects the application’s termination and immediately restarts it. Despite this, we can see that a vanilla Memcached instance will take at least 1 second to bootstrap, and 1.25 additional seconds to build its metadata and hashtables. With RedN, no service disruption is experienced and *get* queries continue to be issued without recovery time.

**OS failure.** We also programmatically induce a kernel panic using `sysctl`, freezing the system. This is a simpler case than process crashes, since we no longer have to worry about the OS freeing RDMA resources. For brevity, we do not show these results, but we experimentally verified that RedN offloads continue operating in the presence of an OS crash.

## 6 Discussion

**Client scalability.** RedN requires servers to manage at least two WQs per client, which is not higher than other RDMA systems. RedN can still introduce scalability challenges with thousands of clients since RNIC cache is limited. However, Mellanox’s dynamically-connected (DC) transport service [5], which allows unused connections to be recycled, can circumvent many such scalability limits.

**Offload for sockets-based applications.** Protocols such as rsocket [16] can be used to transparently convert sockets-based applications to use RDMA, making them possible targets for RedN. Although rsocket does not support popular system calls, such as `epoll`, other extensions have been proposed [29] that support a more comprehensive list of system calls and were shown to work with applications like Memcached and Redis.

**Intel RNICs.** Next-generation Intel RNICs are expected to support atomic verbs, such as CAS—which RedN uses to implement conditionals. To control when WRs can be fetched by the NIC, Intel uses a validity bit in each WR header. This bit can be dynamically modified via an RDMA operation to mimic ENABLE. However, there is no equivalent for the WAIT primitive, meaning that clients cannot trigger a pre-posted chain. One possible workaround for this is to use another PCIe device on the server to issue a doorbell to the RNIC, allowing the WR chain to be triggered. We leave the exploration of such techniques as future work.

**Insights for next-generation RNICs.** Our experience with RedN has shown that keeping WRs in server memory (to allow them to be modified by other RDMA verbs) is a key bottleneck. If the NIC’s cache was made directly accessible via RDMA, WRs can be pre-fetched in advance and unnecessary PCIe round-trips on the critical path can be avoided. We hope future RNICs will support such features.

## 7 Conclusion

We show that, in spite of appearances, commodity RDMA NICs are Turing-complete and capable of performing complex offloads without *any* hardware modifications. We take this insight and explore the feasibility and performance of these offloads. We find that, using a commodity RNIC, we can achieve up to  $2.6 \times$  and  $35 \times$  speed-up versus state-of-the-art RDMA approaches, for key-value get operations under uncontended and contended settings, respectively, while allowing applications to gain failure resiliency to OS and process crashes. We believe that this work opens the door for a wide variety of innovations in RNIC offloading which, in turn, can help guide the evolution of the RDMA standard.

RedN is available at <https://redn.io>.

**Acknowledgements.** This work has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 770889), as well as NSF grant 1751231. Waleed Reda was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC), funded by the European Commission (EACEA) (FPA 2012-0030). We would like to thank Gerald Q. Maguire Jr. and our anonymous reviewers for their comments and feedback as well as Jasmine Murcia. Thanks also go to our shepherd Ang Chen.

## References

- [1] Agilio CX SmartNICs. <https://www.netronome.com/products/agilio-cx/>.
- [2] Catapult. <https://www.microsoft.com/en-us/research/project/project-catapult/>.
- [3] Cavium-Xpliant. <https://www.openswitch.net/cavium/>.
- [4] ConnectX series. <https://www.mellanox.com/products/ethernet/connectx-smartnic>.
- [5] Dynamically Connected (DC) QPs. [https://docs.mellanox.com/display/rdmacore50/DynamicallyConnected\(DC\)QPs](https://docs.mellanox.com/display/rdmacore50/DynamicallyConnected(DC)QPs).
- [6] ibv\_modify\_qp\_rate\_limit(3) - Linux man page. [https://man7.org/linux/man-pages/man3/ibv\\_modify\\_qp\\_rate\\_limit.3.html](https://man7.org/linux/man-pages/man3/ibv_modify_qp_rate_limit.3.html).
- [7] Intel Ethernet 800 Series Network Adapters. <https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-800-series-network-adapters/e810-cqda1-100gbe-brief.html>.
- [8] Intel Optane DC Persistent Memory - Product Brief. <https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf>.
- [9] LibVMA. <https://github.com/Mellanox/libvma/wiki/Architecture>.
- [10] LiquidIO II SmartNICs. <https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html>.
- [11] Mellanox BlueField. <https://www.mellanox.com/products/bluefield-overview>.
- [12] Mellanox PCX. <https://github.com/Mellanox/pcx/tree/master/config>.
- [13] Mellanox store. <http://store.mellanox.com/>.
- [14] NetFPGA platform. <https://netfpga.org/>.
- [15] RDMA RFC. <https://tools.ietf.org/html/rfc5040>.
- [16] rsocket(7) - Linux man page. <https://linux.die.net/man/7/rsocket>.
- [17] Stingray. <https://www.broadcom.com/products/ethernet-connectivity/smarnic>.
- [18] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi. The Impact of RDMA on Agreement. *arXiv preprint arXiv:1905.12143*, 2019.
- [19] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel. Assise: Performance and Availability via NVM Colocation in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.
- [20] O. Cardona. Towards Hyperscale High Performance Computing with RDMA, 2019. [https://pc.nanog.org/static/published/meetings/NANOG76/1999/20190612\\_Cardona\\_Towards\\_Hyperscale\\_High\\_v1.pdf](https://pc.nanog.org/static/published/meetings/NANOG76/1999/20190612_Cardona_Towards_Hyperscale_High_v1.pdf).
- [21] S. Dolan. mov is Turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.
- [22] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.
- [23] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, 2019.
- [24] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.
- [25] M. Gabbielli and S. Martini. *Programming Languages: Principles and Paradigms*, page 145. Undergraduate Topics in Computer Science. Springer London, 2010.
- [26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.
- [27] M. Kazhamiaka, B. Memon, C. Kankanamge, S. Sahu, S. Rizvi, B. Wong, and K. Daudjee. Sift: resource-efficient consensus with RDMA. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 260–271, 2019.
- [28] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 297–312, 2018.

- [29] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socks-Direct: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103. 2019.
- [30] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.
- [31] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.
- [32] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.
- [33] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.
- [34] Mellanox RDMA Aware Networks Programming User Manual. [https://www.mellanox.com/related-docs/prod\\_software/RDMA\\_Aware\\_Programming\\_user\\_manual.pdf](https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf).
- [35] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.
- [36] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.
- [37] M. Poke and T. Hoefer. Dare: High-performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.
- [38] A. Rosenbaum. Multiprocess Sharing of RDMA Resources, 2018. [https://openfabrics.org/images/2018workshop/presentations/103\\_ARosenbaum\\_Multi-ProcessSharing.pdf](https://openfabrics.org/images/2018workshop/presentations/103_ARosenbaum_Multi-ProcessSharing.pdf).
- [39] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart Remote Memory. *Proceedings of the Fifteenth EuroSys Conference*, 2020.
- [40] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.
- [41] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.
- [42] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, 2018.
- [43] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.
- [44] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.
- [45] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with RDMA: decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1571–1586, 2018.
- [46] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 741–758, 2019.

## Appendix A Turing completeness sketch

To show that RDMA is turing complete, we need to establish that RDMA has the following three properties:

1. Can read/write arbitrary amounts of memory.
2. Has conditional branching (e.g., if & else statements).
3. Allows nontermination.

Our paper already demonstrates that these properties can be satisfied using our constructs but, for completeness, we also analogize our system with x86 assembly instructions that have been proven to be capable of simulating a Turing machine. Dolan [21] demonstrated that this is in fact possible using just the x86 `mov` instruction. As such, we need to prove that RDMA has sufficient expressive power to emulate the `mov` instruction.

### A.1 Emulating the x86 `mov` instruction

To provide an RDMA implementation for `mov`, we first need to consider the different addressing modes used by Dolan [21] to simulate a Turing machine. The addressing mode describes how a memory location is specified in the `mov` operands.

Table 7 shows a list of all required addressing modes, their x86 syntax, and one possible implementation for each with RDMA.  $R$  operands denote registers but, since RDMA operations can only perform memory-to-memory transfers, we assume these registers are stored in memory. For simplicity, we only focus on `mov` instructions used to perform *loads* but note that *stores* can be implemented in a similar manner.

For *immediate* addressing, the operand is part of the instruction and is passed directly to register  $R_{dst}$ . This can be implemented simply using an `WRITEIMM` which takes a constant in its *immediate* parameter and writes it to a specified memory location (register  $R_{dst}$  in this case). To perform more complex operations, *indirect* allows `mov` to use the value of

the operand as a memory address. This enables the dynamic modification of the address at runtime, since it depends on the contents of the register when the instruction is executed. To implement this, we use two write operations with doorbell ordering (refer to §3.1 for a discussion of our ordering modes). The first `WRITE` changes the *source address* attribute of the second `WRITE` operation to the value in register  $R_{src}$ . This allows the second `WRITE` operation to write to register  $R_{dst}$  using the value at the memory address pointed to by  $R_{src}$ . Lastly, *indexed* addressing allows us to add an offset ( $R_{off}$ ) to the address in register  $R_{src}$ . This can be done by simply performing an RDMA ADD operation between the two writes with doorbell ordering, in order to add the offset register value  $R_{off}$  to  $R_{src}$ . This allows us to finally write the value  $[R_{src} + R_{off}]$  to  $R_{dst}$ . With these three implementations, we showcase that RDMA can in fact emulate all the required `mov` instruction variants.

### A.2 Allowing nontermination

To simulate a real Turing machine, we need to also satisfy the code nontermination requirement. In the x86 architecture, this can be achieved via an unconditional jump [21] that loops back to the start of the program. For RDMA, this can also be achieved by having the CPU re-post the WRs after they are executed. While this is sufficient for Turing completeness it, nevertheless, wastes additional CPU cycles and can also impact latency if CPU cores are busy or unable to keep up with WR execution. As an alternative, RedN provides a way to loop back without any CPU interaction by relying on `WAIT` and `ENABLE` to recycle RDMA WRs (as described in §3.4). Regardless of which approach is employed, RDMA is capable of performing an unconditional jump to the beginning of the program. This means that we can emulate all x86 instructions used by Dolan [21] for simulating a Turing machine.

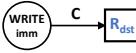
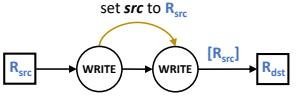
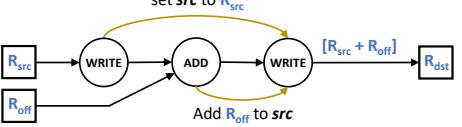
Addressing mode	x86 syntax	RedN equivalent
Immediate	<code>mov R<sub>dst</sub>, C</code>	
Indirect	<code>mov R<sub>dst</sub>, [R<sub>src</sub>]</code>	
Indexed	<code>mov R<sub>dst</sub>, [R<sub>src</sub> + R<sub>off</sub>]</code>	

Table 7: Addressing modes for the x86 `mov` instruction and their RDMA implementation in RedN.



# FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism

Rajath Shashidhara<sup>1</sup>    Tim Stamler<sup>2</sup>  
<sup>1</sup>University of Washington

Antoine Kaufmann<sup>3</sup>    Simon Peter<sup>1</sup>  
<sup>2</sup>UT Austin    <sup>3</sup>MPI-SWS

## Abstract

FlexTOE is a flexible, yet high-performance TCP offload engine (TOE) to SmartNICs. FlexTOE eliminates almost all host data-path TCP processing and is fully customizable. FlexTOE interoperates well with other TCP stacks, is robust under adverse network conditions, and supports POSIX sockets.

FlexTOE focuses on data-path offload of established connections, avoiding complex control logic and packet buffering in the NIC. FlexTOE leverages fine-grained parallelization of the TCP data-path and segment reordering for high performance on wimpy SmartNIC architectures, while remaining flexible via a modular design. We compare FlexTOE on an Agilio-CX40 to host TCP stacks Linux and TAS, and to the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive performance for RPCs, even with wimpy SmartNICs. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS, respectively. FlexTOE’s data-path parallelism generalizes across hardware architectures, improving single connection RPC throughput up to 2.4× on x86 and 4× on BlueField. FlexTOE supports C and XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

## 1 Introduction

TCP remains the default protocol in many networks, even as its CPU overhead is increasingly a burden to application performance [3, 17, 46]. A long line of improvements to software TCP stack architecture has reduced overheads: Careful packet steering improves cache-locality for multi-cores [17, 24, 45], kernel-bypass enables safe direct NIC access from user-space [3, 46], application libraries avoid system calls for common socket operations [17], and fast-paths drastically reduce TCP processing overheads [19]. Yet, even with these optimizations, communication-intensive applications spend up to 48% of per-CPU cycles in the TCP stack and NIC driver (§2.1).

Offload promises further reduction of CPU overhead. While moving parts of TCP processing, such as checksum and segmentation, into the NIC is commonplace [54], full TCP offload engines (TOEs) [6, 7, 33] have so far failed to find widespread adoption. A primary reason is that fixed offloads [56] limit protocol evolution after deployment [9, 29, 36]. Tonic [2] provides building blocks for flexible transport protocol offload to FPGA-SmartNICs, but FPGA development is still difficult and slow.

We present FlexTOE, a high-performance, yet flexible offload of the widely-used TCP protocol. FlexTOE focuses on

scenarios that are common in data centers, where connections are long-lived and small transfers are common [29]. FlexTOE offloads the TCP data-path to a network processor (NPU) based SmartNIC, enabling full customization of transport logic and flexibility to implement data-path features whose requirements change frequently in data centers. Applications interface directly but transparently with the FlexTOE datapath through the *libTOE* library that implements POSIX sockets, while FlexTOE offloads all TCP data-path processing (§2.1).

TCP data-path offload to SmartNICs is challenging. SmartNICs support only restrictive programming models with stringent per-packet time budgets and are geared towards massive parallelism with wimpy cores [26]. They often lack timers, as well as floating-point and other computational support, such as division. Finally, offload has to mask high-latency operations that cross PCIe. On the other hand, TCP requires computationally intensive and stateful code paths to track in-flight segments, for reassembly and retransmission, and to perform congestion control [2]. For each connection, the TCP data-path needs to provide low processing tail latency and high throughput and is also extremely sensitive to reordering.

Resolving the gap between TCP’s requirements and SmartNIC hardware capabilities requires careful offload design to efficiently utilize SmartNIC capabilities. Targeting FlexTOE at the TCP data-path of established connections avoids complex control logic in the NIC. FlexTOE’s offloaded data-path is one-shot for each TCP segment—segments are never buffered in the NIC. Instead, per-socket buffers are kept in per-process host memory where libTOE interacts with them directly. Connection management, retransmission, and congestion control are part of a separate control-plane, which executes in its own protection domain, either on control cores of the SmartNIC or on the host. To provide scalability and flexibility, we decompose the TCP data-path into fine-grained modules that keep private state and communicate explicitly. Like microservices [29], FlexTOE modules leverage a data-parallel execution model that maximizes SmartNIC resource use and simplifies customization. We organize FlexTOE modules into a *data-parallel computation pipeline*. We also *reorder* segments on-the-fly to support parallel, out-of-order processing of pipeline stages, while enforcing in-order TCP segment delivery. To our knowledge, no prior work attempting full TCP data-path offload to NPU SmartNICs exists.

We make the following contributions:

- We characterize the CPU overhead of TCP data-path processing for common data center applications (§2.1). Our analysis shows that up to 48% of per-CPU cycles are spent in TCP data-path processing, even with optimized TCP stacks.

- We present FlexTOE, a flexible, high-performance TCP offload engine (§3). FlexTOE leverages data-path processing with fine-grained parallelism for performance, but remains flexible via a modular design. We show how to decompose TCP into a data-path and a control-plane, and the data-path into a data-parallel pipeline of processing modules to hide SmartNIC processing and data access latencies.
- We implement FlexTOE on the Netronome Agilio-CX40 NPU SmartNIC architecture, as well as x86 and Mellanox BlueField (§4). Using FlexTOE design principles, we are the first to demonstrate that NPU SmartNICs can support scalable, yet flexible TCP data-path offload. Our code is available at <https://tcp-acceleration-service.github.io/FlexTOE>.
- We evaluate FlexTOE on a range of workloads and compare to Linux, the high-performance TAS [19] network stack, and a Chelsio Terminator TOE [6] (§5). We find that the Memcached [32] key-value store scales throughput up to 38% better on FlexTOE than using TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS respectively, 27% higher throughput than Chelsio for bidirectional long flows, and an order of magnitude higher throughput under 2% packet loss than Chelsio. We extend the FlexTOE data-path with debugging and auditing functionality to demonstrate flexibility. FlexTOE maintains high performance when interoperating with other network stacks. FlexTOE’s data-path parallelism generalizes across platforms, improving single connection RPC throughput up to 2.4× on x86 and 4× on BlueField.

## 2 Background

We motivate FlexTOE by analyzing TCP host CPU processing overheads of related approaches (§2.1). We then place FlexTOE in context of this and further related work (§2.2). Finally, we survey the relevant *on-path* SmartNIC architecture (§2.3).

### 2.1 TCP Impact on Host CPU Performance

We quantify the impact of different TCP processing approaches on host CPU performance in terms of CPU overhead, execution efficiency, and cache footprint, when processing common RPC-based workloads. We do so by instrumenting a single-threaded Memcached [32] server application using hardware performance counters (cf. §5 for details of our testbed). We use the popular memtier\_benchmark [51] to generate the client load, consisting of 32 B keys and values, using as many clients as necessary to saturate the server, executing closed-loop KV transactions on persistent connections. Table 1 shows a breakdown of our server-side results, for each Memcached request-response pair, into NIC driver, TCP/IP stack, POSIX sockets, Memcached application, and other factors.

**In-kernel.** Linux’s TCP stack is versatile but bulky, leading to a large cache footprint, inefficient execution, and high CPU overhead. Stateless offloads [54], such as segmentation

Module	Linux		Chelsio		TAS		FlexTOE	
	kc	%	kc	%	kc	%	kc	%
NIC driver	0.71	6	1.28	14	0.18	5	0	0
TCP/IP stack	4.25	35	0.40	4	1.44	43	0	0
POSIX sockets	2.48	21	2.61	29	0.79	23	0.74	44
Application	1.26	10	1.31	16	0.85	26	0.89	53
Other	3.42	28	3.28	37	0.09	3	0.04	3
Total	12.13	100	8.89	100	3.34	100	1.67	100
Retiring	4.60	38	2.43	27	1.66	48	0.77	46
Frontend bound	3.53	29	1.52	17	0.46	13	0.34	21
Backend bound	3.40	28	4.68	53	1.24	36	0.46	27
Bad speculation	0.55	5	0.26	3	0.13	4	0.09	6
Instructions (k)	16.18		8.14		6.26		2.93	
IPC	1.33		0.92		1.85		1.75	
Icache (KB)	47.50		73.43		39.75		19.00	

**Table 1.** Per-request CPU impact of TCP processing.

and generic receive offload [12], reduce overhead for large transfers, but they have minimal impact on RPC workloads dominated by short flows. We find that Linux executes 12.13 kc per Memcached request on average, with only 10% spent in the application. Not only does Linux have a high instruction and instruction cache (Icache) footprint, but privilege mode switches, scattered global state, and coarse-grained locking lead to 62% of all cycles spent in instruction fetch stalls (frontend bound), cache and TLB misses (backend bound), and branch mispredictions (cf. [19]). These inefficiencies result in 1.33 instructions per cycle (IPC), leveraging only 33% of our 4-way issue CPU architecture. Linux is, in principle, easy to modify, but kernel code development is complex and security sensitive. Hence, introducing optimizations and new network functionality to the kernel is often slow [29, 42, 43].

**Kernel-bypass.** Kernel-bypass, such as in mTCP [17] and Arrakis [46], eliminates kernel overheads by entrusting the TCP stack to the application, but it has security implications [52]. TAS [19] and Snap [29] instead execute a protected user-mode TCP stack on dedicated cores, retaining security and performance. By eliminating kernel calls, TAS spends only 800 cycles in the socket API—31% of Linux’s API overhead. TAS also reduces TCP stack overhead to 34% of Linux. TAS reduces Icache footprint, front and back-end CPU stalls, improving IPC by 40% versus Linux, and reducing the total per-request CPU impact to 27% of Linux. However, kernel-bypass still has significant overhead. Only 26% of per-request cycles are spent in Memcached—the remainder is spent in TAS (breakdown in §C).

**Inflexible TCP offload.** TCP offload can eliminate host CPU overhead for TCP processing. Indeed, TOEs [7] that offload the TCP data-path to the NIC have existed for a long time. Existing approaches, such as the Chelsio Terminator [6], hard-wire the TCP offload. The resulting inflexibility prevents data center operators from adapting the TOE to their needs

and leads to a slow upgrade path due to long hardware development cycles. For example, the Chelsio Terminator line has been slow to adapt to RPC-based data center workloads.

Chelsio’s inflexibility shows in our analysis. Despite drastically reducing the host TCP processing cycles to 10% of Linux and 28% of TAS, Chelsio’s TOE only modestly reduces the total per-request CPU cycles of Memcached by 27% versus Linux and inflates them by 2.6× versus TAS. Chelsio’s design requires interaction through the Linux kernel, leading to a similar execution profile despite executing 50% fewer host instructions per request. In addition, Chelsio requires a sophisticated TOE NIC driver, with complex buffer management and synchronization. Chelsio’s design is inefficient for RPC processing and leaves only 16% of the total per-request cycles to Memcached—6% more than Linux and 10% fewer than TAS.

**FlexTOE.** FlexTOE eliminates all host TCP stack overheads. FlexTOE’s instruction (and Icache) footprint is at least 2× lower than the other stacks, leading to an execution profile similar to TAS, where 46% of all cycles are spent retiring instructions. In addition, 53% of all cycles can be spent in Memcached—an improvement of 2× versus TAS, the next best solution. The remaining cycles are spent in the POSIX sockets API, which cannot be eliminated with TCP offload.

FlexTOE is also flexible, allowing operators to modify the TOE at will. For example, we have modified the TCP data-path many times, implementing many features that require TOE modification, including scalable socket API implementations [24, 45], congestion control protocols [1, 34], scalable flow scheduling [53], scalable PCIe communication protocols [44], TCP tracing [13], packet filtering and capture (tcpdump and PCAP), VLAN stripping, programmable flow classification (eBPF [30]), firewalling, and connection splicing similar to AccelTCP [37]. All of these features are desirable in data centers and are adapted frequently.

## 2.2 Related Work

Beyond the TCP implementations covered in §2.1, we cover here further related work in SmartNIC offload, parallel packet processing, and API and network protocol specialization.

**SmartNIC offload.** On-path SmartNICs (§2.3), based on network processor units (NPUs) and FPGAs, provide a suitable substrate for flexible offload. Arsenic [47] is an early example of flexible packet multiplexing on a SmartNIC. Microsoft’s Catapult [48] offloads network management, while Dagger [22] offloads RPC processing to FPGA-SmartNICs. Neither offloads a transport protocol, like TCP. AccelTCP [37] offloads TCP connection management and splicing [28] to NPU-SmartNICs, but keeps the TCP data-path on the host using mTCP [17]. Tonic [2] demonstrates in simulation that high-performance, flexible TCP transmission offload might be possible, but it stops short of implementing full TCP data-path offload (including receiver processing) in a non-simulated environment. LineFS [20] offloads a distributed file system to an off-path

SmartNIC, leveraging parallelization to hide execution latencies of wimpy SmartNIC CPUs and data access across PCIe. Taking inspiration from Tonic and LineFS, but also from actor, and microservice-based approaches presented in iPipe [26], E3 [27], and Click [23, 38], FlexTOE shows how to decompose the TCP data-path into a fine-grained data-parallel pipeline to support full and flexible offload to on-path NPU-SmartNICs.

**Parallel packet processing.** RouteBricks [8] parallelizes across cores and cluster nodes for high-performance routing, achieving high line-rates but remaining flexible via software programmability. Routing relies on read-mostly state and is simple compared to TCP. FlexTOE applies fine-grained parallelization to complex, stateful code paths.

**Specialized APIs and protocols.** Another approach to lower CPU utilization is specialization. R2P2 [21] is a UDP-based protocol for remote procedure calls (RPCs) optimized for efficient and parallel processing, both at the end-hosts and in the network. eRPC [18] goes a step further and co-designs an RPC protocol and API with a kernel-bypass network stack to minimize CPU overhead per RPC. RDMA [49] is a popular combination of a networking API, protocol, and a (typically hardware) network stack. iWARP [50], in particular, leverages a TCP stack underneath RDMA, offloading both. These approaches improve processing efficiency, but at the cost of requiring application re-design, all-or-nothing deployments, and operational issues at scale [11], often due to inflexibility [36, 56]. FlexTOE instead offloads the TCP protocol in a flexible manner by relying on SmartNICs. Upper-layer protocols, such as iWARP, can also be implemented using FlexTOE.

## 2.3 On-path SmartNIC Architecture

On-path SmartNICs<sup>1</sup>, such as Marvell Octeon [5], Pensando Capri [10, 55], and Netronome Agilio [39, 40], support massively parallel packet processing with a large pool of flow processing cores (FPCs), but they lack efficient support for sophisticated program control flow and complex computation [26].

We explore offload to the NFP-4000 NPU, used in Netronome Agilio CX SmartNICs [39]. We show the relevant architecture in Figure 1. Like other on-path SmartNICs, FPCs are organized into islands with local memory and processing resources, akin to NUMA domains. Islands are connected in a mesh via a high-bandwidth interconnect (arrows in Figure 1).

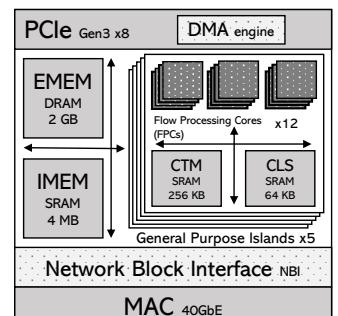


Figure 1. NFP-4000 overview.

<sup>1</sup>Mellanox BlueField [31] and Broadcom Stingray [4] are off-path SmartNICs that are not optimized for packet processing [26].

The PCIe island has up to two PCIe Gen3 x8 interfaces and a DMA engine exposing DMA transaction queues [41]. FPCs can issue up to 256 asynchronous DMA transactions to perform IO between host and NIC memory. The MAC island supports up to two 40 Gbps Ethernet interfaces, accessed via a *network block interface* (NBI).

**Flow Processing Cores (FPCs).** 60 FPCs are grouped into five general-purpose islands (each containing 12 FPCs). Each FPC is an independent 32-bit core at 800 MHz with 8 hardware threads, 32 KB instruction memory, 4 KB data memory, and CRC acceleration. While FPCs have strong data flow processing capabilities, they have small codestores, lack timers, as well as floating-point and other complex computational support, such as division. This makes them unsuitable to execute computationally and control intensive TCP functionality, such as congestion, connection, and complex retransmission control. For example, congestion avoidance involves computing an ECN-ratio (gradient). We found that it takes 1,500 cycles (1.9  $\mu$ s) per RTT to perform this computation on FPCs.

**Memory.** The NFP-4000 includes multiple memories of various sizes and performance characteristics. General-purpose islands have 64KB of island-local scratch (*CLS*) and 256 KB of island target memory (*CTM*), with access latencies of up to 100 cycles from island-local FPCs for data processing and transfer, respectively. The internal memory unit (*IMEM*) provides 4 MB of SRAM with an access latency of up to 250 cycles. The external memory unit (*EMEM*) provides 2 GB of DRAM, fronted by a 3 MB SRAM cache, with up to 500 cycles latency.

**Implications for flexible offload.** The NFP-4000 supports a broad range of protocols, but the computation and memory restrictions require careful offload design. As FPCs are wimpy and memory latencies high, sequential instruction execution is much slower than on host processors. Conventional run-to-completion processing that assigns entire connections to cores [3, 17, 19] results in poor per-connection throughput and latency. In some cases, it is beyond the feasible instruction and memory footprint. Instead, an efficient offload needs to leverage more fine-grained parallelism to limit the per-core compute and memory footprint.

### 3 FlexTOE Design

In addition to flexibility, FlexTOE has the following goals:

- **Low tail latency and high throughput.** Modern datacenter network loads consist of short and long flows. Short flows, driven by remote procedure calls, require low tail completion time, while long flows benefit from high throughput. FlexTOE shall provide both.
- **Scalability.** The number of network flows and application contexts that servers must handle simultaneously is increasing. FlexTOE shall scale with this demand.

To achieve these goals and overcome SmartNIC hardware limitations, we propose three design principles:

1. **One-shot data-path offload.** We focus offload on the TCP RX/TX data-path, eliminating complex control, compute, and state, thereby also enabling fine-grained parallelization. Further, our data-path offload is one-shot for each TCP segment. Segments are never buffered on the NIC, vastly simplifying SmartNIC memory management.
2. **Modularity.** We decompose the TCP data-path into fine-grained, customizable modules that keep private state and communicate explicitly. New TCP extensions can be implemented as modules and hooked into the data-flow, simplifying development and integration.
3. **Fine-grained parallelism.** We organize the data-path modules into a data-parallel computation pipeline that maximizes SmartNIC resource use. We map stages to FPCs, allowing us to fully utilize all FPC resources. We employ TCP segment sequencing and reordering to support parallel, out-of-order processing of pipeline stages, while enforcing in-order segment delivery.

**Decomposing TCP for offload.** We use the TAS host TCP stack architecture [19] as a starting point. TAS splits TCP processing into three components: a data-path, a control-plane, and an application library. The data-path is responsible for scalable data transport of established connections: TCP segmentation, loss detection and recovery, rate control, payload transfer between socket buffers and the network, and application notifications. The control-plane handles connection and context management, congestion control, and complex recovery involving timeouts. Finally, the application library intercepts POSIX socket API calls and interacts with control-plane and data-path using dedicated context queues in shared memory. Data-path and control-plane execute in their own protection domains on dedicated cores, isolated from untrusted applications, and communicate through efficient message passing queues.

**FlexTOE offload architecture.** In FlexTOE we adapt this architecture for offload, by designing and integrating a *data-path running efficiently* on the SmartNIC (§3.1). The FlexTOE control-plane can run on the host or on a SmartNIC control CPU, with the same functionality as in TAS (cf. §D). The FlexTOE control-plane additionally manages the SmartNIC data-path resources. Similarly, our application library (libTOE) intercepts POSIX socket calls and is dynamically linked to unmodified processes that use FlexTOE, and communicates directly with the data-path.

Figure 2 shows the offload architecture of FlexTOE, with a host control-plane (each box is a protection domain). libTOE, data-path, and control-plane communicate via pairs of *context queues* (CTX-Qs), one for each communication direction. CTX-Qs leverage PCIe DMA and MMIO or shared memory for SmartNIC-host and intra-host communication, respectively.

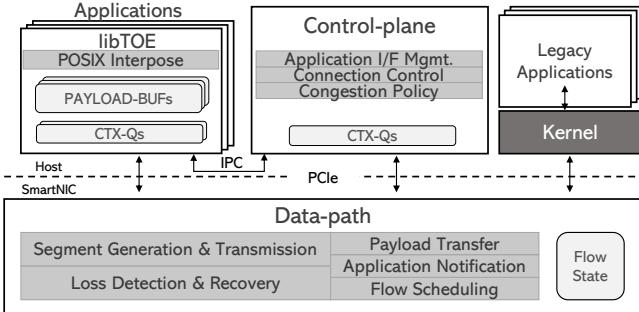


Figure 2. FlexTOE offload architecture (host control-plane).

FlexTOE supports per-thread context queues for scalability. Each TCP socket keeps receive and transmit payload buffers (PAYLOAD-BUFs) in host memory. libTOE appends data for transmission into the per-socket TX PAYLOAD-BUF and notifies the data-path using a thread-local CTX-Q. The data-path appends received segments to the socket’s RX PAYLOAD-BUF after reassembly and libTOE is notified via the same thread-local CTX-Q. Non-FlexTOE traffic is forwarded to the Linux kernel, which legacy applications may use simultaneously.

### 3.1 TCP Data-path Parallelization

To provide high offload performance using relatively wimpy SmartNIC FPCs, FlexTOE has to leverage all available parallelism within the TCP data-path. In this section, we analyze the TAS host TCP data-path to investigate what parallelism can be extracted. In particular, the TCP data-path in TAS has the following three workflows:

- **Host control (HC):** When an application wants to transmit data, executes control operations on a socket, or when retransmission is necessary, the data-path must update the connection’s transmit and receive windows accordingly.
- **Transmit (TX):** When a TCP connection is ready to send—based on congestion and flow control—the data-path prepares a segment for transmission, fetching its payload from a socket transmit buffer and sending it out to the MAC.
- **Receive (RX):** For each received segment of an established connection, the data-path must perform byte-stream reassembly—advance the TCP window, determine the segment’s position in the socket receive buffer, generate an acknowledgment to the sender, and, finally, notify the application. If the received segment acknowledges previously transmitted segments, the data-path must also free the relevant payload in the socket transmit buffer.

Host TCP stacks, such as Linux or TAS, typically process each workflow to completion in a critical section accessing a shared per-connection state structure. HC workflows are typically processed on the program threads that trigger them, while TX and RX are typically triggered by NIC interrupts and processed on high-priority (kernel or dedicated) threads.

For efficient offload, we decompose this data-path into an up to five-stage parallel pipeline of processing modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, and *context queue*.

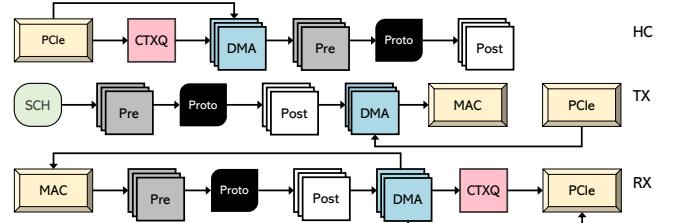


Figure 3. Per-connection data-path workflows. *Protocol* is atomic. Other stages may be replicated for parallelism.

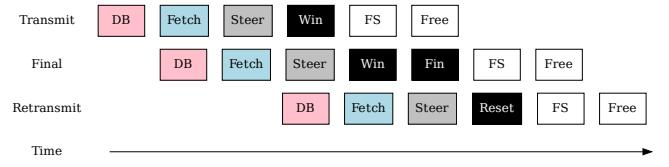


Figure 4. HC pipeline: Transmit, FIN, and retransmit.

(Figure 3). Accordingly, we partition connection state into module-local state (cf. §A). The pipeline stages are chosen to maximize data-path parallelism. Pre-processing accesses connection identifiers such as MAC and IP addresses for segment header preparation and filtering. The post-processing block handles application interface parameters, such as socket buffer addresses and context queues. These parameters are read-only after connection establishment and enable coordination-free scaling. Congestion control statistics are collected by the post-processor, but are only read by forward stages and can be updated out-of-order (updates commute). The protocol stage executes data-path code that must atomically modify protocol state, such as sequence numbers and socket buffer positions. It is the only *pipeline hazard*—it cannot execute in parallel with other stages. The DMA stage is stateless, while context queue stages may be sharded. Both conduct high-latency PCIe transactions and are thus separate stages that execute in parallel and scale independently.

We run pipeline stages on dedicated FPCs that utilize local memory for their portion of the connection state. Pipelining allows us to execute the data-path in parallel. It also allows us to replicate processing-intensive pipeline stages to scale to additional FPCs. With the exception of protocol processing, which is atomic per connection, all pipeline stages are replicated. To concurrently process multiple connections, we also replicate the entire pipeline. To keep flow state local, each pipeline handles a fixed *flow-group*, determined by a hash on the flow’s 4-tuple (the flow’s protocol type is ignored—it must be TCP). We now describe how we parallelize each data-path workflow by decomposing it into these pipeline stages.

**3.1.1 Host Control (HC).** HC processing is triggered by a PCIe doorbell (DB) sent via memory-mapped IO (MMIO) by the host to the context queue stage. Figure 4 shows the HC pipeline for two transmits (the second transmit closes the connection) triggered by libTOE, and a retransmit triggered by the control-plane. HC requests may be batched.



**Figure 5.** TX pipeline sending 3 segments.

The context queue stage polls for DBs. In response to a DB, the stage allocates a descriptor buffer from a pool in NIC memory. The limited pool size flow-controls host interactions. If allocation fails, processing stops and is retried later. Otherwise, the DMA stage fetches the descriptor from the host context queue into the buffer (Fetch). The pre-processor reads the descriptor, determines the flow-group, and routes to the appropriate protocol stage (Steer). The protocol stage updates connection receive and transmit windows (Win). If the HC descriptor contains a connection-close indication, the protocol stage also marks the connection as FIN (Fin). When the transmit window expands due to the application sending data for transmission, the post-processor updates the flow scheduler (FS) and returns the descriptor to the pool (Free).

Retransmissions in response to timeouts are triggered by the control-plane and processed the same as other HC events (fast retransmits due to duplicate ACKs are described in §3.1.3). The protocol stage resets the transmission state (Reset) to the last ACKed sequence number (go-back-N retransmission).

**3.1.2 Transmit (TX).** Transmission is triggered by the flow scheduler (SCH) when a connection can send segments. Figure 5 shows the TX pipeline for 3 example segments.

The pre-processor allocates a segment in NIC memory (Alloc), prepares Ethernet and IP headers (Head), and steers the segment to the flow-group's protocol stage (Steer). The protocol stage assigns a TCP sequence number based on connection state and determines the transmit offset in the host socket transmit buffer (Seq). The post-processor determines the socket transmit buffer address in host memory (Pos). The DMA stage fetches the host payload into the segment (Payload). After DMA completes, it issues the segment to the NBI (TX), which transmits and frees it.

**3.1.3 Receive (RX).** Figure 6 shows the RX pipeline for 3 example segments, where segment #3 arrives out of order.

**Pre-processing.** The pre-processor first validates the segment header (Val). Non-data-path segments<sup>2</sup> are filtered and forwarded to the control-plane. Otherwise, the pre-processor determines the connection index based on the segment's 4-tuple (Id) that is used by later stages to access connection state. The pre-processor generates a *header summary* (Sum), including only relevant header fields required by later pipeline stages and steers the summary and connection identifier to the protocol stage of its flow-group (Steer).

<sup>2</sup>Data-path segments have any of the ACK, FIN, PSH, ECE, and CWR flags and they may have the timestamp option.



**Figure 6.** RX pipeline receiving 3 segments, 1 out of order.

**Protocol.** Based on the header summary, the protocol stage updates the connection's sequence and acknowledgment numbers, the transmit window, and determines the segment's position in the host socket receive payload buffer, trimming the payload to fit the receive window if necessary (Win). The protocol stage also tracks duplicate ACKs and triggers fast retransmissions if necessary, by resetting the transmission state to the last acknowledged position. Finally, it forwards a snapshot of relevant connection state to post-processing.

Out-of-order arrivals (segment #3 in Figure 6) need special treatment. Like TAS [19], we track one out-of-order interval in the receive window, allowing the protocol stage to perform reassembly directly within the host socket receive buffer. We merge out-of-order segments within the interval in the host receive buffer. Segments outside of the interval are dropped and generate acknowledgments with the expected sequence number to trigger retransmissions at the sender. This design performs well under loss (cf. §5.3).

**Post-processing.** The post-processor prepares an acknowledgement segment (Ack). FlexTOE provides explicit congestion notification (ECN) feedback and accurate timestamps for RTT estimation (Stamp) in acknowledgments. It also collects congestion control and transmit window statistics, which it sends to the control-plane and flow scheduler (Stats). Finally, it determines the physical address of the host socket receive buffer, payload offset, and length for the DMA stage. If libTOE is to be notified, the post-processor allocates a context queue descriptor with the appropriate notification.

**DMA.** The DMA stage first enqueues payload DMA descriptors to the PCIe block (Payload). After payload DMA completes, the DMA stage forwards the notification descriptor to the context queue stage. Simultaneously, it sends the prepared acknowledgment segment to the NBI (TX), which frees it after transmission. This ordering is necessary to prevent the host and the peer from receiving notifications before the data transfer to the host socket receive buffer is complete.

**Context queue.** If necessary, the context queue stage allocates an entry on the context queue and issues the context queue descriptor DMA to notify libTOE of new payload (Notify) and frees the internal descriptor buffer (Free).

## 3.2 Sequencing and Reordering

TCP requires that segments of the same connection are processed in-order for receiver loss detection. However, stages in FlexTOE's data-parallel processing pipeline can have varying processing time and hence may reorder segments. Figure 7

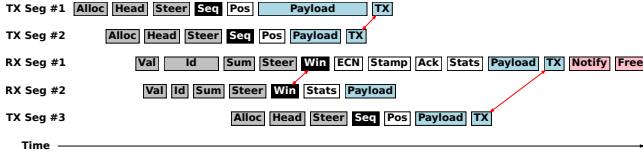


Figure 7. Undesirable pipeline reordering (red arrows).

shows three examples on a bidirectional connection where undesirable segment reordering occurs.

1. **TX.** TX segment #1 stalls in DMA across a congested PCIe link, causing it to be transmitted on the network after TX segment #2, potentially triggering receiver loss detection.
2. **RX.** RX segment #1 stalls in flow identification during pre-processing, entering the protocol stage later than RX segment #2. The protocol stage detects a hole and triggers unnecessary out-of-order processing.
3. **ACK.** TX segment #3 is processed after RX segment #1 in the protocol stage. RX segment #1 generates an ACK, but RX post-processing is complex, resulting in TX segment #3 with a higher sequence number being sent before ACK segment #1.

To avoid reordering, FlexTOE’s data-path pipeline sequences and reorders segments if necessary. In particular, we assign a sequence number to each segment entering the pipeline. The parallel pipeline stages can operate on each segment in any order. The protocol stage requires in-order processing and we buffer and re-order segments that arrive out-of-order before admitting them to the protocol stage. Similarly, we buffer and re-order segments for transmission before admitting them to the NBI. We leverage additional FPCs for sequencing, buffering, and reordering.

### 3.3 Flexibility

Data center networks evolve quickly, requiring TCP stacks to be easily modifiable by operators, not just vendors [29, 42, 43]. Many desirable data center features require TOE modification and are adapted frequently by operators. FlexTOE provides flexibility necessary to implement and maintain these features even beyond host stacks such as TAS, by relying on a programmable SmartNIC. To simplify development and modification of the TCP data-path, FlexTOE provides an extensible, data-parallel pipeline of self-contained modules, similar to the Click [38] extensible router.

**Module API.** The FlexTOE module API provides developers one-shot access to TCP segments and associated meta-data. Meta-data may be created and forwarded along the pipeline by any module. Modules may also keep private state. For scalability, private state cannot be accessed by other modules or replicas of the same module. Instead, state that may be accessed by further pipeline stages is forwarded as meta-data.

The replication factor of pipeline stages and assignment to FPCs is manual and static in FlexTOE. As long as enough FPCs are available, this approach is acceptable. Operators

can determine an appropriate replication factor that yields acceptable TCP processing bandwidth for a pipeline stage via throughput microbenchmarks at deployment. Stages that modify connection state atomically may be deployed by inserting an appropriate steering stage that steers segments of a connection to the module in the atomic stage, holding their state (cf. protocol processing stage in §3.1).

**XDP modules.** FlexTOE also supports eXpress Data Path (XDP) modules [14–16], implemented in eBPF. XDP modules operate on raw packets, modify them if necessary, and output one of the following result codes: (i) XDP\_PASS: Forward the packet to the next FlexTOE pipeline stage. (ii) XDP\_DROP: Drop the packet. (iii) XDP\_TX: Send the packet out the MAC. (iv) XDP\_REDIRECT: Redirect the packet to the control-plane.

XDP modules may use BPF maps (arrays, hash tables) to store and modify state atomically [25], which may be modified by the control-plane. For example, a firewall module may store blacklisted IPs in a hash map and the control-plane may add or remove entries dynamically. The module can consult the hash map to determine if a packet is blacklisted and drop it. XDP stages scale like other pipeline stages, by replicating the module. FlexTOE automatically reorders processed segments after a parallel XDP stage (§3.2).

Using these APIs, we modified the FlexTOE data-path many times, implementing the features listed in §2.1 (evaluation in §5.1). Further, ECN feedback and segment timestamping (cf. §3.1.3) are optional TCP features that support our congestion control policies. Operators can remove the associated post-processing modules if they are not needed.

By handling atomicity, parallelization, and ordering concerns, FlexTOE allows complex offloads to be expressed using few lines of code. For example, we implement AccelTCP’s connection splicing in 24 lines of eBPF code (cf. Listing 1 in the appendix). The module performs a lookup on the segment 4-tuple in a BPF hashmap. If a match is not found, we forward the segment to the next pipeline stage. Otherwise, we modify the destination MAC and IP addresses, TCP ports, and translate sequence and acknowledgment numbers using offsets configured by the control-plane, based on the connection’s initial sequence number. Finally, we transmit. FlexTOE handles sequencing and updating the checksum of the segment. Additionally, when we receive segments with control flags indicating connection closure, we atomically remove the hashmap entry and notify the control-plane.

### 3.4 Flow Scheduling

FlexTOE leverages a work-conserving flow scheduler on the NIC data-path. The flow scheduler obeys transmission rate-limits and windows configured by the control-plane’s congestion control policy. For each connection, the flow scheduler keeps track of how much data is available for transmission and the configured rate. Transmission rates and windows

are stored in NIC memory and are directly updated by the control-plane using MMIO.

We implement our flow scheduler based on Carousel [53]. Carousel schedules a large number of flows using a time wheel. Based on the next transmission time, as computed from rate limits and windows, we enqueue flows into corresponding slots in the time wheel. As the time slot deadline passes, the flow scheduler schedules each flow in the slot for transmission (§3.1.2). To conserve work, the flow scheduler only adds flows with a non-zero transmit window into the time wheel and bypasses the rate limiter for uncongested flows. These flows are scheduled round-robin.

## 4 Agilio-CX40 Implementation

This section describes FlexTOE’s Agilio-CX40 implementation. Due to space constraints, the x86 and BlueField ports are described in detail in §E. FlexTOE’s design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports.

FlexTOE is implemented in 18,008 lines of C code (LoC). The offloaded data-path comprises 5,801 lines of C code. We implement parts of the data-path in assembly for performance. libTOE contains 4,620 lines of C, whereas the control path contains 5,549 lines of C. libTOE and the control plane are adapted from TAS. We use the NFP compiler toolchain version 6.1.0.1 for SmartNIC development.

**Driver.** We develop a Linux FlexTOE driver based on the igb\_uio driver that enables libTOE and the control plane to perform MMIO to the SmartNIC from user space. The driver supports MSI-X based interrupts. The control-plane registers an eventfd for each application context in the driver. The interrupt handler in the driver pings the corresponding eventfd when an interrupt is received from the data-path for the application context. This enables libTOE to sleep when waiting for IO and reduces the host CPU overhead of polling.

**Host memory mapping.** To simplify virtual to physical address translation for DMA operations, we allocate physically contiguous host memory using 1G hugepages. The control-plane maps a pool of 1G hugepages at startup and allocates socket buffers and context queues out of this pool. In the future, we can use the IOMMU to eliminate the requirement of physically contiguous memory for FlexTOE buffers.

**Context queues.** Context queues use shared memory on the host, but communication between SmartNIC and host requires PCIe. We use scalable and efficient PCIe communication techniques [44] that poll on host memory locations when executing in the host and on NIC-internal memory when executing on the NIC. The NIC is notified of new queue entries via MMIO to a NIC doorbell. The context queue manager notifies applications through MSI-X interrupts, converted by the driver to an eventfd, after a queue has been inactive.

### 4.1 Near-memory Processing

An order of magnitude difference exists in the access latencies of different memory levels of the NFP-4000. For performance, it is critical to maximize access to local memory. The NFP-4000 also provides certain near-memory acceleration, including a lookup engine exposing a content addressable memory (CAM) and a hash table for fast matching, a queue memory engine exposing concurrent data structures such as linked lists, ring buffers, journals, and work-stealing queues. Finally, synchronization primitives such as ticket locks and inter-FPC signaling are exposed to coordinate threads and to sequence packets. We build specialized caches at multiple levels in the different pipeline stages using these primitives. Other NICs have similar accelerators.

**Caching.** We use each FPC’s CAM to build 16-entry fully-associative local memory caches that evict entries based on LRU. The protocol stage adds a 512-entry direct-mapped second-level cache in CLS. Across four islands, we can accommodate up to 2K flows in this cache. The final level of memory is in EMEM. When an FPC processes a segment, it fetches the relevant state into its local memory either from CLS or from EMEM, evicting other cache entries as necessary. We allocate connection identifiers in such a way that we minimize collisions on the direct-mapped CLS cache.

**Active connection database.** To facilitate connection index lookup in the pre-processing stage, we employ the hardware lookup capability of IMEM to maintain a database of active connections. CAM is used to resolve hash collisions. The pre-processor computes a CRC-32 hash on a segment’s 4-tuple to locate the connection index using the lookup engine. The pre-processor caches up to 128 lookup entries in its local memory via a direct-mapped cache on the hash value.

**FPC mapping.** FlexTOE’s pipeline fully leverages the Agilio CX40 and is extensible to further FPCs, e.g. of the Agilio LX [40]. For island-local interactions among modules, we use CLS ring buffers. CLS supports the fastest intra-island producer-consumer mechanisms. Among islands, we rely on work-queues in IMEM and EMEM.

We use all but one general-purpose islands for the first three stages of the data-path pipeline (*protocol islands*). Each island manages a *flow-group*. While protocol and post-processing FPCs are local to a flow-group, pre-processors handle segments for any flow. We assign 4 FPCs to pre-/post-processing stages in each flow-group. Each island retains 3 unassigned FPCs that can run additional data-path modules (§5.1).

On the remaining general-purpose island (called *service island*), we host remaining pipeline stages and adjacent modules, such as context queue FPCs, the flow scheduler (SCF), and DMA managers. DMA managers are replicated to hide PCIe latencies. The number of FPCs assigned to each functionality is determined such that no functionality may become a

bottleneck. Sequencing and reordering FPCs are located on a further island with miscellaneous functionality.

**Flow scheduler.** We implement Carousel using hardware queues in EMEM. Each slot is allocated a hardware queue. To add a flow to the time wheel, we enqueue it on the queue associated with the time slot. Note that the order of flows within a particular slot is not preserved. EMEM support for a large number of hardware queues enables us to efficiently implement a time wheel with a small slot granularity and large horizon to achieve high-fidelity congestion control. Converting transmission rates to deadlines requires division, which is not supported on the NFP-4000. Thus, the control-plane computes transmission intervals in cycles/byte units from rates and programs them to NIC memory. This enables the flow scheduler to compute the time slot using only multiplication.

## 5 Evaluation

We answer the following evaluation questions:

- **Flexible offload.** Can flexible offload improve throughput, latency, and scalability of data center applications? Can we implement common data center features? (§5.1)
- **RPCs.** How does FlexTOE’s data-path parallelism enable TCP offload for demanding RPCs? Do these benefits generalize across hardware architectures? Does FlexTOE provide low latency for short RPCs? Does FlexTOE provide high throughput for long RPCs? To how many simultaneous connections can FlexTOE scale? (§5.2)
- **Robustness.** How does FlexTOE perform under loss and congestion? Does it provide connection-fairness? (§5.3)

**Testbed cluster.** Our evaluation setup consists of two 20-core Intel Xeon Gold 6138 @ 2 GHz machines, with 40 GB RAM and 48 MB aggregate cache. Both machines are equipped with Netronome Agilio CX40 40 Gbps (single port), Chelsio Terminator T62100-LP-CR 100 Gbps and Intel XL710 40 Gbps NICs. We use one of the machines as a server, the other as a client. As additional clients, we also use two 2×18-core Intel Xeon Gold 6154 @ 3 GHz systems with 90 MB aggregate cache and two 4-core Intel Xeon E3-1230 v5 @ 3.4 GHz systems with 9 MB aggregate cache. The Xeon Gold machines are equipped with Mellanox ConnectX-5 MT27800 100 Gbps NICs, whereas the Xeon E3 machines have 82599ES 10 Gbps NICs. The machines are connected to a 100 Gbps Ethernet switch.

**Baseline.** We compare FlexTOE performance against the Linux TCP stack, Chelsio’s kernel-based TOE<sup>3</sup>, and the TAS kernel-bypass stack<sup>4</sup>. TAS does not perform well with the Agilio CX40 due to a slow NIC DPDK driver. We run TAS on the Intel XL710 NIC, as in [19], unless mentioned otherwise. We use identical application binaries across all baselines. DCTCP is our default congestion control policy.

<sup>3</sup>Chelsio does not support kernel-bypass.

<sup>4</sup>TAS [19] performs better than mTCP [17] on all of our benchmarks. Hence, we omit a comparison to mTCP and AccelTCP [37], which uses mTCP.

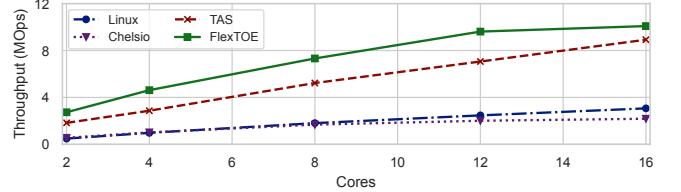


Figure 8. Memcached throughput scalability.

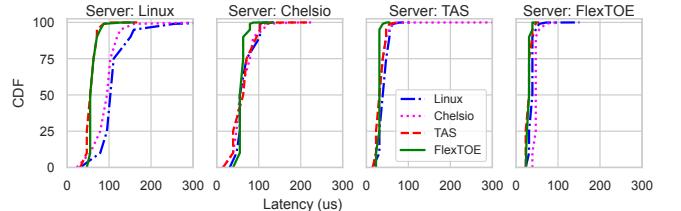


Figure 9. Latency of different server-client combinations.

### 5.1 Benefit of Flexible Offload

**Application throughput scalability.** Offloaded CPU cycles may be used for application work. We quantify these benefits by running a Memcached server, as in §2.1, varying the number of server cores. Figure 8 shows that, by saving host CPU cycles (cf. Table 1), FlexTOE achieves up to 1.6× TAS, 4.9× Chelsio, and 5.5× Linux throughput. FlexTOE and TAS scale similarly—both use per-core context queues. The Agilio CX becomes a compute-bottleneck at 12 host cores. Linux and Chelsio are slow for this workload, due to system call overheads, and do not scale well due to in-kernel locks.

**Low (tail) latency.** We repeat a single-threaded version of the same Memcached benchmark for all server-client network stack combinations. Latency distributions are shown in Figure 9. We can see that FlexTOE consistently provides the lowest median and tail Memcached operation latency across all stack combinations. Offload provides excellent performance isolation by physically separating the TCP data-path, even though FlexTOE’s pipelining increases minimum latency in some cases (cf. §5.2).

**Flexibility.** Unlike fixed offloads and in-kernel stacks, FlexTOE provides full user-space programmability via a module API, simplifying development. Customizing FlexTOE is simple and does not require a system reboot. For example, we have developed logging, statistics, and profiling capabilities that can be turned on only when necessary. We make use of these capabilities during development and optimization of FlexTOE. We implemented up to 48 different tracepoints (including examples from bpftrace [13]) in the data-path pipeline, tracking transport events such as per-connection drops, out-of-order packets and retransmissions, inter-module queue occupancies, and critical section lengths in the protocol module for various event types. Table 2 shows that profiling degrades data-path performance versus the baseline by up to 24% when all 48 tracepoints are enabled. We also implement tcpdump-style traffic logging, including packet filters based on header

Build	Throughput (MOps)
Baseline FlexTOE	11.35
Statistics and profiling	8.67
tcpdump (no filter)	6.52
XDP (null)	10.87
XDP (vlan-strip)	10.83

Table 2. Performance with flexible extensions.

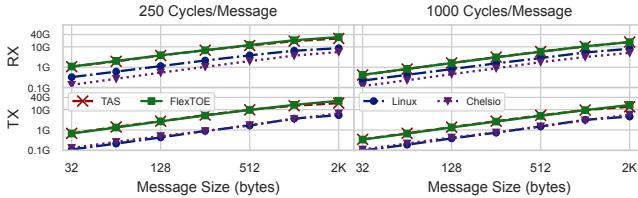


Figure 10. RPC throughput for saturated server.

fields. Logging naturally has high overhead (up to 43% when logging all packets). FlexTOE provides the flexibility to implement these features and to turn them on only when necessary.

Furthermore, new data-plane functionality leveraging the XDP API may be dynamically loaded into FlexTOE as eBPF programs. eBPF programs can be compiled to NFP assembly. This level of dynamic flexibility is hard to achieve with an FPGA as it requires instruction set programmability (overlays [52]). We measure the overhead of FlexTOE XDP support by running a null program that simply passes on every packet without modification. We observe only 4% decline in throughput. Common XDP modules, such as stripping VLAN tags on ingress packets, also have negligible overhead. Finally, connection splicing (cf. Listing 1 in the appendix) achieves a maximum splicing performance of 6.4 million packets per second, enough to saturate the NIC line rate with MTU-sized packets, leveraging only idle FPCs<sup>5</sup>.

## 5.2 Remote Procedure Calls (RPCs)

RPCs are an important but difficult workload for flexible offload. Latency and client scalability requirements favor fast processing engines with large caches, such as found in CPUs and ASICs. Neither are available in on-path SmartNICs. We show that flexible offload can be competitive with state-of-the-art designs. We then show that FlexTOE’s data-path parallelism is necessary to provide the necessary performance.

**Typical RX / TX performance.** We start with a typical server scenario, processing RPCs of many (128) connections, produced in an open loop by multiple (16) clients (multiple pipelined RPCs per connection). To simulate application processing, our server waits for an artificial delay of 250 or 1,000 cycles for each RPC. We run single-threaded to avoid the network being a bottleneck. We quantify RX and TX throughput separately, by switching RPC consumer and producer roles among clients and servers, over different RPC sizes.

<sup>5</sup>We are compute-limited by our Agilio CX. Using an Agilio LX, like AccelTCP, would allow us to achieve even higher throughput.

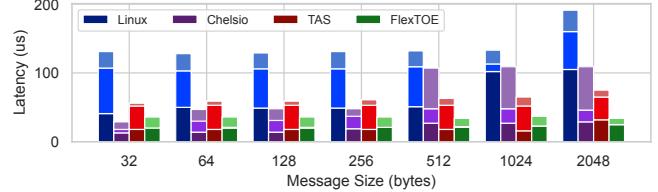


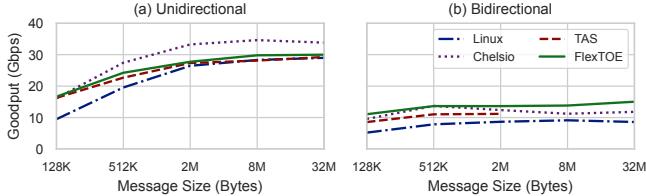
Figure 11. Median, 99p and 99.99p RPC RTT.

Figure 10 shows the results. For 250 cycles of processing overhead, FlexTOE provides up to 4× better throughput than Linux and 5.3× better throughput than Chelsio when receiving. For 2 KB message size, both TAS and FlexTOE reach 40 Gbps line rate, whereas Linux and Chelsio barely reach 10 Gbps and 7 Gbps, respectively. When sending packets, the difference in performance between Linux and FlexTOE is starker. FlexTOE shows over 7.6× higher throughput over both Linux and Chelsio for all message sizes. The gains remain at over 2.2× as we go to 1,000 cycles/RPC. Performance of TAS and FlexTOE track closely for all message sizes. This is expected as the single application server core is saturated by both network stacks (TAS runs on additional host cores).

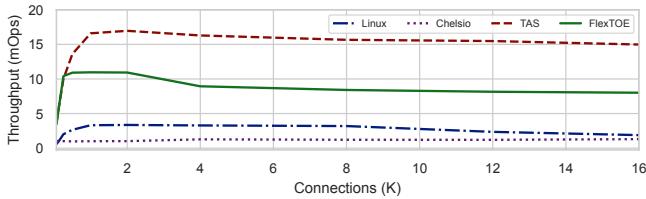
We break down this result by studying the performance sensitivity of each TCP stack, varying each RPC parameter within its sensitive dynamic range. For these benchmarks, we evaluate the raw performance of the stacks, without application processing delays.

**RPC latency.** A client establishes a single connection to the server and measures single RPC RTT. Figure 11 shows the median and tail RTT for various small message sizes (stacked bars). The inefficiency of in-kernel networking is reflected in the median latency of Linux, which is at least 5× worse compared to other stacks. For message sizes < 256 B, FlexTOE’s median latency (20 us) is 1.4× Chelsio’s median latency (14 us) and 1.25× TAS’s median latency (16 us). FlexTOE’s data-path pipeline across many wimpy FPCs increases median latency for single RPCs. However, FlexTOE has an up to 3.2× smaller tail compared to Chelsio and nearly constant per-segment overhead as the RPC size increases. In case of a 2 KB RPC (larger than the TCP maximum segment size), FlexTOE’s latency distribution remains nearly unchanged. FlexTOE’s fine-grain parallelism is able to hide the processing overhead of multiple segments, providing 22% lower median and 50% lower tail latency than TAS.

**Per-connection throughput.** In this setup, a client transfers a large RPC message to the server. In the first case (Figure 12a), the server responds with a 32 B response whereas in the second case (b), the server echoes the message back to the client (TAS performance is unstable with messages > 2 MB in this case—we omit these results). In the short-response case, Chelsio performs 20% better than the other stacks—Chelsio is a 100 Gbps NIC optimized for unidirectional streaming. However, it has 20% lower throughput as compared to FlexTOE in



**Figure 12.** Large RPC throughput with varying RPC size.



**Figure 13.** Connection scalability benchmark.

the echo case. Other stacks cannot parallelize per-connection processing, leading to limited throughput<sup>6</sup>, while FlexTOE’s throughput is limited by its protocol stage. FlexTOE currently acknowledges every incoming packet. For bidirectional flows, this quadruples the number of packets processed per second. Implementing delayed ACKs would improve FlexTOE’s performance further for large flows.

**Connection scalability.** We establish an increasing number of RPC client connections from all 5 client machines to a multi-threaded echo server. To stress TCP processing, each connection leaves a single 64 B RPC in-flight. Figure 13 shows the throughput as we vary the number of connections. This workload is very challenging for FlexTOE as it exhausts fast memory and prevents per-connection batching, causing a cache miss at every pipeline stage for every segment. Up to 2K connections, FlexTOE shows a throughput of 3.3× Linux. TAS performs 1.5× better than FlexTOE for this workload. FlexTOE is compute-bottlenecked<sup>7</sup> at the protocol stage, which uses 8 FPCs in this benchmark. Agilio CX caches 2K connections in CLS memory. Beyond this, the protocol stage must move state among local memory, CLS, and EMEM. EMEM’s SRAM cache is increasingly strained as the number of connections increases. FlexTOE’s throughput declines by 24% as we hit 8k connections and plateaus beyond that<sup>8</sup>. TAS’s fast-path exhibits better connection scalability, as it has access to the larger host CPU cache, while Linux’s throughput declines significantly. Chelsio has poor performance for this workload, as `epoll()` overhead dominates.

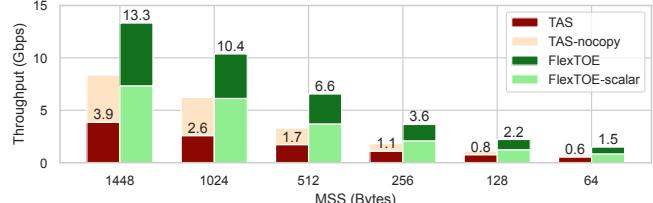
**Benefit of data-path parallelism.** To break down the impact of FlexTOE’s data-parallel design on RPC performance,

<sup>6</sup>With multiple unidirectional flows, all stacks achieve line rate (Figure 15b).  
<sup>7</sup>We expect that running FlexTOE on the Agilio LX with 1.2 GHz FPCs—1.5× faster than Agilio CX—would boost the peak throughput to match TAS performance. Agilio LX also doubles the number of FPCs and islands. It would allow us to exploit more parallelism and cache more connections.

<sup>8</sup>While we evaluate up to 16K connections, FlexTOE can leverage the 2 GB on-board DRAM to scale to 1M+ connections.

Design	Throughput (Mbps)	Latency (us)	
		50p	99.99p
Baseline	79.32	1	1,179
+ Pipelining	3,640.49	46	183
+ Intra-FPC parallelism	8,194.34	103	128
+ Replicated pre/post	11,086.93	140	94
+ Flow-group islands	22,684.69	286	46

**Table 3.** FlexTOE data-path parallelism breakdown.



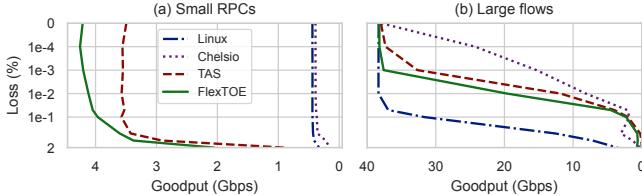
**Figure 14.** FlexTOE benefits on BlueField SmartNIC.

we repeat the echo benchmark with 64 connections, with each connection leaving a single 2 KB RPC in-flight (to be able to evaluate both intra and inter connection parallelism). Table 3 shows the performance impact as we progressively add data-path parallelism. Our baseline runs the entire TCP processing to completion on the SmartNIC before processing the next segment. Pipelining improves performance by 46× over the baseline. As we enable 8 threads on the FPCs (2.25× gain), we hide the latency of memory operations and improve FPC utilization. Next, we replicate the pre-processing and post-processing stages, leveraging sequencing and reordering for correctness, to extract 1.35× improvement and finally, with four flow-group islands, we see a further 2× improvement. We can see that each level of data-path parallelism is necessary, improving RPC throughput and latency by up to 286×.

**Do these benefits generalize?** We investigate whether data-path parallelism provides benefits across platforms. In particular, we investigate single connection throughput of pipelined RPCs across a range of maximum segment sizes (MSS) on a Mellanox BlueField [31] MBF1M332A-ASCAT 25 Gbps SmartNIC and on a 32-core AMD 7452 @ 2.35 GHz host with 128 GB RAM, 148 MB aggregate cache, and a conventional 100 Gbps ConnectX-5 NIC. We use a single-threaded RPC sink application, running on the same platform<sup>9</sup>. We compare TAS’s core-per-connection processing to FlexTOE’s data-parallelism. We replicate each of FlexTOE’s pre and post processing stages 2×, resulting in 9 FlexTOE cores. Further gains may be achievable by more replication. To break down FlexTOE’s benefits, we also compare to a FlexTOE pipeline without replicated stages (FlexTOE-scalar), using 7 cores.

Figure 14 shows BlueField results. FlexTOE outperforms TAS by up to 4× on BlueField (and 2.4× on x86). Depending on RPC size, FlexTOE accelerates different stages of the TCP data path. For large RPCs, FlexTOE accelerates data copy to

<sup>9</sup>BlueField is an off-path SmartNIC that is not optimized for packet processing offload to host-side applications (§2.3).



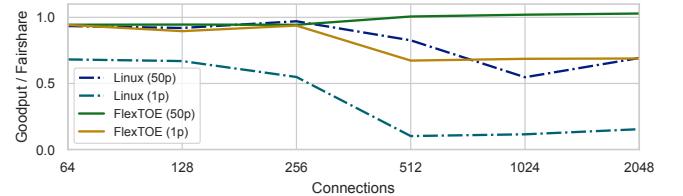
**Figure 15.** Throughput, varying packet loss rate.

socket payload buffers. To show this, we eliminate the step in TAS (TAS-nocopy), allowing TAS to perform at  $0.5\times$  FlexTOE on BlueField (and identical to FlexTOE on x86). For smaller RPCs, TAS-nocopy benefits diminish and FlexTOE supports processing higher packet rates. FlexTOE-scalar achieves only up to  $2.3\times$  speedup over TAS on BlueField (and  $1.47\times$  on x86), showing that only part of the benefit comes from pipelining. Finally, FlexTOE speedup is greater on the wimpier BlueField, resembling our target architecture (§2.3), than on x86. To save powerful x86 cores, some stages may be collapsed, even dynamically (cf. Snap [29]), at little performance cost.

### 5.3 Robustness

**Packet loss.** We artificially induce packet losses in the network by randomly dropping packets at the switch with a fixed probability. We measure the throughput between two machines for 100 flows running 64 B echo-benchmark as we vary the loss probability, shown in Figure 15a. We configure the clients to pipeline up to 8 requests on each connection to trigger out-of-order processing when packets are lost. FlexTOE’s throughput at 2% losses is at least twice as good as TAS and an order of magnitude better than the other stacks for this case. We repeat the unidirectional large RPC benchmark with 8 connections and measure the throughput as we increase the packet loss rate. For this case (b), Chelsio has a very steep decline in throughput even with  $10^{-4}\%$  loss probability. Linux is able to withstand higher loss rates as it implements more sophisticated reassembly and recovery algorithms, including selective acknowledgments—FlexTOE and TAS implement single out-of-order interval tracking on the receiver-side and go-back-n recovery on the sender. FlexTOE’s behavior under loss is still better than TAS. FlexTOE processes acknowledgments on the NIC, triggering retransmissions sooner, and its predictable latency, even under load, helps FlexTOE recover faster from packet loss. We note that RDMA tolerates up to 0.1% losses [35], while eRPC falters at 0.01% loss rate [18]. Unlike FlexTOE, RDMA discards all out-of-order packets on the receiver side [35]. TAS [19] provides further evaluation of the benefits of receiver out-of-order interval tracking.

**Fairness.** To show scalability of FlexTOE’s SCH (§3.4), we measure the distribution of connection throughputs of bulk flows between two nodes at line rate for 60 seconds. Figure 16 shows the median and 1st percentile throughput of FlexTOE and Linux as we vary the number of connections. For FlexTOE, the median closely tracks the fair share throughput and the tail



**Figure 16.** Throughput distribution at line rate.

deg.	# con.	Tpt. (G)		Lat. 99.99p (ms)		JFI	
		on	off	on	off	on	off
4	16	9.51	9.47	5.98	11.58	0.98	0.95
4	64	9.51	9.23	10.75	44.39	0.96	0.73
4	128	9.48	8.96	13.74	64.25	0.99	0.53
10	10	3.66	1.04	2.50	18.26	0.95	0.78
20	20	1.76	0.36	7.35	138.32	0.95	0.46

**Table 4.** FlexTOE congestion control under incast.

is  $0.67\times$  of the median. Linux’s fairness is significantly affected beyond 256 connections. Jain’s fairness index (JFI) drops to 0.36 at 2K connections for Linux, while FlexTOE achieves 0.98. Above 1K connections, Linux’ median throughput is worse than FlexTOE’s 1st percentile.

**Incast.** We simulate incast by enabling traffic shaping on the switch to restrict port bandwidth to various incast degrees and we configure WRED to perform tail drops when the switch buffer is exhausted. In this experiment, the client transfers 64 KB RPCs and the server responds with a 32 B response on each connection. As shown in Table 4, control-plane-driven congestion control in FlexTOE is able to achieve the shaped line rate, maintain low tail latency, and ensure fairness among flows under congestion. Disabling it causes excessive drops, inflating tail latency by  $18.8\times$  and skewing fairness by  $2\times$ .

## 6 Conclusion

FlexTOE is a flexible, yet high-performance TCP offload engine to SmartNICs. FlexTOE leverages fine-grained parallelization of the TCP data-path and segment reordering for high performance on wimpier SmartNIC architecture, while remaining flexible via a modular design. We compare FlexTOE to Linux, the TAS software TCP accelerator, and the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive performance for RPCs, even with wimpier SmartNICs, and is robust under adverse operating conditions. FlexTOE’s API supports XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

**Acknowledgments.** We thank the anonymous reviewers and our shepherd, Brent Stephens, for their helpful comments and feedback. This work was supported by NSF grant 1751231.

## References

- [1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the 2010 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’10, pages 63–74, New York, NY, USA, 2010. Association for Computing Machinery.
- [2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’20, pages 93–110, USA, 2020. USENIX Association.
- [3] 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, USA, 2014. USENIX Association.
- [4] Broadcom. Broadcom Stingray SmartNICs. <https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225>, 2018.
- [5] Cavium. Cavium OCTEON Development Kits. <https://cavium.com/octeon-software-develop-kit.html>, 2018.
- [6] Chelsio Communications. T6 ASIC: High performance, dual port unified wire 1/10/25/40/50/100Gb Ethernet controller. <https://www.chelsio.com/wp-content/uploads/resources/Chelsio-Terminator-6-Brief.pdf>, 2017.
- [7] Andy Currid. TCP offload to the rescue: Getting a toehold on TCP offload engines—and why we need them. *Queue*, 2(3):58–65, May 2004.
- [8] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Kries, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP ’09, pages 15–28, New York, NY, USA, 2009. Association for Computing Machinery.
- [9] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chatuveli, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’18, pages 51–64, USA, 2018. USENIX Association.
- [10] Michael Galles and Francis Matus. Pensando distributed services architecture. *IEEE Micro*, 41(2):43–49, 2021.
- [11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’16, pages 202–215, New York, NY, USA, 2016. Association for Computing Machinery.
- [12] Intel Corporation. Intel 82599 10 GbE controller datasheet. Revision 3.4, November 2019. <https://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html>.
- [13] IO Visor Project, Linux Foundation. bpftace: High-level tracing language for Linux eBPF. <https://github.com/ iovisor/bpftace>, 2021.
- [14] IO Visor Project, Linux Foundation. XDP: express data path. <https://www.iovisor.org/technology/xdp>, 2021.
- [15] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. ebpf hardware offload to smartnics: cls bpf and xdp. [https://www.netronome.com/media/documents/eBPF\\_HW\\_OFFLOAD\\_HNiMne8\\_2\\_.pdf](https://www.netronome.com/media/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf), 2021.
- [16] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. Xdp hardware offload: Current work, debugging and edge cases. [https://www.netronome.com/media/documents/viljoen-xdpofoffload- talk\\_2.pdf](https://www.netronome.com/media/documents/viljoen-xdpofoffload- talk_2.pdf), 2021.
- [17] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’14, pages 489–502, USA, 2014. USENIX Association.
- [18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’19, pages 1–16, USA, 2019. USENIX Association.
- [19] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys ’19, New York, NY, USA, 2019. Association for Computing Machinery.
- [20] Jongyul Kim, InsuJang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, SOSP ’21, pages 756–771, New York, NY, USA, 2021. Association for Computing Machinery.
- [21] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making rpcs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC ’19, pages 863–879, USA, 2019. USENIX Association.
- [22] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, pages 36–51, New York, NY, USA, 2021. Association for Computing Machinery.
- [23] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery.
- [24] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel TCP design and implementation for short-lived connections. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’16, pages 339–352, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] Linux - bpf(2) – linux manual page. <https://man7.org/linux/man-pages/man2/bpf.2.html>, 2021.
- [26] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using IPipe. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’19, pages 318–333, New York, NY, USA, 2019. Association for Computing Machinery.
- [27] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC ’19, pages 363–378, USA, 2019. USENIX Association.
- [28] David A. Maltz and Pravin Bhagwat. TCP splice application layer proxy performance. *Journal of High Speed Networks*, 8(3):225–240, January 2000.
- [29] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin

- Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, pages 399–413, New York, NY, USA, 2019. Association for Computing Machinery.
- [30] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 USENIX Winter Conference*, USENIX ’93, page 2, USA, 1993. USENIX Association.
- [31] Mellanox. Mellanox BlueField Platforms. [http://www.mellanox.com/related-docs/hpu-multicore-processors/PB\\_BlueField\\_Ref\\_Platform.pdf](http://www.mellanox.com/related-docs/hpu-multicore-processors/PB_BlueField_Ref_Platform.pdf), 2018.
- [32] memcached. Memcached, 2020. <https://memcached.org/>.
- [33] Microsoft. Information about the TCP Chimney offload, receive side scaling, and network direct memory access features in Windows Server 2008. <https://docs.microsoft.com/en-US/troubleshoot/windows-server/networking/information-about-tcp-chimney-offload-rss-netdma-feature>.
- [34] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’15, pages 537–550, New York, NY, USA, 2015. Association for Computing Machinery.
- [35] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’18, pages 313–326, New York, NY, USA, 2018. Association for Computing Machinery.
- [36] Jeffrey C. Mogul. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th USENIX Conference on Hot Topics in Operating Systems*, HotOS ’03, page 5, USA, 2003. USENIX Association.
- [37] YoungGyun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’20, pages 77–92, USA, 2020. USENIX Association.
- [38] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP ’99, pages 217–231, New York, NY, USA, 1999. Association for Computing Machinery.
- [39] Netronome. Netronome Agilio CX SmartNIC. <https://www.netronome.com/products/agilio-cx/>, 2018.
- [40] Netronome. Netronome Agilio LX SmartNIC. <https://www.netronome.com/products/agilio-lx/>, 2018.
- [41] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery.
- [42] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making network stack part of the virtualized infrastructure. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC ’20, USA, 2020. USENIX Association.
- [43] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 65–71, New York, NY, USA, 2017. Association for Computing Machinery.
- [44] NVM Express Workgroup. NVM Express: Base specification. [https://nvmeexpress.org/wp-content/uploads/NVM-Express-1\\_4a-2020.03.09-Ratified.pdf](https://nvmeexpress.org/wp-content/uploads/NVM-Express-1_4a-2020.03.09-Ratified.pdf), 2020.
- [45] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys ’12, pages 337–350, New York, NY, USA, 2012. Association for Computing Machinery.
- [46] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI ’14, pages 1–16, Broomfield, CO, October 2014. USENIX Association.
- [47] I. Pratt and K. Fraser. Arsenic: a user-accessible gigabit ethernet interface. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society*, volume 1 of INFOCOM ’01, pages 67–76 vol.1, 2001.
- [48] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA ’14, pages 13–24. IEEE Press, 2014.
- [49] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. <http://www.rdmaconsortium.org/>.
- [50] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007.
- [51] Redis Labs. memtier\_benchmark: Load generation and benchmarking NoSQL key-value databases. [https://github.com/RedisLabs/memtier\\_benchmark](https://github.com/RedisLabs/memtier_benchmark), 2020.
- [52] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the 2021 Workshop on Hot Topics in Operating Systems*, HotOS ’21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery.
- [53] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’17, pages 404–417, New York, NY, USA, 2017. Association for Computing Machinery.
- [54] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaesle. We need to talk about NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS ’13, page 1, USA, 2013. USENIX Association.
- [55] Pensando Systems. Pensando DSC-25 distributed services card. <https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf>, 2020.
- [56] The Linux Foundation. toe. <https://wiki.linuxfoundation.org/networking/toe>.

Field	Bits	Description
<b>Pre-processor</b> (connection identification)—15B:		
peer_mac	48	Remote MAC address
peer_ip	32	Remote IP address
local remote_port	32	TCP ports
flow_group	2	hash(4-tuple) % 4
<b>Protocol</b> (TCP state machine)—43B:		
rx tx_pos	64	RX/TX buffer head
tx_avail	32	Bytes ready for TX
rx_avail	32	Available RX buffer space
remote_win	16	Remote receive window
tx_sent	32	Sent unack. TX bytes
seq	32	TCP seq. number
ack	32	TCP remote seq. number
ooo_start len	64	Out-of-order interval
dupack_cnt	4	Duplicate ACK count
next_ts	32	Peer timestamp to echo
<b>Post-processor</b> (ctx queue, congestion control)—51B:		
opaque	64	App connection id
context	16	Context-queue id
rx tx_base	128	RX/TX buffer base
rx tx_size	64	RX/TX buffer size
cnt_ackb ecnb	64	ACK'd and ECN bytes
cnt_fretx	8	Fast-retransmits count
rtt_est	32	RTT estimate
rate	32	TX rate

**Table 5.** Connection state partitions (total: 108B).

## A TCP Connection State Partitioning

To enable fine-grained parallelism, we partition connection state across pipeline stages. Table 5 shows the per-connection state variables, grouped by pipeline stage. Pre-processor state contains connection identifiers (MAC, IP addresses; TCP port numbers). Protocol state contains TCP windows, sequence and acknowledgment numbers, and host payload buffer positions. Post-processor state contains host payload buffer and context queue locations, and data-path congestion control state. DMA and context queue stages are stateless.

In aggregate, each TCP connection has 108 bytes of state, allowing us to offload millions of connections to the SmartNIC. In particular, we can manage 16 connections per protocol FPC, 512 connections per flow-group, and 16K connections in the EMEM cache. Using all of EMEM, we can support up to 8M connections.

## B Connection Splicing Implementation

We implement AccelTCP’s connection splicing in 24 lines of eBPF code. Listing 1 shows the entire code.

## C TAS TCP/IP Processing Breakdown

Table 6 shows a breakdown of the per-packet TCP/IP processing overheads (summarized as *TCP/IP stack* in Table 1) in TAS for the Memcached benchmark conducted in §2.1. For

```

BPF_MAP_HASH_DECLARE(splice_tbl, SPLICE_MAX_FLOWS, \
    sizeof(struct pkt_4tuple_t), sizeof(struct tcp_splice_t));

int bpf_xdp_prog(struct xdp_md* ctx)
{
    struct tcp_splice_t state;
    struct pkt_hdr_t *hdr = BPF_XDP_ADDR(ctx->data);
    struct pkt_4tuple_t *key = &hdr->ip.src;

    // Filter non-IPv4/TCP segments to control-plane
    if (!segment_ipv4_tcp(hdr))
        return XDP_REDIRECT;

    // Connection Control: Segments with SYN, FIN, RST
    // Atomically remove map entry and forward to control-plane
    if (segment_tcp_ctrlflags(hdr)) {
        BPF_MAP_DELETE_ELEM(splice_tbl, key);
        return XDP_REDIRECT;
    }

    if (BPF_MAP_LOOKUP_ELEM(splice_tbl, key, &state) < 0)
        return XDP_PASS; // Send to data-plane

    patch_headers(hdr, &state);
    return XDP_TX; // Send out the MAC
}

void patch_headers(struct pkt_hdr_t *hdr,
                  struct tcp_splice_t *state)
{
    hdr->eth.src = hdr->eth.dst;
    hdr->eth.dst = state->remote_mac;
    hdr->ip.src = hdr->ip.dst;
    hdr->ip.dst = state->remote_ip;
    hdr->tcp.sport = state->local_port;
    hdr->tcp.dport = state->remote_port;

    hdr->tcp.seq += state->seq_delta;
    hdr->tcp.ack += state->ack_delta;
}

```

**Listing 1.** Connection splicing with XDP in FlexTOE.

each request, TAS performs loss detection (and potentially recovery) that involves processing the incoming request segment, generating an acknowledgement for it, and additionally, processing the acknowledgement for the response segment, consuming 42% of the total per-packet processing cycles. TAS spends 9% of the total cycles to prepare the response TCP segment for transmission and an additional 12% to schedule flows

Function	Cycles	%
Segment generation	130	9
Loss detection (and recovery)	606	42
Payload transfer	10	1
Application notification	381	26
Flow scheduling	172	12
Miscellaneous	141	10
Total	1,440	100

**Table 6.** Breakdown of TCP/IP stack overheads in TAS.

based on the rate configured by the congestion control protocol. TAS spends 26% of per-packet cycles interacting with the application, to notify when a request is received, to admit a response for transmission, and to free the transmission buffer when it is acknowledged. For small request-response pairs (32B in this case), the payload copy overheads are negligible.

## D Control Plane

FlexTOE’s control plane is similar to that of existing approaches that separate control and data-plane activities, such as TAS [19]. Using it, we implement control-plane policies, such as congestion control, per-connection rate limits, per-application connection limits, and port partitioning among applications (cf. [52]). We briefly describe connection and congestion control in this appendix. Retransmissions are described in §3.1.1 and §3.1.3. TAS [19] provides further description and evaluation of the control plane (named “slow-path” in the TAS paper).

**Connection control.** Connection control involves complex control logic, such as ARP resolution, port and buffer allocation, and the TCP connection state machine. The data-path forwards control segments to the control-plane. The control-plane notifies libTOE of incoming connections on listening ports. If the application decides to accept() the connection, the control-plane finishes the TCP handshake, allocates host payload buffers and a unique connection index for the data-path. It then sets up connection state in the data-path at the index location. Similarly, libTOE forwards connect() calls to the control-plane, which establishes the connection. On shutdown(), the control-plane disables the connection and removes the corresponding data-path state.

**Congestion control.** FlexTOE provides a generic control-plane framework to implement different rate and window-based congestion control algorithms, akin to that in TAS [19]. The control-plane runs a loop over the set of active flows to compute a new transmission rate, periodically. The interval between each iteration of the loop is determined by the round-trip time (RTT) of each flow. In each iteration, the control-plane reads per-flow congestion control statistics from the data-path to calculate a new rate or window for the flow. The rate or window is then set in the data-path flow scheduler (§3.4) for enforcement. We also monitor retransmission timeouts in the control iteration. FlexTOE implements DCTCP [1] and TIMELY [34] in this way.

## E FlexTOE x86 and BlueField Ports

We have ported the FlexTOE data-path to the x86 and BlueField platforms. FlexTOE’s design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports. FlexTOE’s decomposition, pipeline parallelism, and per-stage replication all generalize across platforms. Both ports are also almost

identical to the Agilio-CX40 implementation (cf. §4) and were completed within roughly 2 person-weeks, demonstrating the great development velocity of a software TCP offload engine. We describe the implementation differences of each port to the Agilio-CX40 version in this section.

**Hardware cache management.** The hardware-managed cache hierarchies of x86 and BlueField obviate the need for software-managed caching that was implemented on Agilio. Instead of leveraging near-memory processing acceleration of the NFP-4000 (cf. §4.1), our ports implement multi-core ring buffers, flow lookup and packet sequencers in software. The more powerful x86 and BlueField cores make up for the difference in performance.

**Symmetric core mapping.** Unlike the NFP-4000, where FPCs are organized into islands, cores on x86 and BlueField have mostly symmetric communication properties, so the assignment of modules to cores is arbitrary and the manual FPC mapping step is omitted. However, we note that core mapping may still be beneficial, for example to leverage shared caches and node locality on multi-socket x86 systems. Each instance of a module runs on its own core. Apart from the six fine-grained pipeline modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, *context queue*, and *SCH* shown in Figure 3, the ports utilize an additional *netif* module to interface with DPDK NIC queues to receive and transmit packets. Therefore, FlexTOE-scalar uses 7 cores and the FlexTOE-2× configuration uses 2 additional cores to replicate the pre and post-processing stages for a total of 9 cores.

**Context queues use only shared memory.** Our x86 and BlueField ports currently only support applications running on the same platform as FlexTOE. Hence, context queues always use shared memory rather than DMA. The corresponding DMA pipeline stage executes the payload copies in software using shared memory, rather than leveraging a DMA engine.

**Platform-specific parameters.** The replication factor of each pipeline stage is platform dependent. Stage-specific microbenchmarks on each platform can determine it. Our generalization experiments (§5.2) are designed to show that FlexTOE’s data-parallelism can improve single connection throughput. Hence, we configure only one instance of the FlexTOE data-path pipeline in these versions (no flow-group islands—we do not process multiple connections in these experiments). Each port’s pipeline uses the same number of stages as the Agilio-CX40 version, but we set different replication factors for the pre and post processing stages on x86 and BlueField (no replication and 2× replication). We do not attempt to find the optimal replication factor for best performance nor compact stages to reduce wasted CPU cycles.

# Swift: Adaptive Video Streaming with Layered Neural Codecs

Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, Dimitris Samaras  
*Stony Brook University*

## Abstract

Layered video coding compresses video segments into layers (additional code bits). Decoding with each additional layer improves video quality incrementally. This approach has potential for very fine-grained rate adaptation. However, layered coding has not seen much success in practice because of its cross-layer compression overheads and decoding latencies. We take a fresh new approach to layered video coding by exploiting recent advances in video coding using deep learning techniques. We develop *Swift*, an adaptive video streaming system that includes i) a layered encoder that learns to encode a video frame into layered codes by purely encoding *residuals* from previous layers without introducing any cross-layer compression overheads, ii) a decoder that can *fuse* together a subset of these codes (based on availability) and decode them all in one go, and, iii) an adaptive bit rate (ABR) protocol that synergistically adapts video quality based on available network and client-side compute capacity. *Swift* can be integrated easily in the current streaming ecosystem without any change to network protocols and applications by simply replacing the current codecs with the proposed layered neural video codec when appropriate GPU or similar accelerator functionality is available on the client side. Extensive evaluations reveal *Swift*'s multi-dimensional benefits over prior video streaming systems.

## 1 Introduction

Internet video delivery often encounters highly variable and unpredictable network conditions. Despite various advances made, delivering the highest possible video quality continues to be a challenging problem due to this uncertainty. The problem is more acute in wireless networks as the channel conditions and mobility adds to the uncertainty [39, 46]. Interestingly, the next generation wireless networks may even make the problem more challenging (e.g., 60GHz/mmWave [10, 11, 38]).

To counter the challenges posed by such varying network capacity, current video delivery solutions predominantly practice adaptive streaming (e.g., DASH [50]), where a source video is split into segments that are encoded at the server into multiple bitrates providing different video qualities, and a client runs an adaptive bitrate (ABR) algorithm to dynamically select the highest quality that fits within the estimated network capacity for the next segment to be downloaded.

**Need for layered coding.** Most of the current commercial ABR algorithms adopt a monolithic encoding practice (e.g.,

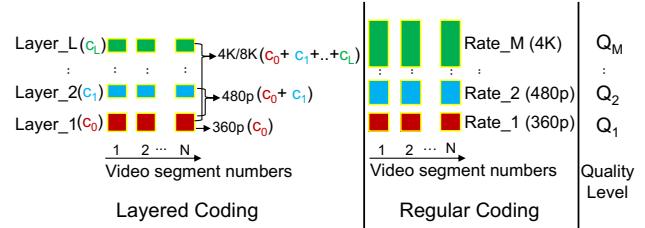


Figure 1: *Layered* vs. *Regular* coding methods. In Regular coding the video segments are coded independently at different qualities. In Layered coding a given quality can be reconstructed by combining codes for multiple layers thus facilitating incremental upgrades or downgrades.

H.265 [53]), where the same video segment is encoded ‘independently’ for each quality level. The decision on fetching a segment at a certain quality is considered *final* once the ABR algorithm makes a determination based on estimating the network capacity. However, these estimations are far from accurate, resulting in either underutilizing or overshooting the network capacity. For example, the ABR algorithm may fetch at a *low quality* by underestimating the network capacity, or it may fetch at a high quality causing *video stalls* by overestimating. Consequently, even the optimal ABR algorithms fail to provide a good quality of experience (QoE), as such rigid methods that do not fit the need of the streaming conditions.

An alternate technique, called *layered coding*, has been long studied [12, 14, 36, 47, 67] that can avoid the above streaming issues. The key idea here is that, instead of *independently* encoding the segment in different qualities, the segment is now encoded into *layers*; the base layer provides a certain video quality, and additional layers improve the video quality when applied over the base layer. See Figure 1. This means that, if the network throughput improves, one can fetch additional layers to improve video quality at a much lower cost compared to a *regular* codec.<sup>1</sup> We use the term *regular* coding to indicate the current practice of independent encoding in multiple qualities (current standards such as H.265/HEVC [53]).

**Challenges with layered coding.** Layered coding, however, faces two nontrivial challenges: *compression overhead*, and *coding latency*. The compression overhead mainly comes from not having the inter-layer frame prediction to avoid reconstruction drift in quality [29, 42, 61, 67]. On the other hand, the decoding latency is a function of the number of layers as

<sup>1</sup>We use terms coding or codec for encoding and decoding together. Also we use the terms encoding/compression, decoding/decompression interchangeably.

each layer needs to be decoded separately. Notwithstanding these issues, some studies have indeed applied layered coding in streaming and have shown slightly better QoE compared to the regular coding methods, benefiting from its ability to do dynamic quality upgrades [31]. However, they do not address either the overhead or the latency issues directly. Industry streaming solutions continue to adopt the regular codecs, shipping these codecs in hardware to avoid computational challenges, making it harder to adopt new innovations.

**Neural video codecs.** A learning approach to video coding has shown tremendous improvement in compression efficiency in just a few years [43, 60, 65]. Figure 2 shows bits-per-pixel vs PSNR plots<sup>2</sup> for several generations of codecs of two types – *neural codecs* that use deep learning and traditional *algorithmic* codecs that use the popular H.26x standards. It took algorithmic codecs 18 years to make the same progress that neural codecs achieved in the last 4 years! One reason for this rapid development is that neural codecs can run in software that can be integrated as part of the application, support agile codec development and provide royalty-free codecs. Further, they run on data parallel platforms such as GPUs that are increasingly available and powerful.

There are several insights in using neural codecs for video coding – **1)** unlike the traditional *layered* coding methods where it is nontrivial to handcraft each layer<sup>3</sup> to have unique information, a neural network’s loss function can be optimized to encode a video frame into unique layered codes by purely encoding *residuals* from previous layers without introducing a reconstruction drift; **2)** a neural network can be trained to accept a subset of the layered codes and decode all of them in a single-shot, which again was traditionally difficult to do with a handcrafted algorithm due to nonlinear relationships among the codes. Additionally, **3)** neural codecs enable software-driven coding. We note here that GPUs or similar accelerators for neural network computation are critical for success with neural codecs. Fortunately, they are increasingly common in modern devices.

**Swift.** Based on the above insights, we present *Swift*, a novel video streaming system using *layered* coding built on the principles of neural video codecs [32, 60, 65].<sup>4</sup> We show that learning can address the challenges of layered coding mentioned earlier – there is no additional compression overhead with further layering and the decoding latency is independent of the number of layers. *Swift* consists of three design components: i) server-side encoder plus decoder, ii) client-side decoder, and iii) ABR protocol adapted to layered coding and varying compute capacity (in addition to varying network capacity).

<sup>2</sup>Bits-per-pixel captures compression efficiency and PSNR (peak signal-to-noise ratio) captures image quality. Both metrics together capture codec performance.

<sup>3</sup>Throughout the paper, the term ‘layer’ refers to compressed code layers, not neural network layers.

<sup>4</sup>The source code of *Swift* is available at the following site:  
<https://github.com/VideoForge/swift>.

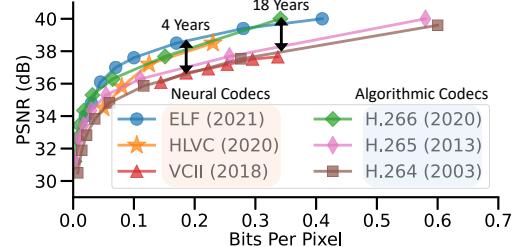


Figure 2: Evolution of neural and algorithmic video codecs showing compression efficiency plots across generations.

We evaluate *Swift* with diverse video content and FCC-released real-world network traces [8]. We compare *Swift* with state-the-art streaming algorithms that combine either regular coding [35, 51, 52] or layered coding [31] with state-of-the-art ABR algorithms. In terms of QoE, *Swift* outperforms the next-best streaming alternative by 45%. It does so using 16% less bandwidth and has a lower reaction time to changing network conditions. In terms of the neural codec, *Swift*’s layered coding outperforms the state-of-the-art layered codec (SHVC [12]) by 58% in terms of compression ratio, and by  $\times 4$  (for six layers) in terms of decoding latency. In summary, our contributions are the following:

- We show how deep learning-based coding can make layered coding both practical and high-performing, while addressing existing challenges that stymied the interest in layered coding.
- We design and build *Swift* to demonstrate a practical layered coding based video streaming system. *Swift* is an embodiment of deep learning-based encoding and decoding methods along with a purpose-built ABR protocol.
- We comprehensively evaluate and showcase the multi-dimensional benefits of *Swift* in terms of QoE, bandwidth usage, reaction times and compression efficiency.

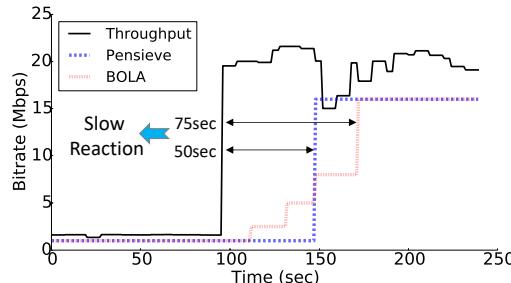
## 2 Motivation

### 2.1 Limitations of Today’s Video Streaming Due to Regular Coding

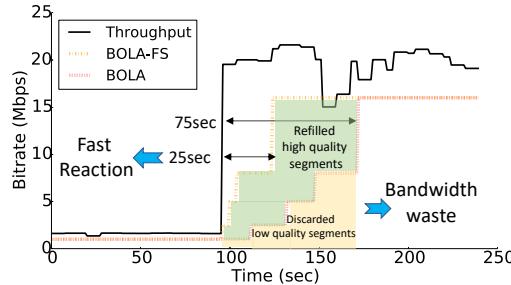
Today’s video providers predominantly use source rate adaptation (e.g., MPEG-DASH [50]) where video segments are encoded at different qualities on the server and an adaptive bitrate (ABR) algorithm chooses the best quality segment to be downloaded based on the network capacity.

The streaming solutions that are widely deployed, use *regular*, standards-driven, algorithmic coding methods such as H.265/HEVC [53] or VP9 [4] for encoding video segments. These coding methods do not allow segments to be upgraded or downgraded based on network conditions.

Figure 3 illustrates this problem using an example experiment (more details about methodology are described in §6.1). The figure shows the quality of segments that are fetched by different state-of-the-art ABR algorithms that use regular



(a) Most ABR algorithms (BOLA, Pensieve) cannot upgrade the quality of a video segment once downloaded and are slow to react to changing network conditions.



(b) BOLA-FS does allow video quality to be upgraded by re-downloading a higher quality segment. However, the previously downloaded segment is wasted.

Figure 3: Limitations of today’s ABR algorithms because of regular coding: either slower reaction to network conditions or bandwidth wastage to achieve faster reaction time to highest quality. The reaction latency includes time to notice throughput increase as well as playing the buffered segments, and hence segment duration (5 sec here) plays a role. Pensieve aggressively controls video quality fluctuations to compensate for incorrect bandwidth prediction, and hence the sudden jump in quality compared to BOLA.

coding. During the experiment, the throughput improves drastically at the 100 second mark. Two state-of-the-art streaming algorithms, Pensieve [35] and BOLA [52], cannot upgrade the quality of a segment once the segment has been downloaded. This causes a slow reaction to adjust to the improved throughput. In Figure 3(b) however, BOLA-FS [51], a version of BOLA, does allow the higher quality segment to be re-downloaded when the network conditions improve. However, the previously downloaded lower quality segment is discarded, resulting in wasted bandwidth.

## 2.2 Layered Coding

A more suitable coding method to address the above issues is layered coding, where a video segment is encoded into a *base layer* (providing the lowest playback quality level) and multiple *enhancement layers* as shown in Figure 1. Clearly, layered coding gives much finer control on rate adaptation compared to regular coding. For example, multiple enhancement layers for the same segment can be fetched incrementally as the esti-

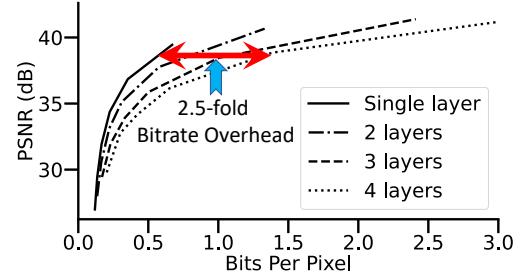


Figure 4: Compression efficiency of traditional layered coding. We use H.265 [53] and its layered extension SHVC [12] to encode the videos (described in §6.1). The single layer bitrate curve is same for both, and the additional layers are for SHVC. As shown, SHVC requires 2.5× more bits for 4 layers of SHVC compared to a single layer for the same quality.

mate of the network capacity improves closer to the playback time, which is not possible in case of regular coding.

## 2.3 Challenges of Adopting Traditional Layered Coding in Video Streaming

Layered coding has typically been developed and implemented as an extension to a *regular* coding technique. Published standards demonstrate this dependency: SHVC [12] has been developed as an extension of H.265 [53], similarly, older SVC [47] as an extension for H.264 [57]. Developing layered coding as an extension on top of a *regular* coding introduces multiple challenges in real-life deployments:

**1) Cross-layer compression overhead:** The key to large compression benefits in current generation video coding standards (e.g.,  $\approx 2000\times$  compression ratio for H.265 [53]) is *inter-frame prediction* – the consecutive frames are similar and so it is efficient to simply encode the difference between consecutive frames. However, using the inter-layer frame prediction across enhancement layers of the current frame with respect to the previous frame makes video quality drift during decoding [29, 42, 61, 67]. To minimize or avoid the drift, most of the *layered* coding methods do not use inter-frame prediction across layers and thus lose out on its compression benefits [11, 17, 31]. In effect, to achieve the same quality, layered coding (e.g., SHVC) requires significantly more bits compared to its *regular* counterpart (e.g., H.265). In our study, we find that a 4-layer SHVC coding method needs 2.5× bits per pixel compared to its *regular* coding counterpart, H.265 (see Figure 4).

**2) High encoding and decoding latency:** The computational complexity of these algorithmic codecs mainly comes from the motion estimation process during inter-frame prediction [53, 57]. During the motion estimation, it is useful - for each pixel - to encode its motion vector, i.e., where its relative location was in the previous frame. The motion vectors are computed for each frame by dividing the frame into thousands of blocks of pixels and searching a similar block in the previous frames. In general, the codecs use a set of previous

frames to search blocks in each frame making it computationally expensive. The process becomes even more complex in case of layered coding because each layer has to be decoded one after the other because of the dependency of a layer on the previous one (to exploit the content redundancy) [11, 27, 30]. This serial process of layered coding makes the latency to be a function of number of layers, and therefore the latency increases progressively as we increase the number of layers.

Figure 5 shows per-frame decoding latency of the state-of-the-art layered coding (i.e., SHVC) of a 1-min video on a desktop with configuration described in §6.1. As shown, it takes more than 100ms to decode each frame for 5 layers, an order of magnitude increase in coding latency compared to its regular counterpart H.265 (an x265 [7] implementation).

Despite several optimizations in the past, such range of latencies makes it infeasible to realize real-time decoding on heterogeneous platforms. Recent studies (e.g., Jigsaw [11]) tackle this challenge by proposing a lightweight *layered* coding method (using GPU implementation), but the latency is still a function of number of layers.

Because of these challenges, traditional layered coding is not used in practice today. In this work, rather than approaching this problem with yet another extension, we seek to explore layered coding via a clean-slate, learning-based approach with a goal towards efficient layered compression by embracing the opportunities of new hardware capabilities (e.g., GPUs and other data parallel accelerators).

## 2.4 Layered Coding using Neural Codecs

Video compression has recently experienced a paradigm shift in the computer vision community due to new advances in deep learning [32, 43, 60, 65]. The compression/decompression here is achieved using neural networks that we refer to as neural video codecs.

The basic idea is the use of an AutoEncoder (AE), a neural network architecture used to learn efficient encodings that has long been used for dimensionality reduction purposes [20]. The AE consists of an encoder and a decoder. The encoder converts an input video to a *code* vector that has a lower dimension than the input size, and the decoder reconstructs (perhaps with a small error) the original input video from the low-dimension *code* vector. The neural network weight parameters ( $W_i$  for encoder and  $W'_i$  for decoder) are trained by minimizing the *reconstruction error*, that is, minimizing the difference between the input and the output of the decoder. The smaller the code, the larger the compression factor but

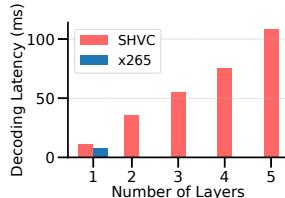


Figure 5: Latency challenges of traditional layered coding. The decoder is run on a high-end Desktop (as described in §5) using a single-threaded implementation of SHVC [3].

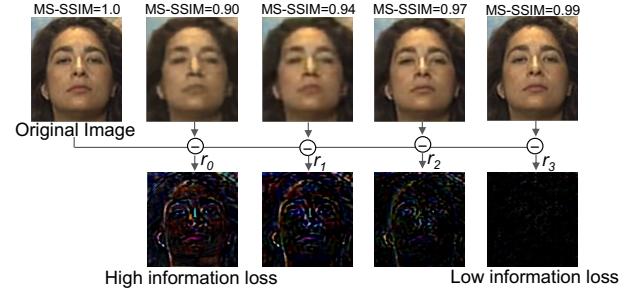


Figure 6: Illustrating the residuals ( $r_0, \dots, r_3$ ) from an original frame to a series of compressed-then-decoded frames. MS-SSIM [56] is a perceptual measure of image quality. A highly compressed frame (lowest MS-SSIM) has more residual information ( $r_0$ ).

higher the reconstruction error.

Our insight in using Autoencoders is that their loss function can be optimized to encode a video frame into unique layered codes by purely encoding residuals from previous layers, unlike the traditional layered coding where it is nontrivial to handcraft each layer to have unique information.

## 3 Swift

### 3.1 Overview

Autoencoders are already shown to provide similar or better performance relative to traditional codecs [32, 43, 60]. Recent work such as Elf-vc [43] is also able to use Autoencoders to provide flexible-rate video coding to fit a target network capacity or achieve a target compression quality. However, current work does not provide a way to encode in the video in incrementally decodable layers. To do this, we make use of *residuals* to form layered codes. A residual is the difference between the input to an encoder and output of the corresponding decoder. Residuals have been used in the past for tasks such as recognition and compression to improve the application’s efficiency (e.g., classification accuracy or compression efficiency) [19, 55, 60].

Swift uses residuals for designing layered codecs for video streaming. The idea is to employ a chain of Autoencoders of identical structure. Each Autoencoder in the chain encodes the residual from the previous layer, with the very first Autoencoder in the chain (implementing the base layer) encoding the input video frames. Figure 6 shows an example, where the residuals are shown from an original frame to a series of progressively compressed-then-decoded frames. The first decoded frame (marked with  $MS\_SSIM = 0.9$ ) has a relatively high loss from the original frame. As a result, the residual  $r_0$  has more information. When this residual information is used for the next layer’s compression, the resulting decoded frame is closer to the original, and in-turn the residual has less information, and so on.

The above ‘iterative’ chaining implicitly represents a layered encoding mechanism. Each iteration (i.e., layer) pro-

duces a compressed version of the original video that we call ‘code.’ These codes encode incremental information such that with more such codes decoded, the reconstruction becomes progressively closer to the original. *Swift* essentially uses this mechanism of residuals to create the layered codes. Such iterative minimization of residual also acts as an implicit regularization to guide the reconstruction (at a given bandwidth), instead of closely-following classical compression methods as in Elf-vc [43].

Figure 7 shows the Autoencoder architecture (more details in §3.2) on the server side. The architecture jointly learns *both* the encoder and decoder in each layer. As before, the Autoencoder’s weight parameters are trained to minimize the reconstruction error between the input and output of the decoder. In this process, the encoder generates a compact code in each layer which is a compressed version of the input video frames. These codes are transmitted to the client, where they can be decoded for progressively better reconstructions.

The decoder learnt at the server is then optimized further (§3.3) to be used at the client side. The client decoder initially reconstructs the base layer from the first code. Then, if more layers/codes are downloaded from the server, the decoder reconstructs the residuals from the second layer onward, and combines with the previous reconstruction(s) to generate the output video frame.

Overall, *Swift* has three main components:

1. A learning-based layered encoder-decoder pair in a single neural network to create residual-based layered codes on the server-side (§3.2).
2. A separate learning-based decoder on the client side. This decoder can decode any combination of layered codes in a single-shot for real-time decoding (§3.3).
3. Extension of an ABR algorithm that can integrate the codec into a complete end-to-end system (§4).

## 3.2 Layered Neural Encoder

We first describe how the encoder and the decoder are trained at the server side. Assume,  $I^t$  is the image or video frame at time  $t$ , for  $t \in \{0, 1, \dots\}$ . The encoder ( $\mathcal{E}$ ) takes each of these frames as input and generates a compact *code vector* ( $c$ ) for each frame, i.e.,  $c^t = \mathcal{E}(I^t)$ . This code for each frame is constructed by exploiting the redundancy across multiple previous frames in the video. Therefore, the encoder takes a set of previous frames as reference in order to encode each frame. The decoder ( $\mathcal{D}$ ) reconstructs the frame  $\hat{I}^t$  given  $c^t$ , i.e.,  $\hat{I}^t = \mathcal{D}(c^t)$ . The optimization problem here is to train  $\mathcal{E}$  and  $\mathcal{D}$  pairs so as to minimize the difference between  $\hat{I}^t$  and  $I^t$ . Since we add our layered coding as a generic extension to any neural codec without changing its internal logic,  $\mathcal{E}$  and  $\mathcal{D}$  can be assumed as blackboxes. An example of a neural codec is presented in Appendix A.

Figure 7 shows the design of our layered encoder-decoder network on the server-side. Here, each iteration (or layer)

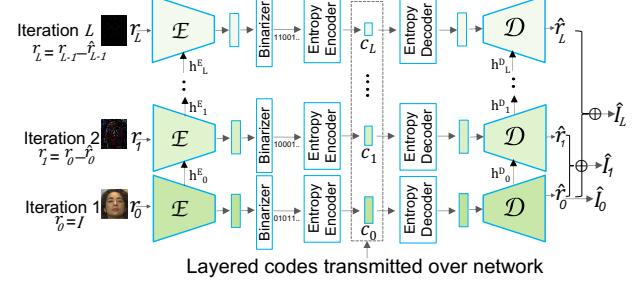


Figure 7: Deep learning based layered coding: a) iterative encoding ( $\mathcal{E}$ ) and decoding ( $\mathcal{D}$ ): in each iteration,  $\mathcal{E}$  encodes a residual into a code ( $c_i$ ) and the decoded output (from  $\mathcal{D}$ ) is used to generate the residual for the next iteration.

encodes a residual  $r_i$  into a code  $c_i$ , where residual  $r_i$  is the difference between the encoder input and decoder output in the previous layers. For the very first iteration, the encoder directly encodes the the original video frame. Representing this mathematically:  $c_i = \mathcal{E}(r_i)$  and  $r_i = r_{i-1} - \hat{r}_{i-1}$  with  $\hat{r}_0 = I$ , for  $i = 0, \dots, L$ , with the exception that for  $i = 0$  (base layer),  $r_0 = I$ .

At each iteration, the decoder can enhance the quality of the video frame with a plain arithmetic sum of the outputs of all previous iterations along with the base layer output. The key here is that both  $\mathcal{E}$  &  $\mathcal{D}$  have separate hidden states ( $h^{\mathcal{E}*}$  and  $h^{\mathcal{D}*}$ ) that get updated iteratively, sharing information between iterations. In fact, this subset of weights shared across iterations, allows better reconstruction of residuals. The entropy (i.e., the information) is very high in the initial layers, but progressively decreases due to the presence of the hidden connections and thus the code size becomes progressively smaller. The training objective for these iterative encoder-decoder pairs is to minimize the  $L_1$  reconstruction loss for the residuals:

$$\mathcal{L}_{\text{rec}} = \frac{1}{L} \sum_{i=0}^{L-1} \|\mathcal{D}(c_i) - r_i\|_1$$

All Autoencoders  $\mathcal{E}$  &  $\mathcal{D}$  in the chain share the same network and thus have identical input and output sizes. They produce the same code sizes for all iterations. *Swift* relies on a separate entropy encoding stage (Figure 7) to create the residual codes that allocate proportional number of bits to match the entropy in each iteration. The fixed length code vector from the output of the encoder  $\mathcal{E}$  is binarized and passed through a traditional entropy encoder similar to CABAC [54].

Note that the learned codec can work with a variety of input video resolutions, and hence we do not need to train a separate model for each video resolution. This is mainly because the Autoencoder here takes one or more video frames as input and extracts the features through convolutions (e.g., Conv2D [41]). Each convolutional kernel (with a fixed size of  $k \times k$  pixels that is much smaller than the input resolution) is applied in a sliding window fashion on  $k \times k$  blocks of pixels to reduce the dimensions and form the Autoencoder’s

compact code vector. In our codec we use 4 downsampling convolution blocks to reduce the dimensions. This makes any input resolution ( $w \times h$ ) to be downsampled to  $(w/16) \times (h/16)$  resolution times the Autoencoder’s bottleneck bits ( $b$ ) after the encoding stage (see Appendix A). Therefore during the testing, the encoder’s output for a  $352 \times 288$  resolution would be  $22 \times 18 \times b$ , while it is  $80 \times 45 \times b$  for  $1280 \times 720$  resolution. Similarly the codec scales with other resolutions during testing.

### 3.3 Layered Neural Decoder

The above iterative coding design already includes the decoder (Figure 7) that can reconstruct the video from the layered codes. In principle, the client can use the same decoder already designed and learned on the server-side. However, the iterative method incurs decoding latency proportional to the number of iterations. This is because the residual codes are created separately in each iteration and the decoder cannot decode a code ( $c_i$ ) unless the previous iteration of encoder encodes  $c_{i-1}$  and the corresponding decoder decodes it to form the residual  $r_i$ .

This latency is acceptable for video servers/CDNs that encode the videos offline and store them ready for on-demand streaming, but clients need to decode the video in real-time ( $\approx 30$  fps). To address this, we develop a separate design of *single-shot* decoder to be used at the clients, that can take any combination of the codes as input and decode the corresponding frames in one shot. See Figure 8.

The codes available at the client are fused and padded with zeros up to a predetermined code length (corresponding to  $L$  levels) to account for unavailable codes. They are then fed into a ‘multi-headed’ decoder ( $H$ ) as shown in the Figure 8. In each head, the padded version of each code  $c_i$  is separately processed through individual neural networks prior to combining them into a common network. When higher layers are unavailable, the corresponding heads will have no effect on reconstruction, and when available, generate a desired residual mapping. Essentially, these multiple heads are lightweight and the common network (after combining the residual codes after multiple-heads) follows the same decoder architecture  $\mathcal{D}$  from §3.2, but within one model. The heads or the common decoder do not share any parameters, in contrast to the iterative decoder which shares hidden states across iterations. Note that for preparing residual codes on the server-side, we

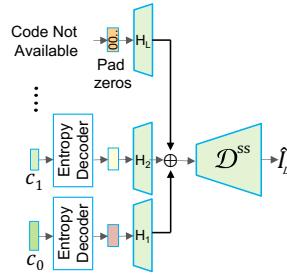


Figure 8: Single-shot decoder ( $\mathcal{D}^{\text{ss}}$ ): a variable sized decoder that takes a subset of the codes to reconstruct a video frame in one go.

still need the iterative  $\mathcal{E}$ - $\mathcal{D}$  architecture as described in §3.2. To distinguish from the server-side decoder ( $\mathcal{D}$ ), we denote this client-side single-shot decoder as  $\mathcal{D}^{\text{ss}}$ .

We train  $\mathcal{D}^{\text{ss}}$  by extending the objective function used at the server-side. Specifically, in addition to the loss function at the server-side decoder ( $\mathcal{D}$ ) which reconstructs a residual, we add a loss function that corresponds to the actual image reconstruction at client-side decoder ( $\mathcal{D}^{\text{ss}}$ ) using the available code layers. Using L1 loss function, both the objectives are as shown below.

$$\mathcal{L}_{\text{rec}} = \frac{1}{L} \sum_{i=0}^{L-1} \left[ \underbrace{\|\mathcal{D}(c_i) - r_i\|_1}_{\text{residual quality loss}} + \underbrace{\|\mathcal{D}^{\text{ss}}(\oplus_{k=0}^i c_k) - I\|_1}_{\text{image quality loss}} \right]$$

Here, the server-side decoder ( $\mathcal{D}$ ) reconstructs the residual image  $r_i$  at each iteration based on the code  $c_i$ , while the client-side  $\mathcal{D}^{\text{ss}}$  reconstructs the original image  $I$  based on the subset of the codes available at the corresponding iteration, i.e.,  $c_0 \dots c_i$ . This function allows us to train the encoder, and both the decoders (server and clients-side) in a single training loop. During the training, all three models  $\mathcal{E}$ ,  $\mathcal{D}$ , and  $\mathcal{D}^{\text{ss}}$  are jointly optimized by summing up the loss computed for  $\mathcal{E}$  and  $\mathcal{D}$  in §3.2 and the direct loss computed for  $\mathcal{D}^{\text{ss}}$  that corresponds to original image reconstruction. This joint training with a more complex objective (i.e., multiple loss functions) does affect the performance of server-side decoder. The iterative decoder from §3.2 has simpler objective than  $\mathcal{D}^{\text{ss}}$ , and hence its performance is better when trained independently (in which case only the second term in the loss function is sufficient for  $\mathcal{D}^{\text{ss}}$ ) compared to trained jointly. In our experiments we observe very little drop in quality on average with the joint training – that would be almost imperceptible to users. Moreover, training each of these models can also incur additional computation costs on servers.

## 4 Streaming with Layered Neural Codecs

Swift’s layered neural codes introduces two challenges to end-to-end streaming. The first challenge arises because Swift’s decoder at the client is expected to be run on the GPU or other similar data-parallel accelerators and run in software, instead of dedicated fixed hardware decoders as is the norm for regular codecs. Even though the software codecs have advantages in terms of on-demand codec upgrades and agile development, it raises the possibility of resource contention with other applications. Since GPU resource availability can vary [33, 45], the client needs to be able to adapt to the available resources.

The second challenge is in bitrate selection. Video streaming protocols encode each video segment into different qualities and uses ABR to select the next best quality video segment to stream. However, the ABR algorithm in Swift has a more complex choice—should one fetch the next segment at the highest possible quality or upgrade the current segment by

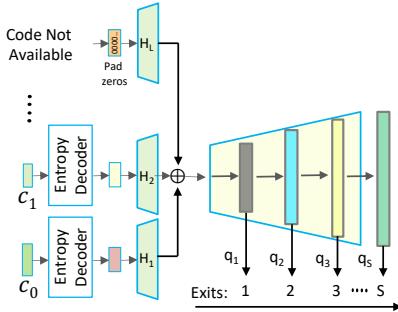


Figure 9: Scalable decoding using multiple exit heads to adapt to dynamic compute capacity. Each exit provides a different trade-off between compute capacity (and in-turn time required to decode) and quality of the video segment.

fetching additional layers? This question is made even more complex because the end-to-end streaming performance is affected both by the network and the compute variability at the client (see above). Traditional bitrate adaptation techniques are designed to only adapt to network variability.

In this section, we describe the design of a scalable decoder and neural-adapted ABR algorithm to tackle the challenges.

#### 4.1 Scaling the Decoder based on Compute Capacity

The decoder architecture (shown in Figure 8) uses network with a certain depth.<sup>5</sup> As is common in Autoencoders, the more the depth, the better is the decoding accuracy, but lower the depth, lower is the compute requirement.

Swift exploits this trade-off by designing multiple, lightweight, output heads at different depths of the network. The decoder then operates at different design points in the accuracy vs compute requirement trade-off by exiting at different depths depending on the GPU capacity. We define GPU capacity as the percentage of time over the past sample period (1 sec in our case) during which one or more cores was executing on the GPU. For example, a 100% GPU utilization means all of the GPU cores are busy with other applications in the last sample period. To this end, we introduce a number of early-exit heads ( $hd_j$ , where  $j = 1..S$ ) in the  $D^{ss}$  decoder that are corresponding to different output video qualities. See Figure 9. For example, if there are 5 exits in the network, then each exit depth outputs  $\times 16$ ,  $\times 8$ ,  $\times 4$  and  $\times 2$  smaller in resolution than the original image, with the final exit as the original reconstruction. Here, the very first early exit outputs a low quality while the final exit outputs higher quality. There has been similar early exit networks used in the literature for various tasks [28, 37].

In Swift, we define a loss function at each exit and optimize the training objective of the decoder at all exits. The decoder is trained by introducing additional L1 reconstruction

losses, so that the outputs of each of the early-exit heads minimizes the difference with the original input ( $I$ ). The objective function is as shown below:

$$\mathcal{L}_{rec} = \frac{1}{L} \sum_{i=0}^{L-1} [\underbrace{\|\mathcal{D}(c_i) - r_i\|_1}_{\text{residual quality loss}} + \underbrace{\frac{1}{S+1} \sum_{j=0}^S \left\| \mathcal{D}_{hd_j}^{ss} (\oplus_{k=0}^i c_k) - I \right\|_1}_{\text{image quality loss}}]$$

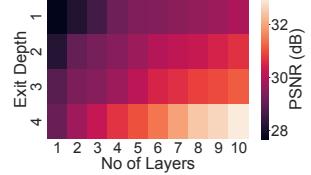


Figure 10: Quality matrix as a function of exit depth and the number of layered codes.

Here,  $S$  is the number of exits. Given a combination of these multiple exits and downloaded codes, the decoder outputs the quality corresponding to both dimensions. Figure 10 shows the heatmap of average video quality when the client decodes different number layered codes while exiting at different depths, for UVG videos described in §6.1. For example, if the client only fetches the base layer (shown as 1 in the figure) and exits at depth 1, the quality of the decoded segment is 28 dB. However, if the client decodes 4 layers and decodes to completion (exit at depth 4), the quality of the decoded segment increases to 32 dB. Note here that the number of layered codes that can be fetched depends on the network capacity while the exit depth depends on the compute capacity.

At runtime, Swift decoder decides on when to exit depending on the GPU capacity. The GPU capacity determines the latency in decoding a segment by computing until different depths. Given a GPU capacity, Swift chooses the maximum depth such that the segment will be decoded without incurring any stalls because the buffer is empty. In §5 we discuss how the client chooses the decoder and obtains the relationship between decode latency and exit depth.

#### 4.2 Adapting ABR for Layered Neural Codecs

A traditional ABR algorithm [31, 35, 52] using regular codecs takes as input the available throughput, buffer levels, and details about the future video segments. The algorithm outputs the quality of the video segment to download next. Swift needs to adapt existing ABR algorithms to work with layered neural codecs. We describe this adaptation in terms of changes to ABR’s output, input, and the objective function. We then discuss how we instantiate the ABR algorithm with these changes. See Figure 11 for an overview. In the discussion below, we assume that the ABR algorithm is run at the server; but it can be adapted to run at the client.

**Output:** The crucial change to Swift’s ABR is that unlike traditional ABR, our algorithm can make one of two choices: download the base layer (i.e., the code with lowest quality) of a future segment or, download an enhancement layer of one of the buffered video segments (that is not played yet) to

<sup>5</sup>here the depth refers to the number of layers in the neural network.

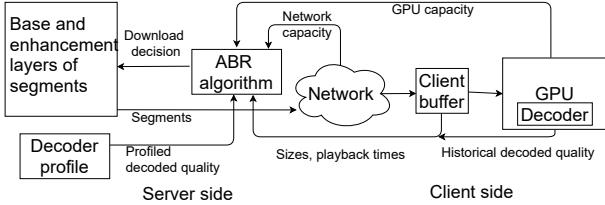


Figure 11: Swift’s video streaming pipeline.

enhance quality. It makes this determination based on a (new) set of inputs and the objective function. This change to the output is needed for streaming algorithms that use layered code, including Grad [31], the state-of-the-art streaming that uses layered coding. We compare the performance of Swift to Grad in §6.

**Input:** Swift introduces two additional inputs to the ABR to take into account the compute variability while decoding. ABR takes as input a matrix that maps the quality of the decoded segment against compute capacity needed for this decoding (similar to Figure 10). This quality matrix is generated offline at the server for each video segment for different compute capacities i.e., for all combinations of layers and exit depths. The next input is the current GPU capacity at the client. Since we run the ABR algorithm at the server, this information is sent by the client in the segment request packet. If the ABR algorithm is run on the client, the GPU capacity is readily available there. In this case the quality matrix at the server can be sent as a part of the manifest file.

The second change is with respect to segments downloaded/buffered at the client but not yet played. Swift’s ABR needs information about these segments to make its choice. Specifically, this includes i) the size of the remaining layers for current/buffered segments, ii) the playback time of all segments in the buffer, to determine if the segment can be enhanced before playback. In addition to that, we input the history of the decoded quality of the last  $k$  displayed segments to reduce variation in quality (this is often called smoothness and is an important metric for improving QoE).

**Objective function:** The video Quality of Experience (QoE) is typically captured using three metrics: i) playback quality ( $Q$ ), i.e., the quality of the downloaded segments and ii) re-buffering ratio ( $R$ ) that measures how often the video stalls because the buffer is empty, iii) smoothness ( $S$ ) that measures fluctuation in quality. Formally,  $QoE = Q - \alpha R - \beta S$ , where  $\alpha$  and  $\beta$  are the coefficients to control the penalties of rebuffering and smoothness [31, 35, 52]. Since Swift’s decoding performance is variable, it may not always provide the best quality that is possible from the downloaded segments. So instead, the objective function in Swift takes into account the quality of the *decoded segment* ( $Q_d$ ) instead of the downloaded segment. Similarly, we compute the smoothness from the quality of decoded segments ( $S_d$ ) instead of downloaded qualities.

## 5 Implementation and System Setup

Swift’s end-to-end implementation includes its layered neural codecs and the ABR protocol presented in §3.1 and §4.

### 5.1 Layered Codec Implementation

We implement our layered coding on top of VCII [60]. VCII is a state-of-the-art neural codec that is learnt over an Autoencoder network. VCII achieves compression efficiency close to state-of-the-art regular (non-neural) video codecs. Similar to regular codecs, VCII does not produce layered codes. Instead, we implement the layered encoder over VCII as described in §3.2. For the decoder at the client, similarly, we modify the loss function to incorporate the single-shot decoding and multi-exit decoder capability. Since our layered technique can be applied as a general extension to any codec, we do not need to change the internal codec logic.

After designing the new encoder/decoder over VCII, the encoder and decoder is retrained for 100K iterations on an Nvidia RTX 2070 GPU. We use ADAM optimizer with a batch size of 16. During the training, we use multiple randomly cropped  $64 \times 64$  image patches from the original images for generalization purposes. The model is trained to produce up to 10 layered codes.

For training, we use the Kinetics dataset [13]. It has 37K videos. We train on 27K, test on 10K videos. For more rigorous testing, we also test on completely different datasets (more details about testing in §6). The training takes around 6 hours. Since the training will be done offline and only once, the training time is reasonable.

### 5.2 Streaming Implementation

We implement our adapted ABR by modifying Pensieve [35]. For training the ABR model, we use  $k = 10$  throughput and compute capacity measurements passing through a 1D-CNN with 128 filters, the quality matrix passed through a 2D-CNN, and aggregated with other inputs described in §4.2. The learning rate and discount factor for the network are 0.001 and 0.99 respectively. We run the ABR algorithm every time a segment or its layer (s) is downloaded. We train the model using simulated network and compute traces, similar to that used in Pensieve [35].

We run the ABR algorithm at the server. Similar to other video streaming servers, the Swift server processes the video segments and encodes them. The server also performs fine-grained profiling of the decoder for two bits of information. First, it creates a matrix of quality levels for different depths. Second, it creates a mapping between GPU capacities and time taken to finish decoding until different depths. Both of this are used as input to the adapted ABR algorithm.

## 6 Swift Evaluation

We evaluate *Swift* both in terms of end-to-end streaming and coding performance. We compare *Swift* with a suite of streaming algorithms and its layered coding with commonly used regular codec (HEVC) [68] and layered codec (SHVC) [3]. Our evaluation shows that:

- Overall QoE with *Swift* improves by 45% at the median compared to the second best streaming performance.
- *Swift* uses 16% and 22% less bandwidth compared to next best streaming algorithms that use regular and layered codecs, respectively.
- *Swift*'s neural layered codec improves compression efficiency by 58% over state-of-the-art layered codec SHVC.

### 6.1 Evaluation Methodology

In this section, we describe the methodology to evaluate *Swift*'s end-to-end streaming performance.

**Experimental set up.** We conduct all experiments on a desktop with Nvidia 2070 RTX GPU as the client. Our evaluation uses FullHD videos from UVG [5] dataset consisting of 7 videos for streaming.<sup>6</sup> Each video is of 5 mins and is divided into 5 second segments. Each experiment runs for all segments in the video emulated over network capacity and compute capacity traces (described below). The performance is reported as an average across all the segments in the video.

**Network and compute conditions.** Most of our evaluation is over real traces collected by FCC [8], similar to recent video streaming works [35, 69]. We use 500 traces and filter the traces to have a minimum bandwidth of 1 Mbps. After filtering, FCC dataset has an average bandwidth of 8.2 Mbps with a standard deviation of 3.6 Mbps. These traces capture real world network throughput variations.

Unlike other video streaming approaches, *Swift* is affected by compute capacity. To stress test our system, we evaluate *Swift* by synthetically varying the client's GPU capacity. We modify the GPU capacity by choosing a random number of the processes to be active in each time slot; the number of processes active is modeled as a Poisson distribution with  $\lambda = 5$ . Each process shares the GPU equally and we constrain the maximum number of processes to 5.

An ideal scenario for *Swift* is when the GPU capacity is fixed and 100% of the GPU is available. For completeness, we run experiments under this condition. We refer to this as *Swift-C* in the graphs. For a fair comparison, we compare *Swift* with existing methods assuming they have hardware accelerated decoding, while varying GPU resources for *Swift*.

**Metrics.** We measure streaming performance using the following metrics: 1) video QoE (as defined in §4.2) normalized

against maximum QoE possible and averaged across all segments for all traces, 2) bandwidth usage, 3) reaction time (as defined in §6.2.3).

**Baselines.** We compare the performance of *Swift* with multiple state-of-the-art streaming algorithms that use different combinations of video coding and ABR algorithms:

- **Grad** [31]: Grad is the state-of-the-art algorithm using layered coding technique (SHVC [12]) combined with ABR. This is the closest system to *Swift*. Grad employs a hybrid coding mechanism with SHVC to minimize the cross layer compression overhead and uses a reinforcement learning-based ABR adapted from Pensieve [35].
- **BOLA** [52]: BOLA and the two alternatives below use regular (not layered) codec H.265. BOLA uses an ABR algorithm that maximizes the quality of the video segment based on the buffer levels at the client. BOLA is commonly used in the industry [2].
- **Pensieve** [35]: Pensieve is also built over H.265 [53] but uses a reinforcement learning-based ABR algorithm.
- **BOLA-FS** [51]: BOLA-FS builds over H.265 [53] and uses buffer levels at the client to choose the next video segment, similar to BOLA. However, different from BOLA, BOLA-FS allows video quality upgrades, where low quality segments in the buffer are replaced with higher quality by re-downloading them, when network conditions improve. The problem is that the previously downloaded segments are not used, resulting in wasted bandwidth.

In all of these cases, when using H.265 [53], we encode each segment into six bitrates: {1Mbps, 5Mbps, 8Mbps, 12Mbps, 16Mbps}. For Grad, which uses scalable coding, we encode the video in six layers to achieve similar quality levels. We note that in both cases, encoding the videos into 6 quality levels provided the best results. In case of H.265, it does not support fine-grained adaptation to work well with more quality levels. In case of Grad/SHVC, the compression overhead is too high when using more quality levels. For *Swift*, we encode up to 10 layers for more flexible adaptation as there is no compression overhead.

## 6.2 End-to-end Streaming Results

### 6.2.1 End-to-end QoE Results

Figure 12 shows the overall QoE of *Swift* compared with the four alternatives, along with *Swift-C*. *Swift-C* represents the best possible performance of *Swift*, when compute capacity does not vary and GPU availability is 100%.

We first compare *Swift* with Grad and BOLA-FS which can both upgrade quality of the buffered video segments when network conditions improve. *Swift* improves QoE by 43% and 48% compared to Grad and BOLA-FS respectively. In the case of Grad, the problem is the high compression overhead incurred in implementing layered coding (§2.3). In case of BOLA-FS, there is a significant bandwidth wastage. When

<sup>6</sup>Note that the compression performance is evaluated using a more diverse and standard set of video sequences (see §6.3.1).

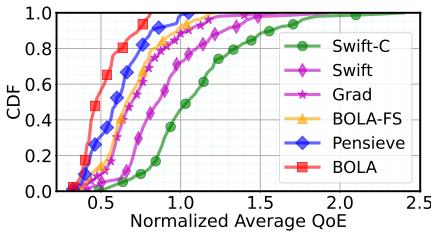


Figure 12: End-to-end QoE. Swift improves QoE by 45% at the median compared to the second best performing algorithm.

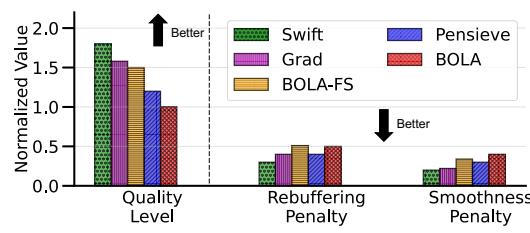


Figure 13: Breakdown of QoE. Overall, Swift has higher quality level while having less rebuffering and smoothness penalty.

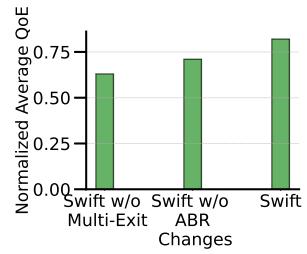


Figure 14: Breakdown Swift’s performance with its individual components.

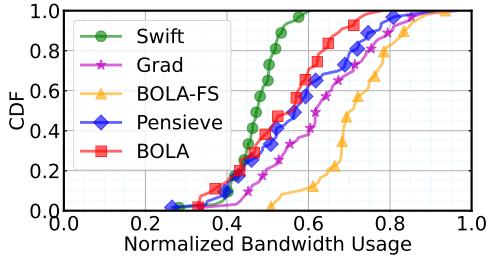


Figure 15: Bandwidth usage of Swift compared to state-of-the-art streaming systems. Swift improves bandwidth usage especially with respect to systems that upgrade quality when the network conditions improve, namely, BOLA-FS and Grad.

compared to Pensieve and BOLA, Swift outperforms by 67% and 74% respectively. Both Pensieve and BOLA do not upgrade video segment quality when the network improves, resulting in poorer quality.

Swift-C (Figure 12) shows the QoE achieved when GPU is not fluctuating and 100% of the GPU is used. As expected, Swift-C outperforms Swift under varying GPU. We also evaluated Swift under WiFi (802.11ac) network (client-server RTT: 20ms) without throttling the bandwidth and Nvidia 2070 GPU at 100%. We find that, Swift still outperforms the next-best-performing algorithm by 28%.

**QoE breakdown** Figure 13 shows the performance of the five streaming algorithms in terms of each QoE component: average quality of the video segments, rebuffering, and smoothness penalty. Swift improves average quality by 19% compared to the next-best streaming alternative. Swift also decreases rebuffering and smoothness penalty by 8% and 11%, respectively, compared to the next best streaming alternative.

**Ablation study** Figure 14 shows the impact of Swift’s components: 1) Swift without multi-exit, 2) Swift without adapted ABR. The figure shows that both components are critical to the performance of Swift. Swift without multi-exit performs poorly compared to the full Swift because the decoder runs through the entire network even when GPU capacity is low rather than exit early. This results in high decoding latency and in-turn high video stalls. In case of Swift without adapting ABR, the system performs poorly because it only adapts to network variations and not compute variations.

### 6.2.2 Bandwidth Benefits

Figure 15 shows the bandwidth benefits of Swift over existing streaming alternatives. Swift uses 16% and 22% less bandwidth compared to Grad and BOLA-FS, incurred due to compression overhead and wasted bandwidth respectively. Pensieve and BOLA results in comparatively less bandwidth waste, but cannot upgrade quality when the network improves resulting in poorer video quality (Figure 12).

### 6.2.3 Reaction to Bandwidth Fluctuations

One key advantage of Swift, or layered coding in general, is that it can adapt to bandwidth fluctuation without wasting bandwidth (see Figure 3). To illustrate this, we use an example network trace that starts with an average low bandwidth of 1 Mbps for 100 seconds and increases to average 18 Mbps for the rest of the trace (250 secs).

To compare the performance of these different streaming techniques, we measure the reaction time in two ways: 1) reaction time to any quality (RTA), which is the elapsed time between when the bandwidth increases to when the user experiences any higher quality video, 2) reaction time to highest quality (RTH), which is the elapsed time between when the bandwidth increases to when the user experiences the highest sustainable video quality.

Figure 16 shows one scenario how the different streaming algorithms adapt to changing network condition for a 250 second sample trace. The black line shows the change in throughput. Swift is the first to react to the change in throughput of all the other alternatives. Figure 17 shows qualitatively that Swift reacts faster, both in terms of RTA and RTH, compared to the alternatives.

Overall, the normalized average video segment quality of Swift throughout the trace was 1.8 compared to the next best alternative, which was 1.6. The reaction time is low even when the throughput decreases instead of increasing (not shown).

## 6.3 Compression Results

We compare Swift’s codec with:

- **HEVC [53]:** This is the most commonly used video codec for video streaming today. We use the libx265 library

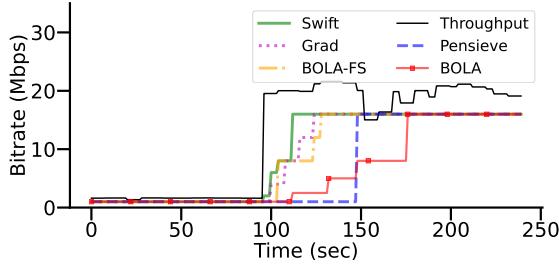


Figure 16: Swift reacts faster compared to all other alternatives. Throughput changes at the 100 second mark.

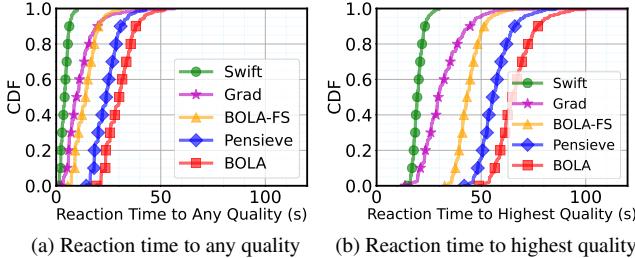


Figure 17: Reaction time. Swift reacts  $2\times$  and  $2.5\times$  faster than Grad and BOLA-FS to reach highest quality, and 25% and 40% faster to reach any high quality.

from FFMPEG [7]. We did not investigate the latest coding standard VVC [59] as it is still in its early stage. We report H.265 results with commonly used codec configuration.<sup>7</sup>

- **Scalable HEVC (SHVC)** [12]: This is a state-of-the-art layered coding method, built as a scalable extension of HEVC also known as SHVC [12]. We evaluate SHVC using a reference implementation from [1].

We present the result averaged over three datasets. One dataset is the test set from the Kinetics dataset (§5). The other two datasets are VTL [6] and UVG [5] that are not used in training. All VTL test videos are in  $352 \times 288$  and UVG videos are in  $1920 \times 1080$  resolution.

### 6.3.1 Compression Efficiency

Figure 18 shows the video quality vs. video size in terms of bits per pixel (BPP) after compression. The metrics we use are: 1) PSNR – this computes the peak signal to noise ratio between two images (higher PSNR indicates better quality of reconstruction), and 2) MS-SSIM (multi-scale structural similarity index method) – a perceptual quality metric taking into account the structural information to weigh more on the spatially close pixels with strong inter-dependencies [56]. For SHVC and Swift’s layered coding a total of 6 layers are used to produce the plots – each point refers to the joint performance of all 6 layers. For HEVC or H.265, each quality point is encoded independently with a different bitrate.

Swift’s layered coding achieves 58% better compression on average compared to traditional layered coding (SHVC).

<sup>7</sup>We use *fast* preset with group of pictures value 30.

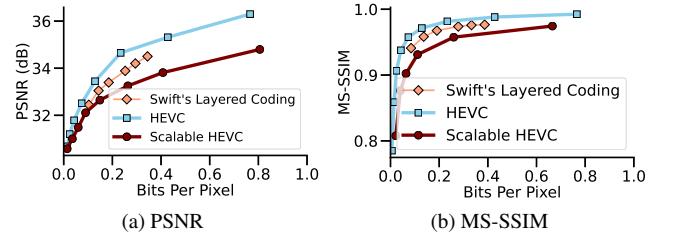


Figure 18: Compression efficiency. Swift’s Layered coding outperforms the traditional layered coding SHVC and performs close to HEVC.

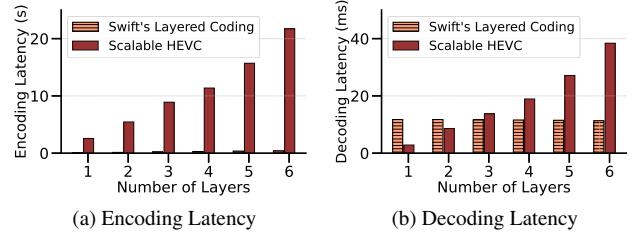


Figure 19: Encoding and Decoding latency. Both the encoding and decoding latency Swift’s layered coding is significantly less than traditional layered coding. More importantly, the decoding latency of Swift’s layered coding is independent of number of layered codes unlike traditional layered coding.

The compression difference is mainly due to the cross-layer overhead incurred by SHVC. Swift’s compression is close to that of HEVC with Swift performing poorer by 0.02 MS-SSIM and 0.8 dB in the median case. However, the QoE when using Swift is still better than the QoE than HEVC because of the fine-grained rate adaptation benefits.

### 6.3.2 Encoding and Decoding Latency

In this set of experiments, we compare the encoding and decoding latency of Swift’s codec and the state-of-the-art layered codec (we omit regular HEVC codec here because it has negligible latencies). The evaluation here benchmarks the latency on a Desktop machine (Intel 12 core CPU with Nvidia RTX 2070 GPU).<sup>8</sup> Figure 19 shows average per-frame encoding and decoding latency as a function of number of layers. There is  $15\times$  increase in encoding latency from layer one to six in case of traditional layered coding (SHVC). Swift’s layered coding has an encoding latency of 20ms for one layer and increases proportionately by  $6\times$  for the sixth layer. While improvements are still needed to get close to the encoding latency of regular (non-layered) codec, the encoding latency in Swift is significantly less than SHVC. More importantly, the decoding latency of Swift’s layered coding is independent of the number of layers, while SHVC increases proportionately similar to encoding latency. This is due to the use of the single-shot decoder of Swift (§3.3).

<sup>8</sup>Of note, HEVC and SHVC are run on the CPU and our layered coding runs on GPU as it is the most efficient on the GPU.

## 7 Discussion

Swift’s layered coding addresses compression overhead challenge highlighted in §2. Below we discuss some of the additional benefits as well as limitations of layered neural codecs.

### 7.1 Additional Opportunities of Neural Codecs

**Flexible data-driven approach:** In a learning-based approach to video coding the learning can be made video specific, for example, customized to video types [9], likely providing an opportunity to better learn video type-specific features. This ultimately leads to streaming quality improvements relative to the one-size-fits-all solution that exists today.

**Software-defined coding:** Unlike existing codecs, neural codecs do not need to be baked into a fixed hardware. They are more easily upgradable. Common ML tools (e.g., Pytorch with CUDA support) ensure that running neural codecs on data parallel co-processors (e.g., GPUs) requires significantly less development cycle compared to porting traditional codecs. The softwareization of video coding gives the content providers flexibility to integrate codec features on demand, support agile codec development, provide royalty-free codecs, and eliminate compatibility issues.

**Design of application-specific codecs:** Various video analytics solutions [21, 24, 62] often apply DNN-based analytics (e.g., object detection and classification) on video streams that are coded using traditional video codecs. However, this results in suboptimal performance because they are originally designed for human perceptual quality. Instead, neural codecs are amenable to training with loss functions more tuned towards appropriate analytics. Similarly, specialized codecs can be designed for conferencing or surveillance that may have constant backgrounds or other commonly appearing features that, once learnt, can be compressed very efficiently.

### 7.2 Limitations of Neural Codecs

One key assumption of Swift is that the client devices need to be equipped with GPUs or other similar accelerators to run neural networks. Otherwise, the decoding latency could become a bottleneck. While such accelerators are expected to be commonplace, they do add to the device cost and energy consumption. Also, the current design of Swift targets on-demand video streaming because the iterative layered coding does not offer real-time encoding. More work is needed on the encoding side for applying Swift to live video applications (such as conferencing or live analytics) to overcome the encoding latency challenges.

Finally, the QoE evaluations for Swift are done using a learning based ABR algorithm based on Pensieve [35] and Grad [31]. It may be challenging to generalize such algorithms for unknown environments that can still occur in practice [34, 63]. However, given our characterization of the input

and output along with the objective function, we expect that other algorithmic ABR approaches (such as BOLA [52] or FUGU [63]) are equally applicable for Swift.

## 8 Related Work

**Video streaming:** There has been an extensive prior work on improving QoE for regular video streaming. Much of the previous work focuses on improving the adaptive bitrate algorithms by better predicting the available throughput. Festive [25] predicts throughput using a harmonic mean. BBA [23] and BOLA [52] take into account the buffer capacity to determine video bitrate. Fugu [63] and MPC [66] use learning-based throughput prediction. There is a recent interest in using reinforcement learning for adaptive bitrate selection (e.g., Pensieve [35] and other follow-up work). Recent solutions such as SENSEI [69] improves QoE by introducing user sensitivity into ABR algorithms. Swift is able to extend existing ABR algorithms for use with neural codecs and can synergistically optimize network and compute resources to improve QoE.

**Video compression:** Traditional compression methods such as H.264/265 [53, 57] employ many algorithms that include frame prediction [64, 70], transform coding and quantization [18, 40, 48, 58], and entropy coding [54]. In the past decade or two, there have been several studies on improving both the compression efficiency and coding latency for these algorithms on an individual basis [15, 16, 26, 49]. Similarly, there have been extensive studies on improving the traditional layered coding, while still facing challenges of compression overhead and high latency [11, 14, 67]. Unlike all these algorithmic codecs, there is a recent shift in codec design using deep learning [32, 43, 44, 60]. Swift belongs to this second category and develops layered coding on top of neural codecs.

## 9 Conclusions

We have described Swift, an adaptive video streaming system using layered neural codecs that use deep learning. Swift’s neural codec achieves efficient layered compression without introducing cross layer compression overheads and eliminates the dependency of decoding latency on the number of layers. Swift extends existing ABR frameworks to accommodate layered neural codecs and demonstrates significant performance benefits compared to state-of-the-art adaptive video streaming systems.

## Acknowledgements

We thank our shepherd Junchen Jiang and the anonymous reviewers for their feedback, which greatly improved the paper. This work was partially supported by the Partner University Fund, the SUNY2020 ITSC, and a gift from Adobe.

## References

- [1] A Reference Implementation of SHVC (Scalable exten-tino to HEVC). <https://hevc.hhi.fraunhofer.de/shvc>.
- [2] Akamai players. <https://players.akamai.com/players/dashjs>.
- [3] HEVC scalability extension. <https://hevc.hhi.fraunhofer.de/shvc>.
- [4] libvpx-vp9. <https://trac.ffmpeg.org/wiki/Encode/VP9>.
- [5] Ultra video group. <http://ultravideo.fi/>.
- [6] Video trace library. <http://trace.eas.asu.edu/index.html>.
- [7] x265. <https://trac.ffmpeg.org/wiki/Encode/H.265>.
- [8] Measuring broadband America, FCC. <https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-eighth>, 2018.
- [9] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *preprint arXiv:1609.08675*, 2016.
- [10] Shivang Aggarwal, Urjit Satish Sardesai, Viral Sinha, Deen Dayal Mohan, Moinak Ghoshal, and Dimitrios Koutsonikolas. LiBRA: learning-based link adaptation leveraging PHY layer information in 60 GHz WLANs. In *ACM Conference on Emerging Networking Experiments and Technologies*, pages 245–260, 2020.
- [11] Ghufran Baig, Jian He, Mubashir Adnan Qureshi, Lili Qiu, Guohai Chen, Peng Chen, and Yinliang Hu. Jigsaw: Robust live 4K video streaming. In *MobiCom*, pages 1–16, 2019.
- [12] Jill M Boyce, Yan Ye, Jianle Chen, and Adarsh K Ramasubramonian. Overview of shvc: Scalable extensions of the high efficiency video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(1):20–34, 2015.
- [13] Joao Carreira and Andrew Zisserman. Quo vadis, Action recognition? A new model and the kinetics dataset. In *CVPR*. IEEE, 2017.
- [14] Jacob Chakareski, Sangeun Han, and Bernd Girod. Layered coding vs. multiple descriptions for video streaming over multiple paths. *Multimedia Systems*, 10(4):275–285, 2005.
- [15] Mei-Juan Chen, Yu-De Wu, Chia-Hung Yeh, Kao-Min Lin, and Shinfeng D Lin. Efficient CU and PU decision based on motion information for interprediction of HEVC. *IEEE Transactions on Industrial Informatics*, 14(11):4735–4745, 2018.
- [16] Santiago De-Luxán-Hernández, Valeri George, Jackie Ma, Tung Nguyen, Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. An intra subpartition coding mode for vvc. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1203–1207. IEEE, 2019.
- [17] Anis Elgabli, Vaneet Aggarwal, Shuai Hao, Feng Qian, and Subhabrata Sen. LBP: robust rate adaptation algorithm for SVC video streaming. *IEEE/ACM Transactions on Networking*, 26(4):1633–1645, 2018.
- [18] Vivek K Goyal. Theoretical foundations of transform coding. *IEEE Signal Processing Magazine*, 18(5):9–21, 2001.
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [20] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.
- [21] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *Usenix OSDI*, pages 269–286, 2018.
- [22] Hanzhang Hu, Debadeepa Dey, Martial Hebert, and J Andrew Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. In *AAAI*, volume 33, pages 3812–3821, 2019.
- [23] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.
- [24] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *ACM SIGCOMM*, pages 253–266, 2018.
- [25] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN)*, 22(1):326–340, 2014.
- [26] Y-H Kim, J-W Yoo, S-W Lee, J Shin, J Paik, and H-K Jung. Adaptive mode decision for H.264 encoder. *Electronics letters*, 40(19):1172–1173, 2004.
- [27] PoLin Lai, Shan Liu, and Shawmin Lei. Low latency directional filtering for inter-layer prediction in scalable video coding using hevc. In *2013 Picture Coding Symposium (PCS)*, pages 269–272. IEEE, 2013.

- [28] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D Lane. Adaptive inference through early-exit networks: Design, challenges and directions. *arXiv preprint arXiv:2106.05022*, 2021.
- [29] Athanasios Leontaris and Pamela C Cosman. Drift-resistant snr scalable video coding. *IEEE transactions on image processing*, 15(8):2191–2197, 2006.
- [30] Weiyao Lin, Krit Panusopone, David M Baylon, and Ming-Ting Sun. A computation control motion estimation method for complexity-scalable video coding. *IEEE transactions on circuits and systems for video technology*, 20(11):1533–1543, 2010.
- [31] Yunzhuo Liu, Bo Jiang, Tian Guo, Ramesh K Sitaraman, Don Towsley, and Xinbing Wang. Grad: Learning for overhead-aware adaptive video streaming with scalable video coding. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 349–357, 2020.
- [32] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. DVC: An end-to-end deep video compression framework. In *CVPR*, pages 11006–11015, 2019.
- [33] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.
- [34] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. *arXiv preprint arXiv:2008.12858*, 2020.
- [35] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*, pages 197–210. ACM, 2017.
- [36] Steven McCanne, Martin Vetterli, and Van Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE journal on selected areas in communications*, 15(6):983–1001, 1997.
- [37] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. ePerceptive: Energy Reactive Embedded Intelligence for Batteryless Sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 382–394, 2020.
- [38] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905, 2020.
- [39] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.
- [40] Tung Nguyen, Philipp Helle, Martin Winken, Benjamin Bross, Detlev Marpe, Heiko Schwarz, and Thomas Wiegand. Transform coding techniques in hevc. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):978–989, 2013.
- [41] Keiron O’Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [42] Amy R Reibman, Leon Bottou, and Andrea Basso. Scalable video coding with managed drift. *IEEE transactions on circuits and systems for video technology*, 13(2):131–140, 2003.
- [43] Oren Rippel, Alexander G Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. Elfvc: Efficient learned flexible-rate video coding. *arXiv preprint arXiv:2104.14335*, 2021.
- [44] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. *preprint arXiv:1811.06981*, 2018.
- [45] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 Usenix ATC 21*, pages 397–411, 2021.
- [46] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11 ad/ac Wireless LANs. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.
- [47] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.
- [48] Heiko Schwarz, Tung Nguyen, Detlev Marpe, and Thomas Wiegand. Hybrid video coding with trellis-coded quantization. In *2019 Data Compression Conference (DCC)*, pages 182–191. IEEE, 2019.
- [49] Mahmut E Sinangil, Vivienne Sze, Minhua Zhou, and Anantha P Chandrakasan. Cost and coding efficient

- motion estimation design considerations for high efficiency video coding (HEVC) standard. *IEEE Journal of selected topics in signal processing*, 7(6):1017–1028, 2013.
- [50] Iraj Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4), 2011.
- [51] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the dash reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 15(2s):1–29, 2019.
- [52] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.
- [53] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.
- [54] Vivienne Sze and Madhukar Budagavi. High Throughput CABAC Entropy Coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1778–1791, 2012.
- [55] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *CVPR*, pages 5306–5314, 2017.
- [56] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402. Ieee, 2003.
- [57] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.
- [58] Mathias Wien. Variable block-size transforms for H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):604–613, 2003.
- [59] Mathias Wien and Benjamin Bross. Versatile video coding—algorithms and specification. In *2020 IEEE International Conference on Visual Communications and Image Processing (VCIP)*, pages 1–3. IEEE, 2020.
- [60] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahnenbuhl. Video compression through image interpolation. In *ECCV*, pages 416–431, 2018.
- [61] Feng Wu, Shipeng Li, and Ya-Qin Zhang. A framework for efficient progressive fine granularity scalable video coding. *IEEE transactions on Circuits and Systems for Video Technology*, 11(3):332–344, 2001.
- [62] Xiufeng Xie and Kyu-Han Kim. Source compression with bounded dnn perception loss for iot edge computer vision. In *ACM MobiCom*, pages 1–16, 2019.
- [63] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.
- [64] Jiheng Yang, Baocai Yin, Yanfeng Sun, and Nan Zhang. A block-matching based intra frame prediction for H.264/AVC. In *2006 IEEE International Conference on Multimedia and Expo*, pages 705–708. IEEE, 2006.
- [65] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with hierarchical quality and recurrent enhancement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6628–6637, 2020.
- [66] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 325–338. ACM, 2015.
- [67] Sangki Yun, Daehyeok Kim, Xiaofan Lu, and Lili Qiu. Optimized layered integrated video encoding. In *INFOCOM*, pages 19–27. IEEE, 2015.
- [68] Alireza Zare, Alireza Aminlou, Miska M Hannuksela, and Moncef Gabbouj. HEVC-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 601–605. ACM, 2016.
- [69] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. Sensei: Aligning video streaming quality with dynamic user sensitivity. In *NSDI*, pages 303–320, 2021.
- [70] Shiping Zhu, Shupei Zhang, and Chenhao Ran. An improved inter-frame prediction algorithm for video coding based on fractal and H.264. *IEEE Access*, 5:18715–18724, 2017.

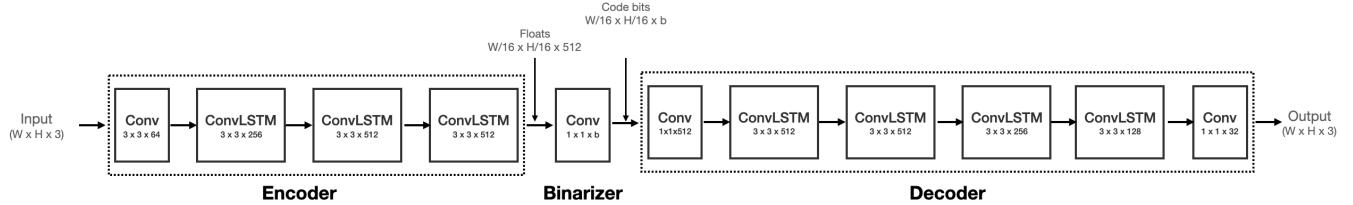


Figure 20: An example of a base neural codec and its internal logic to encode and decode a video frame in a single iteration.

## A Appendix. Example Neural Codec

Swift’s layered neural coding is designed as a generic extension that can be implemented on top of any neural codec, and hence throughout the paper, we considered neural codec as a blackbox without discussing the internal details of codec logic. In this section, we present an example codec for a better understanding of neural coding principles. This example follows the design of VCII [60], the same design we also used in our implementation.

In general, most of the existing neural codecs follow traditional concepts of I, P, and B frames when compressing a video [22, 43, 60]. An I frame is compressed much like an image with no reference, and P/B frames reference other frames for reconstruction as they encode motion and residual information relative to the reference frames. Swift adopts a similar approach of compressing I frames and P/B frames separately by using i) a neural image codec [55] for compressing I frames, and ii) a neural video codec [60] for compressing P/B frames. The output for each of these frames after the encoding stage from the Autoencoder, is a neural representation,

i.e., code bits with floating point values. The code bits for I frames represent directly the frame data, however, the code bits P/B frames represent motion and residual information with respect to the reference frames.

Figure 20 shows an example codec structure followed by [55, 60] as well as Swift. It contains three key parts: encoder, binarizer, and decoder. The encoder takes the original video frame as input and applies convolutions (along with an LSTM block) to downscale the frame into a low dimensional vector. In our example, we have four such blocks, each downscaling the frame resolution by half. For example, when we encode a  $1280 \times 720$  frame, the output of encoding stage contains  $80 \times 45 \times 512$  resolution with floating point representations. After the encoding, a binarizer converts the floats to a binary bitstream with the same resolution but packs each float in  $b$  bits. Optionally, these bits can be further passed through an entropy encoder [54] to compress the bitstream efficiently. During the decoding process, a reverse process is learned by upsampling the frame resolution at each stage in the network, achieving the original resolution at the final stage.

# Ekyा: Continuous Learning of Video Analytics Models on Edge Compute Servers

Romil Bhardwaj<sup>1,2</sup>, Zhengxu Xia<sup>3</sup>, Ganesh Ananthanarayanan<sup>1</sup>, Junchen Jiang<sup>3</sup>, Yuanchao Shu<sup>1</sup>, Nikolaos Karianakis<sup>1</sup>, Kevin Hsieh<sup>1</sup>, Paramvir Bahl<sup>1</sup>, and Ion Stoica<sup>2</sup>

<sup>1</sup>Microsoft, <sup>2</sup>UC Berkeley, <sup>3</sup>University of Chicago

## Abstract

Video analytics applications use edge compute servers for processing videos. Compressed models that are deployed on the edge servers for inference suffer from *data drift* where the live video data diverges from the training data. Continuous learning handles data drift by periodically retraining the models on new data. Our work addresses the challenge of jointly supporting inference and retraining tasks on edge servers, which requires navigating the fundamental tradeoff between the retrained model’s accuracy and the inference accuracy. Our solution Ekyा balances this tradeoff across multiple models and uses a micro-profiler to identify the models most in need of retraining. Ekyा’s accuracy gain compared to a baseline scheduler is 29% higher, and the baseline requires 4× more GPU resources to achieve the same accuracy as Ekyा.

## 1 Introduction

Video analytics applications, such as for urban mobility [2, 5] and smart cars [27], are being powered by deep neural network (DNN) models for object detection and classification, e.g., Yolo [36], ResNet [39] and EfficientNet [61]. Video analytics deployments stream the videos to *edge servers* [14, 15] placed on-premise [13, 38, 81, 84]. Edge computation is preferred for video analytics as it does not require expensive network links to stream videos to the cloud [81], while also ensuring privacy of the videos (e.g., many European cities mandate against streaming their videos to the cloud [11, 87]).

Edge compute is provisioned with limited resources (e.g., with weak GPUs [14, 15]). This limitation is worsened by the mismatch between the growth rate of the compute demands of models and the compute cycles of processors [12, 90]. As a result, edge deployments rely on *model compression* [67, 86, 94]. The compressed DNNs are initially trained on representative data from each video stream, but while in the field, they are affected by *data drift*, i.e., the live video data diverges significantly from the data that was used for training [23, 52, 77, 79]. Cameras in streets and smart cars encounter varying scenes over time, e.g., lighting, crowd densities, and changing object mixes. It is difficult to exhaustively cover all

these variations in the training, especially since even subtle variations affect the accuracy. As a result, there is a sizable drop in the accuracy of edge DNNs due to data drift (by 22%; §2.3). In fact, the fewer weights and shallower architectures of compressed DNNs often make them unsuited to provide high accuracy when trained with large variations in the data. **Continuous model retraining.** A promising approach to address data drift is continuous learning. The edge DNNs are incrementally *retrained* on new video samples even as some earlier knowledge is retained [28, 83]. Continuous learning techniques retrain the DNNs periodically [72, 93]; we refer to the period between two retrainings as the “retraining window” and use a sample of the data that is accumulated during each window for retraining. Such ongoing learning [42, 89, 96] helps the compressed models maintain high accuracy.

Edge servers use their GPUs [15] for DNN inference on many live video streams (e.g., traffic cameras in a city). Adding continuous training to edge servers presents a tradeoff between the live inference accuracy and drop in accuracy due to data drift. Allocating more resources to the retraining job allows it to finish faster and provide a more accurate model sooner. At the same time, during the retraining, taking away resources from the inference job lowers its accuracy (because it may have to sample the frames of the video to be analyzed).

Central to the resource demand and accuracy of the jobs are their *configurations*. For retraining jobs, configurations refer to the hyperparameters, e.g., number of training epochs, that substantially impact the resource demand and accuracies (§3.1). The improvement in accuracy due to retraining also depends on *how much* the characteristics of the live videos have changed. For inference jobs, configurations like frame sampling and resolution impact the accuracy and resources needed to keep up with analyzing the live video [22, 37].

**Problem statement.** We make the following decisions for retraining. (1) in each retraining window, decide which of the edge models to retrain; (2) allocate the edge server’s GPU resources among the retraining and inference jobs, and (3) select the configurations of the retraining and inference jobs. We also constraint our decisions such that the inference ac-

curacy *at any point in time* does not drop below a minimum value (so that the outputs continue to remain useful to the application). Our objective in making the above three decisions is to maximize the inference accuracy *averaged over the retraining window* (aggregating the accuracies during and after the retrainings). Maximizing inference accuracy over the retraining window creates new challenges as it is different from (i) video inference systems that optimize only the instantaneous accuracy [22, 32, 37], (ii) model training systems that optimize only the eventual accuracy [8, 17, 69, 85, 88, 95].

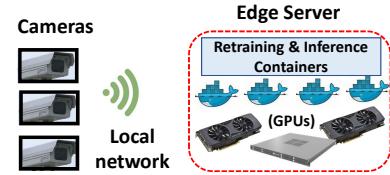
Addressing the fundamental tradeoff between the retrained model’s accuracy and the inference accuracy is computationally complex. First, the decision space is multi-dimensional consisting of a diverse set of retraining and inference configurations, and choices of resource allocations over time. Second, it is difficult to know the performance of different configurations (in resource usage and accuracy) as it requires actually retraining using different configurations. Data drift exacerbates these challenges because a decision that works well in a retraining window may not do so in the future.

**Solution components.** Our solution Ekyा has two main components: a resource scheduler and a performance estimator.

In each retraining window, the **resource scheduler** makes the three decisions listed above in our problem statement. In its decisions, Ekyा’s scheduler prioritizes retraining the models of those video streams whose characteristics have changed the most because these models have been most affected by data drift. The scheduler decides against retraining the models which do not improve our target metric. To prune the large decision space, the scheduler uses the following techniques. First, it simplifies the spatial complexity by considering GPU allocations only in coarse fractions (e.g., 10%) that are accurate enough for the scheduling decisions, while also being mindful of the granularity achievable in modern GPUs [4]. Second, it does not change allocations to jobs *during the re-training*, thus largely sidestepping the temporal complexity. Finally, our micro-profiler (described below) prunes the list of configurations to only the promising options.

To make efficient choices of configurations, the resource scheduler relies on estimates of accuracy after the retraining and the resource demands. We have designed a **micro-profiler** that observes the accuracy of the retraining configurations on a *small subset* of the training data in the retraining window with *just a few epochs*. It uses these observations to extrapolate the accuracies when retrained on a larger dataset for many more epochs. Further, we restrict the micro-profiling to only a small set of *promising* retraining configurations. These techniques result in Ekyा’s micro-profiler being 100× more efficient than exhaustive profiling while still estimating accuracies with an error of 5.8%. To estimate the resource demands, the micro-profiler measures the retraining duration *per epoch* when 100% of the GPU is allocated, and scales for different allocations, epochs, and training data sizes.

**Implementation and Evaluation.** We have evaluated Ekyा



**Figure 1:** Cameras connect to the edge server, with consumer-grade GPUs for DNN inference and retraining containers.

using a system implementation and trace-driven simulation. We used video workloads from dashboard cameras of smart cars (Waymo [68] and Cityscapes [57]) as well as from traffic and building cameras over 24 hours. Ekyा’s accuracy compared to competing baselines is 29% higher. As a measure of Ekyा’s efficiency, attaining the same accuracy as Ekyा will require 4× more GPU resources on the edge for the baseline.

**Contributions:** Our work makes the following contributions.

- 1) We introduce the metric of *inference accuracy averaged over the retraining window* for continuous training systems.
- 2) We design an *efficient micro-profiler* to estimate the benefits and costs of retraining edge DNN models.
- 3) We design a scalable resource scheduler for *joint retraining and inference* on edge servers.
- 4) We release Ekyा’s source code and video datasets with 135 hours of videos and corresponding labels to spur future research in continuous learning at the edge. See [aka.ms/ekya](http://aka.ms/ekya).

## 2 Continuous training on edge compute

### 2.1 Edge Computing for Video Analytics

Video analytics deployments commonly analyze videos on edge servers placed on-premise (e.g., from AWS [14] or Azure [15]). A typical edge server supports tens of video streams [19], e.g., on the cameras in a building, with customized models for each stream [59] (see Figure 1). Video analytics applications adopt edge computing for the following reasons [13, 38, 81].

1) Edge deployments are often in locations where the *up-link network to the cloud is expensive* for shipping continuous video streams, e.g., in oil rigs with expensive satellite network or smart cars with data-limited cellular network.<sup>1</sup>

2) Network links out of the edge locations experience *outages* [76, 81]. Edge compute provides robustness against disconnection to the cloud [26] and prevents disruptions [20].

3) Videos often contain *sensitive and private data* that users do not want sent to the cloud (e.g., many EU cities legally mandate that traffic videos be processed on-premise [11, 87]).

Thus, due to reasons of network cost and video privacy, it is preferred to run both inference and retraining on the edge compute device itself without relying on the cloud. In fact, with bandwidths typical in edge deployments, cloud-based solutions are slower and result in lower accuracies (§6.4).

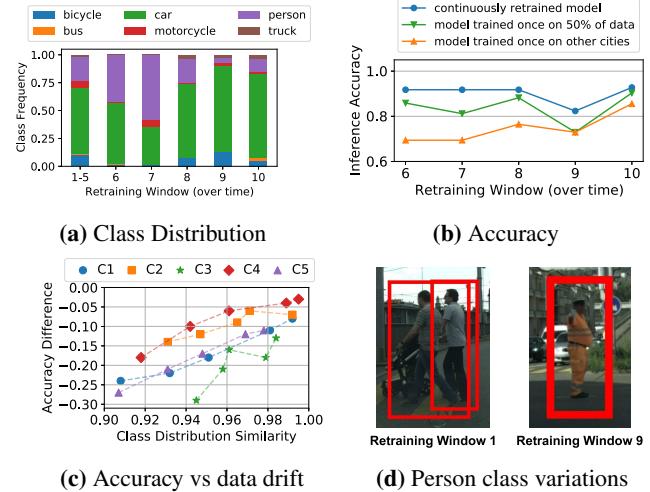
<sup>1</sup>The uplinks of LTE cellular or satellite links is 3 – 10Mb/s [58, 65], which can only support a couple of 1080p 30 fps HD video streams whereas a typical deployment has many more cameras [81].

## 2.2 Compressed DNN Models and Data drift

Advances in computer vision research have led to high-accuracy DNN models that achieve high accuracy with a large number of weights, deep architectures, and copious training data. While highly accurate, using these heavy and general DNNs for video analytics is both expensive and slow [22, 34], which make them unfit for resource-constrained edge computing. The most common approach to addressing the resource constraints on the edge is to train and deploy *specialized and compressed* DNNs [53, 60, 64, 67, 86, 94], which consist of far fewer weights and shallower architectures. For instance, Microsoft’s edge video analytics platform [5] uses a compressed DNN (TinyYOLO [75]) for efficiency. Similarly, Google released Learn2Compress[2] for edge devices to automate the generation of compressed models from proprietary models. These compressed DNNs are trained to only recognize the limited objects and scenes specific to each video stream. In other words, to maintain high accuracy, they forego generality for improved compute efficiency [22, 34, 72].

**Data drift.** As specialized edge DNNs have shallower architectures than general DNNs, they can only memorize limited amount of object appearances, object classes, and scenes. As a result, specialized edge DNNs are particularly vulnerable to *data drift* [23, 52, 77, 79], where live video data diverges significantly from the initial training data. For example, variations in the object pose, scene density (e.g. rush hours), and lighting (e.g., sunny vs. rainy days) over time make it difficult for traffic cameras to accurately identify the objects of interest (cars, bicycles, road signs). Cameras in modern cars observe vastly varying scenes (e.g., building types, crowd sizes) as they move through different neighborhoods and cities. Further, the *distribution* of the objects change over time, which reduces the edge model’s accuracy [93, 99]. Due to their ability to memorize limited amount of object variations, edge DNNs have to be continuously updated with recent data and changing object distributions to maintain a high accuracy.

**Continuous training.** The preferred approach, that has gained significant attention, is for edge DNNs to *continuously learn* as they incrementally observe new samples over time [42, 89, 96]. The high temporal locality of videos allows the edge DNNs to focus their learning on the most recent object appearances and object classes [72, 82]. In Ekyा, we use a modified version of iCaRL[89] learning algorithm to on-board new classes, as well as adapt to the changing characteristics of the existing classes. Since manual labeling is not feasible for continuous training systems on the edge, the labels for the retraining are obtained from a “golden model” - a highly accurate (87% and 84% accuracy on Cityscapes and Waymo datasets, respectively) but expensive model (deeper architecture with large number of weights). The golden model cannot keep up with inference on the live videos and we use it to label only a small fraction of the videos in the retraining window. Our approach is essentially that of supervising a



**Figure 2: Continuous learning in the Cityscapes dataset.** Shift in class distributions (a) across windows necessitates continuous learning (b). Model accuracy is not only affected by class distribution shifts (c), but also by changes in object appearances (d).

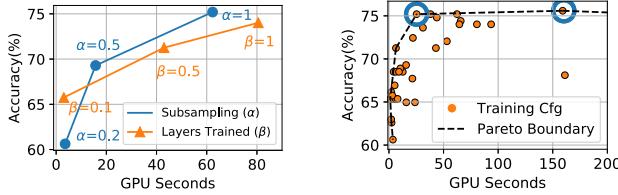
low-cost “student” model with a high-cost “teacher” model (or knowledge distillation [33]), and this has been broadly applied in computer vision literature [42, 72, 93, 96].

## 2.3 Accuracy benefits of continuous learning

To show the benefits of continuous learning, we use the video stream from one example city in the Cityscapes dataset [57] that consists of videos from dashboard cameras in many cities. In our evaluation in §6, we use both moving dashboard cameras as well as static cameras over long time periods. We divide the video data in our example city into ten fixed *re-training windows* (200s in this example).

**Understanding sources of data drift.** Figure 2a shows the change of object class distributions across windows. The initial five windows see a fair amount of persons and bicycles, but bicycles rarely show up in windows 6 and 7, while the share of persons varies considerably across windows 6 – 10. Figure 2c summarizes the effect of this data drift on model accuracy in five independent video streams, C1-C5. For each stream, we train a baseline model on the first five windows, and test it against five windows in the future and use cosine similarity to measure the class distribution shift for each window. Though accuracy generally improves when the model is used on windows with similar class distributions (high cosine similarity), the relationship is not guaranteed (C2, C3). This is because class distribution shift is not the only form of data drift. Illumination, pose and appearance differences also affect model performance (e.g. clothing and angles for objects in the person class vary significantly; Figure 2d).

**Improving accuracy with continuous learning.** Figure 2b plots inference accuracy of an edge DNN (a compressed ResNet18 classifier) in the last five windows using different training options. (1) Training a compressed ResNet18 with video data on all other cities of the Cityscapes dataset does not



(a) Effect of Hyperparameters

(b) Resource-accuracy

**Figure 3: Measuring retraining configurations.** GPU seconds refers to the duration taken for retraining with 100% GPU allocation. (a) varies two example hyperparameters, keeping others constant. Note the Pareto boundary of configurations in (b); for every non-Pareto configuration, there is at least one Pareto configuration that is better than it in *both* accuracy and GPU cost.

result in good performance. (2) Unsurprisingly, we observe that training the edge DNN once using data from the first five windows of *this example city* improves the accuracy. (3) *Continuous retraining* using the most recent data for training achieves the highest accuracy consistently. Its accuracy is higher than the other options by up to 22%.

Interestingly, using the data from the first five windows to train the larger ResNet101 DNN (not graphed) achieves better accuracy than the continuously retrained ResNet18. The substantially better accuracy of ResNet101 compared to ResNet18 when trained *on the same data* of the first five windows also shows that this training data was indeed fairly representative. But the lightweight ResNet18’s weights and architecture limits its ability to learn and is a key contributor to its lower accuracy. Nonetheless, ResNet101 is 13 $\times$  slower than the compressed ResNet18 [21]. This makes the efficient ResNet18 more suited for edge deployments and continuous learning enables it to maintain high accuracy even with data drift. Therefore, the need for continuous training of edge DNNs is ongoing and not just during a “ramp-up” phase.

### 3 Scheduling retraining and inference jointly

We propose *joint retraining and inference* on edge servers. The joint approach utilizes resources better than statically provisioning compute for retraining. Since retraining is periodic [72, 93] with far higher compute demands than inference, static provisioning causes idling. Compared to uploading videos to the cloud for retraining, our approach has advantages in privacy (§2.1), and network costs and accuracy (§6.4).

#### 3.1 Configuration diversity of retraining and inference

**Tradeoffs in retraining configurations.** The hyperparameters for retraining, or “retraining configurations”, influence the resource demands and accuracy. Retraining fewer layers of the DNN (or, “freezing” more layers) consumes lesser GPU resources, as does training on fewer data samples, but they also produce a model with lower accuracy; Figure 3a.

Figure 3b illustrates the resource-accuracy trade-offs for an edge DNN (ResNet18) with various hyperparameters: number of training epochs, batch sizes, number of neurons in the last

Configuration	Retraining Window 1		Retraining Window 2	
	End Accuracy	GPU seconds	End Accuracy	GPU seconds
Video A Cfg1A	75	85	95	90
Video A Cfg2A (*)	70	65	90	40
Video B Cfg1B	90	80	98	80
Video B Cfg2B (*)	85	50	90	70

**Table 1: Hyperparameter configurations for retraining jobs in Figure 4’s example.** At the start of retraining window 1, camera A’s inference model has an accuracy of 65% and camera B’s inference model has an accuracy of 50%. Asterisk (\*) denotes the configurations picked in Figures 4b and 4d.

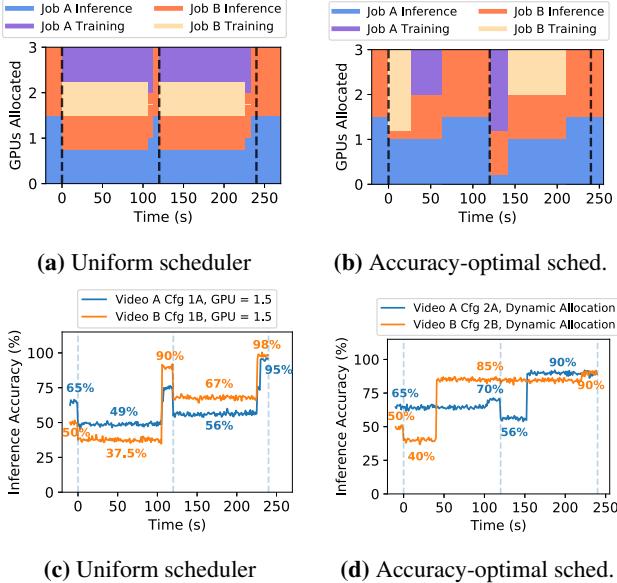
layer, number of frozen layers, and fraction of training data. We make two observations. First, there is a wide spread in the resource usage (measured in GPU seconds), by upto a factor of 200 $\times$ . Second, higher resource usage does not always yield higher accuracy. For the two configurations circled in Figure 3b, their GPU demands vary by 6 $\times$  even though their accuracies are the same ( $\sim 76\%$ ). Thus, careful selection of the configurations considerably impacts the resource efficiency. Moreover, the accuracy spread across configurations is dependent on the extent of data-drift. Retraining on visually similar data with little drift results in a narrower spread. With the changing characteristics of videos, it is challenging to efficiently obtain the resource-accuracy profiles for retraining.

**Tradeoffs in inference configurations.** Inference pipelines also allow for flexibility in their resource demands at the cost of accuracy through configurations to downsize and sample frames [59]. Reducing the resource allocation to inference pipelines increases the processing latency per frame, which then calls for sub-sampling the incoming frames to match the processing rate, that in turn reduces inference accuracy [32]. Prior work has made dramatic advancements in profilers that efficiently obtain the resource-accuracy relationship for *inference configurations* [37]. We use these efficient inference profilers in our solution, and also to ensure that the inference pipelines keep up with analyzing the live video streams.

#### 3.2 Illustrative scheduling example

We use an example with 3 GPUs and two video streams, A and B, to show the considerations in scheduling inference and retraining tasks jointly. Each retraining uses data samples accumulated since the *beginning* of the last retraining (referred to as the “retraining window”).<sup>2</sup> To simplify the example, we assume the scheduler has knowledge of the resource-accuracy profiles, but these are expensive to get in practice (we describe our efficient solution for profiling in §4.3). Table 1 shows the retraining configurations (Cfg1A, Cfg2A, Cfg1B, and Cfg2B), their respective accuracies after the retraining, and GPU cost.

<sup>2</sup>Continuous learning targets retraining windows of tens of seconds to few minutes [72, 93]. We use 120 seconds in this example. Our solution is robust to and works with any given window duration for its decisions (See §6.2).



**Figure 4: Resource allocations (top) and inference accuracies (bottom) over time for two retraining windows (each of 120s).** The left figures show a uniform scheduler which evenly splits the 3 GPUs, and picks configurations resulting in the most accurate models. The right figures show the accuracy-optimized scheduler that prioritizes resources and optimizes for inference accuracy averaged over the retraining window (73% compared to the uniform scheduler’s 56%). The accuracy-optimal scheduler also ensures that inference accuracy never drops below a minimum (set to 40% in this example, denoted as  $\alpha_{\text{MIN}}$ ).

The scheduler is responsible for selecting configurations and allocating resources for inference and retraining jobs.

**Uniform scheduling:** Building upon prior work in cluster schedulers [9, 80] and video analytics systems [32], a baseline solution for resource allocation evenly splits the GPUs between video streams, and each stream evenly partitions its allocated GPUs for retraining and inference tasks; see Figure 4a. Just like model training systems [29, 44, 45], the baseline always picks the configuration for retraining that results in the highest accuracy (Cfg1A, Cfg1B for both windows).

Figure 4c shows the *inference* accuracies of the two live streams. We see that when the retraining tasks take resources away from the inference tasks, the inference accuracy drops significantly ( $65\% \rightarrow 49\%$  for video A and  $50\% \rightarrow 37.5\%$  for video B in Window 1). While the inference accuracy increases *after* retraining, it leaves too little time in the window to reap the benefit of retraining. Averaged across both retraining windows, the inference accuracy across the two video streams is only 56% because the gains due to the improved accuracy of the retrained model are undercut by the time taken for retraining (during which inference accuracy suffered).

**Accuracy-optimized scheduling:** Figures 4b and 4d illustrate an accuracy-optimized scheduler, which by taking a holistic view on the multi-dimensional tradeoffs, provides an average inference accuracy of 73%. In fact, to match the accura-

cies, the above uniform scheduler would require nearly twice the GPUs (i.e., 6 GPUs instead of 3 GPUs).

This scheduler makes three key improvements. First, the scheduler selects the hyperparameter configurations based on their accuracy improvements *relative* to their GPU cost. It selects lower accuracy options (Cfg2A/Cfg2B) instead of the higher accuracy ones (Cfg1A/Cfg1B) because these configurations are substantially cheaper (Table 1). Second, the scheduler *prioritizes* retraining tasks that yield higher accuracy, so there is more time to reap the benefit from retraining. For example, the scheduler prioritizes B’s retraining in Window 1 as its inference accuracy after retraining increases by 35% (compared to 5% for video A). Third, the scheduler controls the accuracy drops during retraining by balancing the retraining time and the resources taken away from inference.

## 4 Eky: Solution Description

Continuous training on limited edge resources requires smartly deciding when to retrain each video stream’s model, how much resources to allocate, and what configurations to use. Making these decisions presents two challenges.

First, the decision space of multi-dimensional configurations and resource allocations is computationally more complex than two fundamentally challenging problems of multi-dimensional knapsack and multi-armed bandits (§4.1). Hence, we design a **thief scheduler** (§4.2), a heuristic that makes the joint retraining-inference scheduling tractable in practice.

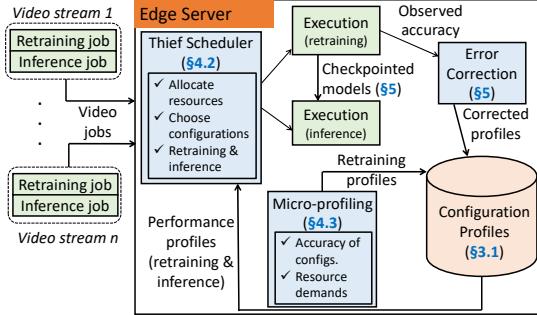
Second, the scheduler requires the model’s exact performance (in resource usage and inference accuracy), but this requires retraining using all the configurations. We address this challenge with our **micro-profiler** (§4.3), which retrains only a few select configurations on a fraction of the data. Figure 5 presents an overview of Eky’s components.

### 4.1 Formulation of joint inference and retraining

The problem of joint inference and retraining aims to maximize overall inference accuracy for all video streams  $\mathcal{V}$  in a retraining window  $T$  with duration  $\|T\|$ . All work must be done in  $\mathcal{G}$  GPUs. Thus, the total compute capability is  $\mathcal{G}\|T\|$  GPU-time. Without loss of generality, let  $\delta$  be the smallest granularity of GPU allocation. Each video  $v \in \mathcal{V}$  has a set of *retraining* configurations  $\Gamma$  and a set of *inference* configurations  $\Lambda$  (§3.1). Table 4 (§A) lists the notations.

**Decisions.** For each video  $v \in \mathcal{V}$  in a window  $T$ , we decide: (1) the retraining configuration  $\gamma \in \Gamma$  ( $\gamma = \emptyset$  means no retraining); (2) the inference configuration  $\lambda \in \Lambda$ ; and (3) how many GPUs (in multiples of  $\delta$ ) to allocate for retraining ( $\mathcal{R}$ ) and inference ( $I$ ). We use binary variables  $\phi_{v,\gamma,\lambda,\mathcal{R},I} \in \{0, 1\}$  to denote these decisions (see Table 4 §A for the definition). These decisions require  $C_T(v, \gamma, \lambda)$  GPU-time and yields overall accuracy of  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$ .  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  is averaged across the window  $T$  (§3.2), and the above decisions determine the inference accuracy at *each point in time*.

**Optimization.** Maximize the inference accuracy averaged



**Figure 5:** Ekyा’s components and their interactions.

across all videos in a retraining window within the GPU limit.

$$\arg \max_{\phi_{v\gamma\lambda\mathcal{R}I}} \frac{1}{\|\mathcal{V}\|} \sum_{\forall v \in \mathcal{V}, \forall \gamma \in \Gamma, \forall \lambda \in \Lambda, \forall \mathcal{R}, \forall I \in \{0, 1, \dots, \frac{G}{\delta}\}} \phi_{v\gamma\lambda\mathcal{R}I} \cdot A_T(v, \gamma, \lambda, \mathcal{R}, I)$$

subject to

$$\begin{aligned} 1. \quad & \sum_{\forall v \in \mathcal{V}, \forall \gamma \in \Gamma, \forall \lambda \in \Lambda, \forall \mathcal{R}, \forall I} \phi_{v\gamma\lambda\mathcal{R}I} \cdot C_T(v, \gamma, \lambda) \leq G \|T\| \\ 2. \quad & \sum_{\forall v \in \mathcal{V}, \forall \gamma \in \Gamma, \forall \lambda \in \Lambda, \forall \mathcal{R}, \forall I} \phi_{v\gamma\lambda\mathcal{R}I} \cdot (\mathcal{R} + I) \leq \frac{G}{\delta} \\ 3. \quad & \sum_{\forall \gamma \in \Gamma, \forall \lambda \in \Lambda, \forall \mathcal{R}, \forall I} \phi_{v\gamma\lambda\mathcal{R}I} \leq 1, \forall v \in \mathcal{V} \end{aligned} \quad (1)$$

The first constraint ensures that the GPU allocation does not exceed the available GPU-time  $G \|T\|$  in the retraining window. The second constraint limits the *instantaneous* allocation (in multiples of  $\delta$ ) to never exceed the available GPUs. The third constraint ensures that at most one configuration is picked for retraining and inference each for a video  $v$ .

Our analysis in §A.1 shows that the above optimization problem is *harder* than the multi-dimensional binary knapsack problem and modeling the uncertainty of  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  is more challenging than the multi-armed bandit problem.

## 4.2 Thief Scheduler

Our scheduling heuristic makes the scheduling problem tractable by decoupling resource allocation (i.e.,  $\mathcal{R}$  and  $I$ ) and configuration selection (i.e.,  $\gamma$  and  $\lambda$ ) (Algorithm 1). We refer to Ekyा’s scheduler as the “thief” scheduler and it iterates among all inference and retraining jobs as follows.

(1) It starts with a fair allocation for all video streams  $v \in \mathcal{V}$  (line 2 in Algorithm 1). In each step, it iterates over all the inference and retraining jobs of each video stream (lines 5-6), and *steals* a tiny quantum  $\Delta$  of resources (in multiples of  $\delta$ ; see Table 4, §A) from each of the other jobs (lines 10-11).

(2) With the new resource allocations ( $\text{temp\_alloc}[]$ ), it then selects configurations for the jobs using the *PickConfigs* method (line 14 and Algorithm 2, §A) that iterates over all the configurations for inference and retraining for each video stream. For inference jobs, among all the configurations whose accuracy is  $\geq a_{\min}$ , *PickConfigs* picks the configuration with the highest

accuracy that can keep up with the inference of the live video stream given current allocation (line 3-4, Algorithm 2, §A).

For retraining jobs, *PickConfigs* picks the configuration that maximizes the accuracy  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  over the retraining window for each video  $v$  (lines 6-12, Algorithm 2, §A). *EstimateAccuracy* (line 7, Algorithm 2, §A) aggregates the instantaneous accuracies over the retraining window for a given pair of inference configuration (chosen above) and retraining configuration. Ekyा’s micro-profiler (§4.3) provides the estimate of the accuracy and the time to retrain for a retraining configuration when 100% of GPU is allocated, and *EstimateAccuracy* proportionately scales the GPU-time for the current allocation (in  $\text{temp\_alloc}[]$ ) and training data size. In doing so, it avoids configurations whose retraining durations exceed  $\|T\|$  with the current allocation (first constraint in Eq. 1).

(3) After reassigning the configurations, Ekyा uses the estimated average inference accuracy (accuracy\_avg) over the retraining window (line 14 in Algorithm 1) and keeps the new allocations only if it improves up on the accuracy from prior to stealing the resources (line 15 in Algorithm 1).

The thief scheduler repeats the process till the accuracy stops increasing (lines 15-20 in Algorithm 1) and until all the jobs have played the “thief”. Algorithm 1 is invoked at the beginning of each retraining window, as well as on the completion of every training job during the window.

**Design rationale:** We call out the key aspects that makes the scheduler’s decision efficient by pruning the search space.

- *Coarse allocations:* The thief scheduler allocates GPU resources in quantums of  $\Delta$ . Intuitively,  $\Delta$  is the step size for allocation used by the scheduler. Thus, the final resource allocation from the thief scheduler is within  $\Delta$  of the optimal allocation. We empirically pick a  $\Delta$  that is coarse yet accurate enough in practice, while being mindful of modern GPUs[4]; see §6.2. Algorithm 1 ensures that the total allocation is within the limit (second constraint in Eq 1).
- *Reallocating resources only when a retraining job completes:* Although one can reallocate GPU resource among jobs at finer temporal granularity (e.g., whenever a retraining job has reached a high accuracy), we empirically find that the gains from such complexity is marginal.
- *Pruned configuration list:* Our micro-profiler (described next) speeds up the thief scheduler by giving it only the more promising configurations. Thus, the list  $\Gamma$  used in Algorithm 1 is significantly smaller than the exhaustive set.

## 4.3 Performance estimation with micro-profiling

Ekyा’s scheduling decisions in §4.2 rely on estimations of post-retraining accuracy and resource demand of the retraining configurations. Specifically, at the beginning of each retraining window  $T$ , we need to *profile* for each video  $v$  and each configuration  $\gamma \in \Gamma$ , the accuracy after retraining using  $\gamma$  and the corresponding time taken to retrain.

**Profiling in Ekyा vs. hyperparameter tuning:** While Ekyा’s profiling may look similar to hyperparameter tuning

---

**Algorithm 1: Thief Scheduler.**

---

**Data:** Training ( $\Gamma$ ) and inference ( $\Lambda$ ) configurations  
**Result:** GPU allocations  $\mathcal{R}$  and  $I$ , chosen configurations  
 $(\gamma \in \Gamma, \lambda \in \Lambda) \forall v \in V$

```
1 all_jobs[] = Union of inference and training jobs of videos  $V$ ;  
  /* Initialize with fair allocation */  
2 best_alloc[] = fair_allocation(all_jobs);  
3 best_configs[], best_accuracy_avg = PickConfigs(best_alloc);  
  /* Thief resource stealing */  
4 for thief_job in all_jobs[] do  
5   for victim_job in all_jobs[] do  
6     if thief_job == victim_job then continue;  
7     temp_alloc[]  $\leftarrow$  best_alloc[];  
8     while true do  
9       /*  $\Delta$  is the increment of stealing */  
10      temp_alloc[victim_job]  $-=$   $\Delta$ ;  
11      temp_alloc[thief_job]  $+=$   $\Delta$ ;  
12      if temp_alloc[victim_job] < 0 then break;  
      /* Calculate accuracy over retraining  
       window and pick configurations. */  
13      temp_configs[], accuracy_avg =  
        PickConfigs(temp_alloc[]);  
14      if accuracy_avg > best_accuracy_avg then  
15        best_alloc[] = temp_alloc[];  
16        best_accuracy_avg = accuracy_avg;  
17        best_configs[] = temp_configs[];  
18      else  
        break;  
19 return best_alloc[], best_configs[];
```

---

(e.g., [46, 48, 62, 85]) at first blush, there are two key differences. First, Ekyा needs the performance estimates of a broad set of candidate configurations for the thief scheduler, not just of the single best configuration, because the best configuration is jointly decided across the many retraining and inference jobs. Second, in contrast to hyperparameter tuning which runs separately of the eventual inference/training, Ekyा’s profiling must share compute resource with all retraining and inference.

**Opportunities:** Ekyा leverages three empirical observations for efficient profiling of the retraining configurations. (i) Resource demands of the configurations are deterministic. Hence, we measure the GPU-time taken to retrain for *each epoch* in the current retraining window when 100% of the GPU is allocated to the retraining. This GPU-time must then be re-scaled for varying number of epochs, GPU allocations, and training data sizes in Algorithm 1. For re-scaling number of epochs and training data sizes, we linearly scale the GPU-time. For re-scaling GPU allocations, we use an offline computed profile of the model throughput for different resource allocations to account for sub-linear scaling. Our real testbed-based evaluation shows that these rescaling functions works well in practice. (ii) Post-retraining accuracy can be roughly estimated by training on a small subset of training data for a handful of epochs. (iii) The thief scheduler’s deci-

sions are not impacted by small errors in the estimations.

**Micro-profiling design:** The above insights inspired our approach, called *micro-profiling*, where for each video, we test the retraining configurations on a *small subset* of the retraining data and only for a *small number* of epochs (well before models converge). Our micro-profiler is 100× more efficient than exhaustive profiling (of all configurations on the entire training data), while predicting accuracies with an error of 5.8%, which is low enough in practice to *mostly* ensure that the thief scheduler makes the same decisions as it would with a fully accurate prediction. We use these insights to now explain the techniques that make Ekyा’s micro-profiling efficient.

1) *Training data sampling*: Ekyा’s micro-profiling works on only a small fraction (say, 5% – 10%) of the training data in the retraining window (which is already a subset of all the videos accumulated in the retraining window). While we considered weighted sampling techniques for the micro-profiling, we find that uniform random sampling is the most indicative of the configuration’s performance on the full training data, since it preserves all the data distributions and variations.

2) *Early termination*: Similar to data sampling, Ekyा’s micro-profiling only tests each configuration for a small number (say, 5) of training epochs. Compared to a full fledged profiling that needs few tens of epochs to converge, such early termination greatly speeds up the micro-profiling process.

After early termination on the sampled training data, we obtain the (validation) accuracy of each configuration at each epoch it was trained. We then fit the accuracy–epoch points to the a non-linear curve model from [70] using a non-negative least squares solver [6]. This model is then used to extrapolate the accuracy that would be obtained by retraining with all the data for larger number of epochs. The use of this extrapolation is consistent with similar work in this space [55, 70].

3) *Pruning bad configurations*: Finally, Ekyा’s micro-profiling also prunes out those configurations for micro-profiling (and hence, for retraining) that have historically not been useful. These are configurations that are significantly distant from the configurations on the Pareto curve of the resource-accuracy profile (see Figure 3b), and thus unlikely to be picked by the thief scheduler. To bootstrap pruning, all configurations are evaluated in the first window. After every 2 windows, a fixed fraction of the worst performing configurations are dropped. While first few retraining windows must explore a big space of configurations, the search space size drops exponentially over time. Avoiding these configurations improves the efficiency of the micro-profiling.

**Annotating training data:** For both the micro-profiling as well as the retraining, Ekyा acquires labels using a “golden model” (§2.2). This is a high-cost but high-accuracy model trained on a large dataset. As explained in §2, the golden model cannot keep up with inference on the live videos and we use it to label only a small subset of the videos for retraining. The delay of annotating training data with the golden model

is accounted by the scheduler as follows: we subtract the data annotation delay from the retraining window and only pass the remaining time of the window to Algorithm 2 (§A).

## 5 Implementation and Experimental Setup

**Implementation:** Ekyा uses PyTorch [66] for running and training ML models, and each component is implemented as a collection of long-running processes with the Ray[63] actor model. The micro-profiler and training/inference jobs run as independent actors which are controlled by the thief scheduler actor. Ekyा achieves fine-grained and dynamic reallocation of GPU between training and inference processes using Nvidia MPS [4], which provides resource isolation within a GPU by intercepting CUDA calls and rescheduling them. Our implementation also adapts to errors in profiling by reactively adjusting its allocations if the actual model performance diverges from the predictions of the micro-profiler. Ekyा’s code and datasets are available at the project page: [aka.ms/ekya](http://aka.ms/ekya)

**Datasets:** We use both on-road videos captured by dashboard cameras as well as urban videos captured by mounted cameras. The dashboard camera videos are from cars driving through cities in the US and Europe, Waymo Open [68] (1000 video segments with in total 200K frames) and Cityscapes [57] (5K frames captured by 27 cameras) videos. The urban videos are from stationary cameras mounted in a building (“Urban Building”) as well as from five traffic intersections (“Urban Traffic”), both collected over 24-hour durations. We use a retraining window of 200 seconds in our experiments, and split each of the videos into 200 second segments. Since the Waymo and Cityscapes dataset do not contain continuous timestamps, we create retraining windows by concatenating images from the same camera in chronological order to form a long video stream and split it into 200 second segments.

**DNNs:** We demonstrate Ekyा’s effectiveness on two machine learning tasks – object classification and object detection – using multiple compressed edge DNNs for each task: (i) object classification using ResNet18[39], MobileNetV2[53] and ShuffleNet[98], and (ii) object detection using TinyYOLOv3[75] and SSD[49]. As explained in §2.2, we use an expensive golden model (ResNeXt 101 [91] for object classification and YOLOv3 [75] for object detection) to get ground truth labels for training and testing.

**Testbed and trace-driven simulator:** We run Ekyा’s implementation on AWS EC2 p3.2xlarge instances for 1 GPU experiments and p3.8xlarge for 2 GPU experiments. Each instance has Nvidia V100 GPUs with NVLink interconnects.

We also built a simulator to test Ekyा under a wide range of resource constraints, workloads, and longer durations. The simulator takes as input the accuracy and resource usage (in GPU time) of training/inference configurations logged from our testbed. For each training job, we log the accuracy over GPU-time. We also log the inference accuracy on the real videos. This exhaustive trace allows us to mimic the jobs with

high fidelity under different scheduling policies.

**Retraining configurations:** Our retraining configurations combine the number of epochs to train, batch size, number of neurons in the last layer, number of layers to retrain, and the fraction of data between retraining windows to use for retraining (§3.1). For the object detection models (TinyYOLO and SSDLite), we set the batch size to 8 and the fraction of layers frozen between 0.7 and 0.9. The resource requirements of the configurations for the detection models vary by 153×.

**Baselines:** Our baseline, called *uniform scheduler*, uses (a) a fixed retraining configuration, and (b) a static retraining/inference resource allocation (these are adopted by prior schedulers [9, 32, 80]). For each dataset, we test all retraining configurations on a hold-out dataset<sup>3</sup> (*i.e.*, two video streams that were never used in later tests) to produce the Pareto frontier of the accuracy-resource tradeoffs (*e.g.*, Figure 3). The uniform scheduler then picks two points on the Pareto frontier as the fixed retraining configurations to represent “high” (Config 1) and “low” (Config 2) resource usage, and uses one of them for all retraining windows in a test.

We also consider two alternatives in §6.4. (1) *offloading re-training to the cloud*, and (2) *caching and re-using a retrained model* from history based on various similarity metrics.

## 6 Evaluation

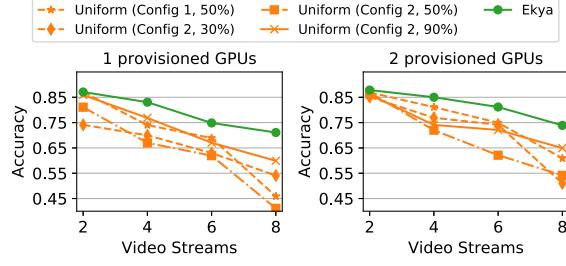
We evaluate Ekyा’s performance, and the key findings are:

- 1) Compared to static retraining baselines, Ekyा achieves upto 29% higher accuracy for compressed vision models in both classification and detection. For the baseline to match Ekyा’s accuracy, it would need 4× additional GPU resources. (§6.1)
- 2) Both micro-profiling and thief scheduler contribute sizably to Ekyा’s gains. (§6.2) In particular, the micro-profiler estimates accuracy with low median errors of 5.8%. (§6.3)
- 3) The thief scheduler efficiently makes its decisions in 9.4s when deciding for 10 video streams across 8 GPUs with 18 configurations per model for a 200s retraining window. (§6.2)
- 4) Compared to alternate designs, including reusing cached history models trained on similar data/scenarios as well as retraining the models in the cloud, Ekyा achieves significantly higher accuracy without the network costs (§6.4).

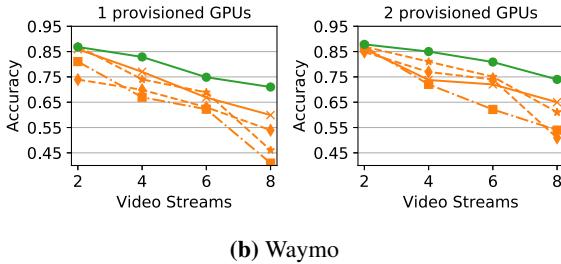
### 6.1 Overall improvements

We evaluate Ekyा and the baselines along three dimensions—*inference accuracy* (% of images correctly classified for object classification, F1 score (measured at a 0.3 threshold for the Intersection-over-Union of the bounding box) for detection), *resource consumption* (in GPU time), and *capacity* (the number of concurrently processed video streams). Note that the evaluation always keeps up with the video frame rate (*i.e.*, no indefinite frame queueing). By default we evaluate the performance of Ekyा on ResNet18 models, but we also show that it generalizes to other model types and vision tasks.

<sup>3</sup>The same hold-out dataset is used to customize the off-the-shelf DNN inference model. This is a common strategy in prior work (*e.g.*, [22]).



(a) Cityscapes



(b) Waymo

**Figure 6:** Effect of adding video streams on accuracy with different schedulers. When more video streams share resources, Ekyा’s accuracy gracefully degrades while the baselines’ accuracy drops faster. (“Uniform (Cfg 1, 90 %)” means the uniform scheduler allocates 90% GPU to inference, 10% to retraining)

**Accuracy vs. Number of concurrent video streams:** Figure 6 shows the ResNet18 model’s accuracy with Ekyा and the baselines when analyzing a growing number of concurrent video streams under a fixed number of provisioned GPUs for Waymo and Cityscapes datasets. The uniform baselines use different combinations of pre-determined retraining configurations and resource partitionings. For consistency, the video streams are shuffled and assigned an id (0-10), and are then introduced in the same increasing order of id in all experiments. This ensures that different schedulers tested for  $k$  parallel streams use the same  $k$  streams, and these  $k$  streams are always a part of any  $k'$  streams ( $k' > k$ ) used for testing.

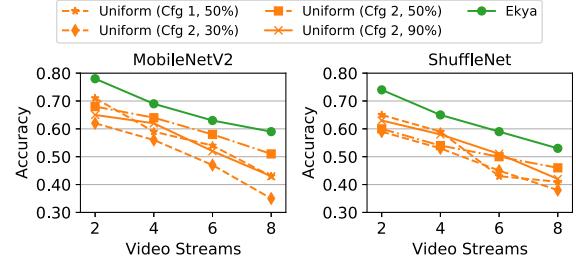
As the number of video streams increases, Ekyा enjoys a growing advantage (upto 29% under 1 GPU and 23% under 2 GPU) in accuracy over the uniform baselines. This is because Ekyा gradually shifts more resource from retraining to inference and uses cheaper retraining configurations. In contrast, increasing the number of streams forces the uniform baseline to allocate less GPU cycles to each inference job, while retraining jobs, which use fixed configurations, slow down and take the bulk of each window.

**Generalizing to other ML models:** Ekyा’s thief scheduler can be readily applied to any ML model and task (e.g., classification or detection) that needs to be fine-tuned continuously on newer data. To demonstrate this, we evaluate Ekyा with:

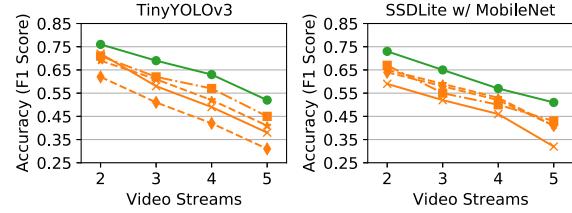
- *Other object classifiers:* Figure 7a shows the performance of Ekyा when running MobileNetV2 and ShuffleNet as the edge models in two independent setups for object classification at the edge. Continuing the trend that we observed for ResNet18 (in Figure 6), Figure 7a shows that Ekyা leads

Scheduler	Capacity		Scaling factor
	1 GPU	2 GPUs	
<b>Ekyा</b>	<b>2</b>	<b>8</b>	<b>4x</b>
Uniform (Config 1, 50%)	2	2	1x
Uniform (Config 2, 90%)	2	4	2x
Uniform (Config 2, 50%)	2	4	2x
Uniform (Config 2, 30%)	0	2	-

**Table 2: Capacity (number of video streams that can be concurrently supported subject to accuracy target 0.75) vs. number of provisioned GPUs. Ekyा scales better than the uniform baselines with more available compute resource.**



(a) Generalize across object classification models



(b) Object Detection Models

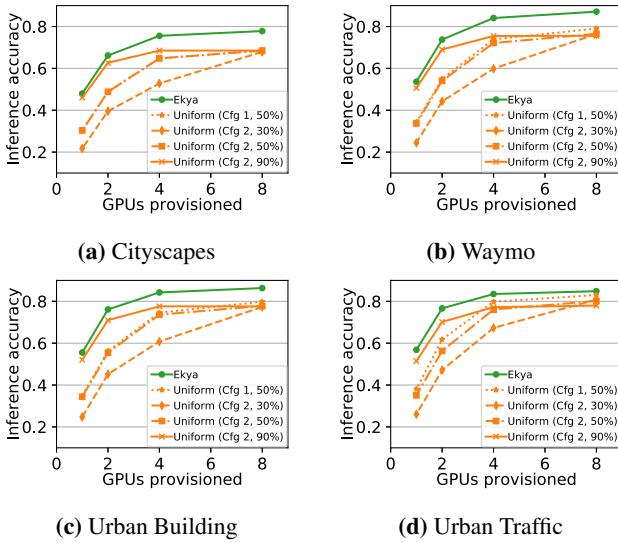
**Figure 7:** Improvement of Ekyा extends to two more compressed DNN classifiers and two popular object detectors.

to up to 22% better accuracy than uniform baselines.

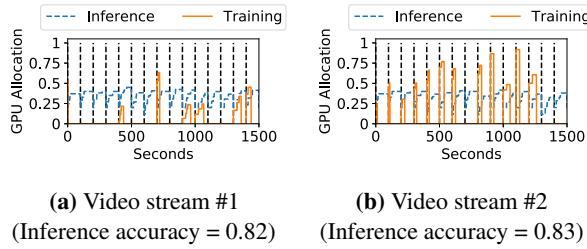
- *Object detection models:* In addition to object classification, we also evaluate using object detection tasks which detect the bounding boxes of objects in the video stream. Figure 7b shows Ekyা outperforms the uniform baseline’s F1 score by 19% when processing same number of concurrent video streams. Importantly, Ekyা’s design broadly applies to new tasks without any systemic changes.

These gains stem from Ekyা’s ability to navigate the rich resource-accuracy space of models by carefully selecting training and inference hyperparameters (e.g., the width multiplier in MobileNetV2, convolution sparsity in ShuffleNet). For the rest of our evaluation, we only present results with ResNet18 though the observations hold for other models.

**Number of video streams vs. provisioned resource:** We compare Ekyা’s *capacity* (defined by the maximum number of concurrent video streams subject to an accuracy threshold) with that of uniform baseline, as more GPUs are available. Setting an accuracy threshold is common in practice, since applications usually require accuracy to be above a threshold for the inference to be usable. Table 2 uses the Cityscapes results (Figure 6) to derive the scaling factor of capacity vs.



**Figure 8: Inference accuracy of different schedulers when processing 10 video streams under varying GPU provisioning.**



**Figure 9: Ekyा's resource allocation to two video streams over time. Ekyा adapts when to retrain each stream's model and allocates resource based on the retraining benefit to each stream.**

the number of provisioned GPUs and shows that with more provisioned GPUs, Ekyा scales faster than uniform baselines.

**Accuracy vs. provisioned resource:** Finally, Figure 8 stress-tests Ekyा and the uniform baselines to process 10 concurrent video streams and shows their average inference accuracy under different number of GPUs. To scale to more GPUs, we use the simulator (§5), which uses profiles recorded from real tests and we verified that it produced similar results as the implementation at small-scale. As we increase the number of provisioned GPUs, we see that Ekyा consistently outperforms the best of the two baselines by a considerable margin and more importantly, with 4 GPUs Ekyा achieves higher accuracy (marked with the dotted horizontal line) than the baselines at 16 GPUs (*i.e.*, 4× resource saving).

The above results show that Ekyा is more beneficial when there is high contention for the GPU on the edge. Under low contention, the room for improvement shrinks. Contention is, however, common in the edge since the resources are tightly provisioned to minimize their idling.

## 6.2 Understanding Ekyा's improvements

**Resource allocation across streams:** Figure 9 shows Ekyा's resource allocation across two example video streams over

several retraining windows. In contrast to the uniform baselines that use the same retraining configuration and allocate equal resource to retraining and inference (when retraining takes place), Ekyा retrains the model only when it benefits and allocates different amounts of GPUs to the retraining jobs of video streams, depending on how much accuracy gain is expected from retraining on each stream. In this case, more resource is diverted to video stream #1 (#1 can benefit more from retraining than #2) and both video streams achieve much higher accuracies (0.82 and 0.83) than the uniform baseline.

**Component-wise contribution:** Figure 10a understands the contributions of resource allocation and configuration selection (on 10 video streams with 4 GPUs provisioned). We construct two variants from Ekyा: *Ekyा-FixedRes*, which removes the smart resource allocation in Ekyा (*i.e.*, using the inference/training resource partition of the uniform baseline), and *Ekyा-FixedConfig* removes the microprofiling-based configuration selection in Ekyा (*i.e.*, using the fixed configuration of the uniform baseline). Figure 10a shows that both adaptive resource allocation and configuration selection has a substantial contribution to Ekyा's gains in accuracy, especially when constrained (*i.e.*, fewer resources are provisioned).

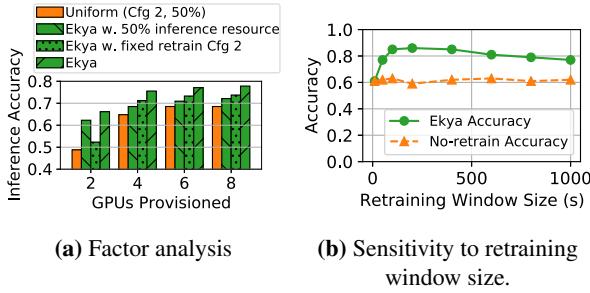
**Retraining window sensitivity analysis:** Figure 10b evaluates the sensitivity of Ekyा to the retraining window size. Ekyा is robust to different retraining window sizes. When the retraining window size is too small (10 seconds), the accuracy of Ekyा is equivalent to no retraining accuracy due to insufficient time and resources for retraining. As the window increases, Ekyा's performance quickly ramps up because the thief scheduler is able to allocate resources to retraining. As the retraining window size further increases Ekyा's performance slowly starts moderately degrading because of the inherent limitation in capacity of compressed models (§2.3).

**Impact of scheduling granularity:** A key parameter in Ekyा's scheduling algorithm (§4.2) is the allocation quantum  $\Delta$ : it controls the runtime of the scheduling algorithm and the granularity of resource allocation. In our sensitivity analysis with 10 video streams, we see that increasing  $\Delta$  from 1.0 (coarse-grained; one full GPU) to 0.1 (fine-grained; fraction of a GPU), increases the accuracy substantially by ~8%. Though the runtime also increases to 9.5 seconds, it is still a tiny fraction (4.7%) of the retraining window (200s).

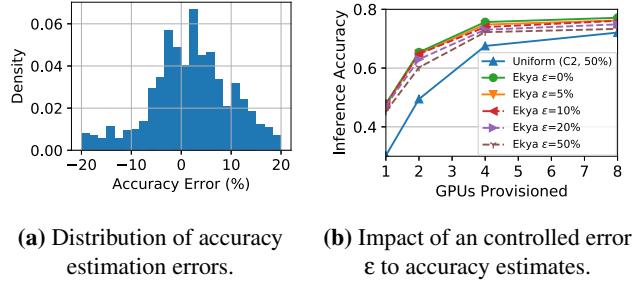
## 6.3 Effectiveness of micro-profiling

The absolute cost of micro-profiling is small; for our experiments, micro-profiling takes 4.4 seconds for a 200s window.

**Errors of microprofiled accuracy estimates:** Ekyा's microprofiler estimates the accuracy of each configuration (§4.3) by training it on a subset of the data for a small number of epochs. To evaluate the micro-profiler's estimates, we run it on all configurations for 5 epochs and on 10% of the retraining data from all streams of the Cityscapes dataset, and calculate the estimation error against the retrained accuracies when trained



**Figure 10:** (a) Component-wise impact of removing dynamic resource allocation (50% allocation) or removing retraining configuration adaptation (fixed Cfg 2). (b) Robustness of Ekyaw to a wide range of retraining window values.



**Figure 11: Evaluation of microprofiling performance.** (a) shows the distribution of microprofiling’s actual estimation errors, and (b) shows the robustness of Ekyaw’s performance against microprofiling’s estimation errors.

on 100% of the data for 5, 15 and 30 epochs. Figure 11a plots the distribution of the errors in accuracy estimation and show that the micro-profiled estimates are largely unbiased with a median absolute error of 5.8%.

**Sensitivity to microprofiling estimation errors:** Finally, we test the impact of accuracy estimation errors (§4.3) on Ekyaw. We add gaussian noise on top of the predicted retraining accuracy when the microprofiler is queried. Figure 11b shows that Ekyaw is robust to accuracy estimate errors: with upto 20% error (which covers all errors in Figure 11a) in the profiler prediction, the maximum accuracy drop is 3%.

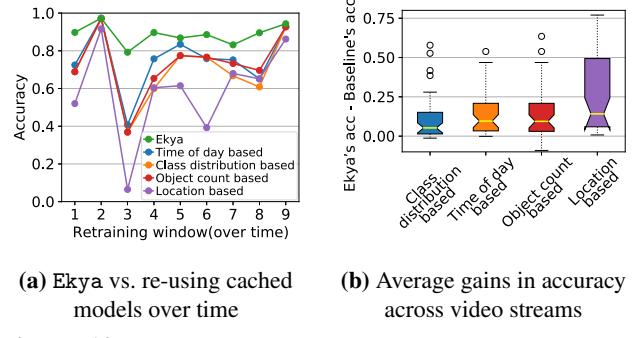
#### 6.4 Comparison with alternative designs

**Ekyaw vs. Cloud-based retraining:** One may upload a sub-sampled video stream to the cloud, retrain the model, and download the model back to the edge [40]. While this solution is not an option for many deployments due to legal and privacy stipulations [11, 87], we still evaluate this option as it lets the edge servers focus on inference. Cloud-based solutions, however, results in lower accuracy due to significant network delays on the constrained networks typical of edges [81].

For example, consider 8 video streams running ResNet18 and a retraining window of 400 seconds. A HD (720p) video stream at 4Mbps and 10% data sub-sampling (typical in our experiments) amounts to 160Mb of training data per camera per window. Uploading 160Mb for each of the 8 cameras over

	Bandwidth (Mbps)	Uplink	Downlink	Acc.	Bandwidth Gap
Cellular	5.1	17.5	68.5%	10.2×	3.8×
Satellite	8.5	15	69.2%	5.9×	4.4×
Cellular (2×)	10.2	35	71.2%	5.1×	1.9×
Ekyaw	-	-	<b>77.8%</b>	-	-

**Table 3: Retraining in the cloud under different networks [58, 65, 81] versus using Ekyaw at the edge.** Ekyaw achieves better accuracy without using expensive satellite and cellular links.



**Figure 12: Ekyaw vs. re-using cached models.** Compared to cached-model selection techniques, models retrained with Ekyaw maintain a consistently high accuracy, since it fully leverages the latest training data and is thus more robust to data-drift.

a 4G uplink (5.1 Mbps [65]) and downloading the trained ResNet18 models (398 Mb each [7]) over the 17.5 Mbps downlink [65] takes 432 seconds (even excluding the model retraining time), which already exceeds the retraining window.

To test on the Cityscapes dataset, we extend our simulator (§5) to account for network delays during retraining, and test with 8 videos and 4 GPUs. We use the conservative assumption that retraining in the cloud is “instantaneous” (cloud GPUs are powerful than edge GPUs). Table 3 lists the accuracies with cellular 4G links (both one and two subscriptions to meet the 400s retraining window) and a satellite link, which are both indicative of edge deployments [81].

For the cloud alternatives to match Ekyaw’s accuracy, we will need to provision additional uplink capacity of 5×-10× and downlink capacity of 2×-4× (of the already expensive links). In summary, Ekyaw’s edge-based solution is better than a cloud alternate for retraining in *both* accuracy and network usage (Ekyaw sends no data out of the edge), all while providing privacy for the videos. However, when the edge-cloud network has sufficient bandwidth, e.g., in an enterprise that is provisioned with a private leased connection, then using the cloud to retrain the models can be a viable design choice.

**Ekyaw vs. Re-using pretrained models:** An alternative to continuous retraining is to cache *pretrained* models and reuse them. We pre-train and cache a few tens of DNNs from earlier windows of the Waymo dataset and test four heuristics for selecting cached models. *Class-distribution*-based selection picks the cached DNN whose training data class distribution has the closest Euclidean distance with the current window’s data. *Time-of-day*-based selection picks the cached

DNN whose training data time matches the current window. *Object-count*-based selection picks the cached DNN based on similar count of objects. *Location*-based selection picks the cached DNNs trained on the same city as the current window.

Figure 12a highlights the advantages of Ekyा over different model selection schemes. We find that since time-of-day-based, object-count-based, and location-based model selection techniques are agnostic to the class distributions of training data of cached models, the selected cached models sometimes do not cover all classes in the current window. Even if we take class distribution into account when picking cached models, there are still substantial discrepancies in the appearances of objects between the current window and the history training data. For instance, object appearance can vary due to pose variations, occlusion or different lighting conditions. In Window 3 (Figure 12a), not only are certain classes underrepresented in the training data, but the lighting conditions are also adverse. Figure 12b presents a box plot of the accuracy difference between Ekyा and model selection schemes, where the edges of the box represent the first and third quartiles, the waist is the median, the whiskers represent the maximum and minimum values and the circles represent any outliers. Ekyा’s continuous retraining of models is robust to scene specific data-drifts and achieves upto 26% higher mean accuracy.

## 7 Limitations and Discussion

**Edge hierarchy with heterogeneous hardware.** While Ekyा allocates GPU resources on a single edge, in practice, deployments typically consist of a *hierarchy* of edge devices [19]. For instance, 5G settings include an on-premise edge cluster, followed by edge compute at cellular towers, and then in the core network of the operator. The compute resources, hardware (e.g., GPUs, Intel VPUs [1], and CPUs) and network bandwidths change along the hierarchy. Thus, Ekyा will have to be extended along two aspects: (a) multi-resource allocation to include both compute and the network in the edge hierarchy; and (b) heterogeneity in edge hardware.

**Privacy of video data.** As explained in §2.1, privacy of videos is important in real-world deployments, and Ekyा’s decision to retrain only on the edge device is well-suited to achieving privacy. However, when we extend Ekyा to a hierarchy of edge clusters, care has to be taken to decide the portions of the retraining that can happen on edge devices that are *not* owned by the enterprise. Balancing the need for privacy with resource efficiency is a subject for future work.

**Generality beyond vision workloads.** Ekyा’s thief scheduler is generally applicable to DNN models since it only requires that the resource-accuracy function be strictly increasing wherein allocation of more resources to training results in increasing accuracy. This property holds true for *most* workloads (vision and language DNNs). However, when this property does *not* hold, further work is needed to prevent Ekyा’s microprofiler from making erroneous estimations and its thief scheduler from making sub-optimal allocations.

## 8 Related Work

**1) ML training systems.** For large scale scheduling of training in the cloud, model and data parallel frameworks [3, 10, 24, 50] and various resource schedulers [30, 31, 56, 69, 95, 97] have been developed. These systems, however, target different objectives than Ekyा, like maximizing parallelism, fairness, or minimizing average job completion. Collaborative training systems [18, 51] work on decentralized data on mobile phones. They focus on coordinating the training between edge and the cloud, and not on training alongside inference.

**2) Video processing systems.** Prior work has built low-cost, high-accuracy and scalable video processing systems for the edge and cloud [22, 32, 37]. VideoStorm investigates quality-lag requirements in video queries [32]. NoScope exploits difference detectors and cascaded models to speedup queries [22]. Focus uses low-cost models to index videos [34]. Chameleon exploits correlations in camera content to amortize *profiling costs* [37]. Reducto [47] and DDS [25] seek to reduce edge-to-cloud traffic by intelligent frame sampling and video encoding. All of these works optimize only the inference accuracy or the system/network costs of DNN inference, unlike Ekyा’s focus on retraining. More recently, LiveNAS[41] deploys continuous retraining to update video upscaling models, but focuses on efficiently allocating client-server bandwidth to different subsamples of a single video stream. Instead, Ekyा focuses on GPU allocation for maximizing retrained accuracy across multiple video streams.

**3) Hyper-parameter optimization.** Efficient exploration of hyper-parameters is crucial in training systems to find the model with the best accuracy. Techniques range from simple grid or random search [17], to more sophisticated approaches using random forests [35], Bayesian optimization [85, 88], probabilistic modelling [71], or non-stochastic infinite-armed bandits [46]. Unlike the focus of these techniques on finding the hyper-parameters with the highest accuracy, our focus is on resource allocation. Further, we are focused on the inference accuracy over the retrained window, where producing the best retrained model often turns out to be sub-optimal.

**4) Continuous learning.** Machine learning literature on continuous learning adapts models as new data comes in. A common approach used is transfer learning [33, 51, 72, 74]. Research has also been conducted on handling catastrophic forgetting [43, 79], using limited amount of training data [73, 89], and dealing with class imbalance [16, 92]. Ekyा builds atop continuous learning techniques for its scheduling and implementation, to enable them in edge deployments.

## 9 Acknowledgements

We thank the NSDI reviewers and our shepherd, Minlan Yu, for their invaluable feedback. This research is partly supported by NSF (CCF-1730628, CNS-1901466), UChicago CERES Center, a Google Faculty Research Award and gifts from Amazon, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

## References

- [1] Azure percept. <https://azure.microsoft.com/en-us/services/azure-percept/>.
- [2] Google ai blog: Custom on-device ml models with learn2compress. <https://ai.googleblog.com/2018/05/custom-on-device-ml-models.html>. (Accessed on 03/09/2021).
- [3] MxNet: a flexible and efficient library for deep learning. <https://mxnet.apache.org/>.
- [4] Nvidia multi-process service. [https://docs.nvidia.com/deploy/pdf/CUDA\\_Multi\\_Process\\_Service\\_Overview.pdf](https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf). (Accessed on 09/16/2020).
- [5] Reducing edge compute cost for live video analytics. <https://techcommunity.microsoft.com/t5/internet-of-things/live-video-analytics-with-microsoft-rocket-for-reducing-edge/b-a-p/1522305>. (Accessed on 03/09/2021).
- [6] scipy.optimize.nnls — scipy v1.5.2 reference guide. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.nnls.html>. (Accessed on 09/17/2020).
- [7] torchvision.models — pytorch 1.6.0 documentation. <https://pytorch.org/docs/stable/torchvision/models.html>. (Accessed on 09/16/2020).
- [8] A Comprehensive List of Hyperparameter Optimization & Tuning Solutions. <https://medium.com/@mikkokotila/a-comprehensive-list-of-hyperparameter-optimization-tuning-solutions-88e067f19d9>, 2018.
- [9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.
- [10] 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.
- [11] Achieving Compliant Data Residency and Security with Azure.
- [12] AI and Compute. <https://openai.com/blog/ai-and-compute/>, 2018.
- [13] G. Ananthanarayanan, V. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. R. Sivalingam, and S. Sinha. Real-time Video Analytics – the killer app for edge computing. *IEEE Computer*, 2017.
- [14] AWS Outposts. <https://aws.amazon.com/outposts/>.
- [15] Azure Stack Edge. <https://azure.microsoft.com/en-us/services/databox/edge/>.
- [16] E. Belouadah and A. Popescu. IL2M: Class Incremental Learning With Dual Memory. In *IEEE ICCV*, 2019.
- [17] J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012.
- [18] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roslander. Towards Federated Learning at Scale: System Design. In *SysML*, 2019.
- [19] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *ACM/IEEE SEC*, 2018.
- [20] CLIFFORD, M. J., PERRONS, R. K., ALI, S. H., AND GRICE, T. A. Extracting Innovations: Mining, Energy, and Technological Change in the Digital Age. In *CRC Press*, 2018.
- [21] cnn-benchmarks. <https://github.com/jcjohnson/cnn-benchmarks#resnet-101>, 2017.
- [22] D. Kang, J. Emmons, F. Abuzaid, P. Bailis and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. In *VLDB*, 2017.
- [23] D Maltoni, V Lomonaco. Continuous learning in single-incremental-task scenarios. In *Neural Networks*, 2019.
- [24] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NeurIPS*, 2012.
- [25] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.
- [26] Edge Computing at Chick-fil-A. <https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2>. 2019.
- [27] Ganesh Ananthanarayanan, Victor Bahl, Yuanchao Shu, Franz Loewenherz, Daniel Lai, Darcy Akers, Peiwei Cao, Fan Xia, Jiangbo Zhang, Ashley Song. Traffic Video Analytics – Case Study Report. 2019.

- [28] GI Parisi, R Kemker, JL Part, C Kanan, S Wermter . Continual lifelong learning with neural networks: A review. In *Neural Networks*, 2019.
- [29] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1487–1495, New York, NY, USA, 2017. Association for Computing Machinery.
- [30] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM*, 2014.
- [31] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *USENIX NSDI*, 2019.
- [32] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodík, Matthai Philipose, Victor Bahl, Michael Freedman. Live video analytics at scale with approximation and delay-tolerance. In *USENIX NSDI*, 2017.
- [33] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. In *NeurIPS Deep Learning and Representation Learning Workshop*, 2015.
- [34] K. Hsieh, G. Ananthanarayanan, P. Bodík, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *USENIX OSDI*, 2018.
- [35] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, 2011.
- [36] Joseph Redmon, Ali Farhadi . Yolo9000: Better, faster, stronger. In *CVPR*, 2017.
- [37] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodík, Siddhartha Sen, Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *ACM SIGCOMM*, 2018.
- [38] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyer, Pillai Padmanabhan, Mahadev Satyanarayanan. Towards scalable edge-native applications. In *ACM/IEEE Symposium on Edge Computing*, 2019.
- [39] K He, X Zhang, S Ren, J Sun . Deep residual learning for image recognition. In *CVPR*, 2016.
- [40] M. Khani, P. Hamadanian, A. Nasr-Esfahany, and M. Alizadeh. Real-time video inference on edge devices via adaptive model streaming. *arXiv preprint arXiv:2006.06628*, 2020.
- [41] J. Kim, Y. Jung, H. Yeo, J. Ye, and D. Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 107–125, New York, NY, USA, 2020. Association for Computing Machinery.
- [42] Konstantin Shmelkov, Cordelia Schmid, Kartek Alahari . Incremental learning of object detectors without catastrophic forgetting. In *ICCV*, 2017.
- [43] J. Lee, J. Yoon, E. Yang, and S. J. Hwang. Lifelong Learning with Dynamically Expandable Networks. In *ICLR*, 2018.
- [44] A. Li, O. Spyra, S. Perel, V. Dalibard, M. Jaderberg, C. Gu, D. Budden, T. Harley, and P. Gupta. A generalized framework for population based training. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 1791–1799, New York, NY, USA, 2019. Association for Computing Machinery.
- [45] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, Jan. 2017.
- [46] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.
- [47] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020.
- [48] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 61–73, New York, NY, USA, 2019. Association for Computing Machinery.
- [49] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016.

- [50] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
- [51] Y. Lu, Y. Shu, X. Tan, Y. Liu, M. Zhou, Q. Chen, and D. Pei. Collaborative learning between cloud and end devices: an empirical study on location prediction. In *ACM/IEEE SEC*, 2019.
- [52] M McCloskey, NJ Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, 1989.
- [53] M Sandler, A Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen . Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.
- [54] M. J. Magazine and M. Chern. A note on approximation schemes for multidimensional knapsack problems. *Math. Oper. Res.*, 9(2), 1984.
- [55] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, Feb. 2020. USENIX Association.
- [56] K. Mahajan, A. Singhvi, A. Balasubramanian, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. In *USENIX NSDI*, 2020.
- [57] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele . The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016.
- [58] Measuring Fixed Broadband - Eighth Report, FEDERAL COMMUNICATIONS COMMISSION OFFICE OF ENGINEERING AND TECHNOLOGY. <https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eighth-report>. 2018.
- [59] Microsoft-Rocket-Video-Analytics-Platform. <https://github.com/microsoft/Microsoft-Rocket-Video-Analytics-Platform>.
- [60] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.
- [61] Mingxing Tan, Quoc V. Le . Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.
- [62] U. Misra, R. Liaw, L. Dunlap, R. Bhardwaj, K. Kanadasamy, J. E. Gonzalez, I. Stoica, and A. Tumanov. *RubberBand: Cloud-Based Hyperparameter Tuning*, page 327–342. Association for Computing Machinery, New York, NY, USA, 2021.
- [63] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI’18*, page 561–577, USA, 2018. USENIX Association.
- [64] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun . Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.
- [65] OPENSIGNAL. Mobile Network Experience Report . <https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience>. 2019.
- [66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [67] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *ICLR*, 2017.
- [68] Pei Sun and Henrik Kretzschmar and Xerxes Dotiwalla and Aurelien Chouard and Vijaysai Patnaik and Paul Tsui and James Guo and Yin Zhou and Yuning Chai and Benjamin Caine and Vijay Vasudevan and Wei Han and Jiquan Ngiam and Hang Zhao and Aleksei Timofeev and Scott Ettinger and Maxim Krivokon and Amy Gao and Aditya Joshi and Yu Zhang and Jonathon Shlens and Zhifeng Chen and Dragomir Anguelov. Scalability in perception for autonomous driving: Waymo open dataset, 2019.
- [69] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *ACM EuroSys*, 2018.

- [70] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.
- [71] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca. HyperDrive: exploring hyperparameters with POP scheduling. In *ACM/IFIP/USENIX Middleware*, 2017.
- [72] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, Kayvon Fatahalian. Online model distillation for efficient video inference. In *ICCV*, 2019.
- [73] S. V. Ravuri and O. Vinyals. Classification accuracy score for conditional generative models. 2019.
- [74] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. In *IEEE CVPR Workshop*, 2014.
- [75] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018.
- [76] Residential landline and fixed broadband services . [https://www.ofcom.org.uk/\\_\\_data/assets/pdf\\_file/0015/113640/landline\\_broadband.pdf](https://www.ofcom.org.uk/__data/assets/pdf_file/0015/113640/landline_broadband.pdf). 2019.
- [77] RM French. Catastrophic forgetting in connectionist networks. In *Trends in cognitive sciences*, 1999.
- [78] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5), 1952.
- [79] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *AAAI*, 2018.
- [80] H. F. Scheduler. <https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>.
- [81] Shadi Noghabi, Landon Cox, Sharad Agarwal, Ganesh Ananthanarayanan. The emerging landscape of edge-computing. In *ACM SIGMOBILE GetMobile*, 2020.
- [82] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *CVPR*, 2017.
- [83] Shivangi Srivastava, Maxim Berman, Matthew B. Blaschko, Devis Tuia . Adaptive compression-based lifelong learning. In *BMVC*, 2019.
- [84] Si Young Jang, Yoonhyung Lee, Byoungheon Shin, Dongman Lee, Dionisio Vendrell Jacinto . Application-aware iot camera virtualization for video analytics edge computing. In *ACM/IEEE SEC*, 2018.
- [85] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.
- [86] Song Han, Huizi Mao, William J. Dally . Accelerating very deep convolutional networks for classification and detection. In *ICLR*, 2017.
- [87] Sweden Data Collection & Processing.
- [88] K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, and R. P. Adams. Scalable Bayesian Optimization Using Deep Neural Networks. In *ICML*, 2015.
- [89] Sylvestre-Alvise Rebiffé, Alexander Kolesnikov, Georg Sperl, Christoph H. Lampert. icarl: Incremental classifier and representation learning. In *CVPR*, 2017.
- [90] The Future of Computing is Distributed. <https://www.datanami.com/2020/02/26/the-future-of-computing-is-distributed/>, 2020.
- [91] H. Wang, A. Kembhavi, A. Farhadi, A. L. Yuille, and M. Rastegari. Elastic: Improving cnns with dynamic scaling policies. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2258–2267, 2019.
- [92] Y. Wu, Y. Chen, L. Wang, Y. Ye, Z. Liu, Y. Guo, and Y. Fu. Large scale incremental learning. In *IEEE CVPR*, 2019.
- [93] Xi Yin, Xiang Yu, Kihyuk Sohn, Xiaoming Liu and Manmohan Chandraker. Feature transfer learning for face recognition with under-represented data. In *IEEE CVPR*, 2019.
- [94] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *IEEE PAMI*, 2016.
- [95] 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 *USENIX OSDI*, 2018.
- [96] Z. Li and D. Hoiem . Learning without forgetting. In *ECCV*, 2016.
- [97] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*, 2017.
- [98] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

Notation	Description
$\mathcal{V}$	Set of video streams
$v$	A video stream ( $v \in \mathcal{V}$ )
$T$	A retraining window with duration $\ T\ $
$\Gamma$	Set of all retraining configurations
$\gamma$	A retraining configuration ( $\gamma \in \Gamma$ )
$\Lambda$	Set of all inference configurations
$\lambda$	An inference configuration ( $\lambda \in \Lambda$ )
$G$	Total number of GPUs
$\delta$	The unit for GPU resource allocation
$A_T(v, \gamma, \lambda, \mathcal{R}, I)$	Inference accuracy for video $v$ for given configurations and allocations
$C_T(v, \gamma, \lambda)$	Compute cost in GPU-time for video $v$ for given configurations and allocations
$\phi_{v\gamma\lambda\mathcal{R}I}$	A set of binary variables ( $\phi_{v\gamma\lambda\mathcal{R}I} \in \{0, 1\}$ ). $\phi_{v\gamma\lambda\mathcal{R}I} = 1$ iff we use retraining config $\gamma$ , inference config $\lambda$ , $\mathcal{R}\delta$ GPUs for retraining, $I\delta$ GPUs for inference for video $v$

**Table 4: Notations used in Ekyा’s description.**

- [99] Ziwei Liu, Zhongqi Miao, Xiaohang Zhan, Jiayun Wang, Boqing Gong, Stella X. Yu . Large-scale long-tailed recognition in an open world. In *CVPR*, 2019.

## A Thief Scheduler

### A.1 Complexity Analysis.

Assuming all the  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  values are known, the above optimization problem can be reduced to a multi-dimensional binary knapsack problem, a NP-hard problem [54]. Specifically, the optimization problem is to pick binary options ( $\phi_{v\gamma\lambda\mathcal{R}I}$ ) to maximize overall accuracy while satisfying two capacity constraints (the first and second constraints in Eq 1). In practice, however, getting all the  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  is *infeasible* because this requires training the edge DNN using all retraining configurations and running inference using all the retrained DNNs with all possible GPU allocations and inference configurations.

The uncertainty of  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  resembles the multi-armed bandits (MAB) problem [78] to maximize the expected

rewards given a limited number of trials for a set of options. Our optimization problem is more challenging than MAB for two reasons. First, unlike the MAB problem, the cost of trials ( $C_T(v, \gamma, \lambda)$ ) varies significantly, and the optimal solution may need to choose cheaper yet less rewarding options to maximize the overall accuracy. Second, getting the reward  $A_T(v, \gamma, \lambda, \mathcal{R}, I)$  after each trial requires "ground truth" labels that are obtained using the large golden model, which can only be used judiciously on resource-scarce edges (§2.2).

In summary, our optimization problem is computationally more complex than two fundamentally challenging problems (multi-dimensional knapsack and multi-armed bandits).

---

### Algorithm 2: PickConfigs

---

```

Data: Resource allocations in temp_alloc[], configurations
       ( $\Gamma$  and  $\Lambda$ ), retraining window  $T$ , videos  $V$ 
Result: Chosen configs  $\forall v \in V$ , average accuracy over  $T$ 
1 chosen_accuracies[]  $\leftarrow \{\}$ ; chosen_configs[]  $\leftarrow \{\}$ ;
2 for  $v$  in  $V[]$  do
3   infer_config_pool[] =  $\Lambda$ .where(resource_cost <
      temp_alloc[v.inference_job] && accuracy  $\geq a_{MIN}$ );
4   infer_config = max(infer_config_pool, key=accuracy);
5   best_accuracy = 0;
6   for train_config in  $\Gamma$  do
7     /* Estimate accuracy of inference/training
        config pair over retraining window */
8     accuracy = EstimateAccuracy(train_config,
       infer_config, temp_alloc[v.training_job], T);
9     if accuracy > best_accuracy then
10       best_accuracy = accuracy;
11       best_train_config = train_config;
12   chosen_accuracies[v] = best_accuracy;
13   chosen_configs[v] = {infer_config, best_train_config};
14 return chosen_configs[], mean(chosen_accuracies[]);

```

---



# YuZu: Neural-Enhanced Volumetric Video Streaming

Anlan Zhang<sup>1</sup> Chendong Wang<sup>1\*</sup> Bo Han<sup>2</sup> Feng Qian<sup>1</sup>  
<sup>1</sup>University of Minnesota, Twin Cities <sup>2</sup>George Mason University

## Abstract

Differing from traditional 2D videos, volumetric videos provide true 3D immersive viewing experiences and allow viewers to exercise six degree-of-freedom (6DoF) motion. However, streaming high-quality volumetric videos over the Internet is extremely bandwidth-consuming. In this paper, we propose to leverage 3D super resolution (SR) to drastically increase the visual quality of volumetric video streaming. To accomplish this goal, we conduct deep intra- and inter-frame optimizations for off-the-shelf 3D SR models, and achieve up to 542× speedup on SR inference without accuracy degradation. We also derive a first Quality of Experience (QoE) model for SR-enhanced volumetric video streaming, and validate it through extensive user studies involving 1,446 subjects, achieving a median QoE estimation error of 12.49%. We then integrate the above components, together with important features such as QoE-driven network/compute resource adaptation, into a holistic system called YuZu that performs line-rate (at 30+ FPS) adaptive SR for volumetric video streaming. Our evaluations show that YuZu can boost the QoE of volumetric video streaming by 37% to 178% compared to no SR, and outperform existing viewport-adaptive solutions by 101% to 175% on QoE.

## 1 Introduction

Volumetric video is an emerging type of multimedia content. Unlike traditional videos and 360° panoramic videos [28, 53] that are 2D, every frame in a volumetric video consists of a 3D scene represented by a point cloud or a polygon mesh. The 3D nature of volumetric video enables viewers to exercise six degree-of-freedom (6DoF) movement: a viewer can not only “look around” by changing the yaw, pitch, and roll of the viewing direction, but also “walk” in the video by changing the translational position in 3D space. This leads to a truly immersive viewing experience. As the key technology of realizing telepresence [49], volumetric video has registered numerous applications. They can be viewed in multiple ways: through VR/MR (virtual/mixed reality) headsets or directly on PCs (similar to how we play 3D games).

Despite the potentials, streaming volumetric videos over the Internet faces a key challenge of high bandwidth consumption. High-quality volumetric content requires hundreds of Mbps bandwidth [27, 71]. To improve the Quality of Experience

(QoE) under limited bandwidth, prior work has mostly focused on viewport-adaptive streaming (*i.e.*, mainly streaming content that will appear in the viewport) [27, 41, 50]. However, they are ineffective when the entire scene falls inside the viewport. They also require 6DoF motion prediction that is unlikely to be accurate for fast motion. Some other proposals explored remote rendering [26, 52] (*e.g.*, having an edge node transcode 3D scenes into regular 2D frames). However, they require not only 6DoF motion prediction, but also edge/cloud-side transcoding that is difficult to scale, as summarized in Table 1.

In this paper, we employ a different and orthogonal approach toward improving the QoE of volumetric video streaming through *3D super resolution* (3D SR). SR was initially designed for improving the visual quality of 2D images [21, 65]. Recently, researchers in the computer vision community developed SR models for point clouds [43, 61, 63, 70]. This inspires us to employ SR for volumetric video streaming, as each frame of a volumetric video is typically either a point cloud or a 3D mesh.<sup>1</sup> Although there have been recent successful attempts on applying SR to 2D video streaming [22, 39, 68], 3D-SR-enhanced volumetric video streaming is unique and challenging due to the following reasons.

- There is a fundamental difference between *pixel-based* 2D frames and volumetric frames consisting of *unstructured 3D points*, making processing volumetric videos (even without SR) vastly different from 2D videos.
- Due to its 3D nature, the computation overhead of 3D SR is very high. We apply off-the-shelf 3D SR models to volumetric videos [1], and find that the runtime performance of 3D SR is unacceptably poor – achieving only ~0.1 frames per second (FPS) on a PC with a powerful GPU. In contrast, 2D SR can achieve line-rate upsampling by simply downscaling the model [68], but we find that only doing model downscaling is far from being adequate for line-rate 3D SR (*i.e.*, at 30+ FPS).
- Given its recent debut, there lacks research on basic infrastructures such as tools and models supporting volumetric video streaming. For example, there is no QoE model for volumetric videos that can guide bitrate adaptation or critical SR parameter selection; the wide range of factors affecting the QoE make constructing such a model quite challenging.
- There are other practical challenges to overcome, such as a lack of color produced by today’s 3D SR models.

To address the above challenges, we begin by developing to

<sup>1</sup>We focus on point-cloud-based volumetric videos in this work, but the key concepts of YuZu also apply to mesh-based volumetric videos.

\* Current affiliation: University of Wisconsin, Madison.

Schemes	Refs	Advantages ( $\oplus$ ) and Disadvantages ( $\ominus$ )
Direct Streaming	N/A	$\oplus$ Easy to implement, best QoE (if bandwidth is sufficient). $\ominus$ Highest network bandwidth (BW) usage.
Direct + VA	[27, 41]	$\oplus$ Lower BW usage. $\ominus$ BW saving depends on user's motion, QoE depends on motion prediction.
Direct + SR	YuZu	$\oplus$ Good QoE, further lower BW usage, adaptively trades compute resource for BW. $\ominus$ Requires training.
Remote Rendering	[26, 52]	$\oplus$ Lowest BW usage. $\ominus$ QoE depends on motion prediction, need edge support (poor scalability).

Table 1: Four categories of volumetric video streaming approaches (VA = Viewport Adaptation; SR = Super Resolution).

our knowledge a first QoE model for assessing SR-enhanced volumetric video streaming. The model takes into account a variety of factors that may affect the QoE, such as video resolution (*i.e.*, point density)<sup>2</sup>, viewing distance, upsampling ratio, SR-incurred distortion, and QoE metrics from traditional video streaming. We validate our model by conducting two IRB-approved user studies involving 1,446 voluntary participants from 40 countries, using a major genre of volumetric content, *i.e.*, portraits of single/multiple people. The validation results confirm its accuracy, with a median QoE estimation error of 12.49%. Our user studies offer definitive evidence that 3D SR can significantly boost the QoE of volumetric video streaming.

Next, we design, implement, and evaluate YuZu, which is to our knowledge a first SR-enhanced volumetric video streaming system. At its core, YuZu deeply optimizes the end-to-end upsampling pipeline in three aspects: *intra-frame SR*, *inter-frame SR*, and *network-compute resource management*, whose synergy helps drastically improve the runtime performance of SR while retaining the inference accuracy.

For **intra-frame SR**, our approaches are not limited to generic optimizations for deep learning models such as modifying SR models' structures for fast-paced SR. More importantly, we consider the factors that are unique to 3D SR and its data representation: we design a mechanism that leverages the low-resolution content (*i.e.*, the input to the SR model, which is typically discarded after being fed into the model) to reduce the SR model complexity; we also trim the pre-processing and post-processing stages of 3D SR and tailor them to volumetric video streaming. Note that these optimizations are generic, applicable to all the 3D SR models we have investigated [43, 61, 63, 70].

For **inter-frame SR**, YuZu speeds up SR by caching and reusing 3D SR results across consecutive frames. Realizing that none of the 2D inter-frame encoding techniques can be directly applied to volumetric videos, we design an effective inter-frame content reference scheme for SR-enhanced point cloud streams, followed by robust criteria determining whether SR results can be reused between two frames. We then extend reusing SR results from two to multiple consecutive frames through a dynamic-programming-based optimization. The synergy of the above intra- and inter-frame acceleration schemes fills the huge gap between off-the-shelf 3D SR models' performance and what is required for line-rate upsampling of point cloud streams.

YuZu further performs **network-compute resource man-**

**agement** through making judicious decisions about the quality level of the to-be-fetched content and its upsampling ratio. These two decision dimensions are subject to the dynamic network bandwidth and limited compute resources, respectively, which need to be jointly considered given their complex trade-offs – a unique challenge compared to traditional adaptive bitrate (ABR) video streaming. YuZu takes a QoE-driven approach by maximizing the utility function derived from our QoE model. To solve the underlying optimization problem in real time, we develop a hybrid, two-stage algorithm that employs coarse-grained and fine-grained search at different time to efficiently find a good approximate solution. In addition, YuZu performs fast colorization of SR results through efficient nearest point search.

We implement the above components and integrate them into YuZu in 10,848 lines of code. Our extensive evaluations indicate that YuZu can achieve line-rate, adaptive, high-quality 3D SR. We highlight key evaluation results as follows.

- Our user study suggests that 3D SR can boost the volumetric video QoE by 37% to 178% compared to no SR.
- Our optimizations speed up 3D SR by 140× to 542× and reduce GPU memory usage by 68% to 90% with no accuracy degradation, compared to the vanilla SR models [43, 61].
- Compared to a recently proposed viewport-adaptive volumetric video streaming system [27], YuZu improves the QoE by 100.6% to 174.9%.

To summarize, we make the following contributions.

- We build an empirical QoE model for SR-enhanced volumetric videos, and validate it through large-scale user studies involving 1,446 participants. We build our models using volumetric content of single/multiple human portraits, a major application of volumetric video streaming. Note that the model can be applied to non-SR volumetric videos belonging to the same genre, with an SR ratio of 1.
- We propose and design YuZu, an SR-enhanced, QoE-aware volumetric video streaming system.
- We implement YuZu, and conduct extensive evaluations for its QoE improvement and runtime performance.

## 2 Background and Motivation

Recently, the computer vision community extended SR to *static* point clouds [43, 61, 63, 70]. When applied to a video  $v$ , SR trains offline a deep neural network (DNN) model  $M$  that *upsamples* low-resolution frames  $L(v)$  to high-resolution ones  $H(v)$ , using the original (high-resolution) frames  $F(v)$  for training. In the online inference, the server sends  $M$  and  $L(v)$  to the client, which infers  $H(v) = M(L(v))$ . SR leverages the overfitting property of DNN to ensure that  $H(v)$  is highly

<sup>2</sup>The resolution of a point cloud is defined as its point density; the resolution of a volumetric video is the avg. resolution of its point cloud frames.

similar to  $F(v)$ . It achieves bandwidth reduction (or QoE improvement when bandwidth remains the same) since the combined size of  $M$  and  $L(v)$  is much smaller than  $F(v)$ .

We start with a straightforward approach: applying PU-GAN [43], a state-of-the-art 3D SR model, to upsample every point cloud frame of a volumetric video. PU-GAN operates by dividing the entire point cloud of a frame into smaller *patches*, each consisting of a subset of points. Both SR training and inference are performed on a per-patch (as opposed to a per-frame) basis, *i.e.*, each patch is upsampled individually. Its DNN model is based on a generative adversarial network (GAN) and realizes three key stages: feature extraction, feature expansion, and point set generation.

We next describe a case study using PU-GAN to motivate YuZu. Our testing video was captured by three depth cameras. It has 3,622 frames, each consisting of ~100K points depicting a performing actor. We use all its frames to train a PU-GAN model. We set the SR ratio (*i.e.*, upsampling ratio) to 4, making the input and output point clouds consist of roughly 25K and 100K points, respectively. We have both positive and negative findings from this case study. On the positive side, the model can accurately reconstruct each individual frame, *i.e.*, each upsampled point cloud is highly similar to the original one in terms of the geometric structure, as quantified by the Earth Mover’s Distance (EMD [54]):

$$\mathcal{L}_{EMD}(I, G) = \min_{\phi: I \rightarrow G} \frac{1}{|I|} \sum_{x \in I} \|x - \phi(x)\|_2 \quad (1)$$

where  $I$  and  $G$  are the upsampled point cloud and the ground truth, respectively;  $\phi: I \rightarrow G$  is a bijection from the points in  $I$  to those in  $G$ . The average EMD value across all frames is 1.47cm, which confirms good upsampling accuracy [43]; it is also verified by our IRB-approved user studies (§4.2). Also encouragingly, we find that SR indeed achieves significant bandwidth savings. For this 2-minute video, the compressed sizes of  $F(v)$ ,  $M$ , and  $L(v)$  are 1.40 GB, 560 KB, and 0.36 GB, respectively, leading to a bandwidth reduction of 74.2%.

Despite the above encouraging results, we notice three major issues from the above case study.

- **A Lack of Quality-of-Experience (QoE) Model.** For traditional 2D video streaming, there exist numerous studies on modeling the viewer’s QoE [15, 18, 69]. In contrast, volumetric videos are still in their infancy. There is a lack of generic QoE models that researchers can leverage, not to mention a lack of understanding of how SR impacts QoE.

- **Unacceptably Poor Runtime Performance.** 3D SR models are computationally much more heavyweight than 2D SR models. When applying PU-GAN to the above video, the runtime performance is extremely poor. On a machine with an NVIDIA 2080Ti GPU, the upsampling FPS is only 0.1, far below the desired FPS of at least 30. Besides, the GPU memory usage of PU-GAN is 7GB (out of the 11GB available memory of 2080Ti). This is one reason why all the off-the-shelf 3D SR models operate on a per-patch basis, as this saves memory compared to processing a full frame.

- **No Color Support.** We find that no existing 3D SR model can restore the color information of upsampled point cloud.

Note that the last two limitations are common in that they also apply to all other 3D SR models for point clouds that we have examined, such as MPU [61] and PU-Net [70].

### 3 YuZu Overview

YuZu is to our knowledge the first SR-enhanced volumetric video streaming system. It streams video-on-demand volumetric content stored on an Internet server to client hosts. On the server side, the volumetric video is divided into *chunks* each consisting of a fixed number of frames (*i.e.*, point clouds encoded by schemes such as Octree [34, 46] and k-d tree [35, 44]). Each chunk is encoded into multiple versions with different resolutions (*i.e.*, point densities). The SR model training and volumetric content preprocessing (*e.g.*, patch reuse computation, see §5.2) are performed offline on the server side. Similar to a typical DASH server, the YuZu server is stateless (and thus scalable), and all the streaming logic runs on the client side. As shown in Figure 1, the client fetches from the server the video chunks, which can possibly be at a low resolution. Since 3D SR models typically operate on a per-patch basis, the client segments each frame into patches, upsamples them through 3D SR, efficiently colors them (§5.4), and renders them to the viewer.

To achieve line rate SR, YuZu employs novel optimizations tailored to SR-enhanced volumetric video streaming. Regarding *intra-frame optimizations*, off-the-shelf 3D SR models are strategically adapted; low-resolution patches before SR are properly leveraged instead of being discarded; and the patch generation is accelerated (§5.1). For *inter-frame optimizations*, previous SR results are judiciously reused (§5.2).

A crucial decision that YuZu must make is to determine what resolution (quality level) to fetch for each chunk, as well as which SR ratio to apply for upsampling each patch, subject to the resource constraints jointly imposed by the network and computation. YuZu addresses this through a principled, efficient, and QoE-driven discrete optimization framework (§5.3). The framework utilizes a first-of-its-kind QoE model that we derive from ratings of 1,446 real users (§4).

## 4 QoE Model for Volumetric Videos

For SR-enhanced volumetric video streaming, its QoE is affected by a wide range of factors. The large space formed by these factors and their interplay make constructing QoE models much more challenging than conventional videos.

### 4.1 An Empirical QoE Model

We first enumerate factors that may affect the QoE for SR-enhanced volumetric video streaming. They are derived based on the domain knowledge of SR and our communication with other volumetric video viewers.

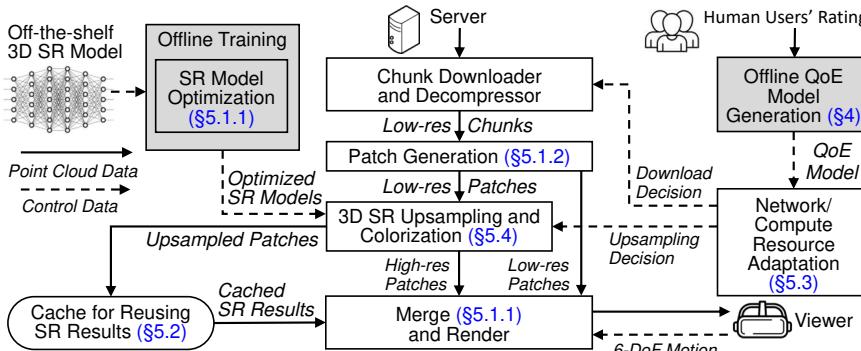


Figure 1: The system architecture of YuZu.

- **Point Density.** Similar to 2D image resolution, a 3D object with a higher point density (resolution) contains more details and thus offers a better QoE.
- **Viewing Distance.** As the viewing distance increases, a rendered 3D object becomes smaller in the displayed view, and is thus less sensitive to quality degradation.
- **SR Ratio and Distortion.** A higher SR ratio leads to a higher point density (and thus more QoE gain), but also potentially higher distortions (and thus more QoE loss).
- **Artifacts caused by Patches.** As described in §2, a typical 3D SR model operates by upsampling individual point subsets called patches. If patches within a frame have non-uniform qualities (caused by different SR ratios), the perceived QoE will be affected.
- **Invisibility due to Finite Viewport and Occlusion.** Due to the 3D nature of volumetric videos, a viewer can see only content that is inside the viewport and not occluded. Outside-viewport or occluded content brings no impact on the QoE.
- **QoE Metrics for Regular Video Streaming.** They include factors such as stall and inter-frame quality switches [69].

Next, we develop an empirical QoE model that considers the above factors. Since SR is performed on a per-patch basis, we first model the QoE for each individual patch as:

$$q_{i,j} = g(d_{i,j}, r_{i,j}, \delta_{i,j}) - h(EMD, \delta_{i,j}) \quad (2)$$

where  $q_{i,j}$  is the quality of patch  $j$  in frame  $i$ ;  $d_{i,j}$  is the patch's original point density before SR;  $\delta_{i,j}$  is the viewing distance to the patch;  $r_{i,j}$  is the SR ratio of the patch. Eq. 2 has two terms:  $g(\cdot)$  considers the patch's perceived density after SR, and  $h(\cdot)$  accounts for the QoE penalty incurred by SR distortion, quantified by the viewing distance and the EMD (Eq. 1) between the upsampled patch and the high-quality patch (ground truth). We empirically define  $g(\cdot)$  and  $h(\cdot)$  as:

$$g(d_{i,j}, r_{i,j}, \delta_{i,j}) = w_1(\delta_{i,j}) \times d_{i,j} \times r_{i,j} \quad (3)$$

$$h(EMD, \delta_{i,j}) = w_2(\delta_{i,j}) \times EMD \quad (4)$$

where  $w_1(\delta_{i,j})$  and  $w_2(\delta_{i,j})$  are weights parameterized on  $\delta_{i,j}$ . Intuitively, in Eq. 3, after SR, the perceived point density improves by a factor of  $r_{i,j}$ ; the QoE gain brought by a higher point density after SR (Eq. 3) and the QoE penalty caused by SR distortion (Eq. 4) depend on the viewing distance.

Age	18-25: 21.8%, 26-30: 29.0%, 31-35: 20.4%, 35+: 28.8%
Gender	Male: 60.3%, Female: 39.2%, Other: 0.5%
Country (40 Total)	US: 55.0%, IN: 28.1%, BR: 5.0%, IT: 2.7%, UK: 1.2%, DE: 1.0%, CA: 0.9%, Other: 6.1%
Education	Bachelor: 59.1%, Master: 23.8%, Other: 17.1%

Table 2: Demographics of the 1,446 subjects in our user studies.

Now given a single frame  $i$ , we define its quality  $Q_i$  as the average of all its visible patches' quality values:

$$Q_i = \frac{\sum_j v_{i,j} q_{i,j}}{\sum_j v_{i,j}} \quad (5)$$

where  $v_{i,j} \in \{0, 1\}$  is 1 iff the patch is visible, i.e., it falls inside the viewport and is not occluded by other patches. To account for the artifacts caused by patches, we define *inter-patch quality switch*  $I_i^{patch}$  as the quality variation across the visible patches within frame  $i$ . To account for inter-frame quality switches, we define *inter-frame quality switch*  $I_i^{frame}$  as the quality change from frame  $i-1$  to frame  $i$ :

$$I_i^{patch} = \text{StdDev}(\{q_{i,j} | \forall j, v_{i,j} > 0\}) \quad (6)$$

$$I_i^{frame} = \|Q_i - Q_{i-1}\| \quad (7)$$

For a volumetric video playback, a possible way to model its overall QoE is a linear combination of  $Q_i$ ,  $I_i^{patch}$ ,  $I_i^{frame}$ , and  $I_i^{stall}$  (the stall of frame  $i$ ). We choose a linear form that is widely used in 2D Internet videos [69]. Thus, we have

$$QoE = \sum_i Q_i - \sum_i \mu_p(\delta_i) I_i^{patch} - \sum_i \mu_f(\delta_i) I_i^{frame} - \sum_i \mu_s(\delta_i) I_i^{stall} \quad (8)$$

Note that depending on the viewing distance, the weights  $\mu_p$ ,  $\mu_f$ , and  $\mu_s$  may differ (e.g., viewers may be more sensitive to stalls when watching a scene at a closer distance), so we parameterize the weights with the viewing distance. In Eq. 8,  $\delta_i$  summarizes the viewing distances to all the patches in frame  $i$ . We empirically choose  $\delta_i = (\sum_j v_{i,j} \delta_{i,j}) / (\sum_j v_{i,j})$ . Also note that the above model is generic and applicable to non-SR-enhanced and non-patch-based volumetric videos as it encompasses special cases without using SR ( $r_{i,j}=1$ ) or patches ( $I_i^{patch}=0$ ).

## 4.2 Model Validation through User Studies

We next conduct user studies with two purposes: validating our QoE model and deriving the model parameters. Our QoE model considers many factors as described in §4.1. The high-level approach of the user study is to let participants subjectively rate the QoE for all the combinations of the above factors' different degrees of impairments, and then use the

Scheme	$1 \times 1$	$1 \times 2$	$1 \times 3$	$1 \times 4$	$2 \times 1$	$2 \times 2$	$3 \times 1$	$4 \times 1$
Pt. density	25%	25%	25%	25%	50%	50%	75%	100%
SR ratio	-	$\times 2$	$\times 3$	$\times 4$	-	$\times 2$	-	-

Table 3: 8 impaired versions (except  $4 \times 1$ ) of a video segment. In scheme  $m \times n$ ,  $m$  is the point density level and  $n$  is SR ratio.

Videos: { <i>Long Dress</i> , <i>Loot</i> [1]; <i>Band</i> , <i>Haggle</i> [36]}
Avg. frame quality $Q_i$ : 7 values uniformly selected from Table 3
Avg. distance $dist_{i,j}$ : {1m, 2m, 3m, 4m}
Avg. inter-patch switch $I_i^{patch}$ : {0.00, 0.45, 0.90}
Avg. inter-frame switch $I_i^{frame}$ : {0.00, 0.45, 0.90}
Avg. stall $I_i^{stall}$ : {0.00, 0.01, 0.03}

Table 4: The factors and their values selected for model validation. subjects’ ratings to train/validate our QoE model. We obtained IRB approvals for our studies. Instead of performing in-person studies, we conduct both studies online by letting users watch pre-generated videos capturing the rendered viewports (with impairments). We take this approach because: (1) it allows vastly scaling up the study, (2) it helps get diverse users worldwide, and (3) the IRB forbids in-person user studies during COVID-19. We have collected responses from 1,446 subjects, whose demographics are shown in Table 2.

We start by studying the QoE gain brought by SR. We have collected 512 subjects’ responses with a total number of 57,344 ratings. The key finding is that SR can effectively boost the QoE. For example, at 1m, compared to  $1 \times 1$ , the (user-rated) QoE increases by 37%, 75%, 150% for  $1 \times 2$ ,  $1 \times 3$ , and  $1 \times 4$ , respectively;  $2 \times 2$  improves the QoE by 178% compared to  $2 \times 1$ . The details can be found in Appendix A.

Next, we validate the overall QoE model (Eq. 8). We choose four videos: *Long Dress* showing a dancing female, *Loot* showing a speaking male, *Band* showing three people playing instruments, and *Haggle* showing three people debating. *Long Dress* and *Loot* are obtained from the 8i dataset [1], each consisting of 800K points per frame for 10 seconds. *Band* and *Haggle* are from the CMU Panoptic dataset [36], each consisting of 300K and 100K points per frame, respectively; we select 10-second segments for our study. For each video, we create 8 versions listed in Table 3. Note that since the participants need to watch a large number of impaired copies, the video length (10 seconds) has to be short. Also note that the videos have different point densities, as we want to make the QoE model generic, applicable to different resolutions. We will experimentally verify this shortly. We use our optimized PU-GAN algorithm (details in §5.1) to perform upsampling and create video clips at 4K resolution for four viewing distances: 1m, 2m, 3m, and 4m, which are determined from a separate IRB-approved user study whose details are described in Appendix B. To maintain a fixed viewing distance  $d$ , we display the viewport at  $d$  meters in front of and facing the viewer. We design a survey using Qualtrics [11] and publish it on Amazon Mechanical Turk (AMT) [2].

We study the impact of all the factors in Eq. 8 on the QoE. Table 4 lists them and their impairment levels. They lead to a total of 756 combinations for each video segment. Since

letting subjects perform  $(756)^2$  pairwise comparisons is infeasible, for each combination, we generate one video clip by putting the impaired version and the high-quality “ground truth” version ( $4 \times 1$ ,  $I_i^{patch} = I_i^{frame} = I_i^{stall} = 0$ , same viewing distance) side by side, in a random order. To generate the impaired version, we randomly add perturbations to the patches’ quality levels to match the corresponding  $I_i^{patch}$  and  $I_i^{frame}$  values, and randomly inject stalls to match  $I_i^{stall}$ . We then ask each subject to watch 100 randomly selected video clips from the 756 clips of a randomly selected video segment. After watching each clip, the subject is asked to rate which side provides a better QoE through 7 choices (“left looks {much better, better, slightly better, similar to, slightly worse, worse, much worse} than right”) If the impaired version is {similar to, slightly worse, worse, much worse} than the ground truth, we give the impaired version a score of {3,2,1,0}, respectively.

We have collected 934 subjects’ responses with a total number of 93,400 ratings for the above survey published on AMT. For each viewing distance, we use the subjects’ ratings to calculate the average score of each of the 756 impaired clips on a scale from 0 to 3, and use it as the QoE ground truth. We then perform 10-fold cross-validation to validate our QoE model (Eq. 8, trained using multi-variable linear regression) for each viewing distance. Figure 2 plots the CDF of the QoE prediction errors at each viewing distance. The median prediction error for 1m, 2m, 3m, 4m is 11.4%, 12.2%, 12.8%, and 12.9%, respectively. The (Person, Spearman) correlation coefficients between the ground-truth QoE score and the predicted QoE score are also high: (0.89, 0.89) at 1m, (0.87, 0.88) at 2m, (0.87, 0.88) at 3m, and (0.85, 0.85) at 4m.

The above QoE models are trained from all four videos. Table 5 shows the Spearman correlation coefficients between the ground-truth QoE and *cross-video* prediction results. We use the data of three videos to train a QoE model and use it to predict the QoE for the remaining video. The results indicate that the same QoE model and its parameters are applicable to volumetric content of the same genre (portraits of people – a major application of volumetric streaming – in our case). We also confirm that most parameters trained from different video segments are indeed quite similar, in spite of the segments’ different point densities. When applied to other genres, the model’s parameters may differ, as to be explored in our future work (the same happens to 2D videos [68]). Table 6 lists our final model’s parameters trained using the entire dataset. The model will be used by YuZu.

## 5 System Design of YuZu

We now detail the system design of YuZu (Figure 1) that addresses the challenges we identified in §2.

### 5.1 Accelerating SR Upsampling

To accelerate 3D upsampling, we take a principled approach by exploring three orthogonal directions:

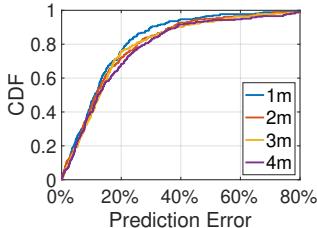


Figure 2: QoE prediction error using our model.

$\delta$	D: Long Dress; L: Loot; B: Band; H: Haggle			
	$DBH \Rightarrow L$	$LBH \Rightarrow D$	$DLB \Rightarrow H$	$DLH \Rightarrow B$
1m	0.80	0.74	0.86	0.85
2m	0.76	0.71	0.87	0.87
3m	0.80	0.73	0.83	0.87
4m	0.78	0.71	0.76	0.80

Table 5: Spearman correlation coefficient between QoE ground truth and cross-video prediction.  $XZY \Rightarrow W$  means using the model trained from videos X, Y, and Z to predict video W’s QoE.

$\delta$	Long Dress + Loot + Band + Haggle				
	$w_1$	$w_2$	$\mu_p$	$\mu_f$	$\mu_s$
1m	0.55	27.80	0.52	0.40	170.5
2m	0.42	39.83	1.05	0.91	149.8
3m	0.27	26.63	1.23	1.04	176.7
4m	0.16	17.17	0.47	0.06	304.1

Table 6: Parameters of the final model used in YuZu.

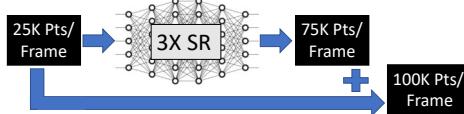


Figure 3: Using a 3× SR model to realize 4× SR.

- **Model Optimization.** How to simplify the upsampling logic while retaining the inference accuracy? (§5.1.1)
- **Data Reduction.** How to strategically feed less data to SR models with negligible impact on QoE? (§5.2)
- **Pre-processing and Post-processing Trimming.** How to simplify the sophisticated pre- and post-processing stages without incurring side effects on inferences? (§5.1.2)

Our optimizations can apply to all 3D SR models we have investigated [43, 61, 63, 70] and they are video-agnostic. In §7, we demonstrate the optimization results for two SR models: PU-GAN [43] and MPU [61].

### 5.1.1 SR Model Optimization

We take a “top-down” approach by first optimizing the model as a whole and then fine-tuning its detailed structure. For most machine learning models (including 2D SR), after performing an inference, the input is no longer needed and will be discarded. Our investigated 3D SR models [43, 61, 63, 70] make no exception. We instead make a fundamental observation regarding 3D point clouds. Different from a 2D image, a point cloud is a set of unstructured points, which means that point clouds can be *merged via a simple set union operation*. We also note that 3D SR’s output points *refine and differ from* the input. Based on this key insight, we propose a simple yet effective optimization: YuZu merges the input low-density point cloud with the SR output in order to improve the visual quality, or to reduce the computation overhead while maintaining the same upsampling ratio. For example, as shown in Figure 3, to achieve 4× upsampling, instead of using a 4× SR model, we can use a (computationally more efficient) 3× SR model and merge the input with the output. Since SR exploits the overfitting nature of DNN, the spatial distributions of upsampled points and the ground truth are expected to be highly similar. By leveraging the input data and downgrading the SR ratio from 4× to 3×, we can achieve an acceleration of up to ∼35% without hurting the SR accuracy (Figure 6). Note that in offline training, the loss function is computed *after* merging the input low-density point cloud with the SR output. This makes the trained models aware of and adaptive

to the merging process, improving the upsampling accuracy compared to computing the loss function before that.

Next, we explore modifying 3D SR model’s DNN structure for inference acceleration. By profiling the inference time of PU-GAN, we find that its three stages, feature extraction, feature expansion, and point set generation, take 78.3%, 19.3%, and 2.4% of execution time, respectively (4× SR). Within the feature extraction stage that dominates the runtime overhead, most operations are *convolutions*. We make the same observation for other 3D SR models that we investigated [61, 63, 70].

To accelerate convolutions, we replace the original feature extraction, which (*e.g.*, in the case of PU-GAN) enhances the solution in PointNet++ [51] through dynamic graph convolution [56], with a recent proposal called spherical kernel function (SKF) [42]. SKF partitions a 3D space into multiple volumetric bins and specifies a learnable parameter to convolve the points in each bin. In contrast to continuous filter approaches (*e.g.*, multilayer perceptron) used in existing SR models, SKF is a *discrete* metric-based spherical convolutional kernel, and is thus computationally attractive for dense point clouds. Moreover, it is applicable to all the 3D SR models we examined. We find that SKF brings no degradation to the upsampling accuracy (§7.3). One reason may be that the kernel asymmetry of SKF facilitates learning fine geometric details of point clouds [42].

In addition to utilizing SKF, we conduct layer-by-layer profiling [22, 66] to fine-tune the SR model’s performance-accuracy tradeoff. Take PU-GAN as an example. We remove the last two dense layers of feature extraction and several heavyweight convolution layers in the feature expansion stage, as they make limited contributions to the upsampling accuracy. We also judiciously remove a small number of expanded features to reduce the GPU memory footprint. For other 3D SR models, their model tuning follows a similar approach.

### 5.1.2 Trimming Pre- and Post-Processing

Recall from §2 that to ensure a manageable model complexity, a 3D SR model divides a point cloud into small patches as basic units for upsampling. We discover that as an important pre-processing step, the patch generation process incurs a high overhead. For example, PU-GAN generates the patches by applying kNN to the seeds created by downsampling. Since the generated patches may overlap, after upsampling, PU-

GAN needs to perform post-processing: it applies the furthest point sampling [48] to remove duplicated points.

To mitigate the above overhead, YuZu adopts a simple patch generation method. It divides the space into cubic cells, and assigns each non-empty cell (*i.e.*, a cell that contains points) to a patch. Compared to the default patch generation approaches used by PU-GAN and other 3D SR frameworks [43, 61], our approach runs very fast; it also brings no overlap among patches, thus eliminating the post-processing step (*i.e.*, overlap removal). In addition, the patches now have a simple geometry shape, so that they can be indexed, searched, and manipulated at runtime. Meanwhile, We find that our patch generation approach does not sacrifice the up-sampling accuracy and may even improve the accuracy compared to vanilla PU-GAN and MPU (§7.3). This is likely because cubic cells provide a more consistent structure for the patches, making it easier to perform SR. We also investigate several other patch generation methods based on Voronoi diagram [24] and 3D SIFT [55], but none outperforms our cubic-cell-based approach from either the performance or the accuracy perspective.

## 5.2 Caching and Reusing SR Results

Videos usually exhibit similarities across frames. We find that volumetric videos make no exceptions. This indicates rich opportunities for caching and reusing SR results.

At a high level, YuZu reuses SR results based on the similarity between patches, which is the basis of inter-frame encoding. Inter-frame similarity has been extensively studied and exploited in 2D videos. However, none of the 2D inter-frame encoding techniques can be directly applied to volumetric videos due to the fundamental difference between pixel-based 2D frames and volumetric frames consisting of unstructured points. There are very few studies on 3D inter-frame encoding [37, 46]; they are incompatible with YuZu’s patch-based upsampling, and incur high complexity hindering line-rate decoding. Due to the above reasons, we design our own SR caching/reusing algorithm. Our algorithm is agnostic of and orthogonal to a specific SR model.

YuZu reuses 3D SR results on a per-patch basis to match the patch-based upsampling procedure. Recall from §5.1.2 that YuZu generates patches using 3D cubic cells. Let  $p(i, j)$  denote patch  $j$  of frame  $i$ , and let  $N(i, j)$  denote the number of points in  $p(i, j)$ . YuZu allows reusing the SR result of  $p(i, j)$  for subsequent consecutive patches at the same location, *i.e.*,  $p(i+1, j), p(i+2, j)$ , and so on. YuZu restricts reusing patches only at the same location due to two considerations. First, we empirically observe that most patch similarities indeed occur at the same cell location; this makes the benefits (in terms of reduced SR overhead) of reusing a patch belonging to a different cell marginal. Second, allowing reusing a patch at a different cell will drastically increase the overhead of pre-computing the caching/reusing decisions.

We now describe YuZu’s SR reuse algorithm. YuZu first

determines offline the similarity of two patches. For each patch pair  $(p(i, j), p(i+1, j))$ , YuZu computes a Weighted Complete Bipartite Graph [17]  $B : p(i, j) \rightarrow p(i+1, j)$ , which we find to be suitable for dealing with unstructured points. In the bipartite graph, there is a directed edge from every point in  $p(i, j)$  to every point in  $p(i+1, j)$ , and the weight of the edge is their Euclidean distance. We then calculate the minimum-weight matching (MWM) [57] for the graph, *i.e.*, finding  $N(i, j)$  edges such that (1) these edges share no common vertices (points), and (2) the sum of their weights is minimized. Intuitively, the MWM identifies a transformation from  $p(i, j)$  to  $p(i+1, j)$  with a minimum moving distance for the points. The Hungarian algorithm [17] that computes the MWM has a complexity of  $O(N^4)$  where  $N = \max\{N(i, j), N(i+1, j)\}$ . We instead employ a faster  $O(N^2)$  approximation algorithm that is found to work well in practice.<sup>3</sup>

We call every edge in the MWM a point motion vector (PMV). A PMV differs from a 2D video’s motion vector, which represents a macroblock in a frame based on the position of the same or a similar macroblock in another reference frame. Leveraging the PMVs, we determine that  $p(i+1, j)$  and  $p(i, j)$  are *similar* if three criteria are satisfied. (1)  $N(i, j)$  and  $N(i+1, j)$  differ by no more than  $\eta_n\%$ ; (2) the average length of all the PMVs is smaller than  $\eta_a$ ; (3) the top 90-percentile of the shortest PMV is smaller than  $\eta_v$ . These three criteria dictate that  $p(i, j)$  and  $p(i+1, j)$  have a similar number of points, and the points’ collective motions are small. Figure 4 shows how  $\eta_a$  impacts EMD and the patch reuse ratio (% of patches that can reuse a previous SR result). As shown, increasing  $\eta_a$  increases the reuse ratio, but meanwhile decreases the accuracy. According to Figure 4, we set  $\eta_a$  to 0.01m to balance the performance and accuracy. Using similar methods, we empirically set  $\eta_n=10$  and  $\eta_v=0.01$ m.

Next, we consider how to reuse an SR result across multiple patches belonging to consecutive frames. We define  $sim_j(i_1, i_2) \in \{0, 1\}$  to be 1 if and only if  $p(i_1, j)$  and  $p(i_2, j)$  are similar, *i.e.*, satisfying the above three criteria where  $i_2 > i_1$ . Figure 5 shows an example of 6 consecutive patches at location  $j$  where  $\forall 1 \leq x < y \leq 6 : sim_j(x, y) = 0$  except that  $sim_j(1, 2), sim_j(2, 3), sim_j(2, 4)$ , and  $sim_j(2, 6)$  are 1. YuZu allows a patch’s SR result to be reused across *consecutive* patches if they are all similar to the first patch. For example, Patches 3 and 4 can reuse Patch 2’s SR result. However, YuZu does not let Patch 6 reuse Patch 2 because  $sim_j(2, 5) = 0$ . We make this design decision for two reasons. First, we observe that non-consecutive patches are unlikely to be similar in real volumetric videos. Second, supporting non-consecutive reuse requires computing  $sim_j(x, y) \forall x < y$ , making offline video processing slow.

We develop an algorithm that *minimizes the number of*

<sup>3</sup>The approximation algorithm sorts all the edges by their weights in ascending order. It then adds the edges to the MWM in that order and skips edges that share points with an existing edge in the matching, until every point in  $p(i, j)$  or every point in  $p(i+1, j)$  is in the MWM.

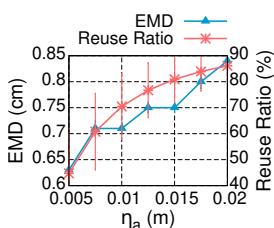


Figure 4: Impact of  $\eta_a$  (using the video in §2, 1x4 SR).

patches to be upsampled, to boost the online SR performance. For example, in Figure 5, the minimum number of patches to be upsampled is 4: Patches 1, 2, 5, and 6. YuZu efficiently and optimally solves this through dynamic programming (DP). Given  $n$  patches  $p(1, j), \dots, p(n, j)$  and their  $sim_j$  information, let  $u(i, j)$  be the minimum number of patches that need to be upsampled in  $\{p(i, j), \dots, p(n, j)\}$  if we decide to upsample  $p(i, j)$ . Then  $u(i, j)$  can be derived through DP as:

$$u(i, j) = \min \left\{ u(i+1, j), \min_{i < k \leq n: \forall i < t \leq k: sim_j(i, t) = 1} \{u(k+1, j)\} \right\} + 1 \quad (9)$$

The RHS of Eq. 9 examines each patch following  $p(i, j)$  and updates  $u(i, j)$  if stopping reusing  $p(i, j)$  at  $p(k+1, j)$  yields a better  $u(i, j)$ . The search continues until hitting a patch that is not similar to  $p(i, j)$ . Eq. 9 can be solved backwards starting from  $u(n+1, j) = 0$ . The solution is  $u(1, j)$ .

Since YuZu streams VoD volumetric content, all the above logic (calculating MWM,  $sim_j$ , and DP) is performed *offline* for each patch location  $j$ . Thus, there is no runtime overhead. The SR reuse decisions are sent to the client as meta data, which is only 0.5KB per frame for our testing video in §2.

### 5.3 Network/Compute Resource Adaptation

YuZu adapts to not only the fluctuating network condition (similar to the job of traditional bitrate adaptation algorithms [45, 64, 69]), but also the available compute resource, due to the high computation overhead of 3D SR. More importantly, these two dimensions incur a tradeoff: given a fixed playback deadline, should YuZu download high-resolution content, or download lower-resolution content and spend time upsampling it? Fortunately, our QoE model (§4.1) dictates how to quantitatively balance this tradeoff.

We first formulate an online network/compute adaptation problem. The video is divided into  $n$  chunks each consisting of  $f$  frames. To achieve fine-grained adaptation, each chunk is further spatially segmented into  $b$  blocks (*e.g.*,  $b=5^3$ ), which are the atomic scheduling units in YuZu’s adaptation algorithm. Each block consists of multiple patches (recall from §5.1.2 that each patch occupies a cubic cell). At runtime, YuZu considers all the blocks belonging to a finite horizon of the next  $w$  chunks, and searches for their quality and SR ratio assignments that maximize the QoE defined in Eq. 8. This formulation extends the model predictive control (MPC) scheme [69] that proves to be effective for traditional 2D

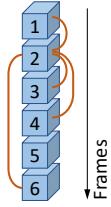


Figure 5: Reusing SR results across consecutive frames.

video streaming. The solution space is  $O(8^{wb})$  (the 8 possible assignments are listed in Table 3).

We consider how to efficiently solve the above discrete optimization problem. An exhaustive search is clearly infeasible. Due to the large solution space, even the memorization approach (FastMPC [69]) is not practical. Another possibility is a learning-based approach such as Pensieve [45]. However, it requires offline training and may incur a non-trivial inference overhead. Moreover, a recent work [64] indicates that reinforcement learning based bitrate adaptation solutions do not necessarily outperform simple buffer-based approaches [33].

To overcome the above challenges, we develop a lightweight approximation algorithm. It executes in two stages: first determine the quality and SR ratios of to-be-downloaded chunks, and then fine-tune the SR ratios before upsampling. Specifically, in the first stage, *before downloading each chunk*, YuZu performs a *coarse-grained search* by assuming that all the blocks in each chunk have the same quality/SR-ratio assignment. The rationale is that, at this moment, the playback deadline is still far away (compared to Stage 2), and thus the network/computation-load uncertainty diminishes the benefits brought by a block-level, fine-grained search. Meanwhile, this reduces the solution space from  $O(8^{wb})$  to  $O(8^w)$ . Specifically, we (1) start with a quasi-optimal solution obtained from an even coarser-grained search at the granularity of every two consecutive chunks, and (2) perform pruning by bounding [19]. After the above two optimizations, for a practical  $w$  (*e.g.*,  $w=10$ ), the search time (for maximizing the QoE in Eq. 8) becomes negligible compared to the downloading and upsampling time. To estimate  $I_i^{start}$  in Eq. 8, at runtime, YuZu continuously estimates (1) the network bandwidth using the method in [29] and (2) the local processing time of a frame using EWMA-based estimation.

The second stage takes place *before upsampling each frame*. At this stage, the playback deadline gets closer and thus a *block-level, fine-grained search* would be beneficial. To reduce the search complexity, YuZu employs Simulated Annealing (SA) [40] – a probabilistic, greedy approach that approximates the global optimum. Our algorithm begins with setting all the blocks’ SR ratios to the lowest (no SR). For each block, the algorithm tries to increase its SR ratio by one level. If the resulting QoE of the finite horizon increases, this change is always accepted; otherwise, we may still accept this change with a probability of  $\exp(-\frac{\Delta}{t})$ , where  $\Delta$  is the decrease of the QoE and  $t$  is the current number of iterations, to avoid a potential local maximum. To speed up the SA algorithm, we reduce the finite horizon to two frames: the previous frame and the current (to be upsampled) frame – we empirically find that conducting frequent adaptations with a short horizon at a per-frame basis outperforms infrequent adaptations with a long horizon at a per-chunk basis in terms of the QoE.

## 5.4 Coloring SR Results

As described in §2, none of the 3D SR models we investigated performs colorization. There are two high-level approaches for colorization. One is augmenting the SR models by adding the color component. This may yield good colorization results, but at the cost of significantly increasing the SR workload. Given this concern, YuZu takes a much more lightweight approach: approximating each upsampled point’s color using the color of the nearest point in the low-density point cloud (*i.e.*, the input to the SR model). In Appendix C, we present the details of our method and experimentally confirm that it can indeed produce good visual quality (with a PSNR >38).

## 6 Implementation

We integrate all the components in §5 into YuZu, a holistic system as shown in Figure 1. Our implementation consists of 10,848 lines of code (LoC), with 8,326 LoC for the client.

For offline SR model training, we modify the source code of PU-GAN [10] and MPU [8] using TensorFlow 1.14 [13] and custom TensorFlow operators from SPH3D-GCN [12]. Our pre-trained models are saved in the ProtoBuf format [9] that is language- and platform-neutral, facilitating future reuse. For online streaming, we implement the client player on Linux in C++. We use the Draco Library [4] for encoding and decoding the point cloud data. We employ Bazel [3] to compile the TensorFlow 1.14 C/C++ library and use the compiled library to load and execute the SR models. The client *pipelines* content fetching (network-bound), point cloud decoding & patch generation (CPU-bound), 3D SR (GPU-bound), and colorization (CPU-bound) of different frames for better performance. The server is also built in C++, with a custom DASH-like protocol over TCP for client-server communication.

## 7 Evaluation

### 7.1 Experimental Setup

**Volumetric Videos.** We use four point-cloud-based volumetric videos throughout our evaluations. (1) Our own video. We capture a volumetric video by ourselves using 3 synchronized depth cameras. It has 3,622 frames (2 min) each consisting of ~100K points. We refer to this video as *Lab*. We have used it to motivate YuZu in §2. (2) The Long Dress (*Dress*) and *Loot* videos (§4.2). They have 300 frames (10 sec) each consisting of ~100K points. Since they are short, we loop them (with cold caches) 10 times in our evaluations. (3) The *Haggle* video (§4.2). It has 7,800 frames (4’20”) each consisting of ~100K points. For all four videos, the eight possible resolution/SR-ratio assignments are listed in Table 3. For each video, we train their SR models separately. All the videos are at 30 FPS, encoded by Draco [4]. Unless otherwise mentioned, the results reported in the remainder of this section are generated using all four videos. The average encoded bitrate of *Lab*, *Dress*, *Loot*, and *Haggle* (4×1) are 96, 108, 118, and 118 Mbps, respectively.

$M_1$	The vanilla 3D SR model (PU-GAN and MPU)
$M_2$	$M_1$ and optimizing patch generation
$M_3$	$M_2$ and layer profiling & pruning
$M_4$	$M_3$ and applying the spherical kernel function (SKF)
$M_5$	$M_4$ and merging SR input with SR output
$M_6$	$M_5$ and caching/reusing SR results

Table 7: SR acceleration methods (cumulative).

**3D SR Models.** We apply our developed model acceleration techniques to two recently proposed 3D SR models: PU-GAN [43] and MPU [61]. The two models usually yield qualitatively similar results, so we show the results of PU-GAN by default. For certain SR-specific experiments (*e.g.*, SR acceleration), we show both models’ results. The models are trained on a per-video basis. For each video, the total size of all its models ( $\times 2$ ,  $\times 3$ , and  $\times 4$ ) is around 1.25 MB.

**Metrics and Roadmap.** We thoroughly evaluate YuZu in terms of performance, QoE, and resource utilization. §7.2 evaluates the QoE improvement brought by our 3D SR optimizations using both subjective (*i.e.*, real-user ratings) and objective (*e.g.*, PSNR [30]) metrics. §7.3 focuses on the performance gain of our 3D SR optimizations, from the perspectives of resource usage, inference time, and upsampling accuracy. §7.4 and §7.5 evaluate the end-to-end performance (*e.g.*, QoE and data usage) of YuZu. §7.6 provides additional micro benchmarks.

**Network Conditions.** We consider the following network conditions that are readily available in today’s wired and wireless networks. (1) *Wired network with stable bandwidth* (*e.g.*, 50, 75, and 100 Mbps) and ~10ms RTT. (2) *Fluctuating bandwidth* captured from real LTE networks. We collect 12 bandwidth traces from a major LTE carrier in multiple U.S. states at diverse locations (campus, malls, streets, *etc.*). Across the traces, their average bandwidth varies from 33.7 to 176.5 Mbps, and the standard deviation ranges from 13.5 to 26.8 Mbps. We use `tcc` [6] to replay these traces (with a 50ms base RTT typically observed in LTE [38]). (3) We also conduct *live LTE experiments* at 9 diverse locations in a U.S. city where the average bandwidth varies from 41.1 to 52.4 Mbps and the standard deviation is between 16.6 and 20.7 Mbps.

**Devices.** We use a commodity machine with an Intel Core i7-9800X CPU @ 3.80GHz and 32GB memory as the YuZu server. We use three client hosts: (1) a desktop with an Intel Core i9-10900X CPU @ 3.70GHz, an NVIDIA GeForce RTX 2080Ti GPU, and 32GB memory (the default client used in our evaluations); (2) a desktop with the same CPU, an NVIDIA GeForce GTX 1660Ti GPU, and 32GB memory; (3) an NVIDIA Jetson TX2 embedded system board with a Pascal-architecture GPU of 256 CUDA Cores, 8GB memory, and a quad-core CPU. They represent a typical high-end PC, a medium-class PC, and a mobile device, respectively.

**User Motion Traces.** We collect 32 users’ 6DoF motion traces when watching the four videos, and replay them in some experiments. The details about how we collect the motion traces can be found in Appendix B.

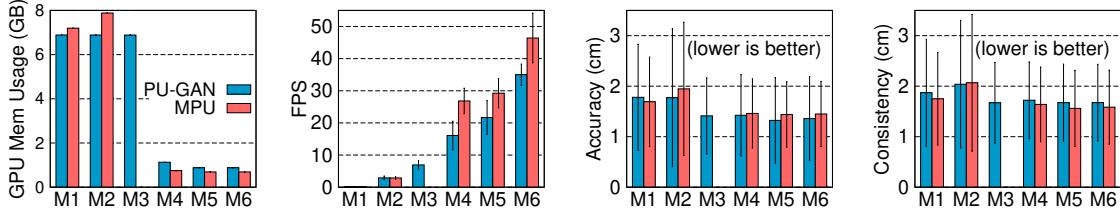


Figure 6: Memory usage, upsampling FPS, upsampling accuracy, and visual consistency of  $M_1$  to  $M_6$  (2080Ti desktop).

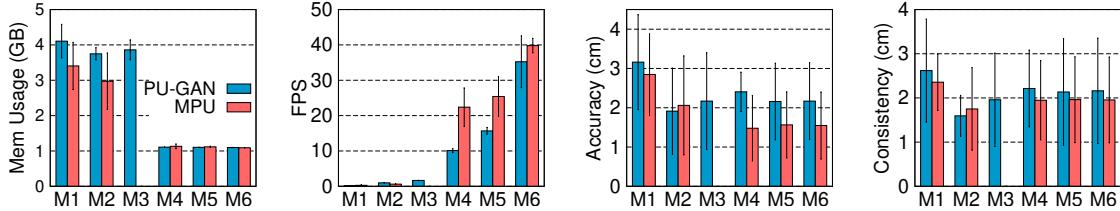


Figure 7: Memory usage, upsampling FPS, upsampling accuracy, and visual consistency of  $M_1$  to  $M_6$  (Jetson TX2 board).

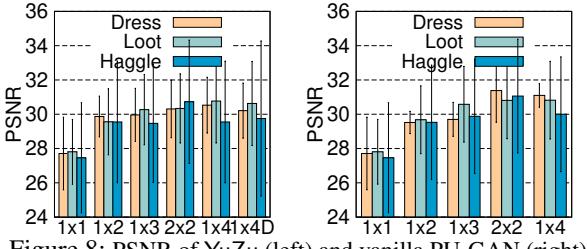


Figure 8: PSNR of YuZu (left) and vanilla PU-GAN (right).

## 7.2 SR Quality

**Subjective Ratings.** Recall that in our user studies, we ask our participants to rate the SR results generated by our optimized SR scheme (§5.1). Figure 15 shows that SR brings a significant boost to the user-perceived QoE. For example, at 1m, compared to 1×1, the user-rated QoE increases by 37%, 75%, and 150% for 1×2, 1×3, and 1×4, respectively; 2×2 improves the QoE by 178% compared to 2×1 (§4.2).

**Objective Metric.** We also examine how SR improves PSNR [30], an objective metric of image quality. The methodology is as follows. We replay the 32 users’ 6DoF motion traces of watching the videos under different SR settings, and save the rendered viewports as images  $\{I_{SR}\}$ . We then repeat the above process using the original videos (4×1), and capture the viewport images  $\{I_{4 \times 1}\}$ . We compute the PSNR values by comparing each image in  $\{I_{SR}\}$  with its corresponding image in  $\{I_{4 \times 1}\}$ . Figure 8 (left) shows the PSNR values for 1×1, 1×2, 1×3, 2×2, 1×4, and 1×4D with reusing SR results (denoted as “1×4D”) across all the captured viewports. We notice a significant increase of PSNR from 1×1 to 1×2. The PSNR also increases marginally from 1×2 to 1×4. Meanwhile, the PSNR change between 1×4 and 1×4D is negligible, indicating that caching and reusing SR results brings little impact on the perceived video quality (but drastic performance gain as shown in §7.3). The results of *Lab* are similar. Note that a PSNR value over 30 typically indicates good visual quality [22, 58]. Figure 8 (right) shows the PSNR values for the unmodified PU-GAN model. The qualitatively similar results

between the left and right plots of Figure 8 indicate that our SR acceleration modifications sacrifice little visual quality. Note the above results include the colorization step, which is described and separately evaluated in Appendix C.

Comparing Figure 15 and Figure 8, we notice disparities between users’ QoE ratings and PSNR values. This indicates that image qualities of rendered 2D content do not directly reflect the perceived QoE of volumetric content. This is a key reason for developing the QoE model for volumetric videos.

## 7.3 SR Performance Breakdown

We now take a closer look at the effectiveness of each of our proposed methods for accelerating SR. As listed in Table 7,  $M_1$  denotes the vanilla 3D SR model as the comparison baseline;  $M_2$  to  $M_6$  are our proposed SR acceleration methods in §5.1 and §5.2. They are presented in a *cumulative* fashion, *i.e.*,  $M_i$  includes every feature of  $M_{i-1}$  plus some new feature. The experiments are conducted using two 3D SR models (PU-GAN [43] and MPU [61]), 100Mbps wired network, 4× SR, with network/compute resource adaptation (§5.3) disabled.

Figures 6 and 7 show the results of PU-GAN and MPU on the PC (2080Ti) and Jetson TX2 board, respectively. On the Jetson board, due to its low compute power (and mobile devices’ small screen size), we reduce the original video’s resolution from 100K to 20K points per frame (*i.e.*, the SR is from 5K to 20K points per frame). We consider four metrics: (1) maximum GPU memory usage (on Jetson TX2 we measure the system memory shared by GPU and CPU), (2) average upsampling speed (in FPS), (3) inference accuracy measured in EMD between each upsampled frame and the ground truth (4×1), and (4) visual consistency measured in EMD between each consecutive pair of upsampled frames.

As shown, on 2080Ti, for PU-GAN (MPU), compared to  $M_1$ ,  $M_6$  reduces the GPU memory usage by 87% (90%), accelerates the upsampling by 307× (542×), improves the average upsampling accuracy by 24% (14%), and slightly improves the consistency. Also, each optimization ( $M_2$  to  $M_6$ ) indi-

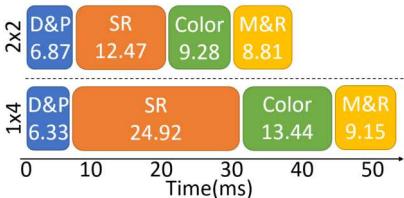


Figure 9: Frame processing time breakdown. D&P: decoding and patch generation; SR: upsampling; Color: colorization, M&R: merging and rendering.

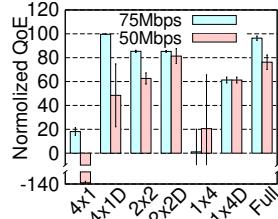


Figure 10: QoE over stable bandwidth (“D”=caching & reusing SR results).

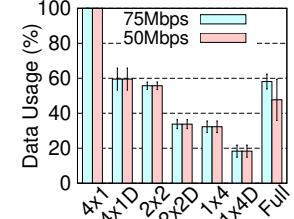


Figure 11: Data usage over stable bandwidth.



Figure 12: QoE vs. Data usage over fluctuating bandwidth (LTE traces).

ividually improves the upsampling speed and possibly other metric(s). The Jetson setup shows a similar trend. The two models (PU-GAN and MPU) we studied exhibit similar performance gains as we progressively apply our optimizations, except that MPU is less sensitive to  $M_5$ . This is because of the network structure difference between PU-GAN and MPU. Note that we do not apply M3 to MPU because our layer-by-layer profiling (§5.1.1) reveals there is no layer that only makes a marginal contribution to the overall upsampling accuracy in the MPU model.

**Latency Breakdown.** Figure 9 shows the latency breakdown of processing an average frame using PU-GAN (*Lab* video, wired 100Mbps, 2080Ti desktop) under two settings: 2x2 and 1x4. As shown, SR remains the most time-consuming component. The breakdown for MPU is similar. The above results indicate the importance of SR acceleration.

## 7.4 Diverse Network Conditions

We evaluate the QoE of YuZu under different network conditions, using the four videos and the associated motion traces.

**Stable Bandwidth.** We first consider two stable bandwidth: 50Mbps and 75Mbps. Under each bandwidth profile, we run the full-fledged YuZu (“Full”) and six statically configured YuZu instances: 4x1, 2x2, and 1x4 with and without SR result reusing. The QoE results are shown in Figure 10. We make several observations. First, when the bandwidth is low (50Mbps), 4x1 (without SR) gives the lowest (and even negative) QoE. This is because the limited bandwidth leads to high *network-incurred* stall when fetching high-resolution content; SR can effectively improve the QoE by using computation to compensate for the low bandwidth. Second, when the bandwidth increases to 75Mbps, 1x4 gives the lowest QoE due to the distortion and *computation-incurred* stall due to the high SR ratio. Instead, when the bandwidth is sufficient, the player should fetch the content with a higher quality (e.g., 4x1D). Third, caching and reusing (C&R) the SR results improves the QoE when either the bandwidth is low (e.g., 4x1 at 50Mbps), or the SR ratio is high (e.g., 1x4). Under these two scenarios, C&R reduces the network and compute resource usage, respectively. The saved resources can be used to improve the content quality for other frames with more heterogeneity.

Figure 11 compares the (normalized) data usage, which is defined as the total downloaded bytes including the SR

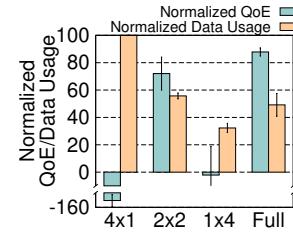


Figure 13: QoE and data usage over live LTE networks.

models and meta data. Compared to 4x1, applying C&R reduces the data usage by 40.5%. Also, increasing the SR ratio reduces the data usage, e.g., 1x4D consumes only 18.3% of the data compared to 4x1. The full-fledged YuZu with adaptation gives the overall best QoE (Figure 10) and low data usage (Figure 11) by balancing the compute and network resource consumption. Compared to 4x1, full YuZu reduces the data usage by 52.3% (50Mbps) and 41.9% (75Mbps) while boosting the QoE by 214% (50Mbps) and 78.3% (75Mbps).

**Fluctuating Bandwidth.** We repeat the above experiment over fluctuating bandwidth emulated using our collected LTE traces (§7.1). The results are shown in Figure 12, which considers both the data usage (x-axis) and the QoE (y-axis). 4x1 yields the highest data usage; further applying C&R (4x1D) not only reduces the data usage by 40.5%, but also increases the QoE by 61.8% due to reduced stall. The full YuZu further improves the QoE by 21.0% and reduces the average data usage by 8.2%. This is achieved through strategically fetching lower-quality blocks and using higher SR ratios. In addition, the full YuZu improves the QoE by 10.4% to 93.7%, compared to 1x4 and 2x2 with and without C&R.

**Live LTE.** We conduct live LTE experiments at 9 locations in a major U.S. city. As shown in Figure 13, the results are largely aligned with those in Figure 12, except for the lower QoE of 4x1. This is because of the lower bandwidth of live LTE throughout the test locations compared to the LTE traces used in Figure 12. Compared to 4x1, the full YuZu improves the QoE by 210.3% and reduces the data usage by 50.8%.

## 7.5 YuZu vs. Existing Approaches

**YuZu vs. Viewport-Adaptive Streaming.** We compare YuZu with ViVo [27], a recently proposed viewport-adaptive approach. Leveraging 6DoF motion prediction, ViVo determines what content to fetch and which quality to fetch based on

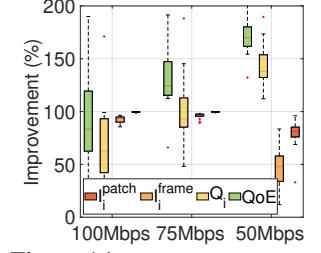


Figure 14: YuZu over ViVo.

predicted viewport and viewing distance. Similar viewport-adaptive approaches are used in the other systems [41, 50].

We develop a custom replication of ViVo on Linux in 7,101 LoC with the same set of configuration parameters. Figure 14 shows the improvement brought by YuZu compared to ViVo in terms of the overall QoE and its three components ( $Q_i$ ,  $I_i^{patch}$ , and  $I_i^{frame}$ , see Eq. 8), using all four videos and the users' motion traces.<sup>4</sup> Note that both systems exhibit negligible stall so  $I_i^{stall}$  is not plotted. As shown, YuZu brings significant improvement on the average QoE (by 100.6% to 174.9%) and on each QoE component. YuZu outperforms ViVo due to three reasons. First, ViVo does not support SR, which YuZu leverages to boost the QoE. Second, ViVo's viewport adaptation approach becomes less effective when the whole scene appears inside the viewport (which oftentimes appears in our motion traces). SR does not suffer from this limitation. Third, to realize viewport adaptive streaming, ViVo has to perform 6DoF motion prediction, which is error-prone. In contrast, YuZu does not require motion prediction, and therefore exhibits more stable performance in particular when the motion is fast. Note that viewport-adaptation and SR are orthogonal approaches and can be *jointly* applied.

**YuZu vs. Simple SR Adaptation.** To demonstrate the efficacy of our network/compute resource adaptation design (§5.3), we compare it with a simple adaptation approach that differs in two aspects. First, unlike YuZu's *two-stage* adaptation, it only performs *single-stage* adaptation before downloading each chunk. Second, it employs a *deterministic* greedy algorithm that increases the SR ratio of each block within the finite horizon (in chronological order) until the QoE does not further improve. In contrast, YuZu employs a *probabilistic* greedy approach that is less vulnerable to a local maximum. We evaluate the simple adaptation algorithm using our LTE traces (§7.4) and plot its result as “Simple” in Figure 12. Compared to it, the full YuZu increases the average QoE by 11.4% and reduces the average data usage by 7.9%.

## 7.6 Micro Benchmarks and Resource Usage

We conduct experiments to show the following. (1) YuZu can work adaptively with different hardware (we compare the results on 2080Ti and 1660Ti; we also ported YuZu to an embedded system, see Figure 7). (2) The main memory (~5GB) and GPU memory (~2GB) usage of YuZu is acceptable. (3) The (one-time) offline training time is non-trivial but acceptable, and the sizes of SR models are negligible (<0.2% of the video size). The details can be found in Appendix D.

## 8 Related Work

**Volumetric Video Streaming.** There exist only a few studies on point-cloud-based volumetric video streaming [25–27, 31,

<sup>4</sup>ViVo does not have the notion of patch; instead its basic adaptation unit is a cubic cell. To ensure fair comparisons, we further divide ViVo's cells into virtual “patches” with the same size as YuZu and assign to them its parent cell's corresponding quality level when calculating  $I_i^{patch}$ .

41, 50, 52, 59]. For example, DASH-PC [31] extends DASH to volumetric videos. PCC-DASH [59] is another DASH-based streaming scheme of compressed point clouds with bitrate adaptation support. ViVo [27] introduces visibility-aware optimizations for volumetric video streaming. GROOT [41] optimizes point cloud compression for volumetric videos. To the best of our knowledge, there is no existing work on applying 3D SR to volumetric video streaming.

**Point Cloud SR.** We can classify existing work on point cloud SR into two categories: optimization-based [16, 32] and learning-based [43, 61, 63, 70]. Most learning-based approaches follow the workflow established in PU-Net [70], which divides a point cloud into patches, learns multi-level point features of each patch, expands the features, and reconstructs the points from the features. All the above methods are designed for a *single* point cloud; they suffer from numerous limitations when applied to volumetric videos (§2).

**Visual Quality Assessment of Point Clouds.** The state-of-the-art visual quality assessment focuses on *static, non-SR* point clouds [23, 47, 60]. For example, using a data-driven approach, Meynet *et al.* [47] present a full-reference visual quality metric for colored point clouds. Different from the above studies, we model the QoE of SR-enhanced volumetric video streaming. We address new challenges on modeling the impact of various factors such as the viewing distance, upsampling ratio, and SR incurred distortion (§4).

**SR for Regular 2D Videos.** NAS [67, 68] is one of the first proposals that apply 2D SR to Internet video streaming. Other recent efforts on 2D SR include PARSEC [22] for 360° panoramic video streaming, LiveNAS [39] for live video streaming, and NEMO [66] for mobile video streaming. In contrast, YuZu addresses numerous unique challenges (§1) on applying 3D SR to volumetric video streaming.

## 9 Concluding Remarks

In this paper, we conduct an in-depth investigation on applying 3D SR to streaming volumetric content. Our proposed QoE model and the YuZu system take a first and important step toward making SR-enhanced volumetric video streaming principled, practical, and affordable. YuZu demonstrates how a series of novel optimizations, which fill a 500× performance gap, as well as judicious network/compute resource adaptation can help significantly improve the QoE for volumetric video streaming.

## Acknowledgments

We thank the anonymous reviewers and our shepherd Anirudh Badam for their insightful comments. The research of Feng Qian was supported in part by a Cisco research award. The research of Bo Han was funded in part by 4-VA, a collaborative partnership for advancing the Commonwealth of Virginia.

## References

- [1] 8i Voxelized Full Bodies (8iVFB v2) - Dynamic Voxelized Point Cloud Dataset. <http://plenodb.jpeg.org/pc/8ilabs>.
- [2] Amazon Mechanical Turk. <https://www.mturk.com/>.
- [3] Bazel. <https://bazel.build/>.
- [4] Draco 3D Data Compression. <https://google.github.io/draco/>.
- [5] ITU-P.913: Methods for the subjective assessment of video quality, audio quality and audiovisual quality of Internet video and distribution quality television in any environment. <https://www.itu.int/rec/T-REC-P.913>.
- [6] Linux TC Man Page. <https://linux.die.net/man/8/tc>.
- [7] Magic Leap One. <https://www.magicleap.com/en-us/magic-leap-1>.
- [8] MPU. <https://github.com/yifita/3PU>.
- [9] Protocol Buffers. <https://developers.google.com/protocol-buffers>.
- [10] PU-GAN. <https://github.com/liruihui/PU-GAN>.
- [11] Qualtrics experience management platform. <https://www.qualtrics.com/>.
- [12] SPH3D-GCN. <https://github.com/hlei-ziyan/SPH3D-GCN>.
- [13] TensorFlow 1.14. <https://github.com/tensorflow/tensorflow/tree/r1.14>.
- [14] The Octree Data Structure. <https://en.wikipedia.org/wiki/Octree>.
- [15] Video Multimethod Assessment Fusion. [https://en.wikipedia.org/wiki/Video\\_Multimethod\\_Assessment\\_Fusion](https://en.wikipedia.org/wiki/Video_Multimethod_Assessment_Fusion), 2016.
- [16] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and Rendering Point Set Surfaces. *IEEE Trans. on Visualization and Computer Graphics*, 9(1):3–15, 2003.
- [17] Armen S Asratian, Tristan MJ Denley, and Roland Hägkvist. *Bipartite graphs and their applications*, volume 131. Cambridge university press, 1998.
- [18] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *Proceedings of ACM SIGCOMM*, 2013.
- [19] Egon Balas and Paolo Toth. Branch and bound methods for the traveling salesman problem. 1983.
- [20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.
- [21] Hong Chang, Dit-Yan Yeung, and Yimin Xiong. Super-Resolution through Neighbor Embedding. In *Proceedings of CVPR*, 2004.
- [22] Mallesham Dasari, Arani Bhattacharya, Santiago Vargas, Pranjul Sahu, Aruna Balasubramanian, and Samir Das. Streaming 360 degree Videos using Super-resolution. In *Proceedings of IEEE INFOCOM*, 2020.
- [23] Rafael Diniz, Pedro Garcia Freitas, and Mylène C.Q. Farias. Towards a Point Cloud Quality Assessment Model Using Local Binary Patterns. In *Proceedings of the 12th International Conference on Quality of Multimedia Experience (QoMEX)*, 2020.
- [24] Steven Fortune. Voronoi diagrams and delaunay triangulations. In *Comp. in Euclidean Geometry*, pages 225–265. World Sci., 1995.
- [25] Serhan Güл, Dimitri Podborski, Thomas Buchholz, Thomas Schierl, and Cornelius Hellge. Low-latency cloud-based volumetric video streaming using head motion prediction. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 27–33, 2020.
- [26] Serhan Güл, Dimitri Podborski, Jangwoo Son, Gurdeep Singh Bhullar, Thomas Buchholz, Thomas Schierl, and Cornelius Hellge. Cloud Rendering-based Volumetric Video Streaming System for Mixed Reality Services. In *Proceedings of ACM MMSys*, 2020.
- [27] Bo Han, Yu Liu, and Feng Qian. ViVo: Visibility-Aware Mobile Volumetric Video Streaming. In *Proceedings of ACM MobiCom*, 2020.
- [28] Jian He, Mubashir Adnan Qureshi, Lili Qiu, Jin Li, Feng Li, and Lei Han. Rubiks: Practical 360° Streaming for Smartphones. In *Proceedings of ACM MobiSys*, 2018.
- [29] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the predictability of large transfer tcp throughput. *ACM SIGCOMM Computer Communication Review*, 35(4):145–156, 2005.

- [30] Alain Hore and Djemel Ziou. Image quality metrics: PSNR vs. SSIM. In *Proceedings of the 20th International Conference on Pattern Recognition*, pages 2366–2369. IEEE, 2010.
- [31] Mohammad Hosseini and Christian Timmerer. Dynamic Adaptive Point Cloud Streaming. In *Proceedings of ACM Packet Video*, 2018.
- [32] Hui Huang, Shihao Wu, Minglun Gong, Daniel Cohen-Or, Uri Ascher, and Hao Zhang. Edge-Aware Point Set Resampling. *ACM Transactions on Graphics*, 32(1):9:1–9:12, 2013.
- [33] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of ACM SIGCOMM*, 2014.
- [34] Yan Huang, Jingliang Peng, C.-C. Jay Kuo, and M. Gopi. A Generic Scheme for Progressive Point Cloud Coding. *IEEE Trans. on Vis. and Computer Graphics*, 14(2):440–453, 2008.
- [35] Erik Hubo, Tom Mertens, Tom Haber, and Philippe Bekaert. The Quantized kd-Tree: Efficient Ray Tracing of Compressed Point Clouds. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 2006.
- [36] Hanbyul Joo, Tomas Simon, Xulong Li, Hao Liu, Lei Tan, Lin Gui, Sean Banerjee, Timothy Scott Godisart, Bart Nabbe, Iain Matthews, Takeo Kanade, Shohei Nobuhara, and Yaser Sheikh. Panoptic studio: A massively multiview system for social interaction capture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.
- [37] Julius Kammerl, Nico Blodow, Radu Bogdan Rusu, Suat Gedikli, Michael Beetz, and Eckehard Steinbach. Real-time Compression of Point Cloud Streams. In *Proceedings of International Conference on Robotics and Automation*, 2012.
- [38] Ali Safari Khatouni, Marco Mellia, Marco Ajmone Marsan, Stefan Alfredsson, Jonas Karlsson, Anna Brumstrom, Ozgu Alay, Andra Lutu, Cise Midoglu, and Vincenzo Mancuso. Speedtest-like measurements in 3g/4g networks: The monroe experience. In *Proceedings of the 29th International Teletraffic Congress (ITC 29)*, pages 169–177, 2017.
- [39] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-Enhanced Live Streaming: Improving Live Video Ingest via Online Learning. In *Proceedings of ACM SIGCOMM*, 2020.
- [40] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.
- [41] Kyungjin Lee, Juheon Yi, Youngki Lee, Sunghyun Choi, and Young Min Kim. GROOT: A Real-Time Streaming System of High-Fidelity Volumetric Videos. In *Proceedings of ACM MobiCom*, 2020.
- [42] Huan Lei, Naveed Akhtar, and Ajmal Mian. Spherical Kernel for Efficient Graph Convolution on 3D Point Clouds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.
- [43] Ruihui Li, Xianzhi Li, Chi-Wing Fu, Daniel Cohen-Or, and Pheng-Ann Heng. PU-GAN: A Point Cloud Upsampling Adversarial Network. In *Proceedings of ICCV*, 2019.
- [44] Jyh-Ming Lien, Gregorij Kurillo, and Ruzena Bajcsy. Multi-camera tele-immersion system with real-time model driven data compression. *The Visual Computer*, 26(3):3–15, 2010.
- [45] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of ACM SIGCOMM*, 2017.
- [46] Rafael Mekuria, Kees Blom, and Pablo Cesar. Design, Implementation and Evaluation of a Point Cloud Codec for Tele-Immersive Video. *IEEE Trans. on Circuits and Systems for Video Technology*, 27(4):828–842, 2017.
- [47] Gabriel Meynet, Yana Nehmé, Julie Digne, and Guillaume Lavoué. PCQM: A Full-Reference Quality Metric for Colored 3D Point Clouds. In *Proceedings of the 12th International Conference on Quality of Multimedia Experience (QoMEX)*, 2020.
- [48] Carsten Moenning and Neil A Dodgson. Fast marching farthest point sampling. Technical report, University of Cambridge, 2003.
- [49] Sergio Orts-Escalano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip Davidson, Sameh Khamis, Mingsong Dou, Vladimir Tankovich, Charles Loop, Qin Cai, Philip Chou, Sarah Mennicken, Julien Valentin, Vivek Pradeep, Shenlong Wang, Sing Bing Kang, Pushmeet Kohli, Yuliya Lutchny, Cem Keskin, and Shahram Izadi. Holoporation: Virtual 3D Teleportation in Real-time. In *Proceedings of ACM UIST*, 2016.
- [50] Jounsup Park, Philip A Chou, and Jenq-Neng Hwang. Volumetric media streaming for augmented reality. In *2018 IEEE Global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

- [51] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Proceedings of Conference on Neural Information Processing Systems (NeurIPS)*, 2017.
- [52] Feng Qian, Bo Han, Jarrell Pair, and Vijay Gopalakrishnan. Toward Practical Volumetric Video Streaming On Commodity Smartphones. In *Proceedings of ACM HotMobile*, 2019.
- [53] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical Viewport-Adaptive 360-Degree Video Streaming for Mobile Devices. In *Proceedings of ACM MobiCom*, 2018.
- [54] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover’s distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.
- [55] Paul Scovanner, Saad Ali, and Mubarak Shah. A 3-dimensional SIFT descriptor and its application to action recognition. In *Proceedings of the 15th ACM International Conference on Multimedia*, pages 357–360, 2007.
- [56] Yiru Shen, Chen Feng, Yaoqing Yang, and Dong Tian. Mining Point Cloud Local Structures by Kernel Correlation and Graph Pooling. In *Proceedings of CVPR*, 2018.
- [57] Steven L Tanimoto, Alon Itai, and Michael Rodeh. Some matching problems for bipartite graphs. *Journal of the ACM (JACM)*, 25(4):517–525, 1978.
- [58] Nikolaos Thomos, Nikolaos V Boulgouris, and Michael G Strintzis. Optimized transmission of jpeg2000 streams over wireless channels. *IEEE Transactions on image processing*, 15(1):54–67, 2005.
- [59] Jeroen van der Hooft, Tim Wauters, Filip De Turck, Christian Timmerer, and Hermann Hellwagner. Towards 6DoF HTTP Adaptive Streaming Through Point Cloud Compression. In *Proceedings of ACM Multimedia*, 2019.
- [60] Irene Viola, Shishir Subramanyam, and Pablo Cesar. A color-based objective quality metric for point cloud contents. In *Proceedings of the 12th International Conference on Quality of Multimedia Experience (QoMEX)*, 2020.
- [61] Yifan Wang, Shihao Wu, Hui Huang, Daniel Cohen-Or, and Olga Sorkine-Hornung. Patch-based Progressive 3D Point Set Upsampling. In *Proceedings of CVPR*, 2019.
- [62] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.
- [63] Huikai Wu, Junge Zhang, and Kaiqi Huang. Point cloud super resolution with adversarial residual graph networks. In *arXiv preprint arXiv:1908.02111*, 2019.
- [64] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *Proceedings of USENIX NSDI*, 2020.
- [65] Jianchao Yang, John Wright, Thomas S. Huang, and Yi Ma. Image Super-Resolution Via Sparse Representation. *IEEE Transactions on Image Processing*, 19(11):2861–2873, 2010.
- [66] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. NEMO: Enabling Neural-enhanced Video Streaming on Commodity Mobile Devices. In *Proceedings of ACM MobiCom*, 2020.
- [67] Hyunho Yeo, Sunghyun Do, and Dongsu Han. How will Deep Learning Change Internet Video Delivery? In *Proceedings of ACM HotNets*, 2017.
- [68] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural Adaptive Content-aware Internet Video Delivery. In *Proceedings of USENIX OSDI*, 2018.
- [69] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of ACM SIGCOMM*, 2015.
- [70] Lequan Yu, Xianzhi Li, Chi-Wing Fu, Daniel Cohen-Or, and Pheng-Ann Heng. PU-Net: Point Cloud Upsampling Network. In *Proceedings of CVPR*, 2018.
- [71] Anlan Zhang, Chendong Wang, Bo Han, and Feng Qian. Efficient volumetric video streaming through super resolution. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 106–111, 2021.

## Appendices

### A Evaluation of QoE Gain Brought by SR

We study the QoE model for  $q_{i,j}$  (Eq. 2) while keeping  $I_i^{patch}$ ,  $I_i^{frame}$ , and  $I_i^{stall}$  as zero. This allows us to measure the impact of SR without interference from other factors.

We use the four videos introduced in §4.2 for the experiment. We apply our optimized PU-GAN algorithm (details

in §5.1) to perform upsampling, and create  $\binom{8}{2} = 28$  video clips where each clip contains 2 out of 8 versions in Table 3 side by side (in a random order). This approach is known as the double stimulus comparison scale (DSCS) method [5] as recommended by ITU (International Telecommunication Union). We repeat the above process for four viewing distances: 1m, 2m, 3m, and 4m, which are determined from a separate IRB-approved user study whose details are described in Appendix B. To maintain a fixed viewing distance  $d$ , we display the viewport at  $d$  meters in front of and facing the viewer. We generate 112 video clips at 4K resolution for each video segment.

Next, we design a survey using Qualtrics [11] and publish it on Amazon Mechanical Turk (AMT) [2]. In the survey, we invite each paid AMT subject to view the 112 clips of a random video segment (out of the 4 videos) in a random order. After watching each clip, the subject is asked to rate which side provides a better QoE through 7 choices (“left looks {much better, better, slightly better, similar to, slightly worse, worse, much worse} than right”). We have collected 512 subjects’ responses with a total number of 57,344 ratings. We show the demographics of the participants in Table 2.

Figure 15 shows the average ratings of the 8 versions across all the users. The four subplots correspond to the four viewing distances. We make four observations. First, when the viewing distance is short, SR can effectively boost the QoE. For example, at 1m, compared to 1×1, the (user-rated) QoE increases by 37%, 75%, 150% for 1×2, 1×3, and 1×4, respectively; 2×2 improves the QoE by 178% compared to 2×1. Second, under the same point density, the upsampled version’s QoE is usually lower than the original content’s QoE, in particular when the SR ratio is large. This is caused by SR’s distortion. However, the gap tends to reduce as the SR ratio decreases. Third, SR’s gain diminishes as the distance increases, because the rendered object becomes smaller in the view. Note that the scores for different distances are not directly comparable. Fourth, the four video segments exhibit similar trends (figure not shown).

**Converting User Ratings to Numerical Scores.** For a given tuple of (user, viewing distance, video segment), we construct a weighted directed graph for the user based on his/her ratings, where the nodes are the 8 schemes. Assume a video clip contains schemes A (on the left) and B (on the right). If the user thinks that the left (right) is much better, better, or slightly better than the right (left), we add an edge from B to A (A to B) with a weight of 3, 2, and 1, respectively. If the user thinks that the left is similar to the right, we add two edges between A and B, one from A to B and the other from B to A, with both edges’ weights set to 0. We then normalize the weights of all the edges to  $[0, 1]$  and apply the PageRank algorithm [20] to each graph to compute the weight of every node. We then use the weights (multiplied by 10 for easy interpretation) as the numerical scores of the 8 schemes for the corresponding (user, viewing distance, video segment)

tuple. Finally, for each of the 8 schemes under a given viewing distance, we average the numerical scores across all the tuples (of that viewing distance) to obtain the results shown in Figure 15. Note that for each viewing distance, the weights of all the schemes (in each of the graphs) add up to 1. As a result, the numerical scores of the same scheme for different viewing distances are not directly comparable.

## B User Study for Collecting 6DoF Motion Traces

We conducted a separate IRB-approved user study for collecting 6DoF motion traces of volumetric videos. Specifically, it captured the viewport trajectories of 32 users who watched the four video segments (*Lab*, *Dress*, *Loot*, *Haggle*) introduced in §2 and §4.2 through either a mixed reality headset (Magic Leap One [7]) or an Android smartphone. We developed custom volumetric video players for both device types. The 6DoF motion data (yaw, pitch, roll, X, Y, Z) was captured at the granularity of 30 Hz. The participants are diverse in terms of their education level (from freshman to Ph.D.), gender (16 females), and age (from 22 to 57). We determine the viewing distances used in §4.2 by analyzing the above traces. As shown in Figure 16, about 70% of the viewing distances are less than 4m. Therefore, we set the maximum viewing distance to be 4m for our user studies, and select the other three distances by evenly dividing this maximum distance into four ranges (*i.e.*, at 1, 2, and 3m).

## C Colorization Algorithm of YuZu and its Evaluation

Recall from §5.4 that YuZu takes a lightweight approach to color the SR results: it approximates each upsampled point’s color using the color of the nearest point in the low-density point cloud (*i.e.*, the input to the SR model).

YuZu employs two mechanisms to speed up the nearest point search. First, the search is performed on an octree [14], which recursively divides a point cloud (as the root node) into eight octants, each associated with a child node. The levels of detail of the point cloud are controlled by the height of the tree. Performing nearest point search on an octree has a low complexity of  $O(\log N)$  where  $N$  is the number of nodes in the tree.

Second, YuZu caches and reuses the results of previously searched points. The cache is indexed by a point’s discretized coordinates, and the cached value is the color looked up from the octree. When coloring an upsampled point, YuZu first performs cache lookup in  $O(1)$ ; upon a hit, the cached color will be directly used as the color of the point; otherwise, YuZu performs a full octree search and adds the search result to the cache. The discretization granularity incurs a tradeoff between colorization performance and quality. We empirically observe that a discretization granularity of  $1\text{cm}^3$  can yield good visual

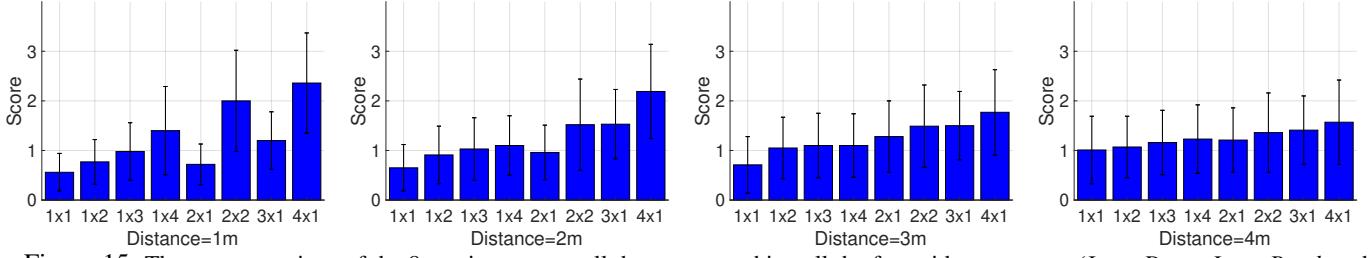


Figure 15: The average ratings of the 8 versions across all the users watching all the four video segments (*Long Dress*, *Loot*, *Band*, and *Haggle*).

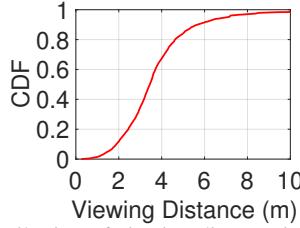


Figure 16: Distribution of viewing distance in our motion traces.

quality under typical viewing distances ( $\geq 1\text{m}$ ).

We also notice opportunities for further improving the colorization quality. For example, the nearest point approach can be generalized into interpolating the nearest  $k$  points' colors; it can also be used in conjunction with DNN-based colorization, which may be more suitable for patches with complex, heterogeneous colors. Nevertheless, these enhancements inevitably increase the runtime overhead. We will explore them in future work.

**Evaluation of Quality of Colorization.** To evaluate the quality of the colorization step alone, we employ the approach in §7.2 where we use PSNR to objectively assess the image quality of rendered viewports. Specifically, we calculate the PSNR values by comparing  $\{I_{4 \times 1}^{\text{NP-Color}}\}$  (defined below) with  $\{I_{4 \times 1}\}$  (defined in §7.2), using the *Dress* and *Loot* videos and the real users' motion traces (Appendix B). The viewport images of  $\{I_{4 \times 1}^{\text{NP-Color}}\}$  are obtained as follows: (1) remove the color from the original ( $4 \times 1$ ) video; (2) apply the above nearest-point (NP) colorization method to the video generated in Step (1), using the  $1 \times 1$  video as the low-resolution point cloud stream from which the colors are picked; (3) replay the same motion traces to render the viewport images for the video colored in Step (2). The PSNR values of  $\{I_{4 \times 1}^{\text{NP-Color}}\}$  are  $38.09 \pm 2.44$  and  $44.15 \pm 2.59$  for *Dress* and *Loot*, respectively, indicating the high fidelity of colors produced by our method. The above numbers are much higher than the PSNR values in Figure 8 (which also includes the colorization step) due to the following reason. PSNR and many other 2D image metrics such as SSIM [62] perform a pixel-wise comparison between two images. In the case of Figure 8, a tiny position shift of a 3D point may result in an also tiny position shift of its projected 2D pixel, leading to a pixel mismatch and thus a decreased PSNR score. This problem does not appear in the

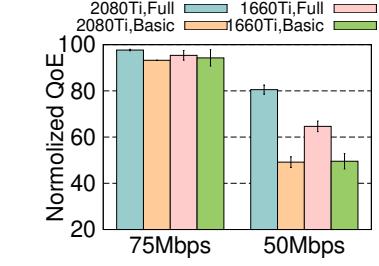


Figure 17: Impact of hardware and computation-aware adaptation.

colorization step.

## D Additional Micro Benchmarks

The following micro benchmark results are generated using the PU-GAN model. The results for the MPU model are qualitatively similar.

**Impact of Computation-aware Adaptation.** 3D SR demands considerable compute resources. Figure 17 demonstrates the impact of hardware and computation-aware adaptation, using the *Lab* video. Figure 17 considers two GPUs: a more powerful 2080Ti GPU and a less powerful 1060Ti GPU. It also considers two adaptation schemes: the full network/compute adaptation scheme described in §5.3 (“Full”) and a computation-agnostic scheme that only adapts according to the network bandwidth (“Basic”). The Basic scheme works as follows. (1) It assumes that SR takes no time to complete; (2) it disables  $2 \times 2$  and  $1 \times 4$  (otherwise the QoE will degrade too much due to excessive stalls). Under the above setup, each bandwidth setting in Figure 17 has four schemes: {2080Ti, 1660Ti}  $\times$  {Full, Basic}. As shown, when there is sufficient bandwidth, the QoE differences among the four schemes are small, because the player is more likely to fetch  $3 \times 1$  and  $4 \times 1$  blocks that do not require SR. However, when the bandwidth becomes low, the difference between 2080Ti and 1060Ti becomes noticeable, and the gap between Full and Basic is even larger. The Basic scheme yields much lower QoE scores because it ignores SR’s computation overhead, leading to excessive stalls.

**Memory Usage.** We measure the client-side memory usage when streaming the *Lab* video over 50Mbps bandwidth (which leads to extensive invocations of SR). On the 2080Ti

(1660Ti) desktop, the peak main memory usage is 5.03GB (5.33 GB); the peak GPU memory usage is 1.97 GB (1.83 GB). YuZu’s GPU memory usage on 2080Ti is higher than the numbers reported in Figure 6 because YuZu loads multiple SR models at runtime. When the available bandwidth is higher, the CPU/GPU memory will reduce because of fewer SR operations.

**Offline Training Time and Model Size.** YuZu incurs non-trivial model training time. For example, on the 2080Ti

desktop, it takes about 88 minutes to train the  $1\times 2$ ,  $1\times 3$ , and  $1\times 4$  models altogether for the *Lab* video consisting of 3,622 frames. However, note that (1) this is a one-time overhead; (2) we did not conduct any performance optimization for training; for a large-scale deployment, the training overhead could potentially be reduced by training one generic model and fine-tuning it for each specific video [68] (left as future work). The SR model size is negligible (< 0.2%) compared to the video size.

# NetVRM: Virtual Register Memory for Programmable Networks

Hang Zhu

*Johns Hopkins University*

Dan R. K. Ports

*Microsoft Research*

Tao Wang

*New York University*

Anirudh Sivaraman

*New York University*

Yi Hong

*Johns Hopkins University*

Xin Jin

*Peking University*

## Abstract

Programmable networks are enabling a new class of applications that leverage the line-rate processing capability and on-chip register memory of the switch data plane. Yet the status quo is focused on developing approaches that share the register memory statically. We present NetVRM, a network management system that supports *dynamic* register memory sharing between multiple *concurrent* applications on a programmable network and is readily deployable on commodity programmable switches. NetVRM provides a *virtual register memory abstraction* that enables applications to share the register memory in the data plane, and abstracts away the underlying details. In principle, NetVRM supports *any* memory allocation algorithm given the *virtual register memory abstraction*. It also provides a default memory allocation algorithm that exploits the observation that applications have diminishing returns on additional memory. NetVRM provides an extension of P4, P4VRM, for developing applications with virtual register memory, and a compiler to generate data plane programs and control plane APIs. Testbed experiments show that NetVRM generalizes to a diverse variety of applications, and that its utility-based dynamic allocation policy outperforms static resource allocation. Specifically, it improves the mean satisfaction ratio (i.e., the fraction of a network application’s lifetime that it meets its utility target) by 1.6–2.2× under a range of workloads.

## 1 Introduction

Programmable networks are a new paradigm that changes how we design, build and manage computer networks. Compared to traditional fixed-function switches, programmable switches allow developers to flexibly change how packets are processed in the switch data plane. The programming model of programmable switches are based on a multi-stage packet processing pipeline [8, 9].

Programmable switches provide different types of stateful objects that preserve states between packets, such as tables, counters, meters and registers. Among them, registers allow packets to read and write various states at line rate, which then affects how the following packets are processed. Such data-plane-accessible *register memory* is one of the defining features of programmable switches, and enables a new class of *reg-stateful* applications which utilize the on-chip register

memory to realize various functionalities. These reg-stateful applications include not only the innovations in traditional network functions like congestion control [45], load balancing [25, 35] and network telemetry [1, 18], but also novel use cases beyond traditional networking, such as caching [23, 32], consensus [13, 14, 22] and machine learning [42, 43].

Given the rise of reg-stateful applications, an important open problem is how to support multiple concurrent reg-stateful applications running efficiently on a programmable network [51]. The utility of reg-stateful applications is usually decided by the amount of allocated register memory and the real-time network traffic [18, 23, 34, 47, 54, 58]. Thus, it is essential to dynamically allocate the limited register memory between multiple applications to optimize the multiplexing benefits. Yet existing approaches of running multiple concurrent applications on programmable networks allocate register memory statically [19, 44, 49, 56, 57]. Changing the amount of register memory for one application would require recompiling and reloading the switch program, which would disrupt the operation of the switch.

In this paper, we propose NetVRM, a network management system that supports *dynamic* register memory sharing between multiple *concurrent* applications on a programmable network. NetVRM advances the status quo with three major features: The first one is a novel *virtual register memory abstraction*, which allows the register memory in the switch data plane to be dynamically allocated between multiple concurrent applications *at runtime*, without recompiling and reloading the data plane program. The second one is a dynamic memory allocation algorithm, which efficiently arbitrates the memory usage between concurrent applications based on the real-time utility measurements. The third one is a language extension and a compiler to generate data plane programs with the virtual register memory abstraction and efficient C++ control plane APIs for high-speed virtual register memory configuration.

The virtualization of register memory allows its dynamic allocation. Our approach is inspired by traditional virtual memory designs in operating systems, but programmable switches introduce two new challenges. First, register memory is distributed over multiple pipeline stages, and each register can be accessed only from one stage. Second, switch applications can access register memory from both the data

plane and control plane. NetVRM’s memory system design is tailored to these characteristics. It places a page table at the front of the virtual register memory’s processing pipeline, using it for memory translation in the data plane. The page table indexes the register memory regions allocated to each application in every stage. The switch control plane manages memory allocation. NetVRM also mediates application accesses to register memory from the control plane to ensure addresses are correctly translated.

NetVRM’s dynamic memory allocation policy exploits the fundamental tradeoff between memory consumption and application utility. In particular, it leverages *diminishing returns*: the observation that, for most reg-stateful applications, the benefit of additional memory decreases with the amount of allocated memory [18, 23, 34, 47, 58]. For example, after a certain point, NetCache [23] cannot further improve the throughput significantly. More importantly, the memory-utility relationship changes both in the *temporal* and *spatial* dimensions based on application characteristics and traffic conditions. For example, the amount of register memory needed by NetCache depends on the request pattern, which can change over time and even vary across different switches. We design an online algorithm that does global memory allocation between applications in the network to maximize multiplexing benefits.

To make it easy to develop applications with NetVRM, we propose P4VRM, an extension to P4 [8]. P4VRM allows developers to virtualize register memory with a few simple modifications to existing P4 code: they mark register arrays to be virtualized and add online utility measurement primitives provided by P4VRM. The compiler takes multiple P4VRM programs as input and outputs a single P4 program with the virtual register memory abstraction and all the applications’ functionalities, and generates the control plane APIs for high-speed virtual memory configuration.

In summary, we make the following contributions.

- We propose NetVRM, a network management system that exposes a virtual register memory abstraction to enable dynamic register memory sharing between multiple concurrent applications on a programmable network at runtime without recompiling and reloading.
- We design a dynamic memory allocation algorithm to efficiently allocate register memory between applications to maximize multiplexing benefits.
- We propose P4VRM, a data plane program extension, and provide a compiler to easily equip the data plane programs with virtual register memory and generate control plane APIs for efficient virtual memory configurations.
- We implement a NetVRM prototype. Testbed experiments on a variety of applications show that compared to static memory allocation, NetVRM improves the mean satisfaction ratio (i.e., the fraction of a network application’s lifetime that it meets its utility target) by 1.6–2.2× under a range of workloads.

## 2 Motivation and Related Work

### 2.1 The Case of Dynamic Register Memory Allocation

**Concurrent reg-stateful network applications.** There are two broad types of objects provided by commodity programmable switches on the data plane—*stateless* objects, such as metadata, packet headers, and *stateful* objects, such as match-action tables, counters, meters, registers. Among them, registers, as one of the defining features of new-generation programmable switches, provide data-plane-accessible *register memory* for packets to read and write various states at line rate and enable much of the latest exciting research [14, 22, 25, 35, 42, 43, 45]. Register memory is implemented with standard SRAM blocks and can be read and written by both the control plane and data plane. Stateful Arithmetic and Logic Unit (ALU) performs register memory access and modification by executing a short program that involves register data, metadata and constant. The register memory is usually organized as register arrays. Each register array consists of several register slots with the same width and can be addressed by index (direct mapping) and hash (hash mapping). We refer to the network applications that use the register memory as *reg-stateful* applications.

Besides the rise and evolution of reg-stateful applications, modern cloud service providers usually serve multiple tenants concurrently [6, 30]. They allow tenants to run different network applications dynamically. For example, Azure and AWS provide a variety of network applications [5, 7] to their tenants, such as network address translation (NAT), load balancer, and network monitoring. We anticipate that the reg-stateful applications will be provided to tenants as programmable switches are being integrated in cloud networks, including both the datacenter networks and the wide area networks that connect the datacenter networks.

**Necessity and potential benefits of network-wide dynamic allocation.** The register memory on programmable switches is fundamentally limited by the hardware. For example, the maximal size of register memory on each stage is only a few Mb on the Intel Tofino switch [50]. Besides the limited register memory, there is a fundamental trade-off between memory consumption and application utility (e.g., its performance or accuracy) in many reg-stateful network applications [18, 23, 34, 47, 58]. Although some applications have a fixed memory requirement, most can operate with different amounts of available memory. Notably, our key observation is that applications generally exhibit *diminishing returns* [18, 23, 34, 47, 58]. The utility improvement decreases with more memory, and for many applications, additional memory has *no* utility after a point. We demonstrate the diminishing returns for four applications in Appendix A, including heavy hitter detection (HH) [54], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [54] and NetCache [23]. The utility is measured using memory hit ratio (§5.1).

In all cases, the amount of memory affects the application utility, and such effects depend heavily on the workload. For example, NetCache [23] needs different amount of register memory with different skewed workload to deliver the same utility (Appendix A). Without dynamic allocation, this presents a formidable deployment challenge because the workload can vary in both temporal and spatial dimensions: different storage clusters see radically different workloads, and even a single cluster’s request pattern changes over time (e.g., on a diurnal cycle) [4].

The diminishing returns and the temporally and spatially dynamic workload together also provide the opportunity to maximize resource multiplexing benefits by efficiently arbitrating the memory usage between concurrent applications.

## 2.2 Target and Scope of NetVRM

**Target applications.** The reg-stateful applications that can benefit from NetVRM must have the following properties.

- **They are elastic** (§5). An inelastic application (e.g., NetChain [22]) that has fixed virtual memory requirement can be supported by NetVRM, but cannot benefit from dynamic memory allocation.
- **The data plane programs have to meet the constraints in P4VRM** (§6), such as stateful ALUs since each operation of one register array must be associated with a specific stateful ALU.
- **The application utility should be obtained instantaneously** (§5.1). It can be computed on the switch (e.g., hit ratio as the default utility) or reported by applications.

We remark that there are a wide range of applications with the above properties, such as measurement applications [18, 39, 47], applications with approximate data structures [20, 34, 54], and caching applications [23, 33].

**Register memory as the scope.** There are a variety of resource types on a programmable switch, such as register memory, SRAM used for tables, TCAM and action units [51]. NetVRM focuses on dynamic allocation for register memory for three reasons. First, we observe that many reg-stateful applications are bottlenecked by register memory. Second, dynamic allocation of other resource types (e.g., match-action tables, TCAM) has been well-studied in the context of Software-Defined Networking (SDN) with traditional fixed-function switches [17, 21, 36, 46]. Third, current switch hardware cannot dynamically reallocate other resource types without rebooting the entire switch [51]. NetVRM is readily deployable on existing programmable switches.

Switch memory available that can be used as virtual register memory could be limited because a certain amount of memory has to be set aside for basic networking functionality, such as L3 routing, and inelastic applications (see §5). The evaluation in §8 shows that NetVRM outperforms the alternatives, regardless of *how much* physical memory is available for virtualization and dynamic allocation. Thus, NetVRM continues to be effective even as the memory for basic net-

working functionality and inelastic applications grows in size, leaving behind less memory for dynamic allocation.

## 2.3 Existing Solutions and Limitations

Recently, several existing works have explored how to support multiple applications on a programmable switch [19, 44, 48, 49, 56, 57]. At a high level, these solutions fail to meet the requirement of dynamic register memory allocation because of at least one of three limitations as follows.

- **Static binding of register memory.** Some of the existing work combine or merge multiple applications into one monolithic data plane program [19, 48, 56, 57] in compilation time. And the binding between register memory allocation and applications is static. Changing the allocation requires the data plane program to be recompiled and reloaded, during which the switch has to be stopped and restarted. This interrupts the operation of all applications on the switch, even the basic ones such as L3 routing.
- **Lack of a real switch environment.** Most of the existing solutions ignore the practical hardware constraints and are not applicable on a real ASIC-based switch (e.g., Intel Tofino [50]). For example, P4VBox [44] provides parallel execution of virtual switch instances on NetFPGA. MTPSA [49] realizes a multi-tenant portable switch architecture on NetFPGA and BMv2, a reference P4 software switch [3]. HyPer4 [19] and HyperV [56] realize the virtualization on software switches (e.g., BMv2, DPDK).
- **Not doing network-wide dynamic allocation.** Network resource allocation has been well studied for SDN with traditional fixed-function switches [16, 17, 21, 36, 37, 46]. For example, DREAM [36] does dynamic allocation for TCAM between measurement applications. However, none of the existing work has disclosed the potential benefit of a network-wide dynamic allocation for the *register memory* on programmable networks.

There are other related works that have explored how to manage and improve network applications on programmable networks. TEA [27] provides external DRAM for storing *table entries*, not *register memory*. Dejavu [52] utilizes the multiple pipelines and resubmission to fit a service chaining in one single switch. RedPlane [28] enables fault-tolerant stateful applications by designing a practical, provably correct replication protocol. NetVRM targets *register memory* and provides a new system for sharing it between multiple concurrent *reg-stateful* applications dynamically.

## 3 NetVRM Overview

NetVRM is a network management system that supports dynamic register memory sharing between multiple concurrent applications on a programmable network. Figure 1 shows an overview of NetVRM. NetVRM includes three critical components: virtual register memory, dynamic memory allocation and the P4VRM compiler. It abstracts away the complexities of allocating physical memory in each application, increases

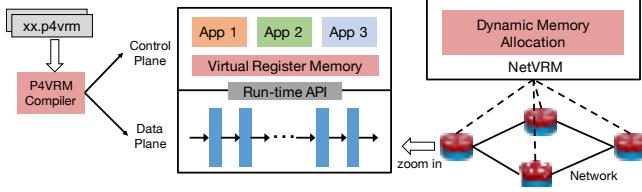


Figure 1: NetVRM overview.

memory utilization via statistical multiplexing, and provides P4VRM as an extension of P4 for developing applications with such virtual register memory.

**Virtual register memory (§4).** NetVRM exposes a virtual register memory abstraction to applications. The virtual register memory component in every switch hides the underlying details of the physical register memory that may span multiple stages and be shared with multiple applications. We design a custom data plane layout and an address translation mechanism to realize the virtual memory. The data plane layout composes the register arrays in multiple stages to one large register array, and allocates the large array to applications. Memory translation contains two page tables. One page table is in the data plane that translates the memory addresses computed from packet headers for memory access during packet processing, and the other is in the control plane for NetVRM to query and update the virtual memory of applications. The two tables are synchronized and managed by NetVRM.

**Dynamic memory allocation (§5).** In principle, NetVRM can support *any* memory allocation algorithm built on top of the virtual register memory. NetVRM also provides a default network-wide memory allocation algorithm for applications *without* knowing the utility functions. The algorithm exploits the diminishing returns between memory usage and application utility to maximize resource multiplexing benefits. We leverage the observation that many applications use the switch as a performance accelerator and deal with insufficient switch memory by having some kind of fallback path, either through the switch control plane or the servers [23, 29, 47]. As such, we cast the resource allocation problem as satisfying as many application’s requirements as possible with respect to available memory size. This allows operators to specify application-specific utility metric and target for each application, avoiding the need to compare different utility functions across applications. NetVRM also provides a default, application-agnostic metric—the memory hit ratio—for applications that do not define their own.

**Language extension and autogeneration** (§6). NetVRM provides P4VRM, an extension to P4 [8] for developers to develop P4 programs with virtual register memory, and a P4VRM compiler to compose and compile individual P4VRM programs of different applications to one single P4 program with virtual register memory abstraction. The compiler also generates C++ APIs for efficient virtual register memory configuration in the control plane.

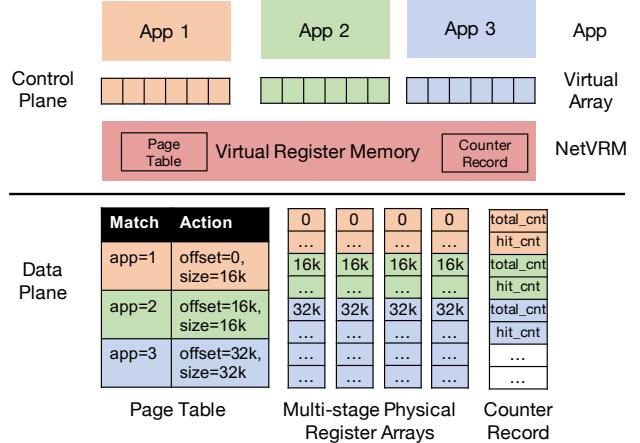


Figure 2: Virtual register memory design.

## 4 Virtual Register Memory

The register memory in the switch data plane is abstracted as register arrays for developers. The main problem of dynamically allocating memory is the coordination between multiple reg-stateful applications. Because register array definitions are hardwired in P4 programs, the code of an application has to be modified when other applications on the switch change, even if the application itself stays the same. NetVRM exposes a *virtual register memory space* to each application, which eliminates the coordination between applications. Each application is implemented with a virtual register array, without explicitly binding the register array to specific stages. As such, the application code does not need to be modified when the memory allocation changes. NetVRM is designed to manage the register memory and does not sacrifice the support of recirculation.

**Page table and counter record.** A key challenge for memory virtualization on a switch, as opposed to a traditional CPU, is that the register memory can be accessed from both the data plane and the control plane (Figure 2). It is straightforward to implement the page table in the control plane. NetVRM simply does the translation in software. Specifically, it intercepts application memory accesses, uses the page table to perform the address translation, and then calls the memory access APIs of the switch driver to update the register memory configuration.

The page table in the data plane is more complicated, because it needs to be implemented using the programmable processing elements in the data plane. Figure 2 shows the design. The page table is implemented with a match-action table, and is placed at the stage before the physical register arrays to be virtualized. The match-action table matches on the application ID and identifies the location and size of the application’s memory region (`offset` and `size`). These parameters are configured by the control plane at runtime as memory is allocated. We remark that *the page table does not introduce register memory overhead in common cases* (§7).

The counter record maintains two counters for each application, which only takes a small amount of memory. One is `total_cnt`, which tracks the total number of packets for an application. The other is `hit_cnt`, which tracks the number of packets that hit the switch register memory for each application. These counters are polled and reset periodically by the control plane to compute real-time memory hit ratios.

**Memory layout.** The memory layout partitions the physical register arrays *horizontally* across the stages. A virtual register array for an application is mapped to multiple blocks with the same start index (`offset` in the page table) and size (`size` in the page table) in each physical array. For example, in Figure 2 application 1 has a virtual array with 64K slots, which is mapped to [0, 16K] in each physical array, and application 3 has a virtual array with 128K slots, which is mapped to [32K, 64K] in each physical array.

This horizontal memory layout has three principal benefits. First, it decouples memory allocation from application code, and eliminates their static binding. The size of a virtual register array and its mapping to the physical arrays are represented by `offset` and `size` in action parameters, which can be dynamically changed at runtime, without recompiling and reloading the code in the data plane. Second, it enables fine-grained memory allocation. Because there are only a few stages (e.g., 10-20 stages) on commodity programmable switches [11, 50], our design can allocate the memory at row granularity (e.g., 8-slot granularity), which is fine-grained enough, compared with the total available slots on the switch (e.g., 512K). Third, it represents the memory layout using a small fixed-sized representation: only two variables (`offset` and `size`) per application. Although a more sophisticated memory layout might be able to achieve better space efficiency, more complex representations such as variable-length block lists would be challenging to implement efficiently in the data plane.

**Address translation.** Let the size of a virtual register array for an application be  $N$ . A virtual address  $VA \in [0, N]$  is the index of the register slot in the virtual array. The physical address  $PA$  is computed by  $PA = (VA/\text{size}, VA\%size + offset)$  after the page table, where  $VA/\text{size}$  denotes the physical array index and  $VA\%size + offset$  denotes the physical slot index in the corresponding stage. Division and modulo on arbitrary integers may not be supported in all switches. In such cases, we allocate virtual arrays with `size` to be a power of two, and implement these two operations with bit operations.

The above translation is sufficient for applications that directly access memory by  $VA$ . Besides these direct accesses, reg-stateful applications on programmable switches often use a lookup table or a hash function to access a register slot. Lookup tables use match-action tables to identify the address corresponding to a key (e.g., to find the memory location of an object in NetCache). We adapt the match-action table to hold a virtual address, then apply the  $VA$  to  $PA$  translation described above. Other applications use a

hash function to map a subset of header fields to a register slot (e.g., hashing the source IP in heavy hitter detection). While in principle the same translation approach can be used, hardware constraints on the Tofino platform mean that hash functions need to be associated with a particular address range, and adding a variable offset to the output requires an additional stage. NetVRM uses a hash function  $h\_size$ , selected during the page table lookup stage, which has output in  $[0, size]$ . Hash lookups first compute  $h\_size(pkt.hdr)$ , then, in a subsequent stage, translate that to the physical slot location:  $PA = (h(pkt.hdr)\%k, h\_size(pkt.hdr) + offset)$ , where  $k$  is the number of physical arrays.

Some applications may need large virtual slots, each of which may be larger than a physical slot. In such cases, we combine multiple physical slots to implement a virtual slot.

## 5 Dynamic Memory Allocation

We classify reg-stateful applications on a programmable network into elastic and inelastic applications based on whether an application can work with a variable amount of register memory. An inelastic application requires a fixed amount of register memory; it cannot work with less (e.g., NetChain [22]). An elastic application does not have a fixed register memory requirement. Our key observation is that most elastic applications overcome insufficient register memory with a fallback mechanism to the network control plane or the servers [23, 47]. The amount of memory typically affects application-level performance metrics (e.g., the system throughout in NetCache [23]). Although it may be possible to transform inelastic applications to elastic ones [29], we leave that to application developers. NetVRM supports both types, while only elastic applications can benefit from NetVRM’s dynamic memory allocation.

Each application is specified with four parameters: the *application type* (e.g., HH); the *subnet* in which the application will run (e.g., 10.0.0.0/8); the *utility metric*, which is either the default metric (i.e., memory hit ratio) or an application-specific one; and the *utility target* (e.g., 0.98 for memory hit ratio). For an inelastic application, the amount of required memory is specified instead of the utility metric and target. NetVRM allocates the memory to it if the requirement can be satisfied, and rejects the application otherwise.

Dynamic memory allocation is only performed for elastic applications. NetVRM periodically polls the counters from the data plane, obtains the utility of each application, and dynamically allocates the register memory between the applications based on their utilities. There is a long line of work related to network utility maximization [26, 38, 40]. NetVRM presents three particular challenges for network utility maximization, including how to define the application utility properly, how to approximate the utility functions, and how to allocate the register memory in the network, which will be demonstrated in detail as follows.

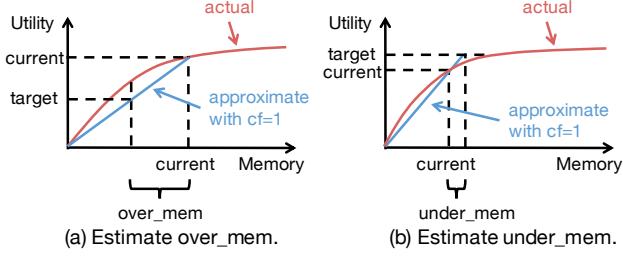


Figure 3: Utility function estimation.

### 5.1 Definition of Application Utility

Finding a proper definition of application utility is challenging, because different applications have their own application-level objectives that cannot be directly compared with each other (e.g., accuracy for a heavy-hitter detector or throughput for NetCache). NetVRM allows applications to compute their own utility metrics and report them to the allocator. Because not all the application-level metrics can be reported online (e.g., accuracy for a heavy-hitter detector), NetVRM also provides a default, generic utility definition. It is based on the observation that for many elastic applications, a register memory miss in packet processing usually affects the application-level performance, e.g., extra latency to process a packet with the fallback mechanism. Therefore, one effective utility definition is the memory hit ratio, which is the ratio of packets directly processed by the register memory in the switch. Besides being application-agnostic, this utility can be computed by tracking counters for memory hits in the data plane by NetVRM itself (§4). Moreover, the memory hit ratio is also a widely-used metric to evaluate the workload reduction for the fallback mechanism in many elastic applications on programmable networks [18, 39, 47].

### 5.2 Problem Formulation

We denote the available virtual register memory size of  $c$  switches in the network as  $M_1, M_2, \dots, M_c$ , respectively. There are  $l$  applications running in the network. Let  $i.\text{target}$  be the utility target of application  $i$ , and  $i.\text{utility}(i.m_1, \dots, i.m_c, i.T)$  be the utility function of application  $i$  where  $i.m_j$  is the memory usage of application  $i$  on switch  $j$  and  $i.T$  is the real-time traffic of application  $i$ . The network resource allocation problem is formulated as follows.

$$\begin{aligned} & \max \sum_{i=1}^l \mathbf{1}(i.\text{utility}(i.m_1, \dots, i.m_c, i.T) \geq i.\text{target}) \\ \text{s.t. } & \sum_{i=1}^l i.m_j \leq M_j, \forall j = 1, \dots, c \end{aligned}$$

The objective is to maximize the number of applications of which the utility targets can be satisfied, and the constraint is to ensure the sum of allocated memory on each switch does not exceed its memory size. We remark that this is one objective that is provided by default and has been used in sev-

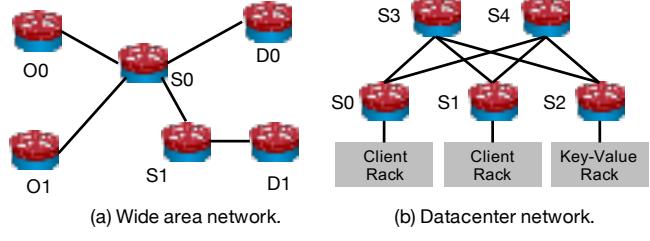


Figure 4: Examples for network-wide allocation.

eral network management scenarios [36, 37]. NetVRM also supports other objectives and memory allocation algorithms.

**Main challenge: unknown and dynamic utility functions.** The main challenge to solve the allocation problem is that the utility functions of the applications are unknown and change over time. It is true that some utility functions can be known as a priori, e.g., the worst-case accuracy and the memory requirement for sketch-based heavy hitter detection (SHH) using count-min sketch [12] can be calculated mathematically [54]. But utility functions for many applications such as HH, NO and SS (§2) are hard to know in advance. More importantly, the solution needs to adapt to real-time traffic and as the applications are started and stopped dynamically.

**Solution: online utility curve estimation without application knowledge.** In order to adapt memory allocation for the applications *without* knowing their utility function, NetVRM leverages the observation that the utility function follows diminishing returns, i.e., that it is concave, which holds for a wide range of reg-stateful network applications [18, 23, 34, 47, 58], and approximates the memory requirement for each application. Let  $i.\text{util}$ ,  $i.\text{target}$  and  $i.\text{mem}$  be the current utility, the utility target and the current memory for application  $i$ , respectively. The utility function is approximated by a polynomial function that intersects the origin. For an application  $i$  above its utility target, we use

$$i.\text{over\_mem} \leftarrow i.\text{mem} - \left(\frac{i.\text{target}}{i.\text{util}}\right)^{cf} * i.\text{mem} \quad (1)$$

to estimate the amount of memory that can be moved from  $i$  to other applications ( $i.\text{over\_mem}$ ). Because of *diminishing returns*, the utility function is *concave* and a linear function (when  $cf = 1$ ) may underestimate  $i.\text{over\_mem}$  (Figure 3(a)). We use a compensation factor  $cf$  which is set to be larger than 1 to compensate this. For an application  $i$  below its utility target, we use

$$i.\text{under\_mem} \leftarrow \left(\frac{i.\text{target}}{i.\text{util}}\right)^{cf} * i.\text{mem} - i.\text{mem} \quad (2)$$

to estimate the amount of memory to be added to  $i$  ( $i.\text{under\_mem}$ ). We use a  $cf$  larger than 1 for faster convergence (Figure 3(b)).

### 5.3 Network-Wide Register Memory Allocation

Based on the approximation in §5.2, NetVRM uses an online algorithm to move memory from over-provisioned applications (those above their utility targets) to under-provisioned applications (those below their utility targets) to maximize

---

**Algorithm 1** Network-wide memory allocation
 

---

```

1:  $new\_plan \leftarrow cur\_plan.copy()$ 
2: for application  $i$  in  $applications$  do
3:   if  $i.util \geq i.target$  then
4:      $satisfied\_list.append(i)$ 
5:      $i.over\_mem \leftarrow i.mem - (i.target/i.util)^{cf} * i.mem$ 
6:     distributed  $i.over\_mem$  to  $i.paths$  proportionally
7:   else
8:      $unsatisfied\_list.append(i)$ 
9:      $i.under\_mem \leftarrow (i.target/i.util)^{cf} * i.mem - i.mem$ 
10:    distributed  $i.under\_mem$  to  $i.paths$  inverse proportionally
11: sort  $satisfied\_list$  by  $over\_mem$  in decreasing order
12: sort  $unsatisfied\_list$  by  $i.under\_mem$  in increasing order
13: for application  $i$  in  $unsatisfied\_list$  do
14:   for path  $p$  in  $i.paths$  do
15:     sort  $p.switches$  based on  $i$ 's existence and  $s.over\_mem$ 
16:     for switch  $s$  in  $p.switches$  do
17:       allocate memory from  $satisfied\_list$  to  $p.under\_mem$ 
18:     if all paths are satisfied then
19:       update  $new\_plan$ 
20:     else
21:       move memory back to  $satisfied\_list$ 
22: return  $new\_plan$ 
  
```

---

the objective. The allocation are performed periodically to handle real-time traffic dynamics and application changes.

**Main challenge: multiple and overlapped paths of an application.** Besides the unknown and dynamic utility functions, the network-wide allocation problem is further complicated by the following two challenges. First, an application may need to handle traffic between multiple origin-destination (OD) pairs, and the traffic between each OD pair may use multiple paths. For example, in a wide area network, the operator may want to detect heavy hitters for flows between multiple OD pairs, e.g., O0-D0 and O1-D1 in Figure 4(a). In a datacenter network, the operator may want to provide in-network caching for traffic from multiple client racks to a key-value store rack, e.g., S0-S2 and S1-S2 in Figure 4(b). Datacenter networks typically use multi-path routing, e.g., path S0-S3-S2 and path S0-S4-S2 for traffic between S0 and S2. Second, different paths of an application may overlap, and thus can share their allocated memory. For example, in Figure 4(b), NetCache can be placed in S2 to save memory instead of in both S3 and S4.

**Solution: network-wide memory allocation.** At a high level, NetVRM performs network-wide memory allocation in two steps. First, NetVRM uses the utility estimation mechanism in §5.2 to estimate the required memory for each application, and decomposes  $over\_mem$  or  $under\_mem$  of each application to multiple paths. Second, it moves the memory from over-provisioned applications to under-provisioned applications. The pseudocode is shown in Algorithm 1.

The first step is to compute and decompose  $over\_mem$  or  $under\_mem$  of each application to multiple paths (line 2-10). NetVRM measures the utility (i.e., the memory hit ratio by default) and the traffic on each path. With the memory hit ratio as the utility, the utility (memory) of application  $i$  is the weighted average of its utilities (memories) by the traffic

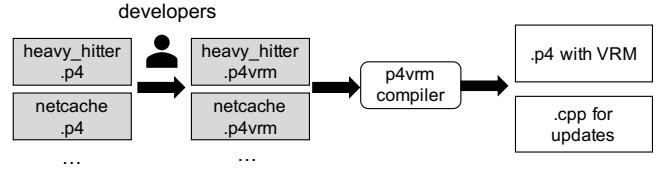


Figure 5: P4VRM compiler compiles P4VRM programs.

volume on its paths. We use the utility estimation mechanism in §5.2 to estimate  $i.over\_mem$  and  $i.under\_mem$ . Then  $i.over\_mem$  is distributed to each path in proportional to their traffic (line 6) and  $i.under\_mem$  is distributed to each path in inverse proportional to their traffic (line 10). We remark that NetVRM also allows disproportional memory allocation.

The second step is to move memory from over-provisioned applications to under-provisioned applications (line 11-21). We use a heuristic that reduces the memory for applications that are more over-provisioned first, and allocates the memory to the applications that are more likely to be satisfied first (line 11-12). For each unsatisfied application, it tries to satisfy the estimated memory requirement on each path (line 13-21). Because each path contains several switches, the algorithm needs to decide which switch to allocate memory from to satisfy the application (line 15-17). Two factors are considered in the decision, which are whether the application already has memory allocated on a switch (i.e.,  $i$ 's existence) and how much extra memory a switch has (i.e.,  $s.over\_mem$ ). These factors aim to avoid small amounts of memory scattering in many switches. If the application's requirement can be satisfied, the plan is updated (line 18-19). Otherwise, the memory is moved back to the satisfied applications (line 20-21).

To accommodate path overlapping, two extensions are required to the algorithm. First, in the utility estimation, the memory on overlapping switches is counted once for each overlapping path. Second, in memory allocation, the memory allocated to an application on overlapping switches is also counted once for each overlapping path.

**Admission control, drop and priority.** Admission control is critical when the total memory requirement exceeds the register memory size in the network. NetVRM admits one application into the network only if there is more available memory on each path than a predefined fraction of the total memory. NetVRM drops one application if it cannot meet the utility target in multiple consecutive allocation epochs. NetVRM targets elastic applications which can work even with no register memory. Thus, *if one application is rejected or dropped, it can turn to the fallback mechanism*. A malicious application with a tough utility target to satisfy would likely be dropped after a few allocation epochs. The operator can also assign custom priorities for the applications. For example, an application can be configured to not be dropped, or be assigned with a minimal amount of memory to avoid starvation when it is under-provisioned.

---

```

<p4_declaration> ::= <vrm_reg_declaration> | <vrm_bb_declaration> | ...
<vrm_reg_declaration> ::= 'vrmReg' <virt_stage> <register_declaration>
<vrm_bb_declaration> ::= 'vrmMergeable' <blackbox_declaration>
| 'vrmNonMergeable' <blackbox_declaration>
<table_declaration> ::= ...
| 'vrmMergeable' <virt_stage> <table_declaration>
| 'vrmNonMergeable' <table_declaration>
<action_function_declaration> ::= ...
| 'vrmMergeable' <action_function_declaration>
| 'vrmNonMergeable' <action_function_declaration>
<control_statement> ::= ...
| 'HIT_COUNTER';
| 'PKT_COUNTER';
<virt_stage> ::= <decimal_value>

```

---

Figure 6: The P4VRM extensions to the P4-14. Gray non-terminal nodes refer to legacy rules in P4-14.

**Memory reallocation process.** At the end of each allocation epoch, NetVRM fetches the counters from the control plane, and computes the online utilities and the new memory allocation plan. Updating the memory allocation plan results in remapping from virtual addresses to physical addresses and moving existing entries because of the remapping. There are general solutions that can be applied to ensure the consistency of memory allocation updates [24, 53]. We apply two optimizations for particular cases in NetVRM. First, network measurement applications periodically reset the state such as counters maintained by the register memory. We align the memory allocation updates with the resetting operations, so that the memory allocation can be updated without moving existing entries and does not sacrifice application correctness. Second, network applications that use lookup-table-based address translation can simply use a delta update when the memory size decreases, and allow more entries when the memory size increases. This ensures consistency because a lookup table is used for maintaining each address mapping.

## 6 Language Extension and Autogeneration

NetVRM provides P4VRM, an extension to the basic syntax and semantics of the P4 programming language [8] that supports virtual register memory abstraction and online utility measurement. Our implementation is based on P4-14, as more existing implementations are implemented in this version, but the same extensions could be applied to P4-16 as well. As shown in Figure 5, to port existing .p4 programs, developers extend them to .p4vrm programs by marking which register arrays are to be virtualized and adding the online utility measurement primitives (HIT\_COUNTER and PKT\_COUNTER) correctly according to the applications. The P4VRM compiler takes multiple .p4vrm programs as input and outputs one merged P4 program (for the data plane) with virtual memory abstraction and online utility measurement, together with the C++ APIs (for the control plane) to configure the virtual register memory efficiently.

```

+ #include "params.p4"
- vrmReg 1 register stg1 {
+ register virtual_stg1 {
    width:32;
- instance_count:8192;
+ instance_count:65536;
}
- vrmNonMergeable blackbox stateful_alu salu_stg1 {
+ blackbox stateful_alu salu_stg1 {
- reg: stg1;
+ reg: virtual_stg1;
...
}
- vrmNonMergeable action act_stg1() {
+ action act_stg1() {
- salu_stg1.execute_stateful_alu_from_hash(hash_1);
+ salu_stg1.execute_stateful_alu(params_md.slot_idx);
}
- vrmNonMergeable table tbl_stg1 {
+ table tbl_stg1 {
+ actions {act_stg1;};
+ default_action:act_stg1();
+ }

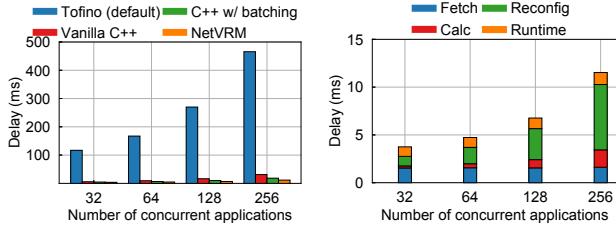
control ingress {
    if (valid(tcp) or valid(udp)) {
+     apply(set_app_id);
+     apply(set_offset_hf);
+     apply(add_offset);
+     if (params_md.app_type==0) {
        apply(tbl_stg1);
...
-         HIT_COUNTER;
+         apply(hit_counter);
...
-         PKT_COUNTER;
+         apply(pkt_counter);
+     }
}

```

Figure 7: An example of P4VRM code transformation by P4VRM compiler. ‘-’ and ‘+’ annotate the change before and after the transformation, respectively.

**Grammar.** As shown in Figure 6, P4VRM extends the P4-14 language specification [2] by introducing new keywords (vrmReg, vrmMergeable and vrmNonMergeable) to tag declarations related to a register array (register, blackbox, action, and table). It marks the register array as virtualized, and marks the related blackboxes, actions and tables that have the same logic as mergeable. It also specifies the stages at which the mergeable tables should be placed (virt\_stage). The two primitive statements (i.e., HIT\_COUNTER and PKT\_COUNTER) are used for online utility measurement. HIT\_COUNTER tracks the number of packets processed by the register memory, and PKT\_COUNTER tracks the total number of packets of the application.

**Generating merged P4 programs and C++ APIs.** To merge parsers, P4VRM compiler abstracts the packet parser of each application as a Finite State Machine (FSM) and merges the identical states into a single FSM. Then, the P4VRM compiler transforms P4VRM-introduced declarations (i.e., vrmReg, vrmMergeable and vrmNonMergeable) to P4-14 declarations (i.e., register, blackbox, action and table), and adds the additional logic for address trans-



(a) Total control loop delay vs. different implementations.  
(b) Delay breakdown for NetVRM.

Figure 8: Analysis of control loop delay.

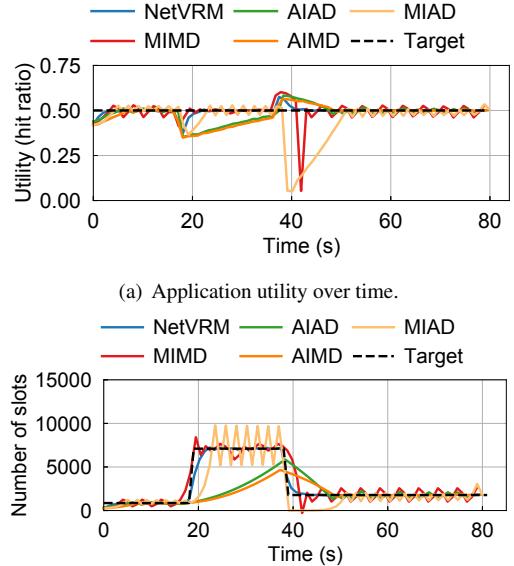
lation, as shown in Figure 7. The compiler also loads a P4 library (`params.p4`) provided by P4VRM, containing additional metadata and logic (e.g., to perform the page table lookup) and adds the appropriate invocations at the beginning of the pipeline. Finally, the compiler generates control plane APIs for resetting counters, fetching counters, resetting virtual memory and configuring the virtual memory.

**Requirement for merge.** Merging multiple reg-stateful applications needs to comply with the same resource constraints as in existing work [19, 56, 57], most notably those related to register memory (e.g., total register memory size per stage, stateful ALUs per stage). If merging violates hardware constraints, the P4VRM compiler would fail and produce no output.

## 7 Implementation

We have implemented a NetVRM prototype on a 6.5 Tbps Intel Tofino switch [50], and used commodity servers to replay traces and generate traffic. The P4 library we provide for virtual register memory support is around 500 lines of P4-14 code. The virtual register memory spans eight physical stages. Other stages are used for necessary functionalities (e.g., routing and enabling concurrent applications). We emulate four switches with the four independent pipelines of the Tofino switch. The data plane program decides which emulated switch one packet enters by checking the ingress port. The implementation batches the data plane updates, and uses multithreading to update the four pipelines simultaneously. The NetVRM control plane implementation consists of around 2200 lines of C++ code. The P4VRM compiler is built on Flex/Bison [31] and parses the .p4vrm files to build an AST. It consists of around 2000 lines of C++ and 900 lines of grammar.

**Overhead of NetVRM.** The address translation needs to be done in two stages (§4), which is realized with two tables to adjust the `slot_idx` (Figure 7). The first table (`set_offset_hf`) can be placed in the same stage with other tables (e.g., `set_app_id`) that are necessary and inevitable for concurrent applications running. The register memory in the second stage where `add_offset` is placed cannot be virtualized, which can be used for basic networking functionality and inelastic applications. In some cases, the register memory in some stages cannot be used even without



(a) Application utility over time.  
(b) Register memory consumption over time.  
Figure 9: Comparison of different algorithms to update memory allocation.

NetVRM because of the indivisibility between the virtual slot size and the number of stages. For example, if there are three physical stages available for virtualization, an application with 2-stage virtual slots can use two stages at most. Then the page table placed in the first stage does not introduce extra register memory overhead. We remark that this is a common case for many applications [18, 23, 34, 47, 54]. The extra resource needed by each application in NetVRM is only one table entry in the page table and two counters for the online utility measurement, without extra stage overhead.

## 8 Evaluation

We evaluate our NetVRM prototype in two scales. We first use microbenchmarks to examine the control loop delay and the properties of the resource allocation algorithm (i.e., stability and convergence speed). With macrobenchmarks, we demonstrate the benefits of NetVRM in combination with a variety of network applications, workload parameters, comparisons with alternative approaches and network topologies.

### 8.1 Microbenchmark

**Control loop delay.** We emulate four switches by the four independent pipelines of the Tofino switch. First, we compare the total control loop delay, i.e., the time to complete a virtual memory reallocation (§5.3), with different implementations, including the default implementation on Tofino switches which uses Python Thrift APIs, a vanilla C++ implementation, a C++ implementation with batching, and NetVRM, which incorporates both batching and multithreading. As shown in Figure 8(a), the C++ implementations are an order of magnitude more efficient than the default implementation of Tofino control plane APIs. NetVRM’s optimizations further reduce the delay by a factor of  $\sim 3$ .

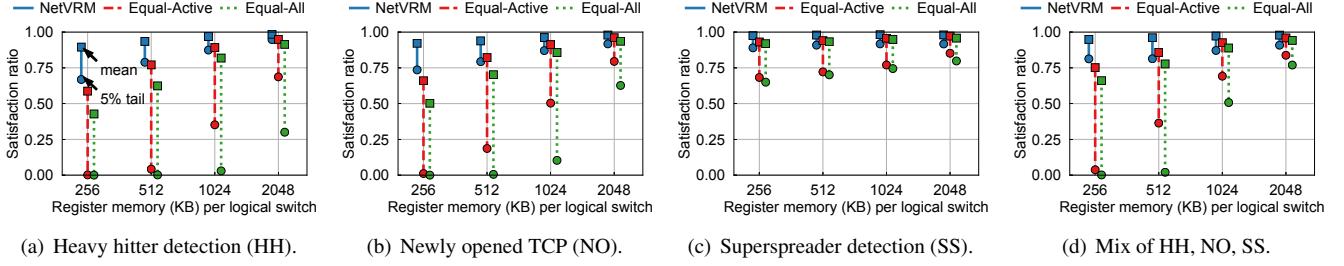


Figure 10: Satisfaction for flow-based applications in the WAN scenario.

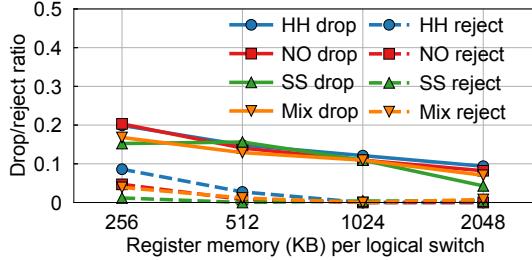


Figure 11: Drop/reject ratio of NetVRM for flow-based applications in the WAN scenario.

We further break down the control loop delay of NetVRM into four parts, i.e., *Fetch*, *Calc*, *Reconfig* and *Runtime*, and measure the latencies with different number of concurrent applications. *Fetch*, *Calc*, *Reconfig* and *Runtime* represent the time of fetching counters, calculating online utility and new memory allocation plan, configuring the page table, and the runtime overhead for resetting the state (e.g., the counters), respectively. As shown in Figure 8(b), the time of *Fetch* remains relatively constant since we use batching to fetch all the counters together where the data size does not influence the latency significantly. The time of *Calc* increases with more applications, due to the heavier overhead to compute the online utility and memory allocation plans. The time of *Reconfig* dominates the control loop delay because of the intensive updates to the data plane for four pipelines.

Due to the limit of our testbed, we only emulate four switches with one Tofino switch in our experiment. We remark that NetVRM can maintain the low control loop delay and scale in real wide area networks and datacenters with a larger number of switches for two reasons. First, *Fetch*, *Reconfig* and *Runtime*, which do not need coordination between multiple switches, can be done in different switches locally and simultaneously. Second, *Calc* needs to compute the online network-wide utility and memory allocation plans for each application which has to be done in a centralized location. Instead of doing it on the switch OS with limited computation capability in our experiment, the time of *Calc* can be reduced easily by running it in a more powerful server.

**Stability and fast convergence of NetVRM.** In this experiment, we compare NetVRM with other alternative approaches which are commonly used in network resource allocation, including AIAD, MIAD, MIMD and AIMD. Those approaches

estimate the memory requirements by increasing (decreasing) the step size additively (A) or multiplicatively (M) when the satisfaction status remains the same (changes) compared with the previous epoch. We run one NetCache [23] application on the switch and set its memory hit ratio target to be 0.5. The workload skewness is Zipf-0.99 at the beginning, then changes to Zipf-0.9 at 18 seconds, and finally changes to Zipf-0.95 at 38 seconds. Figure 9(a) and Figure 9(b) show the utility and memory usage over time, respectively. AIAD and AIMD fail to meet the utility target when the skewness becomes Zipf-0.9 because increasing the memory additively is too slow. MIAD converges slower after 38 seconds because decreasing the step size additively from a large step size is slow. MIMD has the closest performance to NetVRM, but the utility fluctuates around the utility target after convergence. NetVRM estimates the memory requirements based on the online utility (§5.2). Thus, it can react fast and more accurately to the traffic dynamics and maintain the utility above its target most of the time.

## 8.2 Macrobenchmark

**NetVRM configuration and network topology.** The default allocation epoch and measurement epoch are both one second. The default network topology is the Wide Area Network (WAN), where each application has traffic from 4 switches independently. NetVRM drops an application if it cannot meet the utility target in four consecutive epochs and rejects an application if the available memory on the switch is smaller than 1/128 of the total memory.

**Network applications.** NetVRM supports a wide range of network applications. We use five applications in the evaluation, i.e., heavy hitter detection (HH) [47], newly opened TCP connection detection (NO) [55], superspreaders detection (SS) [47], sketch-based heavy hitter detection (SHH) [54] and NetCache [23]. HH, NO and SS are flow-based applications which store precise flow records on the data plane, and evict the existing entries to the control plane upon hash collisions, following the eviction policy in TurboFlow [47]. SHH is a sketch-based application that uses approximate data structures (i.e., count-min sketch [12]) to approximate flow records. NetCache maintains hot key-value pairs on the data plane to serve a request upon a cache hit. For each application type, there can be multiple instances of this application, e.g., belonging to different clients/tenants. Each client/tenant owns

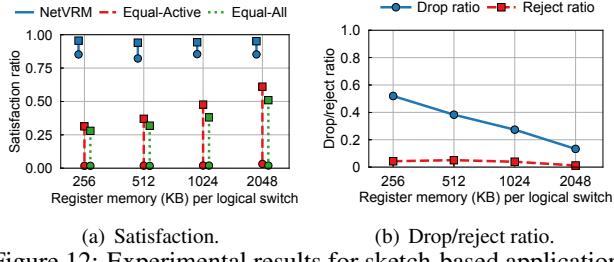


Figure 12: Experimental results for sketch-based applications (SHH) in the WAN scenario.

a /8 subnet of source IP, and can dynamically start or stop application instances within its subnet.

**Traffic traces.** The traces for measurement applications on WAN are the 2019 passive CAIDA traces [10]. The data-center traces are from Facebook’s production clusters [41]. We replay the traces via MoonGen [15]. The NetCache traffic is generated by our DPDK client according to the Zipf distribution with different skewness parameters.

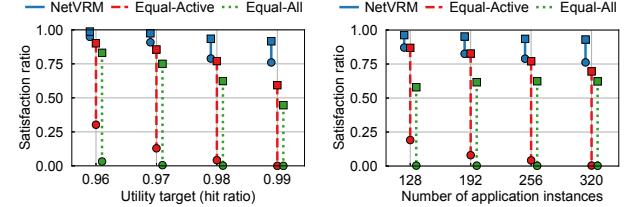
**Alternative approaches.** We compare NetVRM with two alternative approaches. (i) One is *Equal-All*, which *statically* assigns an equal amount of register memory to all applications, active or not. For example, if each application instance runs within a /8 subnet, then there are at most 256 concurrent application instances. Thus, *Equal-All* assigns 1/256 of total memory to each instance. (ii) The other is *Equal-Active*, which only assigns an equal amount of register memory to *active* instances. We emphasize that *Equal-Active* is enabled by the ability of NetVRM to dynamically allocate register memory at runtime. NetVRM further improves *Equal-Active* with the network-wide memory allocation algorithm in §5.

**Performance metric.** We use *satisfaction ratio* as the performance metric for these network applications. Each application instance has a utility target. The satisfaction ratio of an instance is the fraction of time the utility target is met during its lifetime. For each experiment, we compute the satisfaction ratio for every instance, and show the mean and 5th percentile of the satisfaction ratios across all instances. Considering the number of instances is only a few hundreds (i.e., 256), the 5th percentile catches the tail pattern in the last ten instances, while other options (e.g., 1th, 0.1th) are too limited which only show the satisfaction of the last one or two instances.

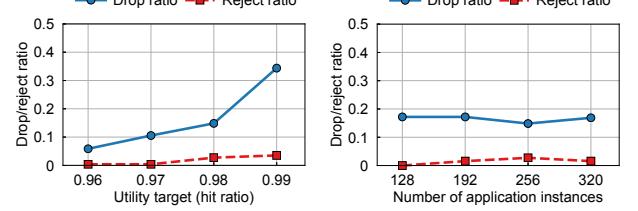
### 8.2.1 Generality

We show that NetVRM is general to a wide range of network application types in the WAN scenario.

**Setup.** We replay the CAIDA traffic on the four emulated switches as in §8.1. We deploy four types of applications including HH, NO, SS and SHH. We omit NetCache as it is not a good use case for the WAN scenario. HH maintains the flow records of the source IP and the corresponding number of packets for all the IP traffic. NO maintains the flow records of the source IP and the corresponding number of packets only for TCP SYN packets. SS records the distinct IP address



(a) Satisfaction vs. utility target. (b) Satisfaction vs. number of application instances.



(c) Drop/reject ratio vs. utility target. (d) Drop/reject ratio vs. number of application instances.

Figure 13: Impact of workload parameters.

pair (source IP and destination IP) for all the IP traffic. SHH maintains the flow records of the source IP and the threshold to be identified as a heavy hitter is set to 200. We do the following extension for a network-wide SHH: one SHH’s utility is defined as the smallest worst-case accuracy across its switches. Since each stage only supports 32-bit read and write from register memory on the data plane, each virtual slot of the three applications spans two physical stages and there are up to 256K virtual slots (i.e., 2048 KB register memory) on each switch.

By default, there are 256 application instances started in 20 minutes based on a Poisson Process and the running time of the instances follows a uniform distribution from 6 minutes to 14 minutes. The utility targets are specified by the operator based on operational requirements. The default utility target for HH, NO, SS, i.e., the memory hit ratio, is 0.98, and the default utility target for SHH, i.e., the worst-case accuracy, is 0.98. On each switch, we use a /8 instance filter and a /2 switch filter to identify the traffic to be processed by each instance. We feed the CAIDA traces into four switches simultaneously and measure the mean and 5th percentile of satisfaction across the 256 instances.

We remark that *this is only one setup of a demanding workload to stress the system*, following the similar workload pattern in [36, 37]. We show that NetVRM outperforms the alternatives with different workload parameters in §8.2.2.

**Results.** Figure 10 shows the satisfaction ratios for flow-based applications (i.e., HH, NO, SS) under different amounts of register memory. For each vertical line, the upper square end is the mean satisfaction ratio, and the lower round end is the 5th percentile satisfaction ratio, among the 256 application instances. Figure 10(a), (b) and (c) show the cases that the instances are from the same application type, and (d) shows the case that the instances are from all the three types.

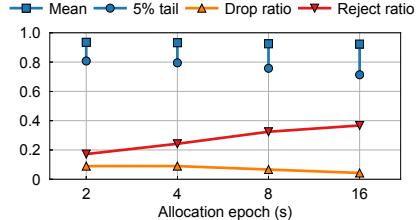


Figure 14: Impact of allocation epochs on NetVRM.

When the register memory is limited (e.g., 256 KB), NetVRM significantly outperforms *Equal-All* and *Equal-Active* on both the mean and the tail. When the register memory is abundant (e.g., 2048 KB), NetVRM is able to maintain both high mean and 5th percentile satisfaction ratios. In contrast, *Equal-All* and *Equal-Active* have comparable mean satisfaction ratios, but suffer from the tail behavior. The advantage of NetVRM over *Equal-All* and *Equal-Active* is consistent across different application types. SS uses src IP and dst IP as the hash key. Thus, it has fewer hash collisions than HH and NO, leading to a higher satisfaction ratio. Figure 11 shows the drop ratios and rejection ratios of NetVRM under the four workloads. Similarly, SS drops and rejects fewer application instances than HH and NO, because it has fewer hash collisions and less memory requirement.

Figure 12 shows that NetVRM outperforms *Equal-All* and *Equal-Active* with the sketch-based applications (i.e., SHH) as well. Compared with flow-based applications, the alternatives have lower satisfaction ratios and NetVRM drops more application instances because SHH needs more memory to guarantee the worst-case accuracy bounds.

The alternatives, *Equal-All* and *Equal-Active*, have close performance for all the applications, which means only having the mechanism of virtual register memory to allocate resources to active applications is not sufficient. The allocation algorithm that decides the memory allocation plan is critical to the performance.

### 8.2.2 Analysis of NetVRM

We analyze NetVRM by showing the impact of workload parameters and the allocation epoch. We use the same setup in §8.2.1 and show the results for the workload of HH. The findings for other application types are similar. We demonstrate the benefits of NetVRM over the local memory allocation approach in Appendix B.

**Impact of workload parameters.** Figure 13(a) shows that NetVRM is able to manage the register memory efficiently with different utility targets. With more strict targets, the three approaches have worse performance as the application instances have higher memory requirements. Figure 13(c) shows the drop ratio and reject ratio increase with more strict targets. Figure 13(b) studies the impact of the number of application instances arriving in each experiment. Fewer instances mean less resource contention, leading to higher satisfaction. NetVRM consistently outperforms the alternatives. Interestingly, Figure 13(d) shows that the drop ratio and

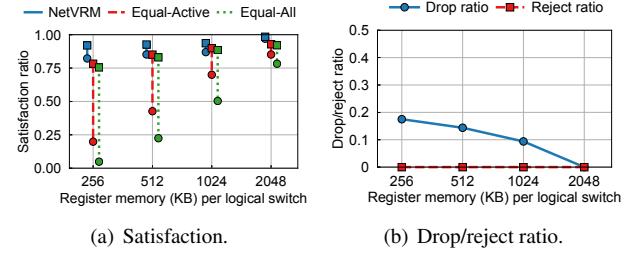


Figure 15: Experimental results in the datacenter scenario.

reject ratio are not significantly influenced by the number of instances in the evaluated range.

**Impact of the allocation epoch.** Figure 14 shows that a shorter allocation epoch leads to a slightly better performance, both in mean and tail. A longer allocation epoch can get a comparable satisfaction ratio but it comes with rejecting more applications. For example, when the allocation epoch is 16 seconds, NetVRM drops and rejects about 40% application instances, while the sum of drop ratio and reject ratio is 25% when the allocation epoch is 2 seconds.

### 8.2.3 NetVRM in Datacenter Network

**Setup.** We use the four independent pipelines of the Tofino switch to emulate four switches, and wire the four switches to build a datacenter network topology (shown in Appendix C). S0, S1 and S2 are ToR switches for client rack 1, client rack 2 and the key-value rack respectively. S3 is a spine switch connecting to them. We run two types of applications, which are HH and NetCache. HH records the number of packets of distinct four tuples (source IP, destination IP, source port, destination port). We use the Cluster-C traffic trace from Facebook’s production datacenters [41]. The trace is anonymized by hashing. The IP addresses are hashed to 64 bits and the port numbers are hashed to 32 bits in the trace. The HH application uses six physical stages to store the four tuples and one extra stage to store the number of packets. We generate pcap files from the Facebook trace, and assign the timestamps of the packets uniformly in one second as the original timestamp is at second granularity. Each application instance owns a /8 subnet. There are 318 HH instances arriving in 20 minutes based on a Poisson process, and the running time of the instances follows a uniform distribution from 6 minutes to 14 minutes. The HH instances use two paths, S0-S3-S2 and S1-S3-S2. The utility target of HH is set to 0.96.

We run two NetCache instances. NetCache1 (NC1) uses path S0-S3-S2, and NetCache2 (NC2) uses path S1-S3-S2. The tenants of NC1 and NC2 are in client rack 1 and client rack 2, respectively, which access different key-value items in the key-value rack, so they cannot share the memory on S2 and S3. NC1 and NC2 run throughout the 30-minute experiment time. The workload skewness changes between Zipf-0.99 and Zipf-0.95 every 6 minutes. The utility target is 0.5. Each virtual slot of NetCache spans 8 physical stages,

resulting in up to 64K virtual slots per switch. The NetCache instances are set to not be dropped.

**Results.** Figure 15(a) shows the satisfaction ratios of the three approaches, and Figure 15(b) shows the drop ratios and reject ratios of NetVRM. Similarly, NetVRM outperforms *Equal-All* and *Equal-Active* consistently under different amounts of register memory. It indicates that NetVRM can multiplex the register memory between different switches in a complicated scenario where applications have multiple paths and measurement applications run along with datacenter-specific applications such as NetCache.

## 9 Conclusion

We present NetVRM, a network management system to support dynamic register memory sharing between multiple concurrent applications on a programmable network. NetVRM provides a *virtual register memory abstraction* that enables register memory sharing in the switch data plane, and dynamically allocates memory for better resource efficiency and application utility. NetVRM also provides P4VRM as an extension of P4 for developing applications with virtual register memory, and a compiler to generate data plane programs and control plane APIs.

**Acknowledgments.** We thank our shepherd Laurent Vanbever and the anonymous reviewers for their valuable feedback on this paper. Xin Jin (xinjin@pku.edu.cn) is the corresponding author. Xin Jin is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. This work is supported in part by NSF grants CNS-1813487, CCF-1918757 and CNS-2008048, and the National Natural Science Foundation of China under the grant number 62172008.

## References

- [1] In-band Network Telemetry (INT) Dataplane Specification. <https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf>.
- [2] P4-14 Language Specification. <https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf>.
- [3] P4 Behavioral Model Repository. <https://github.com/p4lang/behavioral-model>.
- [4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, June 2012.
- [5] Networking and Content Delivery on AWS. <https://aws.amazon.com/products/networking/>.
- [6] Multitenant SaaS on Azure. <https://docs.microsoft.com/en-us/azure/> architecture/example-scenario/multi-saas/multitenant-saas.
- [7] Azure networking services overview. <https://docs.microsoft.com/en-us/azure/networking/fundamentals/networking-overview>.
- [8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, July 2014.
- [9] 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*, August 2013.
- [10] The CAIDA Anonymized Internet Traces 2019 Dataset. <https://data.caida.org/datasets/passive-2019/>.
- [11] Cavium XPliant. <https://www.cavium.com/>.
- [12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.
- [13] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM CCR*, April 2016.
- [14] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *ACM SOSR*, June 2015.
- [15] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *ACM SIGCOMM Conference on Internet Measurement Conference*, 2015.
- [16] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Botatei: Flexible and elastic ddos defense. In *{USENIX} Security*, 2015.
- [17] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM*, August 2013.
- [18] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.
- [19] D. Hancock and J. Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International Conference on emerging Networking EXperiments and Technologies*, 2016.

- [20] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, 2018.
- [21] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVistor: A compositional hypervisor for software-defined networks. In *USENIX NSDI*, May 2015.
- [22] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, April 2018.
- [23] 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 *ACM SOSP*, October 2017.
- [24] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, August 2014.
- [25] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, March 2016.
- [26] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *SIGCOMM CCR*, 2005.
- [27] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, 2020.
- [28] D. Kim, J. Nelson, D. R. Ports, V. Sekar, and S. Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In *ACM SIGCOMM*, 2021.
- [29] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *ACM HotNets Workshop*, 2018.
- [30] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, April 2014.
- [31] J. Levine. *Flex & Bison: Text Processing Tools*. "O'Reilly Media, Inc.", 2009.
- [32] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, April 2017.
- [33] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.
- [34] 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.
- [35] 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 *ACM SIGCOMM*, 2017.
- [36] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, August 2014.
- [37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *ACM CoNEXT*, 2015.
- [38] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *ACM SIGCOMM*, 2016.
- [39] 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*, August 2017.
- [40] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh. End-to-end transport for video qoe fairness. In *ACM SIGCOMM*, 2019.
- [41] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.
- [42] A. Sapiro, I. Abdelaziz, M. Canini, and P. Kalnis. Daiet: a system for data aggregation inside the network. In *ACM Symposium on Cloud Computing*, 2017.
- [43] A. Sapiro, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation, 2019.
- [44] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters*, 2019.
- [45] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI*, March 2017.
- [46] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *USENIX OSDI*, October 2010.

- [47] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *EuroSys*, 2018.
- [48] H. Soni, T. Turletti, and W. Dabbous. P4Bricks: Enabling multiprocessing using linker-based network data plane architecture. 2018.
- [49] R. Stoyanov and N. Zilberman. Mtpsa: Multi-tenant programmable switches. In *Proceedings of the 3rd P4 Workshop in Europe*, 2020.
- [50] Intel Tofino. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html>.
- [51] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud Workshop*, 2020.
- [52] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang. Accelerated service chaining on a single switch asic. In *ACM HotNets Workshop*, 2019.
- [53] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *ACM SIGCOMM*, August 2020.
- [54] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *USENIX NSDI*, 2013.
- [55] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with netqre. In *ACM SIGCOMM*, 2017.
- [56] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.
- [57] P. Zheng, T. Benson, and C. Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.
- [58] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. In *Proceedings of the VLDB Endowment*, November 2019.

## A Diminishing Return Examples

Figure 16 demonstrates the diminishing returns for four applications. The first three are measurement applications: heavy hitter detection (HH) [54], newly opened TCP connection detection (NO) [55], superspreaders detection (SS) [54]. These applications store flow records in the data plane; hash collisions caused by inadequate memory require additional control plane processing. The fourth, NetCache [23] caches hot objects in the switch data plane to improve the throughput of a key-value store. The utility is measured using memory hit ratio. We evaluate the measurement applications (Figure 16(a–c)) on traffic from different subnets of the 2019 passive CAIDA trace [10], and NetCache on a synthetic Zipf workload with different skewness parameters (Figure 16(d)).

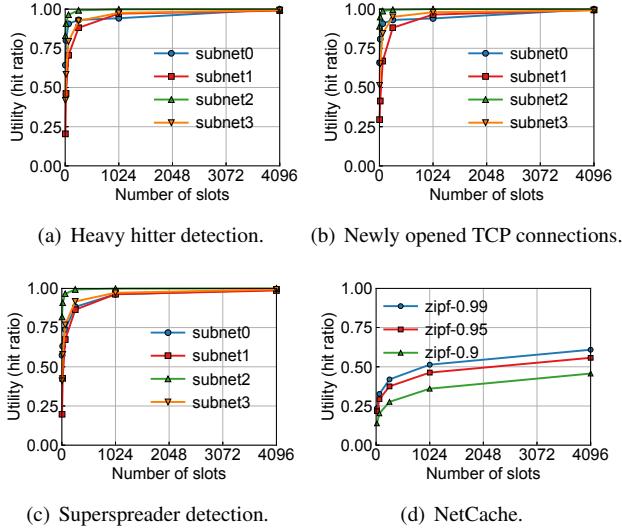


Figure 16: Examples for the diminishing returns of the utility curves in reg-stateful network applications.

## B Additional Evaluation Results

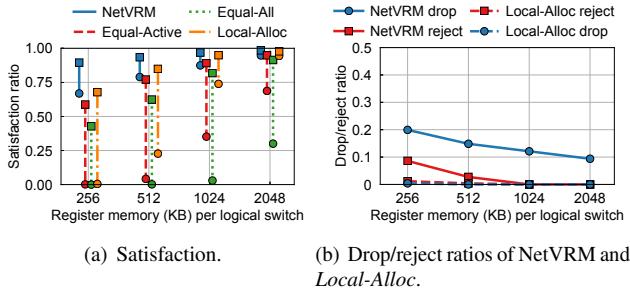


Figure 17: Comparison with *Local-Alloc*.

**Comparison with local memory allocation.** Besides the *Equal-all* and *Equal-Active*, we also compare NetVRM with *Local-Alloc* which only does memory allocation and makes drop/reject decisions on individual switches. One

application is counted as drop/reject only after all the four switches have decided to drop/reject it. We report the results for HH workload. The findings for other application types are similar. Figure 17 shows that *Local-Alloc* has better performance than *Equal-all* and *Equal-Active*, but is still worse than NetVRM because it fails to capture network-wide information and makes sub-optimal allocation and drop/reject decisions.

## C Network Topology in Datacenter Scenario

We wire the four emulated switches to build a datacenter network topology, as shown in Figure 18, to evaluate the performance of NetVRM in the datacenter scenario.

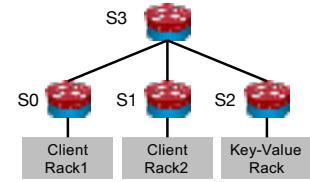


Figure 18: Datacenter topology for evaluation.

# SwiSh: Distributed Shared State Abstractions for Programmable Switches

Lior Zeno<sup>\*</sup> Dan R. K. Ports<sup>†</sup> Jacob Nelson<sup>†</sup> Daehyeok Kim<sup>†</sup> Shir Landau Feibish<sup>‡</sup> Idit Keidar<sup>\*</sup>  
Arik Rinberg<sup>\*</sup> Alon Rashelbach<sup>\*</sup> Igor De-Paula<sup>\*</sup> Mark Silberstein<sup>\*</sup>

<sup>\*</sup>Technion <sup>†</sup>Microsoft Research <sup>‡</sup>The Open University of Israel

## Abstract

We design and evaluate *SwiSh*, a *distributed shared state management* layer for data-plane P4 programs. SwiSh enables running scalable stateful distributed network functions on programmable switches entirely in the data-plane. We explore several schemes to build a shared variable abstraction, which differ in consistency, performance, and in-switch implementation complexity. We introduce the novel *Strong Delayed-Writes (SDW)* protocol which offers consistent snapshots of shared data-plane objects with semantics known as  $r$ -relaxed strong linearizability, enabling implementation of distributed concurrent sketches with precise error bounds.

We implement strong, eventual, and SDW consistency protocols in Tofino switches, and compare their performance in microbenchmarks and three realistic network functions, NAT, DDoS detector, and rate limiter. Our results show that the distributed state management in the data plane is practical, and outperforms centralized solutions by up to four orders of magnitude in update throughput and replication latency.

## 1 Introduction

In recent years, programmable data-plane switches such as Intel’s Tofino, Broadcom’s Trident, and NVIDIA’s Spectrum [9, 33, 60] have emerged as a powerful platform for packet processing, capable of running complex user-defined functionality at Tbps rates. Recent research has shown that these switches can run sophisticated network functions (NFs) that power modern cloud networks, such as NATs, load balancers [40, 57], and DDoS detectors [45]. Such data-plane implementations show great promise for cloud operators, as programmable switches can operate at orders of magnitude higher throughput levels than the server-based implementations used today, enabling a massive efficiency improvement.

A key challenge remains largely unaddressed: realistic data center deployments require NFs to be *distributed over multiple switches*. Multi-switch execution is essential to correctly process traffic that passes through multiple network paths,

to tolerate switch failures, and to handle higher throughput. Yet, building distributed NFs for programmable switches is challenging because most of today’s NFs are *stateful* and need their state to be consistent and reliable. For example, a DDoS detector may need to monitor traffic coming from multiple locations via several switches. However, it cannot be implemented by routing all traffic through a single switch since it is inherently not scalable. Instead, it must be implemented in a distributed manner. Furthermore, in order to detect and mitigate an attack, a DDoS detector must aggregate per-packet source statistics across all switches in order to correctly identify super-spreaders sending to too many destinations. Similarly, in multi-tenant clouds, per-user policies, such as rate limiting, cannot be implemented in a single switch because user’s VMs are often scattered across multiple racks, so the inter-VM traffic passes through multiple switches.

Distributed state management is, in general, a hard problem, and it becomes even harder in the context of programmable data-plane switches. In the “traditional” host-based NF realm, several methods have been proposed to deal with distributed state. These include remote access to centralized state storage [39] and distributed object abstractions [77], along with checkpoints and replication mechanisms for fault tolerance [64, 71]. Unfortunately, few of these techniques transfer directly to the programmable switch environment. These switches have the capability to modify state on *every* packet, allowing them to effectively implement stateful NFs. However, distributing the NF logic across multiple switches is extremely challenging as it requires synchronizing these frequent changes under harsh restrictions on computation, memory and communication.

Existing systems that implement NFs over multiple switches do so by designing ad hoc, application-specific protocols. Recent work on data-plane defense against link flooding [36], argues for data-plane state synchronization among the switches, but provides no consistency guarantees. While applicable in this scenario, it would not be enough in other applications, as we discuss in our analysis (§4). A more common solution, usually applied in network telemetry systems,

is to periodically report the per-switch state to a central controller [1, 6, 18, 25, 26, 29, 49, 78]. Such systems need to manage the state kept on each switch and to determine when and how the central controller is updated – navigating complex trade-offs between frequent updates leading to controller load and communication overhead versus stale data leading to measurement error. In contrast to these approaches, we seek a solution that supports general application scenarios *without relying on a central controller in failure-free runs*, while allowing all switches to take a *consistent action as a function of the global state*, e.g., to block a suspicious source in the DDoS detector.

We describe the design of such a general distributed shared state mechanism for data-plane programs, **SwiSh**. Inspired by distributed shared memory abstractions for distributed systems [41, 48], SwiSh provides several replicated shared variable abstractions with different consistency guarantees. At the same time, SwiSh is tailored to the needs of NFs and co-designed to work in a constrained programmable switch environment.

Our analysis reveals three families of NFs that lend themselves to efficient in-switch implementation, with distinct consistency requirements. For each family we explore the triple tradeoff between consistency, performance, and complexity. We design (1) Strong Read-Optimized (SRO): a strongly consistent variable for read-intensive applications with low update rates, (2) Eventual Write-Optimized (EWO): an eventually-consistent variable for applications that can tolerate inconsistent reads but require frequent writes, and (3) Strong Delayed-Writes (SDW): a novel consistency protocol which efficiently synchronizes multi-variable snapshots across switches while providing a consistency and correctness guarantee known as *r*-relaxed strong linearizability [27].

SDW is ideal for implementing concurrent sketches, which are popular in data-plane programs [12, 13, 24, 30, 35, 38, 51–54, 78, 83]. Unlike eventually consistent semantics, the *r*-relaxed strong linearizability offered by SDW enables principled analysis of concurrent sketches. This property enables the derivation of precise error bounds and generalizes to different sketch types, such as non-commutative sketches [67].

Implementing these abstractions efficiently in a switch is a challenge, and it involves judicious choice of hardware mechanisms and optimization targets. Our main ideas are: (1) *minimizing the buffer space* due to the scarcity of switch memory, even at the expense of higher bandwidth; (2) using the *in-switch packet generator* for implementing reliable packet delivery and synchronization in the data-plane.

We fully implement all the protocols in Tofino switches and devise reusable APIs for data-plane replication. We evaluate the protocols both in micro-benchmarks and in three real-world distributed NFs: a rate limiter, a network address translator (NAT) and a DDoS detector. Our novel SDW protocol achieves micro-second synchronization latency and offers about four orders of magnitude higher update rates compared to a central controller or SRO. We show that SDW (1) achieves

stable 99th percentile replication latency of  $6\mu\text{sec}$  when running on four programmable pipes (two per switch), thus sharing state both among local and remote pipes; (2) scales to 32 switches when executed in a large-scale emulation and fits switch resources even for 4K switches; (3) requires linear number of replication messages in state size which is independent from the number of actual updates to the state.

We show that SDW is instrumental to achieving high performance in applications: the centralized controller fails to scale under growing application load, whereas SDW-based versions show no signs of performance degradation.

This paper extends our workshop paper [82] by introducing the SDW protocol, as well as providing an implementation and evaluation of SRO and EWO.

In summary, this work makes the following contributions:

- Analysis of memory consistency requirements and access patterns of common NFs suitable for in-switch execution,
- Design and implementation of strongly- and eventually consistent shared variables, as well as a new SDW consistency protocol specifically tailored for in-switch implementation, which guarantees consistent snapshots and provably provides *r*-relaxed strong linearizability which facilitates implementation of concurrent sketches,
- An implementation and evaluation of three distributed NFs on Tofino switches demonstrating the practicality and utility of the new abstractions.

## 2 Background: Programmable Switches

The protocol independent switch architecture (PISA) [8] defines two main parts to packet processing. The first is the parser which parses relevant packet headers, and the second is a pipeline of match-and-action stages. Parsed headers and metadata are then processed by the pipeline. The small ( $\sim 10$  MB) switch memory is shared by all pipeline stages. Often, switches are divided into multiple independent pipes [34], each serving a subset of switch ports. From the perspective of in-switch applications, the pipes appear as different switches, so stateful objects are not shared between them.

PISA-compliant devices can be programmed using the P4 language [73]. P4 defines a set of high-level objects that consume switch memory: tables, registers, meters, and counters. While tables updates require control-plane involvement, all other objects can be modified directly from the data-plane.

A data-plane program processes packets, and then can send them to remote destinations to the control-plane processor on the switch, or to the switch itself (called *recirculation*).

Switches process packets atomically: a packet may generate several local writes to different locations, and these updates are atomic in the sense that the next processed packet will not see partial updates. Single-row control-plane table updates are atomic w.r.t. data-plane [74]. These properties allow us to implement complex distributed protocols with concurrent state updates without locks.

Although not a part of PISA, some switches add packet generation support. Packet generators can generate packets directly into the data-plane. For example, the Tofino Native Architecture (TNA) [34] allows generation of up to 8 streams of packets based on templates in switch memory. The packet generator can be triggered by a timer or by matching certain keys in recirculated packets.

### 3 Motivation

#### The Case for Programmable Switches as NF Processors.

The modern data center network incorporates a diverse array of NFs beyond simple packet forwarding. Features like NAT, firewalls, load balancers, and intrusion detection systems are central to the functionality of today’s cloud platforms. These functions are stateful packet processing operations, and today are generally implemented using software middleboxes that run on commodity servers, often at significant cost.

Consider an incoming connection to a data center service. It may pass through a DDoS detection NF [3, 58], which blocks suspicious patterns. This service is stateful; it collects global traffic statistics, e.g., to identify “super-spreader” IPs that attempt to flood multiple targets. Subsequently, traffic may pass through a load balancer, which routes incoming TCP connections to multiple destination hosts. These are stateful too: because subsequent packets in the same TCP connection must be routed to the same server (a property dubbed per-connection consistency), the load balancer must track the connection-to-server mapping. Both DDoS detectors and load balancers are in use at major cloud providers [19, 61], and handle a significant fraction of a data center’s incoming traffic. Implemented on commodity servers, they require large clusters to support massive workload.

Programmable data-plane switches offer an appealing alternative to commodity servers for implementing NFs at lower cost. Researchers have shown that they can be used to implement many types of NFs. For data center operators, the benefit is a major reduction in the cost of NF processing. Whereas a software-based load balancer can process approximately 15 million packets per second on a single server [19], a single switch can process 5 billion packets per second [33]. Put another way, a programmable switch has a price, energy usage, and physical footprint on par with a single server, but can process *several hundred times* as many packets.

**Distributed Switch Deployments.** Prior research focused on showing that NFs can be implemented on a single switch [45, 57]. However, realistic data center deployments universally require multiple switches. We see two possible deployment scenarios. The NF can be placed in switches in the network fabric. For example, in order to capture all traffic, the load balancer would need to run on all possible paths, e.g., by being deployed on every core switch or every aggregation switch. Alternatively, a cluster of switches (perhaps located near the ingress point) could be used to serve as NF accelerators. Both

are inherently distributed deployments: they require multiple switches in order to (1) scale out, (2) tolerate switch failures, and (3) capture traffic across multiple paths.

The challenge of a distributed NF deployment stems from the need to manage the global state shared among the NF instances, which is inherent to distributed stateful applications. Specifically, packet processing at one switch may require reading or updating variables that are also accessed by other switches. For example, the connection-to-server mapping recorded by the load balancer must be available when later packets for that connection are processed – even if they are processed by a different switch, or the original switch fails. Similarly, a rate limiter would need to track and record the total incoming traffic from a given IP, regardless of which switch is processing it.

SwiSh provides a shared state mechanism capable of supporting global state: any global variable can be read or written from any switch. SwiSh transparently replicates state updates to other switches for fault tolerance and remote access. In case of state locality, only a subset of the switches would replicate that state [82].

**The Case for Data-Plane Replication.** Control-plane mechanisms are commonly used for replicating the switch state [7, 11, 43, 56]. However, the scalability limitations of this approach have been well recognized, and several recent works focus on improving it by distributing the control-plane logic across a cluster of machines or switches [43, 81]. SwiSh proposes instead to replicate the state in the data plane.

Data plane replication enables supporting distributed NFs that read or modify switch state on *every packet*. This new capability of programmable data-plane switches allows implementations of more sophisticated data-plane logic than traditional control-plane SDN.

As we will see in §4, applications use state in diverse ways. Some are read-mostly; others update state on every packet. Some require strong consistency among switches to avoid exposing inconsistent states to applications (e.g., a distributed NAT must maintain correct mappings to avoid packet loss), while others can tolerate weak consistency (e.g., rate limiters that already provide approximate results [63]). SwiSh provides replication mechanisms for different classes of data that operate at the speed of the switch data-plane.

At the same time, data-plane replication offers an opportunity to build a more efficient replication mechanism without additional control-plane processing servers. Furthermore, data-plane replication can take advantage of unique programmable hardware characteristics that are not available in a traditional control-plane. For example, the atomic packet processing property enables a multi-location atomic write to the shared state. We leverage this feature to enable fast processing of acknowledgments entirely in the data-plane for our strongly-consistent replication protocol (§6.1).

**Control-plane replication is not enough.** Managing a globally shared state in a programmable data-plane switch requires

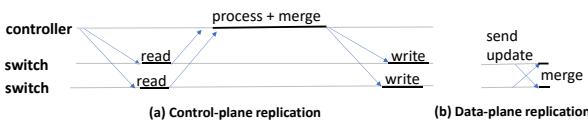


Figure 1: Data-plane vs. control-plane replication

a new approach: replication protocols that run in the control-plane cannot operate at this rate at scale.

Figure 1 shows the cycle performed by a controller to synchronize between switches, and contrasts that with data-plane replication. The controller periodically queries the switches, collects information, processes it, and sends the updates back. Merely reading and updating the register states in switches is quite slow. We measured an average latency of 507 msec to read a sketch with 3 rows each with 64K 4-byte registers from the on-switch control-plane;<sup>1</sup> updates are similar. This latency limits the rate at which the data can be retrieved from switches.

Moreover, the central controller may become the bottleneck quite quickly. For example, recent work on DDoS detection that used a central controller to query switches reported a maximum update rate of once in 5 seconds [53] because it could not accommodate faster updates.

In contrast, data-plane replication reads from and writes to registers much faster: we measured 486  $\mu$ seconds to read the same sketch from the data-plane, which is over three orders-of-magnitude faster than the control-plane access. Further, in-switch processing time is negligible as well.

These properties make data-plane replication an obvious choice for building stateful distributed NFs.

## 4 Application Consistency Requirements

We study the access patterns and consistency requirements of a few typical NF applications that have been built on PISA switches. Table 1 summarizes the results.

We identify three families of consistency requirements:

1. **Strong consistency:** Workloads cannot tolerate inconsistency between switches – a read must see a previous write. These are usually read-intensive workloads that can tolerate infrequent, but expensive writes;
2. **Weak (eventual) consistency:** Mixed read/write workloads tolerate arbitrary inconsistency;
3. **Bounded-delay consistent snapshots:** Mixed read/write workloads that tolerate inconsistency for a bounded time – a read must see all but a bounded number of previous writes, yet require that *all switches* read from a consistent state. These requirements are typical for sketches.

Below, we describe how these consistency requirements arise in several in-switch applications.

<sup>1</sup>We use BfRt API (C++) and average over 100 iterations.

## 4.1 Strong Consistency

**Network Address Translators (NATs)** share the connection table among the NF instances. The table is queried on every packet, but updated when a new connection is opened; table rows require strong consistency, or it may lead to broken client connections in case of multi-path routing or switch failure. Also, NATs usually manage a pool of unassigned ports; however, the pool can be partitioned among the switches into non-overlapping ranges to avoid sharing.

**Stateful firewalls** monitor connection states to enforce context-based rules. These states are stored in a shared table, updated as connections are opened and closed, and accessed for each packet to make filtering decisions. Like the NAT, the firewall NF requires strong consistency to avoid incorrect forwarding behavior.

**L4 load balancers** [57] assign incoming connections to a particular destination IP, then forward subsequent packets to the appropriate destination IP. Per-connection consistency requires that once an IP is assigned to a connection, it does not change, implying a need for strong state consistency.

**Observation 1.** These workloads require strong consistency, but they update state infrequently, making a costly replication protocol more tolerable. Moreover, most of these examples use switch tables that should be modified *through the control-plane*, naturally limiting their update rate. For example, the NAT NF uses control-plane to update the connection table. We leverage this observation when designing the replication protocol for this class of NFs.

## 4.2 Weak (Eventual) Consistency

**Rate limiters** restrict the aggregated bandwidth of flows that belong to a given user. The application maintains a per-user meter that is updated on every packet. The meters are synchronized periodically to identify users exceeding their bandwidth limit and to enforce restrictions. Maintaining an exact network-wide rate across all switches would incur a very high overhead and is therefore unrealistic. So rate limiters can tolerate inconsistencies, but the meters must be synchronized often enough [63] to minimize discrepancy.

**Intrusion prevention systems (IPS)** [47] monitor traffic by continuously computing packet signatures and matching against known suspicious signatures. If the number of matches is above a threshold, traffic is dropped to prevent the intrusion. This application can tolerate transient inconsistencies: it is acceptable for a few malicious packets to go through immediately after signatures are updated.

**Observation 2.** Some NFs tolerate weakly consistent data, potentially affording simpler and more efficient replication protocols. However, as we will describe next, other functions may defer the writes to be once in a window, but do require to have a consistent view of prior writes among all the switches.

	Application	State	Write frequency	Read frequency
Strong consistency	NAT	Translation table	New connection	Every packet
	Firewall	Connection states table	New connection	Every packet
	L4 load-balancer	Connection-to-DIP mapping	New connection	Every packet
Weak consistency	Intrusion prevention system	Signatures	Low	Every packet
	Rate limiter	Per-user meter	Every packet	Every window
Bounded delay consistent snapshot	DDoS detection	Sketch	Every sampled packet	Every packet
	Microburst detection	Sketch	Every packet	Every window

Table 1: NFs classified by their access pattern to shared data and their consistency requirements.

### 4.3 Bounded-Delay Consistent Snapshots

We assign mixed read/write applications that use data sketches to this class. Data sketches are commonly used in data-plane programs [12, 13, 24, 30, 35, 51, 52, 54, 78]. They are probabilistic data structures that efficiently collect approximate statistics about elements of a data stream.

Below we consider two examples of sketch-based NFs.

**Microburst detection** identifies flows that send a lot of data in a short time period. ConQuest [13] is a recent sketch-based system for a single switch, which uses a sliding window mechanism composed of a group of Count-Min sketches (CMS) [14]. At most one sketch is updated on every packet.

**DDoS detection** [45] requires tracking the frequency of source and destination IPs using a CMS with bitsets [80]. The sketch is updated on every packet, but sampled periodically to trigger an alarm when IP frequencies cross a threshold.

Strongly consistent read-optimized protocols are too costly for such workloads due to their write-intensive nature. Fortunately, because a data sketch is inherently approximate, it does not require strong consistency – it is acceptable for a query to miss some updates. Moreover, sketches are typically *stream-order invariant* [67], meaning that the quantity they estimate (such as number of unique sources, heavy hitters, and quantiles) does not depend on the packet order.

At the same time, sketches generally cannot tolerate weak consistency either. With no guarantee of timeliness, sketches might be useless. A DDoS attack might be over by the time it is detected. Moreover, the attack might be detected at one location much earlier than it is detected at another, leading to an inconsistent response. Furthermore, sketches have known error bounds (see [15] and others). These bounds are violated if updates are arbitrarily delayed [27, 66], making it hard to reason about the impact of sketch errors on the application.

**Observation 3.** Sketches require a *bounded-delay consistent snapshot* consistency level. Formally, it provides *r-relaxed strong linearizability* (Appendix A), which supports sketch applications with provably bounded error. Intuitively, *r*-relaxed strong linearizability guarantees that accesses to shared data are equivalent to a sequential execution, except that each query may “miss” up to *r* updates. SwiSh supports this consistency level using its novel Strong Delayed-Write (SDW) protocol,

which provides a consistent snapshot of the sketch at all the replicas, while delaying reads until such a snapshot is constructed.

## 5 SwiSh Abstractions

SwiSh provides the abstraction of shared variables to programmable switches. This section describes the interface and the types of semantics it offers for shared data.

**System model.** We consider a system of many switches, each acting as a replica of shared state. Switches communicate via the network, and we assume a standard failure model: packets can be dropped, duplicated and arbitrarily re-ordered, and links and switches may fail. Since switches are comprised of multiple independent pipes with per-pipe state (§2), we consider a pipe rather than a switch, a node in the protocol. We use the terms pipe and switch interchangeably.

**Data model.** The basic unit of shared state is a *variable*, associated with a unique key, which exposes an API for updating the variable (potentially using general read-modify-write functions), and reading it. The API is thus available on all switches, and variables are read and updated through a distributed protocol. SwiSh supports three types of variables which have different semantics and are accessed through different protocols:

1. *Strong Read-Optimized (SRO)* variables provide strong consistency (linearizability);
2. *Eventual Write-Optimized (EWO)* variables have low cost for both reads and writes, but provide only eventual consistency;
3. *Strong Delayed-Writes (SDW)* variables provide strong consistency (linearizability), but expose writes (even to the local replica) only after their values have been synchronized across the replicas.

We require that, no matter which semantics are used, all variables eventually converge to a common state. To this end, we require that variables be *mergeable*. We consider two merging policies: LWW as a general method, and Conflict-Free Replicated Data Types (CRDTs) as specialized mergeable data types that implement common data structures that are used in NFs. A general way to merge variables is to assign

an order to updates and apply a last-writer-wins (LWW) policy. The merge function applies an update if and only if its version number is larger than the local one. Unique version numbers can be obtained by using a switch ID as a tie breaker in addition to a timestamp attached to each write request.

In some cases, updates can be merged systematically. These are discussed in the literature of Conflict-Free Replicated Data Types (CRDTs), which offer *strong eventual consistency* and *monotonicity* [69]. Monotonicity prevents counter-intuitive scenarios such as an increment-only counter decreasing.

Counters are a natural application for this technique, as they are common in NFs (§4) and have a straightforward CRDT design. An increment-only counter can be implemented by maintaining a *vector* of counter values, one per switch. To update a counter, a switch increments its own element; to read the result, it sums all elements. To merge updates from another switch, a switch takes the largest of the local and received values for each element. Further extensions support decrement operations [69].

Variables may be used to store different data types, such as array entries, read/write variables, sets, and counters. They are implemented using appropriate stateful P4 objects.

## 6 In-Switch Replication Protocols

Below we assume that switches do not fail; we relax this assumption in §6.4.

### 6.1 Strong-Read Optimized (SRO)

The SRO protocol is based on chain replication [76], as shown in Figure 2a, adapted to an in-switch implementation with the following key difference: instead of contacting the tail for its latest version and keeping multiple versions per variable, we forward reads to pending writes to the tail.

SRO provides per-variable linearizability [28], because writes are blocking and reads concurrent to writes are processed by the tail node. Its write throughput is limited by the need to send packets through the control plane.<sup>2</sup> Note, however, that many read-intensive NFs already require control plane involvement for their updates, such as NATs, firewall and load balancers [57].

A variation of this protocol, used in many systems, including CRAQ [72] and ZooKeeper [31], reduces the read latency by performing local reads, yet offers weaker semantics [46].

### 6.2 Eventual Write-Optimized (EWO)

Both variants of the read-optimized protocol have a high write cost. Because supporting both strong consistency and fre-

<sup>2</sup>NetChain [37] implements chain replication entirely in the data plane. The difference is that NetChain is a service and clients are responsible for retrying operations. Our switches are effectively the “clients” and must buffer output packets and retry requests.

quent updates is fundamentally challenging, we offer relaxed-consistency variables. This is acceptable for many write-intensive applications, as discussed in §4.

Reads from EWO variables are performed locally, and writes are applied asynchronously. That is, when a switch receives a packet  $P$  that modifies state, it modifies its local state, emits any output packet  $P'$  immediately, and asynchronously sends a write request to all other switches (Figure 2b). A more sophisticated version can employ batching to avoid flooding the network with updates, and instead send the write request after accumulating several updates.

Unlike SRO, we do not delegate the problem of reliable write delivery to the control plane because it does not scale for write-intensive workloads. Instead, switches periodically synchronize each EWO variable from the data plane. This design choice avoids expensive buffering and re-transmission logic in the data-plane.

Periodic synchronization overcomes the issue of lost packets. As updates to EWO variables are idempotent, packets can be arbitrarily duplicated with no effect. Finally, due to updates being commutative, packet reordering has no effect.

We note that this protocol is simple, but it leads to inconsistent replicas and would incur high bandwidth overheads. With over-subscribed links [23], excessive replication traffic would only worsen the congestion. The following protocol overcomes these limitations.

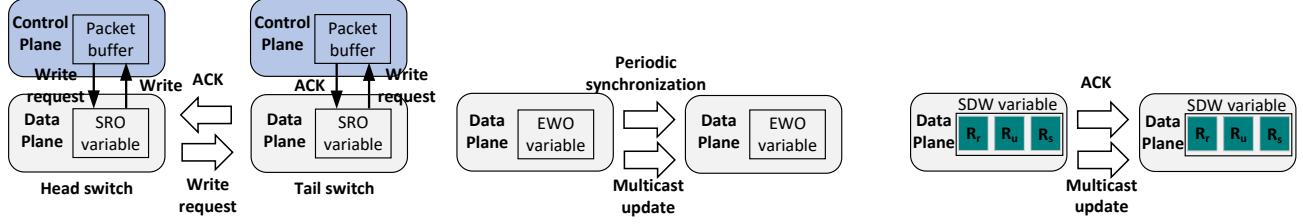
### 6.3 Strong Delayed-Writes (SDW)

As explained in §4, certain NFs tolerate inconsistencies among switches, but require state convergence within a bounded time. For such NFs, SwiSh offers *strong delayed-writes* (SDW) variables, ensuring semantics known as  $r$ -relaxed strong linearizability [27]. These semantics guarantee that every read of a variable observes all but a bounded number of updates. If the variable is used to store a data sketch, then  $r$ -relaxed strong linearizability often directly implies error bounds on the sketch’s estimate [67].

SwiSh batches updates into windows, and synchronizes window advancement (Figure 2c). The complete protocol and its analysis appear in Algorithm 1 in Appendix A; below is an informal overview.

To distribute a variable  $R$ , each switch maintains three objects holding copies of  $R$ :  $R_u$ ,  $R_r$  and  $R_s$ . At any given time,  $R_u$  is updated,  $R_r$  is queried, and  $R_s$  is synchronized (merged) across switches. The objects’ roles are switched in a round-robin manner on window advancement.

All switches run the same protocol. At the start of a window, all switches send the contents of  $R_s$  to all the others. Any (local) update is applied to  $R_u$ , and any query is executed on  $R_r$ . Once a switch receives  $R_s$  from all other switches, and furthermore receives ACKs from all other switches that they received its  $R_s$ , it advances to the next window.



(a) SRO: Based on chain replication. Relies on control-plane for packet buffering.

(b) EWO: Updates are broadcast. Switches periodically send their state for reliability.

(c) SDW: Updates are sent in rounds. Switches advance to the next round after receiving ACKs and updates from others.

Figure 2: A high-level overview of in-switch replication protocols.

On window advancement, the objects are rotated, so  $R_u$  becomes the new synchronization variable  $R'_s$ ,  $R_r$  is merged into  $R_s$  and then cleared – it becomes the new update object  $R'_u$ , and the synchronized buffer  $R_s$  becomes the new read buffer  $R'_r$ . Thus, after the synchronization of window  $w$  completes,  $R'_u$  is empty and ready to accumulate updates of window  $w+1$ ,  $R'_r$  reflects all updates that occurred in all switches in all windows up to  $w-1$ , and  $R'_s$  reflects all updates done in windows up to  $w-1$  in all switches, as well as local updates done in window  $w$ .

Crucially, as we prove in Appendix A, this protocol *guarantees* that a query in some window  $w$  sees all updates occurring in all windows  $\leq w-2$ . We also prove that, by bounding the number of updates in a window to  $B$ , every query sees all but at most  $2NB$  updates that occur before it, where  $N$  is the number of switches.

**Multi-variable snapshots.** Another advantage of the window protocol is that it allows applications to take *consistent snapshots* [59] over a collection of SDW variables by advancing the window simultaneously for all of them. This means that we can support multi-variable queries (for instance, collecting an array of counters as used in a CMS), and ensure that all queries see update batches in a consistent order. Thus, given two updates  $u_1$  and  $u_2$  occurring in different switches, it is impossible for a query at one switch to see a state reflecting only  $u_1$  (and not  $u_2$ ) while a query at another switch sees only  $u_2$  (and not  $u_1$ ).

## 6.4 Handling Failures

We now consider fail-stop switch failures. We assume that a central controller can detect which switches have failed.

**SRO.** When a switch fails, the chain becomes partitioned. First, we reconnect the chain by bypassing the failed node; if the failed switch is the head, the second node in the chain assumes its responsibility. This follows the standard chain replication protocol. A new switch is added to the end of the chain. It starts to process writes, but does not replace the tail

until the data transfer to it is complete. This requires control plane involvement.

The control plane on one of the switches takes a snapshot of its state, and then resends all pending write requests through the normal data plane protocol. These writes contain the sequence number at the time of the snapshot to prevent overwriting newer values with old ones. Once the new switch has acknowledged all writes, it replaces the tail.

**EWO.** Because live replicas regularly synchronize their entire state, this synchronization protocol is inherently robust to switch and link failures. The failed switch is removed from the multicast group. Once a new switch replaces the faulty one, it is added to the multicast group, and begins serving reads after obtaining an initial view of the shared state.

**SDW.** The protocol inevitably stalls once a failure occurs (i.e., the local window ids stop increasing). Denote the maximum window at a correct switch at the time of the failure by  $w_{max}$ . The difference between the local window ids at each pair of switches is at most one. Thus, every stalled switch is in window  $w_{max}$  or  $w_{max}-1$ .

We reconcile the states of the surviving switches as follows: a controller reads the states of all switches. It collects the state of  $R_r$  in some switch that is in window  $w_{max}$  and sends it to all switches that are in window  $w_{max}-1$  (if any), so they advance to window  $w_{max}$ . The controller merges all the  $R_s$  objects to yield the most up-to-date state for window  $w_{max}+1$  and broadcasts it to all switches, thus updating their  $R_s$  objects to the merged state. Then it removes the failed switch from the multicast group and the switches resume the protocol from window  $w_{max}+1$ .

Adding a new replica is a two-stage process: increasing the expected number of ACKs on correct switches and making sure that all switches are in the same window, which stalls window progression, followed by adding the new replica to the multicast group of each correct switch. The new switch begins serving reads after the current window completes.

We note that during the recovery the updates to the live switches are not lost, but rather accumulated in local switch

replicas  $R_u$ . These updates are then synchronized during the recovery. Thus, this protocol is not time critical and can be implemented in control-plane without adding code to the resource-constrained data-plane.

## 7 Design

We explain the messaging mechanism shared by all protocols, and then describe the SDW design. SRO and EWO closely follow their descriptions in §6.

### 7.1 Replication Message Exchange

**Packet format.** Switches exchange replication packets, updates, and acknowledgments with each other to replicate state. Replication packets are IP packets; therefore, by assigning an IP per switch, these packets can be routed using standard L3 routing protocols. Besides Ethernet and IP headers, each packet includes a single bit indicating whether the packet is an update/write request or an ACK, the keys and values accessed by the write, and, in SRO, also a sequence number. For example, in an SRO NAT implementation, the keys are the source IP and source port, and the values are the translated IP and port. In an SDW DDoS application, the keys are sketch indices and the values are counter increments.

**Reliable delivery.** A major challenge in data-plane replication is ensuring delivery of replication packets. Current switches do not provide enough control over internal switch buffers to store and retransmit a packet from the data-plane.

We identify two cases that require buffering. First, there are *replication* packets generated by each switch as part of the replication protocol. Such packets must be reliably delivered in SDW. Second, there are *write* packets that are received from external sources (not from a switch) and update the NF state in a switch. In SRO these packets cannot be externalized until the updated state is synchronized among the switches.

We handle these two cases separately. For SDW replication packets we keep the state being replicated at the application level until acknowledged, instead of buffering the packet. Then an *ACK-check* packet is *periodically* generated by the packet generator. If the sent replication packet has not yet been acknowledged by other switches, the *ACK-check* triggers its retransmission. Here we use the recirculation trigger for the packet generator to initiate a batch of packets at once.

In SRO, the packets themselves must be buffered since their content is not reproducible by the switch. Buffering in the data-plane is an open problem and we leave it for future work. However, since most NFs that use SRO would require the updates to be performed via the control-plane anyway (Observation 1, §4.1), we relay the reliable delivery to the control-plane of the switch that receives the write packet. The cost of buffering and retransmission is negligible, as we show in §9.2. Future switches might enable table updates in the

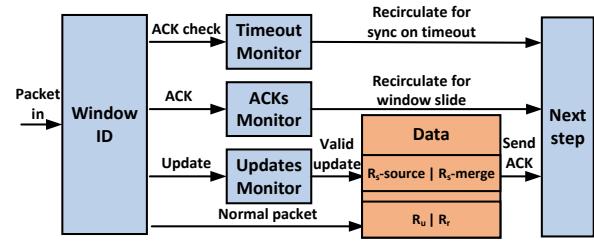


Figure 3: SDW high-level design. Blue boxes are reusable P4 control blocks, while the orange box is application-dependent.

data-plane, motivating data-plane buffering mechanisms to avoid control-plane involvement in replication.

**Packet duplication and reordering.** SRO replication packets are shipped with a sequence number allowing each replica to apply updates in order and to reject updates with sequence numbers lower than those already processed. In EWO, updates are idempotent and monotonic so detecting duplication and reordering is already a part of the merging process. We explain the SDW implementation in detail below.

### 7.2 Strong Delayed-Writes (SDW)

As presented in Figure 3, the data structure used in SDW is organized as two register arrays, each of which holds a 32-bit pair. At any given time, one window is designated for reading and writing, namely, its register arrays used as the  $R_u$  and  $R_r$  objects in the SDW protocol. The other window is used in the sync operation. The sync object  $R_s$  is divided into two register values, one, denoted  $R_s\text{-merge}$ , receives data from other switches, while the other, called  $R_s\text{-source}$ , holds the local state as sent to other switches at the beginning of the window. This separation is important to allow retransmissions (§7.1). **Synchronization.** The alternating window structure enables SwiSh to ensure that the  $R_r$  in each window are consistent across all switches. Each time synchronization is initiated, the content of the  $R_s\text{-source}$  register is sent to all the switches, and the content received from all of the switches is merged in the  $R_s\text{-merge}$  register. Note that each switch also receives (and hence merges) its own update. Once all the updates are received, the content of  $R_s\text{-merge}$  is identical across the switches, so the synchronization for that window is finished.

Unfortunately, a full sketch cannot be read while processing a single packet, so we send the sketch column by column. For simplicity, we first explain handling of a single-column sketch, and then discuss the complete implementation.

Each switch maintains two bitmaps: one, to track ACKs that other switches received its updates, and the other, that it received all updates from them. If an update was lost, the sketch is retransmitted.

**Window advancement.** The last update to complete the bitmaps signifies the completion of the sync round for the

switch. The switch advances the window ID, swaps the roles of the registers, and starts a new sync process again. During this swap the following arrays are swapped:  $R_s$ -merge swaps with  $R_r$ , and  $R_s$ -source swaps with  $R_u$ .

**Ready phase.** Because round advancement is a local event, the switches do not advance their windows in lock-step. Thus, a switch may receive an update for the *next* window, which will be dropped and retransmitted later. Buffering such updates would significantly increase the memory footprint. Instead, we introduce the *ready phase*. Once a switch advances its local window it broadcasts a *ready* packet to all the rest. A switch starts broadcasting its updates only after it receives *ready* packets from all other switches (existing bitmap can be reused for tracking). This phase ensures that an update will not be sent to a switch that is not yet ready to merge it. Ready packets are retransmitted upon timeout, though in the experiments we did not encounter such cases. In our evaluation (§9.2) we show that the *ready* phase is critical to achieving predictable replication latency.

**Multi-column sketches.** Ideally, each switch should track each column being synchronized separately, to filter duplicates and retransmit lost updates. This solution would be too memory-consuming and would limit the sketch size, however. We make two optimizations. First, for  $R_s$ -merge, we retain the original bit-per-switch tracking, so a switch sends an ACK only when a full sketch was received. Thus, we always retransmit a full sketch. Second, we maintain a counter per switch which tracks the index of the next column to be updated. Only updates that match this counter are accepted. This approach is correct: it handles duplicates and packet reorders. However, while it is efficient for duplicates, it would lead to sketch retransmission if packets are reordered. We assume that this is a rare event, however, because IP routing in data centers usually maintains the same path for a given flow.

We implement both approaches. The bitmap-per-column implementation allows using sketches with 3 rows and 64K entries per-row and can scale up to 32 switches. The counter-per-switch implementation can scale to 4K switches for the same sketch size.

Note that changing the communication pattern from an all-to-all to an aggregation tree, e.g. as in SwitchML [68], may also reduce the per-switch state but at the cost of increasing replication latency.

**Register initialization.** There is no way to iterate over all the registers and reset them. Instead, we piggyback initialization on the first write and use a single bit in each register to determine whether the register is initialized. These bits are reset during the processing of sync packets.

**Reducing replication bandwidth.** Recall that SDW is used for a collection of variables, stored in register arrays, over which queries can take consistent snapshots. Our current implementation of the sync protocol exchanges a full state snapshot (including all variables) rather than only the ones that were updated. The challenge for selective updates is that

the switches send a varying number of packets in each window (due to hardware limitations, the state does not fit in one packet), and so the destination does not know when to acknowledge the state receipt. To overcome this challenge, switches count the number of updates that they send in a window and piggyback this number on the last update.

**Recovery.** The recovery protocol follows the algorithm mentioned above (§6.4), but also considers the ready phase and sends ready packets to allow switches to make progress before removing the faulty switch from the replica group. SDW does not rely on a centralized controller in failure-free runs. However, as writes are not lost upon switch failures, recovery is kept off of the critical-path and is not time-sensitive. Therefore, we chose to offload the recovery protocol to a centralized controller which frees switch resources.

## 8 Implementation

We expound the implementation of SRO and EWO, and then we describe the distributed NFs implemented on top of SwiSh. Last, we provide additional implementation details and limitations.

### 8.1 Strong Read-Optimized (SRO)

We run the replication protocol in the control-plane logic. Write packets (packets that modify state) are forwarded to the control plane, which subsequently generates a write request forwards it to the head of the chain.

The way write requests are handled depends on the storage type where the data is stored in the switch. If the data can be modified only from the control-plane, then write requests must be processed by the control-plane at each switch in the chain. Otherwise, write requests can be processed directly in the data-plane. We implement reading from tail by tunneling the reading packets through the tail switch to its destination with an outer IP header (similar to IP-in-IP). While a write is pending, the key is flagged as “read-from-tail”, causing subsequent reading packets to be sent to the tail.

### 8.2 Eventual Write-Optimized (EWO)

The EWO logic uses the following types of packets: (a) Regular packets from applications – read and write to the shared state. (b) Update packets – sent when the local state changes. The recipient merges these updates with its local state. (c) Generated packets – for reliable message delivery. Because each register array can only be accessed once per packet, if the state consists of an array, we generate one packet per array entry. If we maintain multiple register arrays, they can be accessed by a single packet.

Reads are local, while writes require sending an update to other switches. To broadcast updates, we use egress-to-egress mirroring to create a truncated copy of the original

write packet. We use the multicast engine to create a copy of the update packet for each switch in the replica group. Each copy is then modified to carry the updated values.

The application state each switch maintains depends on the particular data structure. For example, to implement a shared counter, each switch maintains a vector of counters, one per switch in the replica group. On the other hand, growing only sets and LWW variables do not require sharding.

In order to ensure eventual consistency in the face of lost update packets, a periodic background task is implemented by using the switch’s packet generator that iterates over the register array, forming write update packets consisting of the indices and values for each register, and forwarding each one to a randomly-selected switch in the replica group.

### 8.3 Distributed NFs

We prototype three multi-switch NFs. We also prototype a distributed version for all of these NFs built using the protocols in SwiSh. In addition, for two of them we also implement a version that uses a central controller for synchronization.

**Network Address Translator (NAT).** This application maps internal source IPs to external source IPs. Each switch maintains two translation tables – one that maps (external source IP, external source port) to (internal source IP, internal source port) and another that performs the inverse mapping. We implement a distributed NAT using the SRO protocol. It requires no changes to the data-plane logic.

**Super-spreaders detection (DDoS).** This application detects source IPs that communicate with more than 1000 unique destination IPs. Inspired by OpenSketch [80], we implement it using a CMS, with a bit set instead of counters. Packets are first sampled based on the (source IP, destination IP) pair. Sampled packets set a single bit in the bitset in each row of the sketch. The bitset is used to estimate the number of unique destinations. Our implementation uses a sketch with 3 rows and 32K 32-bit wide bitsets per row.

We implement two designs based on a central controller. In both, the controller obtains the list of suspicious IPs from each switch, and decides to block IPs if the sum of different destination IPs for that source from both switches exceeds 1000, in which case it inserts an entry to the block list of each switch. However, there are two ways for the controller to obtain this data: (a) pull-based: each switch maintains a gradually growing list of potential IPs to block. The controller periodically *pulls* the delta in the list since the previous pull; (b) push-based: each switch sends a packet when it detects a potential IP to block. For simplicity we mark an IP as suspicious if it sends to more than 500 destinations, and construct the workload to send half of the packets from each source to one switch and another half to the other, thus the implementation works correctly for this case.

The distributed design replicates the sketch using the SDW protocol, each switch unilaterally decides to block a desti-

nation according to the replicated sketch, which essentially holds a global view of the network.

**Rate limiter.** We implement a rate limiter based on the token bucket algorithm [70]. In the single switch design, the controller periodically fetches rate estimations from each switch, calculates the token limit per each user and each switch, and writes it back to the switches. We implement two distributed versions, with EWO and SDW respectively. Switches replicate their own rate estimates for each user, and calculate their limit according to global traffic ratios.

## 8.4 Implementation Details

We implement SwiSh using P4<sub>16</sub> [73] and Intel P4 Studio 9.6.0 [32] for Tofino switches. We implement all protocols as described.

**API.** We expose the building blocks of each protocol’s design as P4 control blocks [73]. We then use this API to implement our NF applications (§8.3).

**Control Plane.** For applications that use SRO variables, we implement the control-plane logic in C++ using the user space packet DMA API (kpkt). For the other protocols, we initialize the switch state using bfrt-python. We also utilize a simple TCP server in C++ for reading register values from the switch for the recovery protocol.

**Limitations.** Our current implementation does not include the required recovery logic for SRO because it is well-known and in-control plane, thus it does not challenge our design. Although independent to the number of switches in the replica group, the major limitation of replicated NFs is the increase in SRAM usage ( $\times 4$ ). We fully implement recovery for SDW.

## 9 Evaluation

We evaluate the protocols and applications on two Tofino switches (each two pipes) and on 32 switches in an emulator. Our key observations are:

- Control-plane replication is too slow.
- SRO has high latency and low throughput.
- SDW is scalable and replicates large sketches in microseconds.
- For a DDoS detector, SDW responds instantly to an attack, blocking malicious packets, while central controller allows almost 50% of the packets to go through.
- For a rate limiter, SDW and EWO respond instantly to traffic changes, while central controller lags behind.

**Setup.** We use two machines with Intel Xeon Silver 4216 2.1 GHz CPUs, connected via two EdgeCore Wedge 100BF-32X programmable switches. The server is dual socket with 192 GB RAM. Hyper-threading and power saving are disabled. One machine acts as a traffic generator/consumer; it has two 100G Intel E800 NICs. The other acts as a central controller; it has two 40G NVIDIA ConnectX-4 Lx EN NICs.

**Topology.** We use the *leaf-spine* topology in which the switches are connected as shown in Figure 4, and run ECMP on one pipe and a NF on the second.

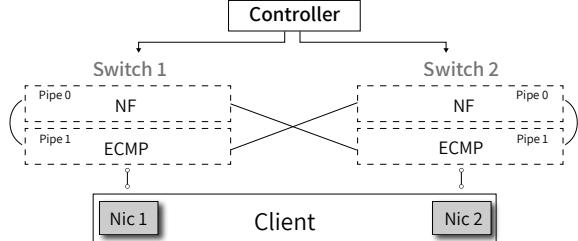


Figure 4: Testbed topology.

**Performance measurement methodology.** We build a DPDK-based packet generator. We evaluate SwiSh on a real packet trace, CAIDA [10], as well as synthetic workloads. Throughput is measured by the NIC and application-level performance counters. Latency is measured in software.

To measure the performance of the in-switch NFs and protocol implementations, we create a line-rate load (100Gbps, unless stated otherwise) on a single switch port. To validate that the performance obtained via this approach is representative of the switch under load over all its ports, we also run one experiment with *a fully loaded switch running at 2.1 Gpps* (§9.2). We show that the performance is almost the same as with a single port traffic, validating our methodology.

## 9.1 End-to-end benchmarks

**NAT.** We replay 10K packets from the CAIDA dataset and measure the per packet latency with and without replication. 21% of the packets are processed by the control plane (update packets), while the rest are processed in the data-plane. Figure 5d shows the latency distribution. SwiSh does not introduce any overheads for read packets, while update packets are taking about twice as long to get processed since they are batched in the control plane until the update is acknowledged by the other switch.

We also compare the throughput of the distributed version with the one on a single switch, while sending 64-byte packets at line rate to a single port. There are *no updates during the test*, as we wait for the handshake to complete. Therefore, both versions achieve line-rate throughput (112 Mpps).

**Super-spreader detection.** DDoS is configured to detect sources (IPs) that communicate with more than 1K different destinations. We create a trace where packets are sent from different source IPs, each with thousands of different destinations. Each source IP sends 10K packets.

In the experiment we replay a trace where we vary the number of packets that have different source IPs sent per second, while maintaining the absolute transfer rate from each source IP constant. This is a reasonable scenario where

an attacker uses a botnet to generate malicious traffic while maintaining the transfer rate of each bot constant.

We compare the number of packets sent by each source IP relative to the number of packets received by the destination IP. Ideally, each source IP should be blocked after the first 1000 packets, therefore the ratio should be about 10%.

We compare the push and pull baselines with the implementation that uses SDW replication. Figure 5b shows that both versions of the centralized controller are quickly becoming overwhelmed and cannot keep up with processing the updates, failing to block packets. At 1.5K source IPs/second the push baseline breaks down because the push requests to block certain IPs from the switch get dropped at the host, thus their respective IPs are left unblocked. The results were obtained after increasing the socket receive buffers to 25MB.

To validate this result, we run the same workload fixed at 4K source IPs/sec. Figure 5a shows the distribution of the ratio of packets received per source IP across all source IPs. We observe that the pull design manages to block up to 30% of all the source IPs, but for each IP different number of packets leaked. Effectively, the pull design was unable to block traffic from 70% of the source IPs. That is because the controller collects batches of requests and handles them together, thus some source IPs manage to send more than others. However, the push design blocks only 5% of all the source IPs. The SDW-based design, shown as a vertical line at 10%, passes the first 10% of each source (which is our super-spreader detection threshold), and then blocks all the packets as expected.

**Per-user rate limiter.** We set a limit of 2Mpps per-user and configure the rate limiter to re-estimate rates every 1ms.

We create a trace where packets are sent from different source IPs (each source defines a different user) with 40 unique users (sending rate is 2Mpps per user). The trace is comprised of alternating phases with a period of 5s. In even phases, all flows of a specific user are split equally between the two switches. While in odd phases, 90% of each user's flows are routed to one of the spine nodes and the rest 10% are forwarded to the other spine node. These alternations results in immediate changes in the per-user rate estimator that each switch maintains.

We compare our EWO and SDW protocols with a pull-based baseline and measure the average throughput per user over time. In the first 5 seconds of the experiment, the traffic is balanced so each switch runs at 1Mpps and the controller sets a per-user limit of 1Mpps on each switch. At the 5th second of the experiment, we change phases, and now one switch measures 1.8Mpps and the other switch measures 0.2Mpps. Because each switch was set to limit each user to 1Mpps, the first switch forwards only 1Mpps and the other switch forwards 0.2Mpps resulting in 1.2Mpps aggregate throughput. Figure 5c shows the average received throughput per-user over time at a sampling period of 200 ms. The baseline misses the phase changing point and allows the throughput to reduce to

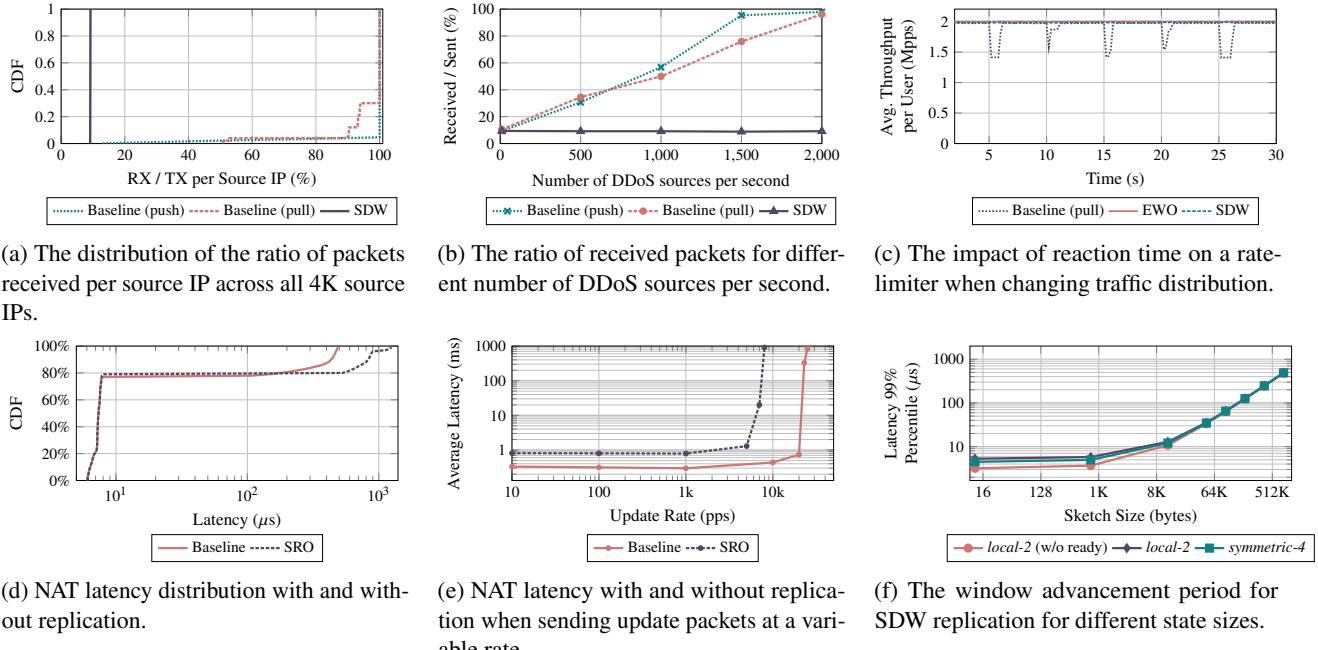


Figure 5: End-to-end and analysis results.

below 1.5Mpps, slightly more than the expected value due to sampling. SDW and EWO perform comparably. They both react immediately to traffic changes without throughput drops. EWO eagerly sends 40 updates every 1ms, allowing the other switch to immediately change its limit. SDW replicates such a small state (40 32bit values) under 10 microseconds (5f).

## 9.2 Analysis

**SRO: Update rate.** We measure the overhead introduced to update packets with replication. For this experiment we run NAT on spine switches and send update packets that are all processed by the control plane. We measure the per packet latency and report the average. Figure 5e shows that compared to the single switch, in the replicated setting the update rate is reduced by a factor of  $\times 2.5$ . This is expected since each update packet generates a write request and an ACK that has to be processed on the other switch.

We observe that *the update rate of SRO is limited by the throughput of the control plane on a single switch*. SRO variables cannot sustain more than 20K updates per second (160Kbps). Update latency increases with the update rate because packets are buffered in the control-plane until acknowledged.

**SDW: Window advancement latency.** We measure the time each window absorbs updates before being advanced. We vary the state size being replicated, so for the smallest sketch the time to advance the window is the upper bound on the replication rate, constrained by the latency of updates between switches. There are no retransmissions in this experiment.

We replicate a sketch with 3 rows and vary each row size to up to 64K counters (total of 768KB in each sketch). We store the global timestamps of the first 10K window increments and read them at the end of the run.

We use the following topologies: (a) *local-2*: two local pipes on a single switch (we measure identical results compared to two remote pipes); (b) *symmetric-4* - four pipes, two in each switch, with a dedicated link between each pipe.

Figure 5f shows the 99th percentile latency to advance the window for each state size. As we see, the window can be advanced as fast as every 3 $\mu$ sec for the sketch of 4 bytes. The current bottleneck is the packet sending rate which, even within the switch (Recirculation), takes a few hundred nanosecs. This window advancement rate implies that the updates become visible after 6 $\mu$ sec (since  $R_u$  becomes  $R_r$  after two window advancements). For the sketch larger than 1K, the actual replication rate is about 13Gbps between each pair of switches, which is about *five orders of magnitude* faster than SRO. We note that this rate is limited by the maximum packet rate ( $\sim 160$ Mpps) of a single port. This is because replication packets hold only 12 bytes of data, which in turn is due to limited per-packet memory accesses imposed by the hardware. Optimizing the effective bandwidth is left for future work.

We observe negligible increase in the window advancement latency when adding two additional switches. This is because each switch updates all the others concurrently, hence no additional delay. The ready phase adds a constant latency overhead of 2 $\mu$ sec to each replication round.

**SDW: Performance under full switch load.** We generate

traffic on all switch ports as follows. We saturate a single port using our packet generation machine and let that traffic travel through each port in the switch by connecting ports in a chain and forming a “snake” (a similar methodology was used in NetCache [38]). We reserve ports that are used for replication. We use the symmetric-2 topology. We saturate the switch with 130B packets, each updating the sketch. For a sketch with 64K entries per-row we measure 486  $\mu$ sec window advancement latency, at a total packet rate of 1.8Gpps. For a sketch size of 1 entry per-row we measure 3.2  $\mu$ sec window advancement latency at a total packet rate of 2.1Gpps.<sup>3</sup> In both extremes, we could not measure any impact on window advancement latency, which is expected as the switch logic is guaranteed to perform packet processing at a switch line rate.

**SDW: Retransmissions and the ready phase.** The ready phase ensures that the switches do not send updates after advancing their window before all others advanced to that same window. Without this guarantee, an update from the consequent window that arrives too early will be dropped, and later retransmitted after a timeout. We now show that this phase is essential to avoid retransmissions and maintain low latency when scaling to more switches.

We first run the protocol without the ready phase (Figure 5f, local-2 no ready) on two pipes on the same switch. The protocol runs in lockstep on both of the pipes, so we do not see any update retransmissions. However, with four pipes (symmetric-4 topology) there are many retransmission (not shown in the Figure). For example, we measure an average of 2934 update retransmissions in the first 10K window advancements across 100 runs. We observe a similar behavior in an asymmetric topology four pipes connected using the leaf-spine topology. Adding the ready phase completely eliminates such retransmissions and allows the system to progress effectively as fast as a two-pipes system, with stable latency guarantees.

**SDW: Recovery.** We measure the total recovery time of the protocol from the time pipes fail to the time the system makes progress, i.e. windows are advancing again. We run four pipes in the 4-symmetric topology that replicate a sketch and shut down random pipes. We disable the failed pipes’ ports to other switches in a random order. We repeat this experiment 20 times for each data point, and vary sketch sizes and failure counts. We report the average recovery time.

Figure 6 shows that recovery time is dominated by the time it takes to synchronize the sketches of correct switches. Therefore, recovery time increases as sketch size increases, and decreases as the failure count increases. As expected, for the 3 pipe failures setup, only a single correct switch remains live, thus recovery time is independent from the sketch size.

As explained in §6.4, updates sent to live switches during the recovery are not lost but accumulated, so the recovery time minimization is a secondary goal. Nevertheless, recovery time can be further reduced by applying additional optimizations,

<sup>3</sup>1-entry per-row requires lower replication load and frees certain resources affording higher packet rate.

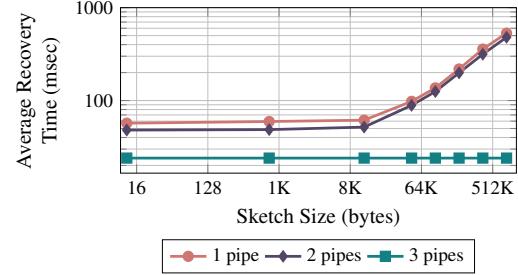


Figure 6: The impact of number of failures and sketch size on the total SDW recovery time.

e.g. parallelizing the currently serial controller-to-switch communication and batching requests, and by writing the logic using a more efficient programming language.

**SDW: Scalability.** We emulate a large replica group of switches running the SDW protocol by connecting together 32 Tofino model instances running in Docker containers. The switches are connected together via another switch that runs L3 forwarding. We verify that the protocol runs correctly and that there are no update retransmissions.

## 10 Related Work

**In-switch NFs.** Previous studies have shown that offloading NFs to programmable switches, such as load balancers [57] and DDoS detectors [45], enables very high performance. However, these projects were designed for a single switch. SwiSh aims to facilitate the deployment of these applications in a distributed fashion. RedPlane [42] enables switch state replication to servers for fault tolerance, but does not support state modification on multiple switches concurrently, as our work does.

**In-switch acceleration.** Previous works suggested in-switch acceleration for general-purpose applications such as key-value caches [38, 50], replicated key-value stores [37], query processing [24] and aggregations [68, 75]. SwiSh can be useful for such general-purpose applications too. For example, SwiSh could be used to implement the cache invalidation mechanism in DistCache. We note, however, that due to the general-purpose nature of these applications, some of them feature a complex state, and require strong semantics together with frequent updates, which SwiSh does not provide. Such requirements are less common in NFs; thus, we target SwiSh to facilitate the development of distributed NFs.

**State management for NFs.** State management and fault-tolerance for NFs on servers have been well studied [20, 64, 65, 71, 77]. However, these techniques are infeasible in the context of programmable switches. For example, FTMB [71] suggests a rollback-recovery technique for fault-tolerance in which packets are logged and replayed upon failures. However, due to the high processing rate of the switch, it is impractical to log

every packet to external storage or through the control-plane.

**In-switch coordination.** NetChain [37] and P4xos [16] implement coordination protocols running in the data plane to provide reliable storage as a network service. We apply data plane replication as an internal building block for NFs, a task for which it is well suited as the data-plane properties (e.g., limitations to  $\sim$ 100 byte objects) are better matched for replicating NF state registers than arbitrary applications.

**Distributed network state.** Managing distributed network state has been well studied. Onix [43] distributes network-wide state among multiple controllers. DIFANE [81] offloads forwarding decisions to authority switches to alleviate load on the controller and to reduce per-flow memory usage in network switches. Mahajan *et al.* [55] explore consistency semantics during network state updates. While previous works focus on control-plane managed state, SwiSh specifically targets replication of mutable state of data-plane programs.

**Distributed network monitoring.** Network-wide monitoring requires coordinated, distributed computation across switches [25, 26, 63]. Harrison *et al.* [25, 26] propose a distributed heavy-hitter detection algorithm that combines local counters with a centralized controller. SwiSh can be used to implement similar algorithms without a centralized controller, potentially providing faster response. Ripple [36] replicates state in data-plane for link-flooding defense but does not provide consistency guarantees. Ripple can be implemented using SwiSh. Distributed computation is also needed if the resources of a single switch are insufficient, e.g. Demianiu *et al.* [18] partition state across switches for flow metric computation.

**Relaxing consistency for availability.** Many systems have traded consistency for increased availability and performance [4, 17, 21, 44, 62, 72, 79]. For example, TACT [79] aims to provide a middle-ground between strong and eventual consistency. However, TACT may block read and write operations to enforce consistency bounds which is unsustainable in the switch environment. Additionally, TACT maintains a single version of the data while SDW maintains multiple versions of the state and seamlessly switches to the up-to-date one as soon as the previous synchronization round is completed. Therefore, the protocol advances as fast as the network conditions allow while providing consistent snapshots to every replica. On the other hand, the combination of dynamic system behavior and consistent snapshots cannot be expressed using TACT’s consistency metrics.

## 11 Conclusions

SwiSh offers a systematic approach to state sharing among programmable switches. We analyze the requirements of in-switch stateful NFs and implement three protocols for data-plane replication. We introduce a novel SDW protocol that achieves high update rate and low update latency, while providing strong consistency guarantees, which are particularly

useful for implementing sketches. We show experimentally that data-plane is practical and fast, and achieves orders of magnitude higher performance than the traditional centralized controller designs. We believe that this work will pave the way for building distributed stateful NFs entirely in data-plane.

## Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Dejan Kostic, for their insightful comments and constructive feedback. Lior Zeno was partially supported by the HPI-Technion Research School. We gratefully acknowledge support from Israel Science Foundation (grants 980/21 and 1027/18) and Technion Hiroshi Fujiwara Cyber Security Research Center.

## References

- [1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the SDN match and action model. *Comput. Networks*, 136:1–12, 2018.
- [2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–28, 2013.
- [3] Amazon Web Services. AWS Shield. <https://aws.amazon.com/shield>.
- [4] Mary Baker and John Ousterhout. Availability in the Sprite distributed file system. In *Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–4, 1990.
- [5] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.
- [6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *IFIP Networking*, pages 449–457, 2020.
- [7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O’Connor, Pavlin Radoslavov, William Snow, et al. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6, 2014.
- [8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica,

- and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.
- [9] Broadcom. Trident 3. <https://www.broadcom.com/products/ethernet-connectivity/switching-strataxgs/bcm56870-series/>.
- [10] CAIDA. The CAIDA UCSD Anonymized Internet Traces - 2019. [https://www.caida.org/catalog/datasets/passive\\_dataset](https://www.caida.org/catalog/datasets/passive_dataset).
- [11] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.
- [12] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 22–28, New York, NY, USA, 2018. Association for Computing Machinery.
- [13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking EXperiments and Technologies*, pages 15–29. ACM, 2019.
- [14] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.
- [15] Graham Cormode and S. Muthu Muthukrishnan. Approximating data with the count-min sketch. *IEEE Software*, 29(1):64–69, 2012.
- [16] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking*, pages 1–13, 2020.
- [17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.
- [18] V. Demianiuk, S. Gorinsky, S. Nikolenko, and K. Kogan. Robust Distributed Monitoring of Traffic Flows. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11, 2019.
- [19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jin-nah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.
- [20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery.
- [21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.
- [22] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 373–382, 2011.
- [23] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, August 2009. ACM.
- [24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.
- [25] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephants: Seize the global heavy hitters. In *Proceedings of the 2020 ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure, SPIN@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, pages 15–21, 2020.
- [26] Harrison, Rob and Cai, Qizhe and Gupta, Arpit and Rexford, Jennifer. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research*, SOSR ’18, New York, NY, USA, 2018. Association for Computing Machinery.

- [27] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–328, 2013.
- [28] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [29] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.
- [30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. SketchLearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.
- [31] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.
- [32] Intel. P4 Studio. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html>.
- [33] Intel. Tofino. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html>.
- [34] Intel. Tofino Native Architecture. <https://github.com/barefootnetworks/Open-Tofino>.
- [35] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 285–291, 2019.
- [36] Jiarong Xing and Wenqing Wu and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3865–3881. USENIX Association, August 2021.
- [37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.
- [38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP ’17*, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.
- [39] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, March 2017. USENIX Association.
- [40] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the 2016 Symposium on SDN Research (SOSR ’16)*, Santa Clara, CA, USA, March 2016. ACM.
- [41] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 USENIX Winter Technical Conference*, San Francisco, CA, USA, January 1994. USENIX.
- [42] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling Fault-Tolerant Stateful in-Switch Applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM ’21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.
- [43] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *OSDI*, volume 10, pages 1–6, 2010.
- [44] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [45] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspari. Offloading Real-time DDoS Attack Detection to Programmable Data Planes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 19–27, 2019.
- [46] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Composing ordered sequential consistency. *Information Processing Letters*, 123:47–50, 2017.

- [47] B. Lewis, M. Broadbent, and N. Race. P4ID: P4 Enhanced Intrusion Detection. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–4, 2019.
- [48] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [49] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 311–324, Santa Clara, CA, March 2016.
- [50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST’19, page 143–157, USA, 2019. USENIX Association.
- [51] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.
- [52] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114, 2016.
- [53] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *Proc. USENIX Security*, 2021.
- [54] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems*, APOCS, pages 31–44, 2020.
- [55] Mahajan, Ratul and Wattenhofer, Roger. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.
- [56] McKeown, Nick and Anderson, Tom and Balakrishnan, Hari and Parulkar, Guru and Peterson, Larry and Rexford, Jennifer and Shenker, Scott and Turner, Jonathan. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.
- [57] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan 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*, pages 15–28, 2017.
- [58] Microsoft Azure. Azure DDoS Protection. <https://azure.microsoft.com/en-us/services/ddos-protection/>.
- [59] Robert HB Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and distributed Systems*, 6(2):165–169, 1995.
- [60] NVIDIA. Spectrum. <https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/>.
- [61] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.
- [62] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, oct 1997.
- [63] Raghavan, Barath and Vishwanath, Kashi and Ramabhadran, Sriram and Yocum, Kenneth and Snoeren, Alex C. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’07, page 337–348, New York, NY, USA, 2007. Association for Computing Machinery.
- [64] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC ’13, New York, NY, USA, 2013. Association for Computing Machinery.
- [65] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew 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, Lombard, IL, 2013. USENIX.

- [66] Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12–16, 2020, Virtual Conference*, volume 179 of *LIPics*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [67] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’20*, pages 117–129, 2020.
- [68] Amedeo Sazio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.
- [69] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS’11*, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.
- [70] S. Shenker and J. Wroclawski. RFC2215: General Characterization Parameters for Integrated Service Network Elements, 1997.
- [71] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM ’15*, page 227–240, New York, NY, USA, 2015. Association for Computing Machinery.
- [72] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference, USENIX’09*, page 11, USA, 2009. USENIX Association.
- [73] The P4 Language Consortium. P4<sub>16</sub> Language Specification. <https://p4.org/p4-spec/docs/P4-16-v1.2.0.html>.
- [74] The P4.org Architecture Working Group. P4<sub>16</sub> Portable Switch Architecture (PSA). <https://p4.org/p4-spec/docs/PSA-v1.1.0.html>.
- [75] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD ’20*, page 2407–2422, New York, NY, USA, 2020. Association for Computing Machinery.
- [76] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation, OSDI’04*, page 7, USA, 2004. USENIX Association.
- [77] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, April 2018. USENIX Association.
- [78] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.
- [79] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. *OSDI’00*, USA, 2000. USENIX Association.
- [80] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 29–42, April 2013.
- [81] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.
- [82] Zeno, Lior and Ports, Dan R. K. and Nelson, Jacob and Silberstein, Mark. SwiShmem: Distributed Shared State Abstractions for Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks, HotNets ’20*, page 160–167, New York, NY, USA, 2020. Association for Computing Machinery.
- [83] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

## A Theoretical Analysis

The SDW protocol supports stream-order invariant data types like sketches. Variables of this type support three API functions: (1)  $\text{UPDATE}(v)$  – handling a single addition of element  $v$ , (2)  $\text{QUERY}()$  – returns a value based on the internal state, and  $\text{MERGE}(R')$  – merges the state of  $R'$  with that of the current variable. A requirement of any variable  $R$  fitting this model is that the  $\text{QUERY}$  result depends only on the set of elements that were ingested before it (either by an  $\text{UPDATE}$  or a  $\text{MERGE}$ ), and not their order. We say that a query *reflects* an update, if the update altered the state before the query executed.

An execution of an algorithm renders a *history*  $H$ , which is a series of *invoke* and *response* events of the three API functions. In a *sequential history* each invocation is immediately followed by its response. The *sequential specification*  $\mathcal{H}$  of a variable is its set of allowed sequential histories.

A *linearization* of a concurrent execution  $\sigma$  is a history  $H \in \mathcal{H}$  such that after adding responses to some pending invocations and removing others,  $H$  and  $\sigma$  consist of the same invocations and responses and  $H$  preserves the order between non-overlapping operations [28]. If every concurrent execution has a linearization, we say that the variable is linearizable. For randomized variables we require a stronger property, called *strong linearizability*. The qualifier “strong” means that the linearization points are not determined post-facto, which is necessary in randomized variables [22].

A relaxed property of a variable is an extension of its sequential specification to allow for more behaviors. We adopt the notion of  $r$ -relaxed strong linearizability from [67], a variant of the relaxation defined by Henzinger et al. [27], brought here for completeness. Intuitively, an  $r$ -relaxed variable allows a query to return a result based on all but at most  $r$  updates that happened before it.

**Definition A.1.** A sequential history  $H$  is an  $r$ -relaxation of a sequential history  $H'$ , if  $H$  is comprised of all but at most  $r$  of the invocations in  $H'$  and their responses, and each invocation in  $H$  is preceded by all but at most  $r$  of the invocation that precede the same invocation in  $H'$ . The  $r$ -relaxation of  $\mathcal{H}$  is the set of histories that have  $r$ -relaxations in  $\mathcal{H}$ , denoted  $\mathcal{H}^r$ .

Our SDW protocol is described in §6.3, and its pseudo-code is presented in Algorithm 1. To prove that Algorithm 1 is  $r$ -relaxed strongly linearizable, we first prove a helper lemma:

**Lemma 1.** Consider a history  $H$  arising from a concurrent execution of Algorithm 1, and some completed update  $u \in H$  executed by  $p_i$ . Let  $w$  be the value of  $\text{win}$  during  $u$ . Update  $u$  is reflected by every query  $q$  on any  $p_j$ , in every window  $w' \geq w + 2$ .

*Proof.* Let  $H$  be a history arising from a concurrent execution of Algorithm 1, and let  $u \in H$  be some completed update

executed by  $p_i$ . Let  $w$  be the value of  $\text{win}$  during the update’s execution on  $p_i$ .

Update  $u$  is added to  $\text{objs}[w \bmod 3]$  on Line 12. On Line 39,  $\text{objs}[(w+2) \bmod 3]$  is broadcast to all switches, specifically to some switch  $p_j$  (as  $p_i$  retains the update in the same place that is merges received variables, this holds for  $j = i$ ).

The next time  $p_j$  advances on Line 35, it enters window  $w' = w + 2$ . Note that the variable that was queried in the previous window ( $w' - 1$ ) is the same variable that reflected  $u$ . This variable is the one queried in round  $w'$ , therefore reflected in round  $w' = w + 2$ .

We now prove by induction that in round  $w'' = w' + k$ ,  $u$  is reflected by a query in round  $w''$  on  $p_j$ . The base is for  $k = 0$ , and has been proved.

Assume the hypothesis holds for  $w'+l$ , we prove for  $w'+l+1$ . In round  $w'+l$ ,  $u$  is reflected by  $\text{obj}[(w'+l+1) \bmod 3]$ . On Line 37,  $p_j$  merges this variable into  $\text{obj}[((w'+l+1)+1) \bmod 3]$ , which is the variable queried in this round.

As this induction is true for all  $k \geq 0$ , it holds for any  $w'' \geq w'$ , proving the lemma.  $\square$

The following corollary follows directly from Lemma 1:

**Corollary 1.1.** Let  $H$  be a history arising from a concurrent execution of Algorithm 1, and let  $q \in H$  be some query completed by  $p_i$ . Let  $w$  be the value of  $\text{win}$  during its execution. Query  $q$  reflects all updates occurring in any window  $w' \leq w - 2$ .

Note: A system where linearizability holds for sub histories including a single query is sometimes called *Ordered Sequential Consistency (OSC)* [46], this is commonly used in systems, e.g., ZooKeeper [31].

Finally, we define the *operation projection* of a history  $H$  and a set of operations  $O$  as the same history containing only invocations and responses of operations in  $O$ . We denote this  $H|_O$ . Using these formalisms we can prove the following theorem:

**Theorem 2.** Consider a history  $H$  arising from a concurrent execution of Algorithm 1, and some query  $q \in H$ . Let  $U$  be the set of updates in  $H$ . The history of  $H|_{U \cup \{q\}}$  is  $r$ -relaxed strongly linearizable.

*Proof.* Let  $H$  be a history arising from a concurrent execution of Algorithm 1, let  $q \in H$  be some query by  $p_i$ , and let  $U$  be the set of all updates in  $H$ . Denote  $H|_{U \cup \{q\}}$  as  $H'$ . We show that  $H'$  is  $r$ -relaxed strongly linearizable with respect to  $\mathcal{H}^r$ , for  $r = 2NB$ . To prove this, we show the existence of two mappings,  $f$  and  $g$ , such that  $f$  maps operations in  $H'$  to *visibility points*, and  $g$  maps operations in  $H'$  to linearization points. Intuitively, visibility points are the time in the execution when an update is visible to a query, i.e., the query reflects the update. Bounding the number of preceding but not yet visible updates gives the relaxation.

We show that (1)  $f(H') \in \mathcal{H}$ , and (2)  $g(H')$  is an  $r$  relaxation of  $f(H')$ . Together, this implies the theorem.

The visibility points ( $f(H')$ ) are as follows:

- For the query, its visibility point is its return.
- For an update returning *false* at time  $t$ , its visibility point is  $t$ .
- For an update returning *true* at time  $t$ , let  $w$  be  $p_i$ 's value of *win* at time  $t$ . The visibility point is the first time after  $t$  that  $p_i$ 's value of *win* is  $w+2$ .

Note that in the latter case, the visibility point is after the update returns, so  $f$  does not preserve real-time order.

The linearization points ( $g(H')$ ) are as follows:

- An update's linearization point is its return, either *true* or *false*.
- A query's linearization point is its return.

By definition, the linearization points as defined by  $g(H')$  aren't decided post-facto – rather the linearization is a pre-determined point in the execution.

Consider some update  $u \in H'$  executed on some  $p_j$  that returns *true*. Let  $w$  be  $p_j$ 's value of *win* during its execution. Let  $w'$  be  $p_i$ 's value of *win* during  $q$ 's execution. We show that if  $w \leq w' - 2$ , then  $q$  observes  $u$ , and if  $w > w' - 2$ , then  $q$  doesn't observe  $u$ .

From the definition of Algorithm 1, for any  $win_i$  on  $p_i$  and  $win_j$  on  $p_j$ ,  $|win_i - win_j| \leq 1$ .

If  $w = w' - 2$ , then when  $p_j$  added  $u$  to its local buffers, it did so to  $obj[w \bmod 3]$ . As  $|win_i - win_j| \leq 1$ ,  $p_j$  advanced at least 1 window from  $w$ . When it did so, it sent  $obj[w \bmod 3]$  to  $p_i$ . In window  $w' - 1$ ,  $p_i$  merges the update into  $obj[w' + 1 \bmod 3]$ . In window  $w'$  this same variable is queried, thus  $q$  observes  $u$ . If  $w \leq w' - 3$ , then the update is merged into some index of the variables array, and is copied over until it is reflected in all 3 of them, and specifically reflected in  $obj[w' + 1 \bmod 3]$  in window  $w'$ .

If  $w \geq w' - 1$ , then when  $p_j$  added  $u$  into its local buffer it did so to  $obj[w \bmod 3]$ . This update is sent to  $p_i$  only in window  $w + 1$ , and therefore isn't reflected in  $obj[w' + 1 \bmod 3]$  in window  $w'$ .

Therefore,  $q$  reflects all updates that return true that happened during any window  $w \leq w' - 2$ . As there are at most  $B$  updates that return true in any window,  $q$  reflects all but at most  $2NB$  updates that precede it in  $H$ . Therefore,  $g(H')$  is an  $2NB$ -relaxation of  $f(H')$ .

As the query returns a value based on the updates that happened before it, and each access to the process local state is down sequentially,  $q$  returns a value that reflects all successful updates that happen before it in  $f(H')$ . Therefore,  $f(H') \in \mathcal{H}$ .  $\square$

Intuitively, every query returns a value reflecting a sub-stream of its preceding and concurrent updates, consisting of all but at most  $r$  successful ones. The upper bound  $r$  on the number of “missing” updates is of vast importance, without it

the drift between one switch and another can grow in an unbounded fashion. For example, consider a counter distributed among two switches running an eventually synchronous algorithm. One switch can increment the counter an arbitrarily large number of times, while the other returns 0 on every query – the promise of eventual synchrony is too weak.

Theorem 2 ensures that every history consisting of a single query and all updates is  $r$ -relaxed strongly linearizable, which in many cases preserves some relaxation of the error bounds. For example, Rinberg et al. [67] show that, under a weak adversary, a K-Minimum Value (K MV)  $\theta$  sketch [5] has an error of at most twice that of the sequential one. Another example is a relaxed Quantiles sketch [2], which has an additive error of  $r/n - (r\epsilon)/n$  with some tuning parameter  $\epsilon$ , where  $r$  is the relaxation and  $n$  is the stream size. Thus, the impact of the relaxation diminishes as the stream size grows.

---

**Algorithm 1:** Algorithm running on switch  $p_i$ .

---

```
1 initialization:
2 win ← 0
3 count ← 0
4 objs ← [ $obj.init()$ ,  $obj.init()$ ,  $o.init()$ ]
5 buf ← {}
6 rcvs ← {}
7 acks ← {}
8 Function Update( $v$ ):
9   if count ==  $B$  then
10    return false                                // Write variable is full
11   else
12    objs [win mod 3].update( $v$ )
13    count ← count + 1
14    return true                               // Add to the write variable
15
16 Function Query():
17  return objs [(win + 1) mod 3].query()      // Serve query from read variable
18
19 on receive “( $o'$ ,  $w'$ )” from  $p_j$ :
20   if  $w' > win$  then                         // Sync
21    buf ← buf ∪ {( $o'$ ,  $w'$ )}
22   else
23    rcvs ← rcvs ∪ { $j$ }
24    objs [(win + 2) mod 3].merge( $o'$ )          // Merge into sync buffer
25    send “ack” to  $p_j$ 
26    check_done()
27
28 on receive “ack” from  $p_j$ :
29  acks ← acks ∪ { $j$ }
30  check_done()
31
32 Function check_done():
33  if |rcvs| ==  $n$   $\&\&$  |acks| ==  $n$  then
34   count ← 0
35   win ← win + 1                                // Rotate right
36    $o' \leftarrow$  objs [win mod 3]
37   objs [(win + 1) mod 3].merge( $o'$ )          // Add the updates from window  $w$  to the current state
38   objs [win mod 3] ←  $o.init()$                   // Clear write variable
39   broadcast “(objs [(win + 2) mod 3], win)”     // Send sync message
40   rcvs ← { $i$ }
41   acks ← { $i$ }
42   forall ( $o', w'$ ) in buf do
43    rcvs ← rcvs ∪ { $j$ }                          // Handle buffered messages
44    objs [(win + 2) mod 3].merge( $o'$ )
45    send “ack” to  $p_j$ 
46   buf ← {}
```

---



# Modular Switch Programming Under Resource Constraints

Mary Hogan<sup>1</sup>, Shir Landau-Feibish<sup>2</sup>, Mina Tahmasbi Arashloo<sup>3</sup>, Jennifer Rexford<sup>1</sup>, and David Walker<sup>1</sup>

<sup>1</sup>Princeton University

<sup>2</sup>The Open University of Israel

<sup>3</sup>Cornell University

## Abstract

Programmable networks support a wide variety of applications, including access control, routing, monitoring, caching, and synchronization. As demand for applications grows, so does resource contention within the switch data plane. Cramming applications onto a switch is a challenging task that often results in non-modular programming, frustrating “trial and error” compile-debug cycles, and suboptimal use of resources. In this paper, we present P4All, an extension of P4 that allows programmers to define *elastic* data structures that stretch automatically to make optimal use of available switch resources. These data structures are defined using *symbolic primitives* (that parameterize the size and shape of the structure) and *objective functions* (that quantify the value gained or lost as that shape changes). A top-level optimization function specifies how to share resources amongst data structures or applications. We demonstrate the inherent modularity and effectiveness of our design by building a range of reusable elastic data structures including hash tables, Bloom filters, sketches, and key-value stores, and using those structures within larger applications. We show how to implement the P4All compiler using a combination of dependency analysis, loop unrolling, linear and non-linear constraint generation, and constraint solving. We evaluate the compiler’s performance, showing that a range of elastic programs can be compiled to P4 in few minutes at most, but usually less.

## 1 Introduction

P4 has quickly become a key language for programming network data planes. Using P4, operators can define their own packet headers and specify how the data plane should parse and process them [7]. In addition to implementing traditional forwarding, routing, and load-balancing tasks, this flexibility has enabled new kinds of in-network computing that can accelerate distributed applications [26, 27] and perform advanced monitoring and telemetry [10, 11, 17, 30].

All of these applications place demands on switch resources, but for many, the demands are somewhat flexible:

additional resources, typically memory or stages in the PISA pipeline, improve application performance, but do not necessarily make or break it. For instance, NetCache [27] improves throughput and latency for key-value stores via in-network computing. Internally, it uses two main data structures: a count-min sketch (CMS) to keep track of popular keys, and a compact key-value store (KVS) to maintain their corresponding values. Increasing or decreasing the size of those structures will have an impact on performance, but does not affect the correctness of the system—a cache miss may increase latency, but the correct values will always be returned for a given key. Other applications, such as traffic-monitoring infrastructure, have similar properties. Increasing the size of the underlying hash tables, Bloom filters, sketches, or key-value stores may make network monitoring somewhat more precise but does not typically result in all-or-nothing decisions.

Because resource constraints for these components are flexible, network engineers can, in theory, squeeze multiple different applications onto a single device. Unfortunately, however, doing so using today’s programming language technology is a challenging and error-prone task: P4 forces programmers to hardcode their decisions about the size and shape of their data structures. If the data structure is too large, the program simply fails to compile and little feedback is provided; if it is too small, it will compile but the resources will be used suboptimally. Moreover, structures are not reuseable: a cache, that fits just fine on a switch alongside a table for IP forwarding, is suddenly too large when a firewall is added. To squeeze the cache in, programmers may have to rewrite the internals of their cache, manually adjusting the number or sizes of the registers or match-action tables used. To test their work, they resort to a tedious trial-and-error cycle of rewriting their applications, and invoking the compiler to see if it can succeed in fitting the structures into the available hardware resources.

This manual process of tweaking the *internal* details of data structures, and checking whether the resulting structures satisfy *global* constraints, is inherently non-modular: Programmers tasked with implementing separate applications cannot do so independently. Indeed, while the same data structures

Data Structure	Used in
Key-value store/ hash table	Precision [6], Sonata [17], Network-Wide HH [19], Carpe [20], Sketchvisor [23], LinearRoad [25], NetChain [26], NetCache [27], FlowRadar [30], Hash-Pipe [41], Elastic Sketch [46]
Hash-based matrix (Sketch)	AROMA [4], Sketchvisor [23], Sketch-learn [24], NetCache [27], Nitrosketch [31], UnivMon [32], Sharma et al. [38], Fair Queueing [39], Elastic Sketch [46]
Bloom filter	NetCache [27], FlowRadar [30], SilkRoad [34], Sharma et al. [38]
Multi-value table	BeauCoup [10], Blink [22]
Sliding window sketch	PINT [5], Conquest [11]
Ring buffer	NetLock [47], Netseer [48]

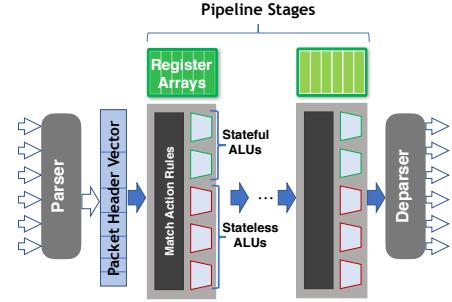
**Figure 1:** PISA data structures

appear again and again (see Figure 1 for a selection), the varying resource constraints makes it difficult to reuse these structures for different targets or applications.

**Elastic Switch Programming.** We extend P4 with the ability to write *elastic* programs. An elastic program is a single, compact program that can “stretch” to make use of available hardware resources or “contract” to squeeze in beside other applications. Elastic programs can be constructed from any number of elastic components that each stretch arbitrarily to fill available space. An elastic NetCache program, for example, may be constructed from an elastic count-min sketch and an elastic key-value store. The programmer can control the relative stretch of these modules by specifying an objective function that the compiler should maximize. For example, the NetCache application could maximize the cache “hit rate” by prioritizing memory allocation for the key-value store (to store more of the “hot” keys) while ensuring that enough remains for the count-min sketch to produce sufficiently accurate estimates of key popularity. In addition to memory, programs could simultaneously maximize the use of other switch resources such as available processing units and pipeline stages.

To implement these elastic programs, we present P4All, a backward-compatible extension of the P4 language with several additional features: (1) symbolic values, (2) symbolic arrays, (3) bounded loops with iteration counts governed by symbolic values, (4) local objective functions for data structures, and (5) global optimization criteria. Symbolic values make the sizes of arrays and other state flexible, allowing them to stretch as needed. Loops indexed by symbolic values make it possible to construct operations over elastic data structures. Objective functions provide a principled way for the programmer to describe the relative gain/loss from growing/shrinking individual data structures. Global optimization criteria make it possible to weight the relative importance of each structure or application residing on a shared device.

We have implemented a compiler for P4All that operates



**Figure 2:** Protocol Independent Switch Architecture (PISA)

in two main stages. First, it computes an upper bound on the number of possible iterations of loops, so it can produce a simpler optimization problem over unrolled, loop-free code. This upper bound is computed by conservatively analyzing the dependency structure of the loop bodies and their resource utilization. Next, the compiler unrolls the loops to those bounds and generates a constraint system that optimizes the resource utilization of the loop-free code for a particular target. We use the Intel Tofino chip as our target. We evaluate our system by developing a number of reusable, elastic structures and building several elastic applications using these structures. Our experiments show that the P4All compiler runs in a matter of minutes (or less) and produces P4 programs that are competitive with hand-optimized code. This paper builds on our earlier workshop paper [21] by extending the language for nonlinear objective functions over multiple variables. We also implement the optimization problem and compiler outlined in the workshop paper, along with evaluating it with a variety of data structures.

In summary, we make the following contributions.

- The design of P4All, a backward-compatible extension to P4 that enables elastic network programming.
- The implementation of an optimizing compiler for P4All.
- A library of reusable elastic data structures, including their objective functions, and examples of combining them to create sophisticated applications.
- An evaluation of our system on a range of applications.

## 2 P4 Programming Challenges

Programming PISA devices is difficult because the resources available are limited and partitioned across pipeline stages. The architecture forces programmers to keep track of implicit dependencies between actions, lay out those actions across stages, compute memory requirements of each task, and fit the jigsaw pieces emerging from many independent tasks together into the overall resource-constrained puzzle of the pipeline.

## 2.1 Constrained Data-Plane Resources

P4 is designed to program a *Protocol Independent Switch Architecture* (PISA) data plane (Figure 2). Such an architecture contains a programmable packet parser, processing pipeline, and deparser. When a packet enters the switch, the parser extracts information from the packet and populates the *Packet Header Vector* (PHV). The PHV contains information from the packet’s various fields, such as the source IP, TCP port, etc. that are relevant to the switch’s task, whether it be routing, monitoring, or load balancing. The PHV also stores additional per-packet data, or *metadata*. Metadata often holds temporary values or intermediate results required by the application. Finally, the deparser reverses the function of the parser, using the PHV to reconstitute a packet and send it on its way.

Between parser and deparser sits a packet-processing pipeline. A program may recirculate a packet by sending it back to the beginning, but too much recirculation decreases throughput. Each stage contains a fixed set of resources.

- **Pipeline stages.** The processing pipeline is composed of a fixed number ( $S$ ) of stages.
- **Packet header vector (PHV).** The PHV that carries information from packet fields and additional per-packet metadata through the pipeline has limited width ( $P$  bits).
- **Registers.** A stage is associated with  $M$  bits of registers (of limited width) that serve as persistent memory.
- **Match-action rules.** Each stage stores match-action rules in either TCAM or SRAM ( $T$  bits).
- **ALUs.** Actions are performed by ALUs associated with a stage. Each stage has  $F$  stateful ALUs (that perform actions requiring registers) and  $L$  stateless ALUs (that do not).
- **Hash units.** Each stage can perform  $N$  hashes at once.

The P4 language helps manage data-plane resources by providing a layer of abstraction above PISA. A P4 compiler maps these higher-level abstractions down to the PISA architecture and organizes the computation into stages. However, experience with programming in P4 suggests, that while a good start, the language is simply *not abstract enough*. It asks programmers to make fixed choices ahead of time about the size of data structures and the amount of computation the programmer believes the compiler can squeeze onto a particular PISA switch. To do this well, programmers must recognize dependencies between actions, estimate the stages available and consider the memory layout and usage of their programs—in short, they must redo many of the jobs of the compiler. These are difficult jobs to do well, even for world-experts, and next to impossible for novices. Inevitably, attempts at estimating resource bounds leads to some amount of trial and error. In summary, the current development environment requires a lot of fiddly, low-level work and takes human time and energy away from innovating at a high level of abstraction.

## 2.2 Example: Implementing NetCache in P4

To illustrate some of the difficulties of programming with P4, consider an engineer in charge of upgrading their network to include a new caching subsystem, based on NetCache [27], which is designed to accelerate response times for web services. NetCache contains two main data structures, a *count-min sketch* (CMS) for keeping track of the popularity of the keys, and a *key-value store* (KVS) to map popular keys to values. Like any good programmer, our engineer constructs these two data structures modularly, one at a time.

First, the engineer implements the CMS, a probabilistic data structure that uses multiple hash functions to keep approximate frequencies for a stream of items in sub-linear space. Intuitively, the CMS is a two-dimensional array of  $w$  columns and  $r$  rows. For each packet ( $x$ ) that enters the switch, its flow ID ( $f_x$ ) is hashed using  $r$  different hash functions ( $\{h_i\}$ ), one for each row, that range from  $(1 \dots w)$ . In each row, the output of the hash function determines which column in the row is incremented for  $f_x$ . For example, in the second row of the CMS, hash function  $h_2$  determines that column ( $h_2(f_x)$ ) is incremented. To approximate the number of times flow  $f_x$  has been seen, one computes the minimum of the values stored in columns  $h_i(f_x)$  for all  $r$  rows.

The CMS may overestimate the number of occurrences of a packet  $x$  if there are hash collisions. Increasing the size of the sketch in any dimension—either by adding more rows (*i.e.*, additional, different hash functions) or by increasing the range of the hash functions—can improve accuracy. Our engineer must decide how to assign resources to the CMS, including how much memory to allocate and how to divide memory into rows. This allocation becomes even harder when grappling with dividing resources between multiple structures.

Figure 3 presents a fragment of a P4 program that implements a CMS. Lines 1-7 declare the metadata used by the CMS to store a count at a particular index (a hash of a flow id). Lines 10-12 declare the low-level data structures (registers) that actually make up the CMS—four rows ( $r = 4$ ) of columns ( $w = 2048$ ) that can each store values represented by 32 bits. Lines 14-16 and 18-20 declare the actions for hashing/incrementing and for updating the metadata designed to store the global minimum. Both actions use metadata, another constrained resource that must be accounted for. The hashing action is a complex action containing several atomic actions: (1) an action to hash the key to an index into a register array, (2) an action to increment the count found at the index, and (3) an action to write the result to metadata for use later in finding the global minimum. Such multi-part actions can demand a number of resources, including several ALUs. As our engineer adds more of these actions to the program, it becomes increasingly difficult to estimate the resource requirements. In the *apply* fragment of the P4 program (lines 22-30), the program first executes all the hash actions, computing and storing counts for each hash function, and then compares those counts

```

1 struct custom_metadata_t {
2     bit<32> min;
3     bit<32> index0;
4     bit<32> count0;
5     ...
6     bit<32> index3;
7     bit<32> count3; }
8 control Ingress( ... ) {
9     /* a register array for each hash table */
10    register<bit<32>>(2048) counter0;
11    ...
12    register<bit<32>>(2048) counter3;
13    /* an action to update each hash table */
14    action incr_0() { ... }
15    ...
16    action incr_3() { ... }
17    /* an action to set the minimum */
18    action min_0(){meta.min = meta.count0;}
19    ...
20    action min_3(){ . . . }
21    /* execute the following on each packet */
22    apply {
23        meta.min = 0; /*initialize global min*/
24        /* compute hashes */
25        incr_0(); ... incr_3();
26        /* compute minimum */
27        if (meta.count0 < meta.min) { min_0();}
28        ...
29        if (meta.count3 < meta.min) { min_3();}
30    } }
```

**Figure 3:** Count-Min Sketch in P4<sub>16</sub>

to each other looking for the minimal one.

Upon reviewing this code, some of the deficiencies of P4 should immediately be apparent. First, there is a great deal of repeated code: Repeated data-structure definitions, action definitions, and invocations of those action definitions in the apply segment of the program. Good programming languages make it possible to avoid repeated code by allowing programmers to craft reusable abstractions. Avoiding repetition in programming has all sorts of good properties including the fact that when errors occur or when changes need to be made, they only need to be fixed/made in one place. Effective abstractions also help programmers change the number or nature of the repetitions easily. Unfortunately, P4 is missing such abstractions. One might also notice that the programmer had to choose magic constants (like 2048) and test whether such constants lead to programs that can be compiled or not.

### 3 Elastic Programming in P4All

P4All improves upon P4 by making it possible to construct and manipulate *elastic data structures*. These data structures may be developed modularly and combined, off-the-shelf, to build efficient new applications. In this section, we illustrate language features by building an elastic count-min sketch and using it in the NetCache application (see also Figure 4).

```

1 /* Count-min sketch module */
2 symbolic rows;
3 symbolic cols;
4 assume cols > 0;
5 assume 0 <= rows && rows < 4;
6 struct custom_metadata_t {
7     bit<32> min;
8     bit<32>[rows] index;
9     bit<32>[rows] count; }
10 register<bit<32>>(cols)[rows] cms;
11 action incr(){int index] { ... }
12 action min(){int index] { ... }
13 control hash_inc( ... ) {
14     apply {
15         for (i < rows) { incr()[i]; } } }
16 control find_min( ... ) {
17     apply {
18         for (i < rows) {
19             if (meta.count[i] < meta.min) {
20                 min()[i]; } } } }
21 objective cms_obj {
22     function: scale (3.0/cols);
23     step: 100; }
24
25 /* Key-value module */
26 symbolic k; /* number of items */
27 assume k > 0;
28 control kv(...){....}
29 /* NetCache module */
30 control NetCache( ... ) {
31     apply {
32         hash_inc.apply();
33         find_min.apply();
34         kv.apply(); } }
35 objective kvs_obj {
36     function: scale (sum(map(lambda y: 1.0/
37         y,range(1,k+1))));
38     step: 100; }
39 maximize 0.8*kvs_obj-0.2*cms_obj
```

**Figure 4:** NetCache and Count-Min Sketch in P4All

### 3.1 Declare the Elastic Parameters

The first step in defining an elastic data structure is to declare the parameters that control the “stretch” of the structure. In the case of the count-min sketch there are two such parameters: (1) the number of rows in the sketch (*i.e.*, the number of hash functions), and (2) the number of columns (*i.e.*, the range of the hash). Such parameters are defined as *symbolic values*:

```

symbolic rows;
symbolic cols;
```

Symbolic integers like `rows` and `cols` should be thought of as “some integer”—they are placeholders that are determined (and optimized for) at compile time. In other words, as in other general-purpose, solver-aided languages like Boogie [29], Sketch [42], or Rosette [43], the programmer leaves the choice of value up to the P4All compiler.

Often, programmers know constraints that are unknown to the compiler. For instance, programmer experience might suggest that count-min sketches with more than four hash

functions offer diminishing returns. Such constraints may be written as assume statements as follows:

```
assume 0 <= rows && rows < 4;
```

An assume statement is related to the familiar assert statement found in languages like C. However, an assert statement *fails* (causing program termination) when its underlying condition evaluates to false. An assume statement, in contrast, always *succeeds*, but adds constraints to the system, guaranteeing the execution can depend upon the conditions assumed.

## 3.2 Declare Elastic State

The next step in defining an elastic data structure is to declare elastic state. P4 data structures are defined using a combination of the packet-header vector (metadata associated with each packet), registers (updated within the data plane), or match-action tables (rules installed by the control plane). The same is true of P4All. However, rather than using constants to define the extent of the state, one uses symbolic values, so the compiler can optimize their extents for the programmer.

In the count-min sketch, each row may be implemented as a register array (whose elements, in this case, are 32-bit integers used as counters). The number of registers in each register array is the number of columns in a row. In P4All, we define this matrix as a symbolic array of register arrays:

```
register<bit<32>>(cols)[rows] cms;
```

In this declaration, we have a symbolic array `cms`, which contains `rows` instances of the register type. Each register array holds `cols` instances of 32-bit values.

One can also define elastic metadata. For instance, for each row of the CMS, we need metadata to record an index and count for that row. To do so, we define symbolic arrays of metadata as follows. Each element of each array is a 32-bit field. The arrays each contain `rows` items.

```
bit<32>[rows] index;
bit<32>[rows] count;
```

## 3.3 Define Elastic Operations

Because elastic data structures can stretch or contract to fit available resources, elastic operations over those data structures must do more or less work in a corresponding fashion. To accommodate such variation, P4All extends P4 with loops whose iteration count may be controlled by symbolic values.

The count-min sketch of our running example consists of two operations. The first operation hashes the input `rows` times, incrementing the result found in the CMS at that location, and storing the result in the metadata. The second iterates over this metadata to compute the overall minimum found at all hash locations. Each operation is implemented using symbolic loops and is encapsulated in its own control block. The code below illustrates these operations.

```
/* actions used in control segments */
action incr()[int i] { ... }
action min()[int i] { ... }
/* hash and increment */
control hash_inc( ... ) {
    apply {
        for (i < rows) {
            incr()[i]; } } }
/* find global minimum */
control find_min( ... ) {
    apply {
        for (i < rows) {
            if (meta.count[i] < meta.min) {
                min()[i]; } } } }
```

These simple symbolic iterations (`for i < rows`) iterate from zero up to the symbolic bound (`rows`), incrementing the index by one each time. The overarching NetCache algorithm can now call each control block in the ingress pipeline.

```
control NetCache( ... ) {
    apply {
        hash_inc.apply(...);
        find_min.apply(...);
        ... } }
```

## 3.4 Specify the Objective Function

Data structures written for programmable switches are valid for a range of sizes. In the CMS example above, multiple assignments to `rows` and `cols` might fit within the resources of the switch. Finding the right parameters becomes even harder when a program has multiple data structures. In the case of NetCache, after defining a CMS, the programmer still needs to define and optimize a key-value store.

To automate the process of selecting parameters, P4All allows programmers to define an objective function that expresses the relationship between the utility of the structure and its size (as defined by symbolic values). For example, the CMS gains utility as one increases the `cols` parameter, because CMS error rate decreases. The P4All compiler should find instances of the symbolic values that optimize the given user-defined function subject to the constraint that the resulting program can fit within the switch resources.

For example, we can define the hit ratio for the key-value store as a function of its size for a workload with a Zipfian distribution. Suppose the key-value store has  $k$  items. The probability of a request to the  $i^{th}$  most popular item is  $\frac{1}{\alpha}$  [9]. In this case,  $\alpha$  is a workload-dependent parameter that captures the amount of skew in the distribution. Then, for  $k$  items, the probability of a cache hit is the sum of the probabilities for each item in the key-value store:  $\sum_{i=1}^k \frac{1}{\alpha}$ . Hence, in P4All, for  $\alpha = 1$ , we might define the following objective function.

```
sum(map(lambda y: 1.0/y, range(1, k+1)))
```

In practice, we have found that non-linear optimization functions that use division can generate poor quality solutions, perhaps due to rounding errors (at least for the solver,

Gurobi [18], that we use). Hence, we *scale* such functions up, which results in the following optimization function.

```
scale(sum(map(lambda y: 1.0/y, range(1,k+1))))
```

Because we supply programmers with a library of reusable structures and optimization functions for them, non-expert programmers who use our libraries do not have to concern themselves with such details.

Similarly, we can define CMS error,  $\epsilon$ , in terms of the number of columns,  $w$ , in the sketch. For a workload with parameter  $\alpha$ , we can set  $w = 3(1/\epsilon)^{1/\alpha}$  [13]. The number of rows in the CMS does not affect  $\epsilon$ , so we may choose to leave it out of the objective function. However, we can incorporate constraints to guarantee a minimum number of rows. The number of rows,  $d$ , in a CMS is used to determine a bound on the confidence,  $\delta$ , of the estimations in the sketch ( $d = 2.5 \ln 1/\delta$ ) [13]. For  $\alpha = 1$ , this objective function is  $3.0/\text{cols}$ .

In NetCache, the programmer must decide if either data structure should receive a higher proportion of the resources. If the CMS is prioritized, it can more accurately identify heavy hitters. However, the key-value store may not have sufficient space to store the frequently requested items. Conversely, if the CMS is too small, it cannot accurately measure which keys are popular and should be stored in the cache.

To capture the balance between data structures, a programmer can combine the objectives of each data structure into a weighted sum. For the NetCache application, this means creating an objective function that slightly prioritizes the hit rate of the key-value store over the error of the CMS:

```
maximize 0.8*kvs_obj - 0.2*cms_obj
```

Figure 5 presents the symbolic values and possible objective functions for different data structures. Each structure has symbolic values and an objective function derived from the purpose of the structure, which may vary across applications. For example, the key-value store used in NetCache [27] acts as a cache, and the main goal of the algorithm is to maximize the cache hits. In the case of a collision in the hash table used in BeauCoup [10], only one of the values is kept, and the other is discarded, resulting in possible errors. Therefore, the main goal of the algorithm is to minimize collisions. The programmer can define the objective function of each structure based on the specific needs of the system. Existing analyses of common data structures can assist in defining these functions. For example, for the Bloom filter, the probability for false positives in Zipfian-distributed traffic has been analyzed by Cohen and Matias [12].

**Complex Objectives.** Some objective functions (*e.g.*, CMS) may only include a single symbolic variable, while others are a function of multiple variables (*e.g.*, Bloom filter in Figure 5). Because our compiler uses Gurobi [18] in the back end to solve optimization problems, it is bound by

Gurobi’s constraints. In particular, Gurobi cannot solve complex, non-linear objectives that are functions of multiple variables directly. As a consequence, we tackle these objectives in two steps. First, we transform objectives in multiple variables (say,  $x$  and  $y$ ) into objectives in a single variable (say  $x$ ), by choosing a set of possible values of  $y$  to consider. We create a different Gurobi instance for each value of  $y$ , solve all the instances independently (a highly parallelizable task) and find the global optimum afterwards. Second, we use Gurobi to implement piece-wise linear approximations of the non-linear functions. Both of these steps benefit from some user input, and we have extended P4All to accommodate such input.

To reduce objectives with multiple variables to a single variable, we allow users to provide a set of points at which to consider evaluating certain symbolic values. Doing so provides users some control over the number of Gurobi instances generated and hence the compilation costs of solving complex optimization problems. Such sets can be generated via “range notation” (optionally including a stride, not shown here). For example, a possible objective function for a Bloom filter depends on the number of bits in the filter as well as the number of hash functions used. To eliminate the second variable from the subsequent optimization objective, a programmer can define the symbolic variable `hashes` as follows.

```
symbolic hashes [1..10]
```

On processing such a declaration, the compiler generates ten separate optimization problems, one for each potential value of the hash functions. The compiler chooses the solution from the instance that generated the optimal objective, and it outputs the program layout and the concrete values for the number of hashes and number of bits in the filter.

To reduce non-linear functions to linear ones, piecewise linear approximations are used. By default, the compiler will use the simplest such approximation: a single line. Doing so results in fast compile times, but can lead to suboptimal solutions. To improve the quality of solution, we allow programmers to specify the number of linear pieces using a “step” annotation on their objective function. For instance, on lines 21-23 of Figure 4, the objective for the CMS is defined with a simple function and a “step” of 100, indicating that a linear component is created between every 100th value. Increasing the number of linear components in the approximation can increase the cost of solving these optimization problems. By providing programmers with optional control, we support a “pay-as-you-go” model that allows programmers to trade compile time for precision if they so choose.

## 4 Compiling Elastic Programs

Inputs to the P4All compiler include a P4All program and a specification of the target’s resources (*i.e.*, the PISA resource parameters defined Section 2.1 and the capabilities of the ALUs). The compiler outputs a P4 program with a concrete

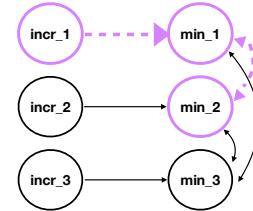
Module	Symbolic values	Intuition	Objective Function
Key-value store/ hash table	Number of rows $k$	NetCache [27]: Maximize cache hits	maximize $\sum_{i=1}^k \frac{1}{i^\alpha}$
Hash-based matrix (Sketch)	Num rows $d$ , num columns $w$	NetCache [27] (CMS): Minimize heavy hitter detection error	minimize $\epsilon = (\frac{3}{w})^\alpha$
Bloom filter	Num bits $m$ , num hash functions $k$	NetCache [27]: Minimize false positives. Expected number of items in stream $n$	minimize $(1 - e^{-\frac{kn}{m}})^k$
Multi-value table	Number of rows $k$	BeauCoup [10]: Minimize collisions. BeauCoup parameter set $B$ : Probability to insert to table $p = f(\alpha, B)$ ; Expected number of items in stream $n$	minimize $(\frac{1}{k})^{n \cdot p}$
Sliding window sketch	Num rows $d$ , num columns $w$ , num epochs $t$	ConQuest [11]: Maximize epochs and minimize error	maximize $t(1 - (\frac{3}{w})^\alpha)$
Ring buffer	Buffer length $b$	Netseer [48]: Maximize buffer capacity	maximize $b$

**Figure 5:** Symbolic values and objective functions for Zipfian distributed traffic with (constant) parameter  $\alpha$ .

assignment for each symbolic value, and a mapping of P4 program elements to stages in the target’s pipeline. The output program is a *valid instance* of the input when the concrete values chosen to replace symbolic ones satisfy the user constraints (*i.e.*, assume statements) as well as the constraints of the PISA model that is targeted. In addition, loops are unrolled as indicated given the chosen concrete values. The output program is an *optimal instance*, when in addition to being valid, it maximizes the given objective function.

The P4All compiler first analyzes the control and data dependencies between actions in the program to compute an *upper bound* on the number of times each loop can be unrolled without exhausting the target’s resources (§4.1). For example, a for-loop with a dependency across successive iterations cannot run more times than the number of pipeline stages ( $S$ ). The unrolled program also cannot require more ALUs than exist on the target  $((F + L) * S)$ .

Next, the compiler generates an integer linear program (ILP) with variables and constraints that govern the quantity and placement of actions, registers, and metadata relative to the target constraints (§4.2). The upper bound ensures this integer linear program is “large enough” to consider all possible placements of program elements that can maximize the use of resources. However, the ILP is more accurate than the coarse unrolling approximation we use. Hence, it may generate a solution that excludes some of the unrolled iterations—some of the later iterations may ultimately not “fit” in the data plane or may not optimize the user’s preferred objective function when other constraints are accounted for. The resulting ILP solution is a layout of the program on the target, including the stage placement and memory allocation, and optimal concrete assignments for the symbolic values. Throughout this section, we use the CMS program in Figure 4 as a running example. For the sake of the example, we assume that the target has three pipeline stages ( $S = 3$ ), 2048b memory per stage ( $M = 2048$ ), two stateful and two stateless ALUs per stage ( $F = L = 2$ ), and 4096 bits of PHV ( $P = 4096$ ).



**Figure 6:** An example dependency graph used for computing upper bounds for loop unrolling (§4.1).

## 4.1 Upper Bounds for Loop Unrolling

In its first stage, the P4All compiler finds upper bounds for symbolic values bounding the input program’s loops. To find an upper bound for a symbolic value  $v$  governing the number of iterations of some loop, the compiler first identifies all of the loops bounded by  $v$ . It then generates a graph  $G_v$  that captures the dependencies between the actions in each iteration of each loop and between successive iterations. It uses the information represented in  $G_v$  and the target’s resource constraints to compute the upper bound.

**Determining dependencies.** When a loop is unrolled  $K$  times, it is replaced by  $K$  repetitions of the code in its body such that in repetition  $i$ , each action  $a$  in the original body of the loop is renamed to  $a_i$ . The compiler constructs the dependency graph  $G_v$  based on the actions in the unrolled bodies of for-loops bounded by  $v$ . Each node  $n$  in the dependency graph  $G_v$  represents a set  $A_n$  of actions that access the same register and thus *must* be placed in the same stage.

Dependency graphs can have (1) *precedence edges*, which are one-way, directed edges, and (2) *exclusion edges*, which are bidirectional. There is a precedence edge from node  $n_1$  to node  $n_2$  (indicated with the notation  $n_1 \rightarrow n_2$ ) if there is a data or control dependency from any of the actions represented by  $n_1$  to any of the actions represented by  $n_2$ . The presence of the edge  $n_1 \rightarrow n_2$  forces all actions associated with  $n_1$  to be placed in a stage that strictly precedes the stage where actions of  $n_2$  are placed. In contrast, an exclusion edge  $(n_1 \leftrightarrow n_2)$  indicates the actions of  $n_1$  must be placed in a

separate stage from the actions of  $n_2$  but  $n_1$  need not precede  $n_2$ . In general, when actions are commutative, but cannot share a stage, they will be separated by exclusion edges. For instance, if actions  $a_1$  and  $a_2$  both add one to the same metadata field, they cannot be placed in the same stage, but they commute:  $a_1$  may precede  $a_2$  or  $a_2$  may precede  $a_1$ .

Figure 6 shows the dependency graph for `rows` from our CMS example. Only the `incr_i` actions access register arrays, and they all access different arrays. Thus, each node represents only one action. There is a precedence edge from `incr_i` to `min_i` as the former writes to the same metadata variable read by the latter. Thus, `incr_i` must be placed in a stage preceding `min_i`. There are exclusion edges between each pair of `min_i` and `min_j` because they are commutative but write to the same metadata fields: `min_i` sets the metadata variable tracking the global minimum `meta.min` to the minimum of its current value and the  $i$ th row of the CMS (`meta.count[i]`).

**Computing the upper bound.** To compute an upper bound for loops guarded by  $v$ , our compiler unrolls for-loops bounded by  $v$  for increasing values of  $K$ , generating a graph  $G_v$  until one of the following two criteria are satisfied:

1. the length of the longest simple path in  $G_v$  exceeds the total number of stages  $S$ , or
2. the total number of ALUs required to implement actions across all nodes in  $G_v$  exceeds the total number of ALUs on the target (*i.e.*,  $(F + L) * S$ ).

Once either of the above criteria are satisfied, the compiler can use the current value of  $K$ , *i.e.*, the number of times the loops have been unrolled, as an upper bound for  $v$ . This is because any simple path in  $G_v$  represents a sequence of actions that must be laid out in disjoint stages. Hence, a simple path longer than the total number of stages cannot be implemented on the switch (*i.e.*, criteria 1). Likewise, the switch has only  $(F + L) * S$  ALUs and a computation that requires more cannot be implemented (*i.e.*, criteria 2).

Figure 6 presents an analysis of a CMS loop bounded by `rows`. Notice that the length of the longest simple path in  $G_{rows}$  will exceed the number of stages ( $S = 3$ ) when three iterations of the loop have been unrolled. On the other hand, when only two iterations of the loop are unrolled, the longest simple path has length 3 and will fit. Thus, the compiler computes 2 as the upper bound for this loop.

**Nested loops.** To manage nested loops, we apply the algorithm described above to each loop, making the most conservative assumption about the other loops. For instance, suppose the program has a loop with nesting depth 2 in which the outer loop bounded by  $v_{out}$  and the inner loop is bounded by  $v_{in}$ . Assume also the valid range of values for both  $v_{in}$  and  $v_{out}$  is  $(1, \infty]$ . The compiler sets  $v_{in}$  to one, unrolls the inner loop, and computes an upper bound for  $v_{out}$  as described above. Next, the compiler sets  $v_{out}$  to one, unrolls the outer loop, and proceeds to compute the upper bound for  $v_{in}$  as described

Variables	
Actions	#1 $\{x_{a_i,s} \mid 0 \leq s < S\}$
Registers	#2 $\{m_{r_i,s} \mid 0 \leq s < S\}$
Match-Action Tables	#3 $\{tm_{t_i,s} \mid 0 \leq s < S\}$
Metadata	#4 $\{d_i \mid i \leq U_v\}$
Constraints	
<b>Dependencies</b>	#5 $x_{a_i,s} = x_{b_i,s} \quad 0 \leq s < S$
	#6 $x_{a_i,s} \leq 1 - x_{b_i,s}$ $s < S$
	#7 $x_{b_i,y} \leq 1 - x_{a_i,z}$ $y, z < S, y \leq z$
	#8 $\sum_{0 \leq s < S} x_{a_i,s} = \sum_{0 \leq s < S} x_{b_i,s}$ $0 \leq i \leq U_v$
<b>Resources</b>	#9 $\sum_i m_{r_i,s} \cdot w_{r_i} \leq M \quad \forall s < S$
	#10 $m_{r_i,s} \leq x_{a_i,s} \cdot M \quad 0 \leq s < S$
	#11 $m_{r_i,s} \cdot w_0 = m_{0,s} \cdot w_{r_i}$ $\forall s < S, r \geq 1$
	#12 $\sum_i tm_{t_i,s} \cdot tw_{t_i} \leq T \quad \forall s < S$
	#13 $\sum_i H_f(a_i) \cdot x_{a_i,s} \leq F$ $\forall 0 \leq s < S$
	#14 $\sum_i H_l(a_i) \cdot x_{a_i,s} \leq L$ $\forall 0 \leq s < S$
	#15 $\sum_i d_i \cdot bits_d \leq P - P_{fixed}$ #16 $d_i = \sum_{0 \leq s < S} x_{a_i,s}$ if accesses( $a, d$ )
<b>Others</b>	#17 $\sum_i h_{ha_i,s} \leq N \quad \forall s < S$
	#18 $\sum_{0 \leq s < S} x_{a_i,s} \leq 1$
	#19 $\sum_{0 \leq s < S} x_{a_{ne},s} = 1$

Figure 7: ILP Summary

above. In theory, heavily nested loops could lead to an explosion in the complexity of our algorithm, but in practice, we have not found nested loops common or problematic. Only our SketchLearn application requires nested loops and the nesting depth is just 2, which is easily handled by our system.

## 4.2 Optimizing Resource Constraints

After unrolling loops, the compiler has a loop-free program it can use to generate an integer linear program (ILP) to optimize. Figure 7 summarizes the ILP variables and constraints. Below, we use the notation # $k$  to refer to the ILP constraint or variable labeled  $k$  in Figure 7.

**Action Variables.** To control placement of actions, the compiler generates a set of ILP variables named  $x_{a_i,s}$  (#1). The variable  $x_{a_i,s}$  is 1 when the action  $a_i$  appears in stage  $s$  of the pipeline and is 0 otherwise. For instance, in the count-min sketch, there are two actions (`incr` and `min`). If we unroll a loop containing those actions twice and there are three stages in the pipeline, we generate the following action variable set.

$$\{x_{a_i,s} \mid a \in \{\text{incr, min}\}, 1 \leq i \leq 2, 0 \leq s < S\}$$

**Register Variables.** In a PISA architecture, any register accessed by an action must be placed within the same stage. Thus placement (and size) of register arrays interact with placement of actions. For each register array  $r$  and pipeline

stage  $s$ , the ILP variable  $m_{r,s}$  contains the amount of memory used to represent  $r$  in stage  $s$  (#2). This value will be zero in any stage that does not contain  $r$  and its associated actions. For instance, to allocate the `cms` registers, the compiler uses:

$$\{m_{\text{cms}_i,s} \mid 1 \leq i \leq 2, 0 \leq s < S\}$$

**Match-Action Table Variables.** These variables represent the resources used by match-action tables. Similar to register variables, the variable  $tm_{t_i,s}$  represents the amount of TCAM used by table  $t_i$  in stage  $s$  (#3). Note that in our current ILP, we assume that all tables, ones with and without ternary matches, use TCAM. We plan to extend the ILP so that it can choose to implement tables without ternary matches in SRAM.

**Metadata Variables.** The amount of metadata needed is also governed by symbolic values. If  $U_v$  is the upper bound on the symbolic value that governs the size of a metadata array, then the compiler generates a set of metadata variables  $d_i$  for  $1 \leq i \leq U_v$  (#4). Each such variable will have value 1 in the ILP solution if that chunk of metadata is required and constraints described later will bound the total metadata to ensure it does not exceed the target size limits. In our running example, the bound  $U_v$  corresponds to the number of iterations of the loop that finds the global minimum value in the CMS.

**Dependency Constraints.** If a set of actions use the same register, they must be placed on the same stage. To do so, the compiler adds a *same-stage constraint* (#5). Similarly, if an action has a data or control dependency on another action, the two must be placed in separate stages. If there is an exclusion edge between actions  $a_i$  and  $b_i$ , the compiler creates a constraint to prevent these actions from being placed in the same stage (#6). If there is a precedence edge between actions  $a_i$  and  $b_i$ , the compiler creates a constraint forcing  $a_i$  to be placed in a stage before  $b_i$  (#7).

**Conditional Constraints.** In some cases, as it happens in our CMS example, multiple loops are governed by the same symbolic values. Hence, iterations of one loop (and the corresponding actions/metadata) exist if and only if the corresponding iterations of the other loop exist. Moreover, if any action within a loop iteration cannot fit in the data plane, then the entire loop iteration should not be instantiated at all. Conditional constraints (#8) enforce these invariants.

**Resource Constraints.** We generate ILP constraints for each of the resources listed in §2.1. Our ILP constraints reflect the memory limit per stage (#9) and the fact that memory and corresponding actions must be co-located (#10). The compiler also generates constraints to ensure that each register array in an array of register arrays has the same size (#11). Moreover, the ILP includes a constraint to guarantee that the TCAM tables in a stage fit within a stage’s resources (#12).

To enforce limits on the number of stateful and stateless ALUs used in each stage, we assume that the target provides two functions  $H_f(a_i)$  and  $H_l(a_i)$  as part of the target specification. These functions specify the number of stateful and stateless ALUs, respectively, required to implement a given

action  $a_i$  on the target. Given that information, the compiler generates constraints to ensure that the total number of ALUs used by actions in the same stage do not exceed the available ALUs in a stage (#13, #14).

To track the use of PHV, constraint #15 ensures  $d_i$  is 1 whenever the action  $a_i$  (which accesses data  $d_i$ ) is used in loop iteration  $i$ . To limit the total number of PHV bits, constraint #16 sums the size in bits ( $bits_d$ ) of the metadata  $d$  associated with iteration  $i$  and enforces it to be within the PHV bits available to elastic program components ( $P - P_{fixed}$ , where  $P_{fixed}$  is the amount of metadata not present in elastic arrays). Finally, each stage in the PISA pipeline can perform a limited number of hash functions. To capture that, the compiler generates constraint #17, which ensures that the number of actions including a hash function  $h$  in each stage does not exceed the available number of hashing units  $N$ .

**Other Constraints.** The compiler generates a constraint so that each action  $a_i$  is placed at most once (#18). Moreover, the compiler ensures that each *inelastic* action  $a_{ne}$  (*i.e.*, an action not encapsulated in a loop bounded by a symbolic value) must be placed in the pipeline (#19). Finally, any assume statements appearing in the P4All program are included in the ILP.

### 4.3 Limitations

Our current ILP formulation assumes each register array and match-action table can be placed in at most one stage. However, a PISA target could conceivably spread a single array or table across multiple pipeline stages. To accommodate multi-stage arrays or tables, we can relax the ILP constraint on placing actions in at most one stage (#18).

Moreover, some compilers further optimize the use of the PHV. For example, after a metadata field has been accessed, the PHV segment storing that field could be overwritten in later stages if the metadata were never accessed again. Our prototype does not yet capture PHV field reuse.

P4All optimizes with mostly static criteria. We do not consider any dynamic components, unless a programmer incorporates a workload-dependent parameter in their objective function. P4All also does not support elastic-width fields or parameterized packet recirculation. We leave these features, as well as PHV reuse, for future work.

## 5 Prototype P4All Compiler

In this section, we describe our prototype P4All compiler, written in Python.

**Target specification.** We created a target specification for the Intel Tofino switch, based on product documentation. The specification captures the parameters in Section 2.1 and the  $H_f$  and  $H_l$  functions that specify the number of ALUs required to implement a given action. Since the Tofino design is proprietary, our specification unquestionably omits some low-level constraints not described in the documentation; with

Applications	P4All Code	Compile Time (sec)	ILP (Var, Constr)
<b>Linear Objective</b>			
IPv4 Forwarding + Stateful Firewall	217	0.4	(192, 1026)
BeauCoup	541	0.1	(672, 7511)
Precision	166	25.7	(1316, 18969)
NetChain	242	27.9	(252, 3278)
Elastic Switch.p4	804	0.2	(1080, 21581)
<b>Non-Linear Objective</b>			
Key-value store (KVS)	127	15.4	(168, 857)
Count-min sketch (CMS)	82	1.8	(396, 1994)
KVS + CMS (Section §3)	170	27.9	(586, 2815)
Non-Elastic Switch.p4 + CMS	853	17.5	(1498, 23575)
SketchLearn	445	2.4	(768, 880)
ConQuest	362	5.8	(612, 3734)
<b>Multivariate Objective</b>			
Bloom filter	70	513.6 (longest) 170.0 (avg)	(240, 308) (132, 191)
CMS + Bloom	223	67.3 (longest) 38.1 (avg)	(658, 2266) (550, 2149)

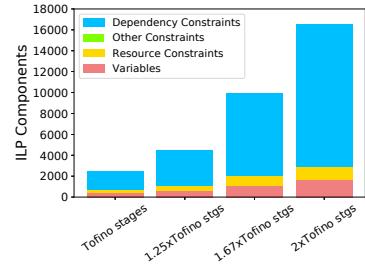
**Figure 8:** P4All applications, showing the lines of code in the P4All implementation. For structures with multiple instances, the last two columns give statistics for the single instance with the *longest* compile time and the *average* of all instances.

knowledge of such constraints, we could augment our target specification and optimization framework to handle them.

**Compute upper bounds for symbolic values.** To compute upper bounds and unroll loops, our prototype must analyze P4 dependencies. To facilitate this, we use the Lark toolkit [1] for parsing. We have also written a Python program that finds dependencies between actions and tables and outputs the information in a format our ILP can ingest. At the moment, we only produce precedence edges. As a result, we do not process exclusion edges, treating all edges as precedence edges. We plan to upgrade this in the future.

**Generate and solve ILP.** Our prototype generates the ILP with variables and constraints in Figure 7, as well as the objective function. We then invoke the Gurobi Optimizer [18] to compute a concrete assignment for each symbolic value. We then use these values to generate the unrolled P4 code.

**P4 compiler.** After the compiler converts the P4All program into a P4 program, we invoke the (black box) Tofino compiler to compile the P4 program for execution on the underlying Tofino switch. If our experiments initially fail to compile to the Tofino switch because of proprietary constraints, we adjust our target specification and added `assume` statements to further constrain the memory allocated to register arrays. Ideally, the P4All compiler would be embedded within a target-specific compiler to automatically incorporate the proprietary constraints, without our needing to infer them.



(a) Number of ILP variables and constraints for CMS as stages increase.

Num Stages	ILP Time (s)
Tofino	1.8
1.25xTofino	4.5
1.67xTofino	53.1
2xTofino	216.0

(b) ILP completion time for CMS as stages increase.

**Figure 9:** ILP performance as number of available stages increases.

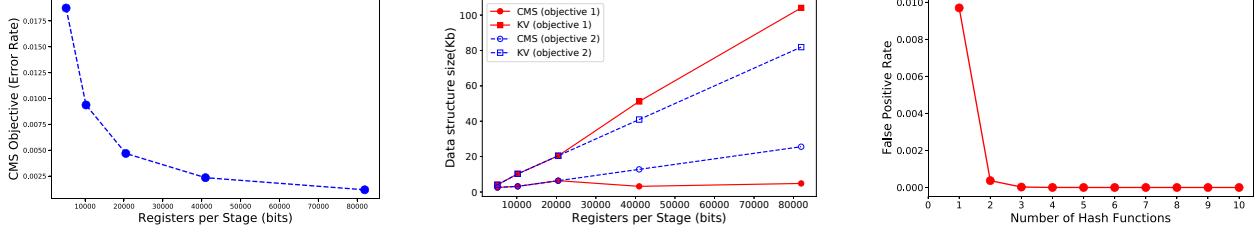
## 6 Performance Evaluation

### 6.1 Compiler Performance

Figure 8 reports the sizes of the constraint systems, and the compile times, for benchmark applications when compiled against our Tofino resource specification. We choose applications with a variety of features, including elastic TCAM tables (Switch.p4), multivariate objectives (Bloom filter), elastic and non-elastic components (IPv4 forwarding and stateful firewall), and multiple elastic components (KVS and CMS, CMS and Bloom filter). In our experiments, we found that the choice of objective function greatly impacts performance. For example, a non-convex objective function results in a mixed integer program (MIP) instead of an ILP, which significantly increases solving time. On the other hand, our applications with linear objective functions (e.g., Switch.p4, BeauCoup) typically had smaller compile times. Additionally, increasing the step size for an objective (i.e., reducing the number of values provided to the ILP) decreases compile time.

For the data structures we evaluated with objective functions with multiple variables (e.g., Bloom filter), our compiler created multiple instances of the optimization problem. We report the average compile time and the average number of ILP variables and constraints *for each instance*, along with the statistics for the largest instance. Our prototype compiler is not parallelized, but could easily be in the future, allowing us to solve many (possibly all) instances at the same time. Compile times of each ILP instance for the Bloom filter application range from roughly one second to 8.5 minutes.

Compile time increases as we increase the number of elastic elements in a P4All program. We evaluate ILP performance by observing the solving time as we increase the number of elastic elements in a program. Compilation for a single elastic sketch completed in about 10 seconds, while compilation for



(a) CMS error rate as memory increases.

(b) CMS and KVS sizes for different objectives. (c) Bloom filter false positive vs. # hash functions.

**Figure 10:** Elasticity of P4All

four sketches took over 30 minutes.

The number of constraints also affects compile time. The Bloom filter had the fewest ILP constraints, as it had no dependent components, and it alone had the largest compilation time. The reason for this is that the smaller number of constraints may lead to a more difficult optimization problem.

When we increase the available resources on the target, we generate a larger optimization problem, with more variables and constraints. Figure 9a shows the change in the number of constraints and variables as we increase the number of available stages on the target. Most of the resource and other constraints (*e.g.*, TCAM size, hash units, at most once, etc.) are linearly proportional to the stages. The dependency constraints are the only constraints that do not increase linearly with the stages. For a single P4All action, we create an ILP variable for each stage. However, the variables for CMS are not linearly proportional to the stages because as we increase stages, the upper bound on the actions also increases, resulting in more variables. Similarly, the ILP completion time increases super linearly with the number of stages (Figure 9b).

Some applications may have both elastic and non-elastic components. In our evaluations, we found that this did not significantly impact compile time. When we combined an elastic CMS and Switch.p4 (with fixed-size TCAM tables), the compile time was 17 seconds. Our compiler requires that all non-elastic portions of the program get placed on the switch, or the program will fail to compile.

**Hand-written vs P4All-generated P4** To investigate whether P4All-generated P4 was competitive with hand-written P4, we examined a few P4 programs written by hand by other programmers and compared those programs with the P4 code generated from P4All. When we compare the number of registers used by the manually-written BeauCoup and the P4All-generated BeauCoup, we find they are exactly the same. ConQuest is made up of sketches, so we use the same objective function described in §3. With that function, our compiler tries to allocate as many registers as possible, and allocates all available space to sketches, as more registers means lower error. Examining the ConQuest paper in more depth, however, shows that the accuracy gains are minimal

after a certain point (2048 columns). To account for this, we easily adjust the objective function, and as a result, the compiled code uses exactly 2048 columns as in the original. This experiment illustrates the power of P4All beautifully. On one hand, our first optimization function is highly effective—it uses up all available resources. On the other hand, when new information arrives, like the fact that empirically, there are diminishing returns beyond a certain point, we need only adjust the objective function to reflect our new understanding of the utility. None of the implementation details need change. While this analysis is admittedly ad hoc, our findings here suggest that P4All does not put programmers at a disadvantage when it comes to producing resource-efficient P4.

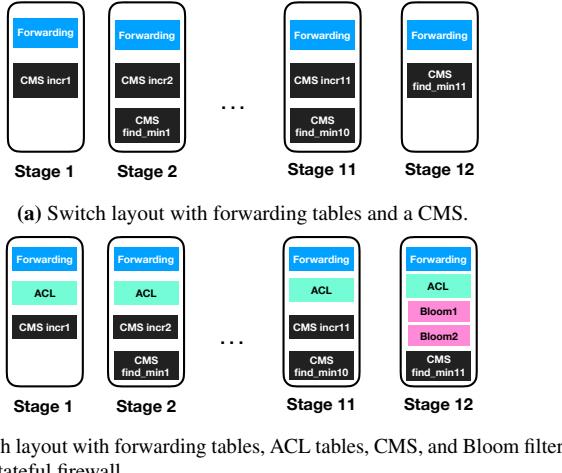
## 6.2 Elasticity

In this section we measure how utility of data structures vary as resources are made available. Figure 10a shows how the error rate of a CMS decreases as we increase the available registers in each stage. Figure 10b shows how the sizes of a KVS and CMS change for different objective functions. We use the objective functions for KVS hit rate and CMS error rate as described in Figure 5. The first objective function  $0.8 * (kv\_obj) - 0.2 * (cms\_obj)$  gives a higher weight to the KVS hit rate, while the second  $0.2 * (kv\_obj) - 0.8 * (cms\_obj)$  gives a higher weight to the CMS error rate.

For multi-variate functions, the compiler generates multiple instances of the optimization problem, and chooses the solution to the instance with the best objective. In Figure 10c, we show the objective (false positive rate) from the instances of optimization for a Bloom filter. In each instance, the compiler increases the number of hashes used. The objective decreases for each instance, but not by much after the first instance.

## 6.3 Case Study

In a conversation with a major cloud provider, the researchers expressed interest in hosting a multiple applications on the same network device, which must include forwarding logic. We designed P4All for exactly such scenarios—elastic structures allow new applications to fit onto a shared device. We consider a simple case study oriented around this problem.



**Figure 11:** Switch program layouts.

To do so, we started with the IPv4 forwarding code from `switch.p4`, but the size of the table is defined symbolically in P4All. We then added a CMS for heavy hitter detection. Figure 11a illustrates the layout: The forwarding tables utilize all of the TCAM resources, and the CMS uses registers.

Next, to demonstrate the flexibility and modularity of our framework, we add access control lists (ACLs), which use match-action tables, and squeeze in a stateful firewall, using Bloom filters, similar to the P4 tutorials [2]. Using P4, the programmer would manually resize the CMS and forwarding tables so the new applications could fit on the switch, but by using P4All, we do not have to change our existing code at all. To write ACLs with elastic TCAM tables, we modify the code in `switch.p4` to include symbolic table sizes. Our compiler automatically resizes the elastic structures to fit on the switch, resulting in the layout in Figure 11b. The forwarding tables and ACLs now share the match-action table resources, and the registers in the Bloom filter fit alongside the CMS.

## 7 Related Work

**Languages for network programming.** There has been a large body of work on programming languages for software defined networks [3, 14, 37, 44] targeted towards OpenFlow [33], a predecessor to P4 [7, 36]. OpenFlow only allows for a fixed set of actions and not control over registers in the data plane, and so these abstractions are not sufficient for P4. While P4 makes it possible to create applications over a variety of hardware targets, it does not make it easy. Domino [40] and Chipmunk [16] use a high-level C-like language to aid in programming switches. P4All also aims to simplify this process, but we enhance P4 with elastic data structures. Domino and Chipmunk optimize the data-plane layout for static, fixed-sized data structures, and P4All optimizes the data structure itself to make the most effective use of resources.

**Using synthesis for compiling to PISA.** The Domino compiler extracts “codelets”, groups of statements that must execute in the same stage. It then uses SKETCH [42] program synthesis to map a codelet to ALUs (atoms in the paper’s terminology) in each stage. If any codelet violates target constraints, the program is rejected. To improve Domino, Chipmunk [16] uses syntax-guided synthesis to perform an exhaustive search of all mappings of the program to the target. Thus, it can find mappings that are sometimes missed by Domino. Lyra [15], extends this notion to a one-big-pipeline abstraction, allowing the composition of multiple algorithms to be placed across several heterogeneous ASICs. Nevertheless, Domino, Chipmunk and Lyra map programs with fixed-size data structures, while P4All enables elastic data structures.

**Compiling to RMT.** Jose et al. [28] use ILPs and greedy algorithms to compile programs for RMT [8] and FlexPipe [35] architectures. These ILPs are part of an all-or-nothing compiler which attempts to place actions on a switch based on the dependencies and the sizes of match-action tables. In contrast, the P4All compiler allows for elastic structures, which can stretch or compress according to a target’s available resources.

**Programmable Optimization.** P<sup>2</sup>GO [45] uses profile-guided optimization (*i.e.*, a sample traffic trace, not a static objective function) to reduce the resources required in a P4 program. P<sup>2</sup>GO can effectively prune components that are not used in a given environment; however, if unexpected traffic turns up later, P<sup>2</sup>GO may have pruned needed functionality!

## 8 Conclusion

In this paper, we introduce the concept of *elastic data structures* that can expand to use the resources on a hardware target. Elastic switch programs are more modular than their inelastic counterparts, as elastic pieces can adjust depending on the resource needs of other components on the switch. They also are portable, as they can be recompiled for different targets.

P4All is a backwards-compatible extension of P4 that includes symbolic values, arrays, loops and objective functions. We have developed P4All code for a number of reusable modules and several applications from the recent literature. We also implement and evaluate a compiler for P4All, demonstrating that compile times are reasonable and that auto-generated programs make efficient use of switch resources. We believe that P4All and our reusable modules will make it easier to implement and deploy a range of future data-plane applications.

## Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Costin Raiciu for their valuable feedback. This work is supported by DARPA under Dispersed Computing HR0011-17-C-0047, NSF under FMiTF-1837030 and CNS-1703493 and the Israel Science Foundation under grant No. 980/21.

## References

- [1] Lark parser. <https://github.com/lark-parser/lark>.
- [2] Stateful firewall in P4. <https://github.com/p4lang/tutorials/tree/master/exercises/firewall>.
- [3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–126. ACM, 2014.
- [4] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *IFIP Networking*, pages 449–457, 2020.
- [5] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, pages 662–680, 2020.
- [6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE International Conference on Network Protocols*, pages 313–323, Sep. 2018.
- [7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.
- [8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, pages 99–110, 2013.
- [9] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, pages 126–134, 1999.
- [10] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, pages 226–239, 2020.
- [11] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking EXperiments and Technologies*, pages 15–29. ACM, 2019.
- [12] Saar Cohen and Yossi Matias. Spectral bloom filters. In *ACM SIGMOD*, pages 241–252. ACM, 2003.
- [13] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In Hillo Kargupta, Jaideep Srivastava, Chandrika Kamath, and Arnold Goodman, editors, *SIAM International Conference on Data Mining*, pages 44–55. SIAM, 2005.
- [14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming*, pages 279–291. ACM, 2011.
- [15] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM*, pages 435–450, 2020.
- [16] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Praveen Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, page 44–61, 2020.
- [17] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, pages 357–371. ACM, 2018.
- [18] Gurobi Optimization. Gurobi optimizer reference manual. <http://www.gurobi.com>, 2019.
- [19] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM Symposium on SDN Research*, 2018.
- [20] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephans: Seize the global heavy hitters. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, pages 15–21, 2020.
- [21] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *ACM Workshop on Hot Topics in Networks*, page 168–174, 2020.

- [22] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 161–176, Boston, MA, February 2019.
- [23] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.
- [24] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketch-learn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.
- [25] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *ACM Symposium on SDN Research*, 2018.
- [26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 35–49, Renton, WA, April 2018.
- [27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Symposium on Operating System Principles*, 2017.
- [28] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *USENIX Conference on Networked Systems Design and Implementation*, pages 103–115, 2015.
- [29] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327. Springer Berlin Heidelberg, 2010.
- [30] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 311–324, Santa Clara, CA, March 2016.
- [31] Zaoxing Liu, Ran Ben Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.
- [32] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114, 2016.
- [33] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [34] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*, pages 15–28, 2017.
- [35] Recep Ozdag. Intel® Ethernet Switch FM6000 Series—Software Defined Networking, 2012. [goo.gl/Anv0vX](http://goo.gl/Anv0vX).
- [36] P4 Language Consortium. P4<sub>16</sub> language specifications, 2018. <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf>.
- [37] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent NetCore: From policies to pipelines. In *ACM SIGPLAN International Conference on Functional programming*, pages 11–24, 2014.
- [38] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Networked Systems Design and Implementation*, pages 67–82, March 2017.
- [39] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–16, Renton, WA, April 2018.
- [40] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, pages 15–28, 2016.
- [41] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rotenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research*, pages 164–176, 2017.
- [42] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for*

- Programming Languages and Operating Systems*, pages 404–415, 2006.
- [43] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 530–541, 2014.
  - [44] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM*, volume 43, pages 87–98, August 2013.
  - [45] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *ACM Workshop on Hot Topics in Networks*, page 146–152, 2020.
  - [46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM*, pages 561–575, 2018.
  - [47] Zhiulong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*, pages 126–138, 2020.
  - [48] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, pages 76–89, 2020.



# Privid: Practical, Privacy-Preserving Video Analytics Queries

Frank Cangialosi , Neil Agarwal , Venkat Arun , Junchen Jiang , Srinivas Narayana , Anand Sarwate , Ravi Netravali   
 MIT CSAIL  Princeton University  University of Chicago  Rutgers University  
privid@csail.mit.edu

## Abstract

Analytics on video recorded by cameras in public areas have the potential to fuel many exciting applications, but also pose the risk of intruding on individuals’ privacy. Unfortunately, existing solutions fail to practically resolve this tension between utility and privacy, relying on perfect detection of all private information in each video frame—an elusive requirement. This paper presents: (1) a new notion of differential privacy (DP) for video analytics,  $(\rho, K, \epsilon)$ -event-duration privacy, which protects all private information visible for less than a particular duration, rather than relying on perfect detections of that information, and (2) a practical system called PRIVID that enforces duration-based privacy even with the (untrusted) analyst-provided deep neural networks that are commonplace for video analytics today. Across a variety of videos and queries, we show that PRIVID increases error by 1-21% relative to a non-private system.

## 1 Introduction

High-resolution video cameras are now pervasive in public settings [1, 3–5, 10], with deployments throughout city streets, in our doctor’s offices and schools, and in the places we shop, eat, or work. Traditionally, these cameras were monitored manually, if at all, and used for security purposes, such as providing evidence for a crime or locating a missing person. However, steady advances in computer vision [32, 51, 53, 55, 65] have made it possible to automate video-content analytics (both live and retrospective) at a massive scale across entire networks of cameras. While these trends enable a variety of important applications [2, 11, 13, 14] and fuel much work in the systems community [26, 30, 40, 43, 44, 47, 48, 54, 73], they also enable privacy intrusions at an unprecedented level [7, 64].

As a concrete example, consider the operator for a network of city-owned cameras. Different organizations (i.e., “analysts”) want access to the camera feeds for a range of needs: (1) health officials want to measure the fraction of people wearing masks and following COVID-19 social distancing orders [38], (2) the transportation department wants to monitor the density and flow of vehicles, bikes, and pedestrians to determine where to add sidewalks and bike lanes [21], and (3) businesses are willing to pay the city to understand shopping behaviors for better planning of promotions [19].

Unfortunately, freely sharing the video with these parties may enable them to violate the privacy of individuals in the scene by tracking where they are, and when. For example, the “local business” may actually be a bank or insurance company that wants to track individuals’ private lives for their risk models, while well-known companies [17] or government agencies may succumb to mission creep [18, 20]. Further, any organiza-

tions with good intentions could have employees with malicious intent who wish to spy on a friend or co-worker [15, 16].

There is an *inherent tension between utility and privacy*. In this paper, we ask: is it possible to enable these (untrusted) organizations to use the collected video for analytics, while also guaranteeing citizens that their privacy will be protected? Currently, the answer is no. As a consequence, many cities have outright banned analytics on public videos, even for law enforcement purposes [9, 12].

While a wide variety of solutions have been proposed (§3), ranging from computer vision (CV)-based obfuscation [23, 60, 68, 70] (e.g., blurring faces) to differential privacy (DP)-based methods [66, 67], they all use some variant of the same strategy: find *all* private information in the video, then hide it. Unfortunately, the first step alone can be unrealistic in practice (§3.1); it requires: (1) an explicit specification of all private information that could be used to identify an individual (e.g., their backpack), and then (2) the ability to spatially *locate* all of that information in *every* frame of the video—a near impossible task even with state-of-the-art CV algorithms [6]. Further, if these approaches cannot find some private information, they fundamentally cannot *know* that they missed it. Taken together, they can provide, at best, a conditional and brittle privacy guarantee such as the following: if an individual is only identifiable by their face, and their face is detectable in every frame of the video by the implementation’s specific CV model in the specific conditions of this video, then their privacy will be protected.

This paper takes a pragmatic stance and aims to provide a definitively achievable privacy guarantee that captures the aspiration of prior approaches (i.e., individuals cannot be identified in any frame or tracked across frames) despite the limitations that plague them. To do this, we leverage two key observations: (1) a large body of video analytics queries are aggregations [47, 49], and (2) they typically aggregate over durations of video (e.g., hours or days) that far exceed the duration of any one individual in the scene (e.g., seconds or minutes) [47]. Building on these observations, we make three contributions by jointly designing a new notion of duration-based privacy for video analytics, a system implementation to realize it, and a series of optimizations to improve utility.

**Duration-based differential privacy.** To remove the dependence on spatially locating all private information in each video frame, we reframe the approach to privacy to instead focus on the temporal aspect of private information in video data, i.e., *how long* something is visible to a camera. More specifically, building on the differential privacy (DP) framework [37], we propose a new notion of privacy for

video,  $(\rho, K, \epsilon)$ -event-duration privacy (formalized in §4.1): anything visible to a camera less than  $K$  times for less than  $\rho$  seconds each time (“ $(\rho, K)$ -bounded”) is protected with  $\epsilon$ -DP. The video owner expresses their privacy policy using  $(\rho, K)$ , which we argue is powerful enough to capture many practical privacy goals. For example, if they choose  $\rho = 5\text{min}$ , anyone visible for less than 5 minutes is protected with  $\epsilon$ -DP, which in turn prevents tracking them. We discuss other policies in §4.2.

This notion of privacy has three benefits. First, it decouples the definition of privacy from its enforcement. The enforcement mechanism does not need to make any decisions about what is private or find private information to protect it; everything (private or not) captured by the bound is protected. Second, a  $(\rho, K)$  bound that captures a set of individuals implicitly captures and thus protects any information visible for the same (or less) time without specifying it (e.g., an individual’s backpack, or even their gait). Third, protecting all individuals in a video scene requires only their maximum duration, and estimating this value is far more robust to the imperfections of CV algorithms than precisely locating those individuals and their associated objects in each frame. For example, even if a CV algorithm misses individuals in some frames (or entirely), it can still capture a representative sample and piece together trajectories well enough to estimate their duration (§4.2).

**Privid: a differentially-private video analytics system.** Realizing  $(\rho, K, \epsilon)$ -privacy (or more generally, any DP mechanism) in today’s video analytics pipelines faces several challenges. In traditional database settings, implementing DP requires adding random noise proportional to the *sensitivity* of a query, i.e., the maximum amount that any one piece of private information could impact the query output. However, bounding the sensitivity is difficult in video analytics pipelines because (1) pipelines typically operate as bring-your-own-query-implementation to support the wide-ranging applications described earlier [22, 25, 26, 28, 29, 39, 41], and (2) these implementations involve video processing algorithms that increasingly rely on deep neural networks (DNNs), which are notoriously hard to inspect or vet (and thus, trust).

To bound the sensitivity necessary for  $(\rho, K, \epsilon)$ -privacy while supporting “black-box” analyst-provided query implementations (including DNNs), PRIVID only accepts analyst queries structured in the following *split-process-aggregate* format (§5.2): (i) videos are split into temporally-contiguous chunks, (ii) each chunk of video is processed by an arbitrary analyst-provided processing program to produce an (untrusted) table, (iii) values in the table are aggregated (e.g. averaged) to compute a result, and (iv) noise is added to the result before release. The key in this pipeline is step (ii): we treat the analyst-provided program as an arbitrary Turing machine with restricted inputs (a single chunk of video frames and some metadata) and restricted outputs (rows of a table). As a result, only one chunk can contribute to the value of each row, and we know which chunk generated each row. If an individual is  $(\rho, K)$ -bounded, the number of chunks they appear

in is bounded, and thus the number of rows their presence can affect is bounded as well. With a bound on the number of rows, we can apply classic differential privacy techniques (§5.5).

**Optimizations for improved utility.** To further enhance utility, PRIVID provides two video-specific optimizations to lower the required noise while preserving an equivalent level of privacy: (i) the ability to mask regions of the video frame, (ii) the ability to split frames spatially into different regions, and aggregate results from these regions. These optimizations result in limiting the portion of the aggregate result that any individual’s presence can impact, enabling a “tighter”  $(\rho, K)$  bound and in turn a higher quality query result.

**Evaluation.** We evaluate PRIVID using a variety of public videos and a diverse range of queries inspired by recent work in this space. PRIVID increases error by 1-21% relative to a non-private system, while satisfying an instantiation of  $(\rho, K, \epsilon)$ -privacy that protects all individuals in the video. We discuss ethics in §9. Source code and datasets for PRIVID are available at <https://github.com/fcangialosi/privid>.

## 2 Problem Statement

### 2.1 Video Analytics Background

Video analytics pipelines are employed to answer high-level questions about segments of video captured from one or more cameras and across a variety of time ranges. Example questions include “how many people entered store X each hour?” or “which roads suffered from the most accidents in 2020?” (see §7.2 and Table 3 for more specific examples). A question is expressed as a *query*, which encompasses all of the computation necessary to answer that question.<sup>1</sup> For example, to answer the question “what is the average speed of red cars traveling along road Y?”, the “query” would include an object detection algorithm to recognize cars, an object tracking algorithm to group them into trajectories, an algorithm for computing speed from a trajectory, and logic to filter only the red cars and average their speeds.

### 2.2 Problem Definition

Video analytics pipelines broadly involve four logical roles (though any combination may pertain to the same entity):

- **Individuals**, whose behavior and activity are observed by the camera.
- **Video Owner (VO)**, who operates the camera and thus owns the video data it captures.
- **Analyst**, who wishes to run queries over the video.
- **Compute Provider**, who executes the analyst’s query.

In this work, we are concerned with the dilemma of a VO. The VO would like to enable a variety of (untrusted) analysts to answer questions about its videos (such as those in §2.1), as long as the results do not infringe on the privacy of the individuals who appear in the videos. Informally, privacy

<sup>1</sup>Our definition is distinct from related work, which defines a query as returning intermediate results (e.g., bounding boxes) rather than the final answer to the high-level question.

“leakage” occurs when an analyst can learn something about a specific individual that they did not know before executing a query. To practically achieve these properties, a system must meet three concrete goals:

1. **Formal notion of privacy.** The system’s privacy policies should formally describe the type and amount of privacy that could be lost through a query. Given a privacy policy, the system should be able to provide a *guarantee* that it will be enforced, regardless of properties of the data or query implementation.
2. **Maximize utility for analysts.** The system should support queries whose final *result* does not infringe on the privacy of any individuals. Further, if accuracy loss is introduced to achieve privacy for a given query, it should be possible to bound that loss (relative to running the same query over the original video, without any privacy preserving mechanisms). Without such a bound, analysts would be unable to rely on any provided results.
3. **“Bring Your Own Model”.** Computer vision models are at the heart of modern video processing. However, there is not one or even a discrete set of models for all tasks and videos. Even the same task may require different models, parameters, or post-processing steps when applied to different videos. In many cases, analysts will want to use models that they trained themselves, especially when training involves proprietary data. Thus, a system must allow analysts to provide their own video-processing models.

It is important to note that the class of analytics queries we seek to enable are distinct from *security-oriented* queries (e.g., finding a stolen car or missing child), which *require* identification of a particular individual, and are thus directly at odds with individual privacy. In contrast, analytics queries involve searching for patterns and trends in large amounts of data; intermediate steps may operate over the data of specific individuals, but they do not distinguish individuals in their final aggregated result (§2.1).

### 2.3 Threat Model

The VO employs a privacy-preserving system to handle queries about a set of cameras it manages; the system retains full control over the video data, analysts can only interact with it via the query interface. The VO does not trust the analysts (or their query implementation code). Any number of analysts may be malicious and may collude to violate the privacy of the same individual. However, analysts trust the VO to be honest. Analysts are also willing to share their query implementation (so that the VO can execute it). The VO views this code as an untrusted blackbox which it cannot yet.

Analysts pose queries adaptively (i.e., the full set of queries is not known ahead of time, and analysts may utilize the results of prior queries when posing a new one). A single query may operate over video from multiple cameras. We assume the VO has sufficient computing resources to execute

the query, either via resources that they own, or through the secure use of third-party resources [62].

The system releases some per-camera metadata publicly (§8.1), including a sample video clip. The resulting leak is interpretable and can be minimized by the VO. The system protects all other information with a formal guarantee of  $(\rho, K, \epsilon)$ -privacy (Def 4.3).

## 3 Limitations of Related Work

Before presenting our solution, we consider prior privacy-preserving mechanisms (both for video and in general). Unfortunately, each fails to satisfy at least one of the goals in §2.2.

### 3.1 Denaturing

The predominant approach to privacy preservation with video data is *denaturing* [23, 34, 60, 68, 70, 72], whereby systems aim to obscure (e.g., via blurring [23] or blocking [68] as in Fig. 1) any private information in the video before releasing it for analysis. In principle, if nothing private is left in the video, then privacy concerns are eliminated.

The fundamental issue is that denaturing approaches require *perfectly* accurate and comprehensive knowledge of the spatial locations of private information in *every frame* of a video. Any private object that goes undetected, even in just a single frame, will not be obscured and thus directly leads to a leakage of private information.

To detect private information, one must first semantically define *what* is private, i.e., what is the full set of information linked, directly or indirectly, to the privacy of each individual? While some information is obviously linked (e.g., an individual’s face), it is difficult to determine *all* such information for all individuals in all scenarios. For instance, a malicious analyst may have prior information that a VO does not, such as knowledge that a particular individual carries a specific bag or rides a unique bike (e.g., Fig. 1-B). Further, even with a semantic definition, detecting private information is difficult. State-of-the-art computer vision algorithms commonly miss objects or produce erroneous classification labels in favorable video conditions [74]; performance steeply degrades in more challenging conditions such as poor lighting, distant objects, and low resolution, all of which are common in public video. Taken together, the problem is that denaturing systems cannot guarantee whether or not a private object was left in the video, and thus fail to provide a formal notion of privacy (violating Goal 1).

Denaturing also falls short from the analyst’s perspective. First, it inherently precludes (safe) queries that aggregate over private information (violating Goal 2). For example, an urban planner may wish to count the number of people that walk in front of camera A and then camera B. Doing so requires identifying and cross-referencing individuals between the cameras (which is not possible if they have been denatured), but the ag-



**Figure 1:** A video clip after (silhouette) denaturing exemplifying some of its shortcomings: (A) entirely missed detections, (B) potentially-identifying objects not incorporated in privacy definition, (C) silhouette may reveal gait.

gregate count may be large and safe to release.<sup>2</sup> Second, obfuscated objects are not naturally occurring and thus video processing pipelines are not designed to handle them. If the analyst’s processing code and models have not been trained explicitly on the type of obfuscation the VO is employing, it may behave in unpredictable and unbounded ways (violating Goal 2).

### 3.2 Differential Privacy

Differential Privacy (DP) is a strong formal definition of privacy for traditional databases [37]. It enables analysts to compute aggregate statistics over a database, while protecting the presence of any individual entry in the database. DP is not a privacy-preserving mechanism itself, but rather a goal that an algorithm can aim to satisfy. Informally speaking, an algorithm satisfies DP if adding or removing an individual from the input database does not noticeably change the output of computation, almost as if any given individual were not present in the first place. More precisely,

**DEFINITION 3.1.** Two databases  $D$  and  $D'$  are *neighboring* if they differ in the data of only a single user (typically, a single row in a table).

**DEFINITION 3.2.** A randomized algorithm  $\mathcal{A}$  is  $\epsilon$ -differentially private if, for all pairs of neighboring databases  $(D, D')$  and all  $S \subseteq \text{Range}(\mathcal{A})$ :

$$\Pr[\mathcal{A}(D) \in S] \leq e^\epsilon \Pr[\mathcal{A}(D') \in S] \quad (3.1)$$

A non-private computation (e.g., computing a sum of bank balances) is typically made differentially private by adding random noise sampled from a Laplace distribution to the final result of the computation [37]. The scale of noise is set proportional to the sensitivity ( $\Delta$ ) of the computation, or the maximum amount by which the computation’s output could change due to the presence/absence of any one individual. For instance, suppose a database contains a value  $v_i \in V$  for each user  $i$ , where  $l \leq v_i \leq u$ . If a query seeks to sum all values in  $V$ , any one individual’s  $v_i$  can influence that sum by at most  $\Delta = u - l$ , and thus adding noise with scale  $u - l$  would satisfy DP.

**Challenges.** Determining the sensitivity of a computation is the key ingredient of satisfying DP. It requires understanding

<sup>2</sup>As a workaround, the VO could annotate denatured objects with query-specific information, but this would conflict with Goal 3.

(a) how individuals are delineated in the data, and (b) how the aggregation incorporates information about each individual. In the tabular data structures that DP was designed for, these are straightforward. Each row (or a set of rows sharing a unique key) typically represents one individual, and queries are expressed in relational algebra, which describes exactly how it aggregates over these rows. However, these answers do not translate to video data; we next discuss the challenges in the context of several applications of DP to video analytics.

Regarding *requirement (a)*, as described in §3.1, it is difficult and error-prone to determine the full set of pixels in a video that correspond to each user (including all potentially identifying objects). Accordingly, prior attempts of applying DP concepts to video analytics [66, 67] that rely on perfectly defined and detected private information (via CV) fall short in the same way as denaturing approaches (violating Goal 1).

Regarding *requirement (b)*, typical video processing algorithms (e.g., ML-based CV models) are not transparent about how they incorporate private objects into their results. Thus, without a specific query interface, the “tightest” possible bound on the sensitivity of an arbitrary computation over a video is simply the entire range of the output space. In this case, satisfying DP would add noise greater than or equal to any possible output, precluding any utility (violating Goal 2).

Given that DP is well understood for tables, a natural idea would be for the VO to use their own (trusted) model to first convert the video into a table (e.g., of objects in the video), then provide a DP interface over *that table*<sup>3</sup> (instead of directly over the video itself). However, in order to provide a guarantee of privacy, the VO would need to completely trust the model that creates the table. This entirely precludes using a model created by the *untrusted* analyst (violating Goal 3).

## 4 Event Duration Privacy

We will first formalize  $(\rho, K, \epsilon)$ -privacy, then provide the intuition for what it protects and clarify its limitations.

### 4.1 Definition

We consider a video  $V$  to be an arbitrarily long sequence of frames, sampled at  $f$  frames per second, recorded directly from a camera (i.e., unedited). A “segment”  $v \subset V$  of video is a contiguous subsequence of those frames. The “duration” of a segment  $d(v)$  is measured in real time (seconds), as opposed to frames. An “event”  $e$  is abstractly *anything* that is visible within the camera’s field of view.

As a running example, consider a video segment  $v$  in which individual  $x$  is visible for 30 seconds before they enter a building, and then another 10 seconds when they leave some time later. The “event” of  $x$ ’s visit is comprised of one 30-second segment, and another 10-second segment.

<sup>3</sup>This would be equivalent to adding DP to an existing video analytics interface, such as [30, 47], which treat the video as a table of objects.

**DEFINITION 4.1**  $((\rho, K)$ -bounded events). An event  $e$  is  $(\rho, K)$ -bounded if there exists a set of  $\leq K$  video segments that completely contain<sup>4</sup> the event, and each of these segments individually have duration  $\leq \rho$ .

(Ex). The tightest bound on  $x$ 's visit is  $(\rho = 30s, K = 2)$ . To be explicit,  $x$ 's visit is also  $(\rho, K)$ -bounded for any  $\rho \geq 30s$  and  $K \geq 2$ .

**DEFINITION 4.2**  $((\rho, K)$ -neighboring videos). Two video segments  $v, v'$  are  $(\rho, K)$ -neighboring if the set of frames in which they differ is  $(\rho, K)$ -bounded.

(Ex). One potential  $v'$  is a hypothetical video in which  $x$  was never present (but everything else observed in  $v$  remained the same). Note this is purely to denote the strength of the guarantee in the following definition, the VO does not actually construct such a  $v'$ .

**DEFINITION 4.3**  $((\rho, K, \epsilon)$ -event-duration privacy). A randomized mechanism  $\mathcal{M}$  satisfies  $(\rho, K, \epsilon)$ -event-duration privacy<sup>5</sup> iff for all possible pairs of  $(\rho, K)$ -neighboring videos  $v, v'$ , any finite set of queries  $Q = \{q_1, q_2, \dots\}$  and all  $S_q \subseteq Range(\mathcal{M}(\cdot, q))$ :

$$Pr[(\mathcal{M}(v, q_1), \dots, \mathcal{M}(v, q_n)) \in S_{q_1} \times \dots \times S_{q_n}] \leq e^\epsilon Pr[(\mathcal{M}(v', q_1), \dots, \mathcal{M}(v', q_n)) \in S_{q_1} \times \dots \times S_{q_n}]$$

**Guarantee.**  $(\rho, K, \epsilon)$ -privacy protects all  $(\rho, K)$ -bounded events (such as  $x$ 's visit to the building) with  $\epsilon$ -DP: informally, if an event is  $(\rho, K)$ -bounded, an adversary cannot increase their knowledge of whether or not the event happened by observing a query result from  $\mathcal{M}$ . To be clear,  $(\rho, K, \epsilon)$ -privacy is *not* a departure from DP, but rather an extension to explicitly specify what to protect in the context of video.

## 4.2 Choosing a Privacy Policy

The VO is responsible for choosing the parameter values  $(\rho, K)$  (“policy”) that bound the class of events they wish to protect. They may use domain knowledge, employ CV algorithms to analyze durations in past video from the camera, or a mix of both. Regardless, they express their goal to PRIVID solely through their choice of  $(\rho, K)$ .

**Automatic setting of  $(\rho, K)$ .** The primary reason  $(\rho, K, \epsilon)$ -privacy is *practical* is that, despite their imperfections, today's CV algorithms are capable of producing good estimates of the maximum duration any individuals are visible in a scene. We provide some evidence of this intuition over three representative videos from our evaluation. For each video, we chose a 10-minute segment and manually annotate the duration of each individual (person or vehicle), i.e., “Ground Truth”, then use

<sup>4</sup>A set of segments is said to completely contain an event if the event is not visible in any frames outside of those segments.

<sup>5</sup>We chose to use  $\epsilon$ -DP rather than the more general  $(\epsilon, \delta)$ -DP for simplicity, since the difference is not significant to our definition. Our definition could be extended to  $(\epsilon, \delta)$ -DP without additional insights.



**Figure 2:** The results of a state-of-the-art object detection algorithm (filtered to “person” class) on one frame of urban. The algorithm misses 76% of individuals in the frame, but is *still* able to produce a conservative bound on the maximum duration of all individuals (Table 1).

Video	Maximum Duration		% Objects CV Missed
	Ground Truth	CV Estimate	
campus	81 sec	83 sec	29%
highway*	316 sec	439 sec	5%
urban	270 sec	354 sec	76%

**Table 1:** Despite the imperfection of current CV algorithms (exemplified by % objects they failed to detect), they still produce a conservative estimate on the duration of any individual’s presence. \*For the purposes of this experiment, we ignored cars that were parked for the entire duration of the segment.

state-of-the-art object detection and tracking to estimate the durations and report the maximum (“CV”). Our results, summarized in Table 1, show that, while object detection misses a non-trivial fraction of bounding boxes, the tracking algorithm is able to fill in the gaps for enough trajectories to capture a conservative estimate of the maximum duration. In other words, for our three videos, using these algorithms to parameterize a  $(\rho, K, \epsilon)$ -private system would successfully capture the duration of, and thus protect the privacy of, *all* individuals, while using them to implement any prior approach would not.

**Relaxing the set of private individuals.** Sometimes protecting *all* individuals is unnecessary. Consider a camera in a store; employees will appear significantly longer and more frequently than customers (e.g., 8 hours every day vs. 30 minutes once a week), but if the fact that the employees work there is public knowledge, the VO can pick a policy (with smaller  $\rho$  and  $K$ ) that only bounds the appearance of customers.

**Generic policies.** Alternatively, the VO can choose a policy to place a generic limit on the (temporal) granularity of queries. Consider a policy  $(\rho = 5\text{min}, K = 1)$ . Suppose individual  $x$  stops and talks to a few people on their way to work each morning, but each conversation lasts less than 5 minutes. Although the policy does not protect  $x$ 's presence or even the fact that they often stop to chat on their way to work, it *does* protect the timing and contents of each conversation.

## 4.3 Privacy Guarantees in Practice

In PRIVID’s implementation of  $(\rho, K, \epsilon)$ -privacy (described in the following section), the policy provides a relative reference point: events that exactly match the policy (i.e., made up of *exactly*  $K$  segments each of duration  $\rho$ ) are protected

with  $\epsilon$ -DP, while events that are visible for shorter or longer durations are protected with a proportionally (w.r.t. the duration) stronger or weaker guarantee, respectively.

**Theorem 4.1.** Consider a camera with a fixed policy  $(\rho, K, \epsilon)$ . If an individual  $x$ 's appearance in front of the camera is bound by some  $(\hat{\rho}, \hat{K})$ , then PRIVID effectively protects  $x$  with  $\hat{\epsilon}$ -DP, where  $\hat{\epsilon}$  is  $O(\frac{\hat{\rho}\hat{K}}{\rho K})\epsilon$ , which grows (degrades) as  $(\hat{\rho}, \hat{K})$  increase while  $(\rho, K, \epsilon)$  are fixed, and the constants do not depend on the query. We provide a formal proof in §A.1.

For example, given  $(\rho = 1\text{hr}, K = 1)$ , PRIVID would protect an a single 2-hour appearance with  $\sim 2\epsilon$ -DP (weaker) or a single half-hour appearance with  $\sim \frac{1}{2}\epsilon$ -DP (stronger).

**Graceful degradation.** An important corollary of this theorem is that privacy degrades “gracefully”. As an event's  $\hat{\rho}$  increases further from  $\rho$  (or  $\hat{K}$  from  $K$ ), its effective  $\hat{\epsilon}$  increases linearly, yielding a progressively weaker guarantee. (The reverse is true, as  $\hat{\rho}$  and  $\hat{K}$  decrease, it yields a stronger guarantee). Thus, if  $\hat{\rho}$  (or  $\hat{K}$ ) is only *marginally* greater than  $\rho$  (or  $K$ ), then the event is not immediately revealed in the clear, but rather is protected with  $\hat{\epsilon}$ -DP, which is still a DP guarantee, only marginally weaker: a malicious analyst has only a marginally higher probability of detecting  $x$  in the worst case. This in effect *relaxes* the requirement that  $(\rho, K)$  be set strictly to the maximum duration an individual could appear in the video to achieve useful levels of privacy. We generalize and provide a visualization of this degradation in §A.2.

**Repeated appearances.** The larger the time window of video a query analyzes, the more instances an individual may appear within the window, even if each appearance is itself bounded by  $\rho$ . Consider our example individual  $x$  and policy  $(\rho = 30\text{s}, K = 2)$  from §4.1. In the query window of a single day  $d$ ,  $x$  appears twice; they are properly  $(\rho, K)$ -bounded and thus the event “ $x$  appeared on day  $d$ ” is protected with  $\epsilon$ -DP. Now, consider a query window of one week;  $x$  appears 14 times (2 times per day), so the event “ $x$  appeared sometime this week” is  $(\rho, 7K)$ -bounded and thus protected with (weaker)  $7\epsilon$ -DP. However, the more specific event “ $x$  appeared on day  $d'$ ” (for any  $d'$  in the week) is *still*  $(\rho, K)$ -bounded, and thus still protected with the same  $\epsilon$ -DP. In other words, while an analyst may learn that an individual appeared *sometime* in a given week, they cannot learn on which day they appeared. Thus, in order to get greater certainty, the analyst must give up temporal granularity.

**Multiple cameras.** When an individual appears in front of multiple cameras, their privacy guarantees are analogous to the previous case of repeated appearances in a single camera. If they appear in front of  $N$  different cameras, where the event of their appearance in camera  $i$  is protected with  $\hat{\epsilon}_i$ -DP, then the event of their appearance across all the cameras is protected with  $\sum_i \hat{\epsilon}_i$ -DP. Suppose for 10 cameras,  $\sum_{i=1}^N \hat{\epsilon}_i$  is large enough for the adversary to detect their appearance with high

confidence. Then while the adversary can infer that a person appeared *somewhere* across the 10 cameras, the adversary cannot learn *which* cameras they appeared in or when; appearances within individual cameras are still protected by  $\epsilon$ -DP.

## 5 PRIVID

In this section, we present PRIVID, a privacy-preserving video analytics system that satisfies  $(\rho, K, \epsilon)$ -privacy (§2.2 Goals 1 and 2) and provides an expressive query interface which allows analysts to supply their own (untrusted by PRIVID) video-processing code (Goal 3).

### 5.1 Overview

PRIVID supports *aggregation* queries, which process a “large” amount of video data (e.g., several hours/days of video) and produce a “small” number of bits of output (e.g., a few 32-bit integers). Examples of such tasks include counting the total number of individuals that passed by a camera in one day, or computing the average speed of cars observed. In contrast, PRIVID does not support a query such as reporting the location (e.g., bounding box) of an individual or car within the video frame. PRIVID can be used for one-off ad-hoc queries or standing queries running over a long period, e.g., the total number of cars per day, each day over a year.

The VO decides the level of privacy provided by PRIVID. The VO chooses a privacy policy  $(\rho, K)$  and privacy budget ( $\epsilon$ ) for each camera they manage. Given these parameters, PRIVID provides a guarantee of  $(\rho, K, \epsilon)$ -privacy (Theorem 5.2) for all queries over all cameras it manages.

To satisfy the privacy guarantee, PRIVID utilizes the standard Laplace mechanism from DP [37] to add random noise to the aggregate query result before returning the result to the analyst. The key technical pieces of PRIVID are: (i) providing analysts the ability to specify queries using arbitrary untrusted code (§5.2), (ii) adding noise to results to guarantee  $(\rho, K, \epsilon)$ -privacy for a single query (§5.5), and (iii) extending the guarantee to handle multiple queries over the same cameras (§5.6).

### 5.2 PRIVID Query Interface

**Execution model.** PRIVID requires queries to be expressed using a *split-process-aggregate* model in order to tie the duration of an event to the amount it can impact the query output. The target video is split temporally into chunks, then each chunk is fed to a separate instance of the analyst's processing code, which outputs a set of rows. Together, these rows form a traditional tabular database (untrusted by PRIVID since it is generated by the analyst). The aggregation stage runs a SQL query over this table to produce a raw result. Finally, PRIVID adds noise (§5.5) and returns *only* the noisy result to the analyst, not the raw result or the intermediate table.

**Query contents.** A PRIVID query must contain (1) a block of statements in a SQL-like language, which we introduce below and call PRIVIDQL, and (2) video processing executables.

- (1) PRIVIDQL statements.** A valid query contains one or more of *each* of the 3 following statements. We provide an example in §5.7.1 and the full grammar in §E of [33].
- SPLIT statements choose a segment of video (camera, start and end datetime) as input, and produce a set of video chunks as output. They specify how the segment should be split into chunks, i.e., the chunk duration and stride between chunks.
  - PROCESS statements take a set of SPLIT chunks as input, and produce a traditional (“intermediate”) table. They specify the executable that should process the chunks, the schema of the resulting table, and the maximum number of rows a chunk can output (`max_rows`, necessary to bound the sensitivity, §5.5). Any rows output beyond the max are dropped.
  - SELECT statements resemble typical SQL SELECT statements that operate over the tables resulting from PROCESS statements and output a  $(\rho, K, \epsilon)$ -private result. They must have an aggregation as the final operation. PRIVID supports the standard aggregation functions (e.g., COUNT, SUM, AVG) and the core set of typical operators as internal relations. An aggregation must specify the range of each column it aggregates (just as in related work on DP for SQL [50]). Each SELECT constitutes at least one data release: one for a single aggregation or multiple for a GROUPBY (one for each key). Each data release receives its own sample of noise and consumes additional privacy budget (§5.6). In order to aggregate across multiple video sources (separate time windows and/or multiple cameras), the query can use a SPLIT and PROCESS for each video source, and then aggregate using a JOIN and GROUPBY in the SELECT.

**(2) PROCESS executables.** Executables take one chunk as input, and produce a set of rows (e.g., one per object) as output.

### 5.3 Providing Privacy Despite Blackbox Executables

When running a PRIVID query, an analyst can observe only two pieces of information: (1) the query result, and (2) the time it takes to receive the result.

**Query result.** In order to link an event’s duration to its impact on the output, PRIVID ensures that the output of processing a chunk  $i$  can *only* be influenced by what is visible in chunk  $i$  (not any other chunk  $j$ ). Then, an individual can *only* impact the outputs of chunks in which they appear, and the duration of their appearance is directly proportional to their contribution to the output table.

To achieve this, PRIVID processes each chunk using a separate instance of the analyst’s executable, each running in its own isolated environment. This environment enforces that the executable can read *only* the video chunk, camera metadata, and a random number generator, and can output *only* values formatted according to the PROCESS schema. However, the executable may use arbitrary operations (e.g., custom ML models for CV tasks).

**Execution time.** To prevent the execution time from leaking any information, we must add two additional constraints. First, each chunk must complete and return a value within a pre-determined time limit  $T$ , otherwise a default value is

returned for that chunk (both  $T$  and the default value are provided by the analyst at query time).<sup>6</sup> Second, PRIVID only returns the final aggregated query result after  $|chunks| \cdot T$ . By enforcing these constraints, the observed return time is only a property of the query itself, not the data.

**Implementation.** Our prototype implementation (described in §D of [33]) satisfies these requirements using standard Linux tools. Alternatively, a deployment of PRIVID could use related work [8, 24, 35] on strong isolation with low overhead.

### 5.4 Interface Limitations

The main limitation of PRIVID’s query interface is the inability to write queries that maintain state across separate chunks. However, in most cases this does not preclude queries, it simply requires them to be expressed in a particular way. One broad class of such queries are those that operate over *unique* objects. Consider a query that counts cars. A straightforward implementation might detect `car` objects, output one row for each object, and count the number of rows. However, if a car enters the camera view in chunk  $i$  and is last visible in chunk  $i+n$ , the PROCESS table will include  $n$  rows for the same car instead of the expected 1. To minimize overcounting, the executable can incorporate a license plate reader, output a `plate` attribute for each car, and then count (`DISTINCT plate`) in the SELECT (as in §5.7.1).

Suppose instead the query were counting people, who do not have globally unique identifiers. To minimize overcounting, the PROCESS executable could choose to output a row only for people that *enter* the scene *during that chunk* (and ignore any people that are already visible at the start of a chunk).

PRIVID’s aggregation interface imposes some limitations beyond traditional SQL (detailed in §E of [33], e.g., the SELECT must specify the range of each column), but these are equivalent to the limitations of DP SQL interfaces in prior work.

### 5.5 Query Sensitivity

The sensitivity of a PRIVID query is the maximum amount the final query output could differ given the presence or absence of any  $(\rho, K)$ -bounded event in the video. This can be broken down into two questions: (1) what is the maximum number of rows a  $(\rho, K)$ -bounded event could impact in the analyst-generated intermediate table, and (2) how much could each of these rows contribute to the aggregate output. We discuss each in turn.

**Contribution of a  $(\rho, K)$  event to the table.** An event that is visible in even a single frame of a chunk can impact the output of that chunk arbitrarily, but due to PRIVID’s isolated execution environment, it can *only* impact the output of that chunk, not any others. Thus, the number of rows a  $(\rho, K)$ -bounded event could impact is dependent on the number of chunks it spans (an event spans a set of chunks if it is visible in at least one frame of each).

---

<sup>6</sup>Timeouts can impact query accuracy, hence analysts should first profile their code to select a conservative limit  $T$ .

In the worst case, an event spans the most contiguous chunks when it is first visible in the last frame of a chunk. Given a chunk duration  $c$  (same units as  $\rho$ ) a single event segment of duration  $\rho$  can span at most  $\text{max\_chunks}(\rho)$  chunks:

$$\text{max\_chunks}(\rho) = 1 + \lceil \frac{\rho}{c} \rceil \quad (5.1)$$

**DEFINITION 5.1** (Intermediate Table Sensitivity). Consider a privacy policy  $(\rho, K)$ , and an intermediate table  $t$  (created with a chunk size of  $c_t$  and maximum per-chunk rows  $\text{max\_rows}_t$ ). The *sensitivity* of  $t$  w.r.t  $(\rho, K)$ , denoted  $\Delta_{(\rho, K)}$ , is the maximum number of rows that could differ given the presence or absence of any  $(\rho, K)$ -bounded event:

$$\Delta_{(\rho, K)}(t) \leq \text{max\_rows}_t \cdot K \cdot \text{max\_chunks}(\rho) \quad (5.2)$$

*Proof.* In the worst case, none of the  $K$  segments overlap, and each starts at the last frame of a chunk. Thus, each spans a separate  $\text{max\_chunks}(\rho)$  chunks (Eq. 5.1). For each of these chunks, all of the  $\text{max\_rows}$  output rows could be impacted.  $\square$

**Sensitivity propagation for  $(\rho, K)$ -bounded events.** Prior work [45, 50, 57] has shown how to compute the sensitivity of a SQL query over *traditional* tables. Assuming that queries are expressed in relational algebra, they define the sensitivity recursively on the abstract syntax tree. Beginning with the maximum number of rows an individual could influence in the input table, they provide rules for how the influence of an individual propagates through each relational operator and ultimately impacts the aggregation function.

Unlike prior work on propagating sensitivity recursively, the intermediate tables in PRIVID are untrusted, and thus require careful consideration to ensure the privacy definition is rigorously guaranteed. In this work, we determined the set of operations that can be enabled over PRIVID’s intermediate tables, derived the sensitivity for each, and proved their correctness. Many rules end up being analogous or similar to those in prior work, but JOINs are different. We provide a brief intuition for these differences below. Fig. 9 in §B contains the complete definition for sensitivity of a PRIVID query.

**Privacy semantics of untrusted tables.** As an example, consider a query that computes the size of the intersection between two cameras, PROCESS’d into intermediate tables  $t_1$  and  $t_2$  respectively. If  $\Delta(t_1) = x$  and  $\Delta(t_2) = y$ , it is tempting to assume  $\Delta(t_1 \cap t_2) = \min(x, y)$ , because a value needs to appear in both  $t_1$  and  $t_2$  to appear in the intersection. However, because the analyst’s executable can populate the table arbitrarily, they can “prime”  $t_1$  with values that would only appear in  $t_2$ , and vice versa. As a result, a value need only appear in either  $t_1$  or  $t_2$  to show up in the intersection, and thus  $\Delta(t_1 \cap t_2) = x + y$ .

**Theorem 5.1.** PRIVID’s sensitivity definition (Fig. 9, §B) provides  $(\rho, K, \epsilon)$ -privacy for a query  $Q$  over  $V$ .

We provide the formal proof in §B.

## 5.6 Handling Multiple Queries

In traditional DP, the parameter  $\epsilon$  is viewed as a “privacy budget”. Informally,  $\epsilon$  defines the total amount of information that may be released about a database, and each query consumes a portion of this budget. Once the budget is depleted, no further queries can be answered.

Rather than assigning a single global budget to an entire video, PRIVID allocates a separate budget of  $\epsilon$  to each frame of a video. When PRIVID receives a query  $Q$  over frames  $[a, b]$  requesting budget  $\epsilon_Q$ , it only accepts the query if *all* frames in the interval  $[a - \rho, b + \rho]$  have sufficient budget  $\geq \epsilon_Q$ , otherwise the query is denied (Alg. 1 Lines 1-3). If the query is accepted, PRIVID then subtracts  $\epsilon_Q$  from each frame in  $[a, b]$ , but *not* the  $\rho$  margin (Alg. 1 Lines 4-5). We require sufficient budget at the  $\rho$  margin to ensure that any single segment of an event (which has duration at most  $\rho$ ) cannot span two temporally disjoint queries (§B).

Note that since each SELECT in a query represents a separate data release, the total budget  $\epsilon_Q$  used by a query is the sum of the  $\epsilon_i$  used by each of the  $i$  SELECTs. The analyst can specify the amount of budget they would like to use for each release (via a CONSUMING clause, defined in §E of [33], see example in §5.7.1).

**Putting it all together.** Algorithm 1 presents a simplified (single video) version of the PRIVID query execution process. We provide the full algorithm in §G of [33].

---

### Algorithm 1: PRIVID Query Execution (simplified)

---

```

Input : Query  $Q$ , video  $V$ , interval  $[a, b]$ , policy  $(\rho, K, \epsilon)$ 
Output: Query answer  $A$ 
1 foreach frame  $f \in V[a - \rho : b + \rho]$  do
2   if  $f.\text{budget} < \epsilon_Q$  then
3     return DENY
4 foreach frame  $f \in V[a : b]$  do
5    $f.\text{budget} -= Q.\text{budget}$ 
6    $\text{chunks} \leftarrow \text{Split } V[a : b] \text{ into chunks of duration } c$ 
7    $T \leftarrow \text{Table(schema)}$ 
8   foreach chunk  $\in \text{chunks}$  do
9      $\text{rows} \leftarrow F(\text{chunk}) // \text{in isolated environment}$ 
10     $T.\text{append}(\text{rows})$ 
11    $r \leftarrow \text{execute PrividQL query } S \text{ over table } T$ 
12    $\Delta_{(\rho, K)} \leftarrow \text{compute recursively over the structure of } S$  (§5.5)
13    $\eta \leftarrow \text{Laplace}(\mu=0, b=\frac{\Delta}{\epsilon_Q})$ 
14    $A \leftarrow r + \eta$ 

```

---

**Theorem 5.2.** Consider an adaptive sequence (§2.3) of  $n$  queries  $Q_1, \dots, Q_n$ , each over the same camera  $C$ , a privacy policy  $(\rho_C, K_C)$ , and global budget  $\epsilon_C$ . PRIVID (Algorithm 1) provides  $(\rho_C, K_C, \epsilon_C)$ -privacy for all  $Q_1, \dots, Q_n$ .

We provide the formal proof in §B.

## 5.7 Example Queries

### 5.7.1 Benevolent Query

Suppose a VO provides access to `camA` via PRIVID, with a policy ( $\rho=60s, K=2$ ). The city transportation department wishes to collect statistics about vehicles passing `camA` during October 2021. We formulate two questions as a PRIVID query:

```
-- Select 1 month time window from camera, split into chunks
SPLIT camA
BEGIN 10-01-2021/12:00am END 11-01-2021/12:00am
BY TIME 10sec STRIDE 0sec
INTO chunksA;
-- Process chunks using analyst's code, store outputs in tableA
PROCESS chunksA USING traffic_flow.py TIMEOUT 1sec
PRODUCING 20 ROWS
WITH SCHEMA (plate:STRING="", type:STRING "", speed:NUMBER=0)
INTO vehiclesA;
-- S1: Number of unique cars per day
SELECT day,COUNT(DISTINCT plate) FROM vehiclesA WHERE type=="car"
GROUP BY day CONSUMING eps=0.5;
-- S2: Average speed of trucks
SELECT AVG(range(speed, 30, 60)) FROM vehiclesA WHERE type=="truck"
CONSUMING eps=0.5;
```

The `SPLIT` selects 1 month of video from `camA`, then divides the frames into a list of 10-second-long chunks (267k chunks total). The `PROCESS` first creates an empty table based on the `SCHEMA` (3 columns). Then, for each chunk, it starts a fresh instance of `traffic_flow.py` inside a restricted container, provides the chunk as input, and appends the output as rows to `vehiclesA`. The executable `traffic_flow.py` contains off-the-shelf object detection and tracking models, a license plate reader, and a speed estimation algorithm (source in §F of [33]).

The first `SELECT` filters all cars, then counts the “distinct” license plates to estimate the number of *unique* cars per day. Each day is a separate data release with an independent sample of noise. The second `SELECT` filters all trucks, then computes the average speed across the entire month of footage. It uses the same input video as the first select, and thus draws from the same budget, so in aggregate the two `SELECT`s consume  $\epsilon = 1.0$  budget from all frames in October 2021.

### 5.7.2 Malicious Query Attempt

Now consider a malicious analyst *Mal* who wishes to determine if individual  $x$  appeared in front of `camA` each day. Assume  $x$ ’s appearance is bound by the VO’s  $(\rho, K)$  policy.

To hide their intent, *Mal* disguises their query as a traffic counter, mimicking  $S_1$  from the previous example. They write identical query statements, but their “`traffic_flow.py`” instead includes specialized models to detect  $x$ . If  $x$  appears, it outputs 20 rows (the maximum) with random values for each of the columns, otherwise it outputs 0 rows. This adds 20 rows to the corresponding daily count for each chunk  $x$  appears.

**Amplification attempt.** Due to the isolated environment (§5.3), the `PROCESS` executable can only output rows for a chunk if  $x$  *truly appears*. It has no way of saving state or communicating between executions in order to artificially output rows for a chunk in which  $x$  does not appear. It could output more than 20 rows for a single chunk, but PRIVID ignores any rows beyond the `PROCESS`’s explicit max (20), so this would not

increase the count. Increasing the rows per chunk parameter would also be pointless: PRIVID would compute a proportionally higher sensitivity and add proportionally higher noise.

**Side channel attempt.** The executable could try to encode the entire contents of a frame in a row of the table, either by encoding it as a string, or a very large number of individual integer columns. But in either case, the analyst cannot view the table directly or even a single row directly, it can only compute noisy aggregations over entire columns.

**Summary.** PRIVID would compute the sensitivity of  $S_1$  (identical in both the benevolent and malicious cases) as  $\Delta_{(60,2)}(Q) \leq 20 \cdot 2 \cdot (1 + \lceil \frac{60}{10} \rceil) = 280$  rows, meaning it would add noise with scale 280 to each daily count. Regardless of how *Mal* changes her executable, it cannot output more than 280 rows based on  $x$ ’s presence. Thus, even if she observed a non-zero value  $\sim 280$ , she could not distinguish whether it is a result of the noise or  $x$ ’s appearance.

*Mal*’s query gets a useless result, because her target ( $x$ ’s appearance) was close in duration to the policy. In contrast, the benevolent query can get a useful result because the duration of its target (the set of *all cars*’ appearances) far exceeds the policy. PRIVID’s noise will translate to  $\mathcal{L}^{-1}(p=0.99, u=0, b=\frac{\Delta}{\epsilon} = \frac{280}{0.5}) \leq 2200$  cars with 99% confidence. If, for example, there are an average of 10 cars in each chunk (and thus 86000 in one day), 2200 represents an error of  $\pm 2.5\%$ .

## 6 Query Utility Optimization

The noise that PRIVID adds to a query result is proportional to both the privacy policy  $(\rho, K)$  and the range of the aggregated values (the larger the range, the more noise PRIVID must add to compensate for it). In this section we introduce two optional optimizations that PRIVID offers analysts to improve query accuracy while maintaining an equivalent level of privacy: one reduces the  $\rho$  needed to preserve privacy (§6.1), while the other reduces the range for aggregation (§6.2).

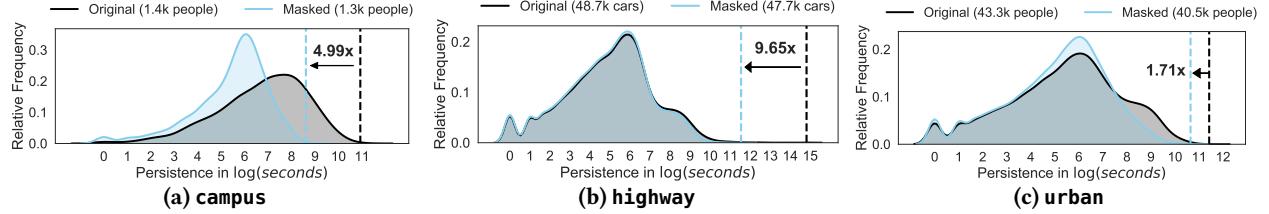
### 6.1 Spatial Masking

**Observation.** In certain settings, a few individuals may be visible to a camera for far longer than others (e.g., those sitting on a bench or in a car), creating a heavy-tailed distribution of presence durations. Fig. 3 (top row) provides some representative examples. Setting  $(\rho, K)$  to the maximum



(a) campus      (b) highway      (c) urban

**Figure 3:** (Top) Heatmap measuring the maximum time any object spent in each pixel, noramlized to the max (yellow) per video. (Bottom) The resulting masks used for our evaluation, chosen from the list of masks automatically generated using the algorithm in §I of [33].



**Figure 4:** The distribution of private objects’ durations (persistence) is heavy tailed. Applying the mask from Fig. 3 significantly lowers the maximum duration, while still allowing most private objects to be detected. The key denotes the total number of private objects detectable before and after applying the mask. The dotted lines highlight the maximum persistence, and the arrow text denotes the relative reduction.

duration in such distributions would result in a large amount of noise needed to protect just those few individuals; all others could have been protected with a far lower amount of noise. We observe that, in many cases, lingering individuals tend to spend the majority of their time in one of a few fixed regions in the scene, but a relatively short time in the rest of the scene. For example, a car may be parked in a spot for hours, but only visible for 1 minute while entering/leaving the spot.

**Opportunity.** Masking fixed regions (i.e., removing those pixels from all frames prior to running the analyst’s video processing) in the scene that contain lingering individuals would drastically reduce the *observable* maximum duration of individuals’ presence, e.g., the parked car from above would be observable for 1 minute rather than hours. This, in turn, would permit a policy with a smaller  $\rho$ , but an equivalent level of privacy—all appearances would still be bound by the policy. Of course, this technique is only useful to an analyst when the remaining (unmasked) part of the scene includes all the information needed for the query at hand, e.g., if counting cars, masking sidewalks would be reasonable but masking roads would not.

**Optimization.** At camera-registration time, instead of providing a single  $(\rho, K)$  policy per camera, the VO can provide a (fixed) list of a few frame masks and, for each, a corresponding  $(\rho, K)$  policy that would provide equivalent privacy when that mask is applied. At query time, the analyst can (optionally) choose a mask from the list that would minimally impact their query goal while maximizing the level of noise reduction (via the tighter  $(\rho, K)$  bound). If a mask is chosen, PRIVID applies it to all video frames *before* passing it to the analyst’s PROCESS executable (the analyst only “sees” the masked video), and uses the corresponding  $(\rho, K)$  in the sensitivity calculation (§5.5).

To aid the analyst in discovering a useful set of masks (i.e., those that reduce  $(\rho, K)$  as much as possible using the fewest pixels), we provide an algorithm in §I.2 of [33]. Regardless of how they are chosen, the masks themselves are static (i.e., the same pixels are masked in every frame regardless of its contents), and the set of available masks is fixed. Neither depend on the query or the target video. Further, the mask itself does not reveal how the analyst generated it or which specific objects contributed to it, it only tells the analyst that some objects appear for a long duration in the masked region.

**Noise reduction.** We demonstrate the potential benefit of masking on three queries (Q1–Q3) from our evaluation

Video	Max(frame)	Max(region)	Reduction
campus	6	3	2.00×
highway	40	23	1.74×
urban	37	16	2.25×

**Table 2:** Reduction in max output range from splitting each video into distinct regions. Reduction shows the factor by which the noise could be reduced.  $2\times$  cuts the necessary privacy level in half.

(Table 3). Given the query tasks (counting unique people and cars), we chose masks that would maximally reduce  $\rho$  without impacting the object counts; the bottom row of Fig. 3 visualizes our masks. Fig. 4 shows that these masks reduce maximum durations by 1.71–9.65 $\times$ . In §I.1 of [33] we show that masking provides similar benefits for 7 additional videos evaluated by BlazeIt [47] and MIRIS [30].

**Masking vs. denaturing.** Although masking is a form of denaturing, PRIVID uses it differently than the prior approaches in §3.1, in order to sidestep their issues. Rather than attempting to dynamically hide individuals as they move through the scene, PRIVID’s masks cover a *fixed* location in the scene and are publicly available so analysts can account for them in their query implementation. Also, masks are used as an optional modification to the input video; the rest of the PRIVID pipeline, and thus its formal privacy guarantees, remain the same.

## 6.2 Spatial Splitting

**Observation.** (1) At any point in time, each object typically occupies a relatively small area of a video frame. (2) Many common queries (e.g., object detections) do not need to examine the entire contents of a frame at once, i.e., if the video is split spatially into regions, they can compute the same total result by processing each of the regions separately.

**Opportunity.** PRIVID already splits videos temporally into chunks. If each chunk is further divided into spatial regions and an individual can only appear in one of these chunks at a time, then their presence occupies a relatively smaller portion of the intermediate table (and thus requires less noise to protect). Additionally, the maximum duration of each individual region may be smaller than the frame as a whole.

**Optimization.** At camera-registration time, PRIVID allows VOs to manually specify boundaries for dividing the scene into regions. They must also specify whether the boundaries are soft (individuals may cross them over time, e.g., between two crosswalks) or hard (individuals will never cross them, e.g., between opposite directions on a highway). At query

time, analysts can optionally choose to spatially split the video using these boundaries. Note that this is in addition to, rather than in replacement of, the temporal splitting. If the boundaries are soft, tables created using that split must use a chunk size of 1 frame to ensure that an individual can always be in at most 1 chunk. If the boundaries are hard, there are no restrictions on chunk size since the VO has stated the constraint will always be true.

**Noise reduction.** We demonstrate the potential benefit of spatial splitting on three videos from our evaluation (Q1-Q3). For each video, we manually chose intuitive regions: a separate region for each crosswalk in campus and urban (2 and 4, respectively), and a separate region for each direction of the road in highway. Table 2 compares the range necessary to capture all objects that appear within one chunk in the entire frame compared to the individual regions. The difference ( $1.74\text{-}2.25\times$ ) represents the potential noise reductions from splitting: noise is proportional to  $\max(\text{frame})$  or  $\max(\text{region})$  when splitting is disabled or enabled, respectively.

**Grid split.** To increase the applicability of spatial splitting, PRIVID could allow analysts to divide each frame into a grid and remove the restrictions on soft boundaries to allow any chunk size. This would require additional estimates about the max size of any private object (dictating the max number of regions they could occupy at any time), and the maximum speed of any object across the frame (dictating the max number of regions they could move between). We leave this to future work.

## 7 Evaluation

The evaluation highlights of PRIVID are as follows:

1. PRIVID supports a diverse range of video analytics queries, including object counting, duration queries, and composite queries; for each, PRIVID increases error by 1-21% relative to a non-private system, while protecting all individuals with  $(\rho, K, \epsilon)$ -privacy (§7.2).
2. PRIVID enables VOs and analysts to flexibly and formally trade utility loss and query granularity while preserving the same privacy guarantee (§7.3).

### 7.1 Evaluation Setup

**Datasets.** We evaluated PRIVID primarily using three representative video streams (campus, highway and urban, screenshots in Fig. 3) that we collected from YouTube spanning 12 hours each (6am-6pm). For one case study (multi-camera), we use the Porto Taxi dataset [58] containing 1.7mil trajectories of all 442 taxis running in the city of Porto, Portugal from Jan. 2013 to July 2014. We apply the same processing as [42] to emulate a city-wide camera dataset; the result is the set of timestamps each taxi would have been visible to each of 105 cameras over the 1.5 year period.

**Implementation.** We implemented PRIVID in 4k lines of Python. We used the Faster-RCNN [63] model in Detectron-v2 [71] for object detection, and DeepSORT [69] for object tracking. For these models to work reasonably given the di-

verse content of the videos, we chose hyperparameters for detection and tracking on a per-video basis (details in §H of [33]).

**Privacy policies.** We assume the VO’s underlying privacy goal is to “protect the appearance of all individuals”. For each camera, we use the strategy in §6.1, to create a map between masks and  $(\rho, K)$  policies that achieve this goal.

**Query parameters.** For each query, we first chose a mask that covered as much area as possible (to get the minimal  $\rho$ ) without disrupting the query. The resulting  $\rho$  values are in Table 3. We use a budget of  $\epsilon = 1$  for each query. We chose query windows sizes ( $W$ ), chunk durations ( $c$ ), and column ranges to best approximate the analyst’s expectations for each query (as opposed to picking optimal values based on a parameter sweep, which the analyst is unable to do).

**Baselines.** For each query, we compute `error` by comparing the output of PRIVID to running the same exact query implementation without PRIVID. We execute each query 1000 times, and report the mean accuracy value  $\pm 1$  standard deviation.

### 7.2 Query Case Studies

We formulate five types of queries to span a variety of axes (target object class, number of cameras, aggregation type, query duration, standing vs. one-off query). Fig. 5 displays results for Q1-Q3. Table 3 summarizes the remaining queries (Q4-Q13).

**Case 1: Q1-Q3 (Counting private objects over time).** To demonstrate PRIVID’s support for standing queries and short (1 hour) aggregation durations, we SUM the number of *unique* objects observed *each hour* over the 12 hours.

**Case 2: Q4-Q6 (Aggregating over multiple cameras with complex operators).** We utilize UNION, JOIN, and ARGMAX to aggregate over cameras in the Porto Taxi Dataset. Due to the large aggregation window (1 year), PRIVID’s noise addition is small (relative to the other queries using a window on the order of hours) and accuracy is high.

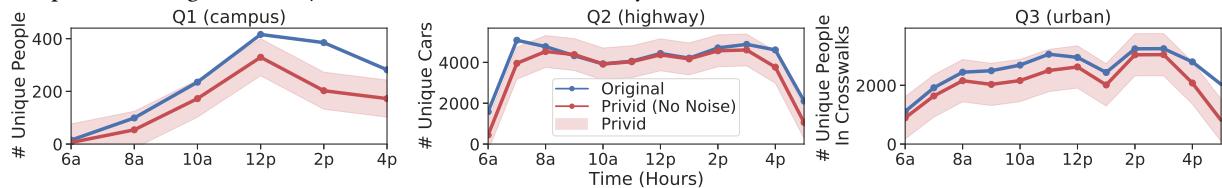
**Case 3: Q7-Q9 (Counting non-private objects, large window).** We measure the fraction of trees (non-private objects) that have bloomed in each video. Executed over an entire network of cameras, such a query could be used to identify the regions with the best foliage in Spring. Relative to Case 1, we achieve high accuracy by using a longer query window of 12 hours (the status of a tree does not change on that time scale), and minimal chunk size (1 frame, no temporal context needed).

**Case 4: Q10-Q12 (Fine-grained results using aggressive masking).** We measure the average amount of time a traffic signal stays red. Since this only requires observing the light itself, we can mask *everything else*, resulting in a  $\rho$  bound of 0 (no private objects overlap these pixels), enabling high accuracy and fine temporal granularity.

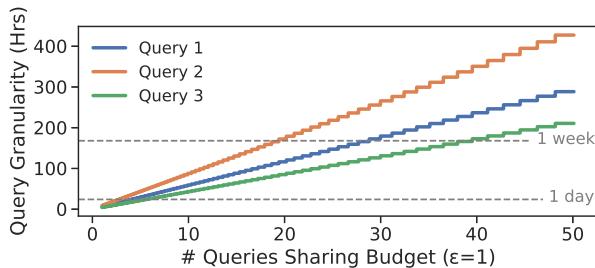
**Case 5: Q13 (Stateful query).** We count only the individuals that enter from the south and exit at the north. It requires a larger chunk size (relative to Q1-Q3) to maintain enough state within a single chunk to understand trajectory.

Case #	Q#	Query Description	Query Parameters	Video	$\rho$	Query Output	Error
Case 2	Q4	Average Taxi Driver Working Hours (union across 2 cameras)	$ W =365$ days, $c=15$ sec, Agg = avg, range = (0,16)	porto10, porto27	[45, 195] sec	5.87 hrs	$5.86\%\pm 0.18\%$
Case 2	Q5	Average # Taxis Traversing 2 Locations on Same Day (intersection across 2 cameras)	$ W =365$ days, $c=15$ sec, Agg = avg, range = (0,300)	porto10, porto27	[45, 195] sec	131 taxis	$0.20\%\pm 0.13\%$
Case 2	Q6	Identifying Camera with Highest Daily Traffic (argmax across all 105 cameras)	$ W =365$ days, $c=15$ sec, Agg = argmax	porto0, ..., porto104	[15, 525] sec	porto20	0%
Case 3	Q7	Fraction of trees with leaves (%)	$ W =12$ hrs, $c=1$ frame, Agg = avg, range = (0,100)	campus	49 sec	$15/15 = 1.00$	$0.10\%\pm 0.11\%$
	Q8			highway	6.21 min	$3/7 = 0.43$	$1.76\%\pm 1.90\%$
	Q9			urban	3.34 min	$4/6 = 0.67$	$0.61\%\pm 0.66\%$
Case 4	Q10	Duration of Red Light (seconds)	$ W =12$ hrs, $c=30$ min, Agg = avg, range = (0,300)	campus	1 frame	75 sec	$0\%\pm 1.4\times 10^{-4}\%$
	Q11			highway	1 frame	50 sec	$0\%\pm 2.1\times 10^{-4}\%$
	Q12			urban	1 frame	100 sec	$0\%\pm 1.0\times 10^{-4}\%$
Case 5	Q13	# Unique People (Filter: trajectory moving towards campus)	$ W =12$ hrs, $c=10$ sec, Agg = sum, range = (0,5)	campus	49 sec	576 people	$20.31\%\pm 2.60\%$

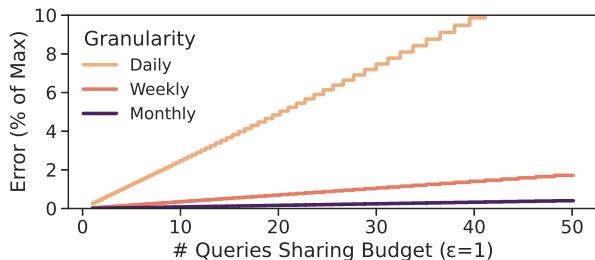
**Table 3:** Summary of query results for Q4-Q13. For Case 3 and 5, we use the same masks (and thus  $\rho$ ) from Fig. 3. For Case 4, we mask all pixels except the traffic light to attain  $\rho=0$ . For Case 2 we do not use any masks.



**Figure 5:** Time series of PRIVID’s output for Case 1 queries. “Original” is the baseline query output without using PRIVID. “Privid (No Noise)” shows the raw output of PRIVID before noise is added. The final noisy output will fall within the range of the red ribbon 99% of the time.



**Figure 6:** Given a fixed query and accuracy target, decreasing the amount of budget used by each query allows more queries to be executed over the same video segment, but requires a proportionally coarser granularity. The  $x$ -axis plots the number of queries evenly sharing a budget of  $\epsilon=1$ , thus  $x=10$  means 10 instances of the same exact query over the same video segment, each using a budget of  $\frac{1}{10}$ . We fix the accuracy target to be 99% of values having error  $\leq 5\%$ .



**Figure 7:** Given a fixed query and granularity, decreasing the amount of budget used by each query allows more queries to be executed over the same video segment, but results in proportionally higher error. The  $x$ -axis is the same as Fig. 6. Each line corresponds to Q1 using a different granularity. The  $y$ -axis plots the error for 99% of values. Error is the amount of noise added relative to the maximum query output. For example, in Q4, the final output is the average number of working hours in the range [0,16]. Thus an error of 1% would mean the noisy result is within 0.16 hours of the true result.

### 7.3 Budget-Granularity Tradeoff

Analysts have two main knobs for each query  $Q$  to navigate the utility space: (1) the fraction  $\epsilon_Q$  of the total budget  $\epsilon$  used by that query, and (2) the duration (granularity) of each aggregation (i.e., “one value per day for a month” has a granularity of one day). The query budget is inversely proportional to both the query granularity and error (the expected value of noise PRIVID adds relative to the output range). Thus, to decrease the amount of budget per query (or equivalently, increase the number of queries sharing the budget), an analyst must choose a (temporally) coarser result, a larger expected error bound, or both. Fig. 6 shows that, for example, 5 instances of Query 3 could release results daily or 40 instances of Query 3 could release results weekly, while achieving the same expected accuracy. Fig. 7 shows that, for example, 20 separate instances of Query 1 ( $x=20$ ) executed over the same target video could each expect 4.8% error if they release one result daily, 0.7% error if they release one weekly, or 0.16% error if they release one monthly. Importantly, this tradeoff is transparent to analysts: Figs. 6 and 7 rely only on information that is publicly available to analysts and did not require executing any queries.

### 7.4 Analyzing Sources of Inaccuracy

PRIVID introduces two sources of inaccuracy to a query result: (1) intentional noise to satisfy  $(\rho, K, \epsilon)$ -privacy, and (2) (unintentional) inaccuracies caused by the impact of splitting and masking videos before executing the video processing. Fig. 5 shows these two sources separately for queries Q1-Q3 (Case 1): the discrepancy between the two curves demonstrates the impact of (2), while the shaded belt shows the relative scale of noise added (1). In summary, the scale of noise added by PRIVID allows the final result to preserve the trend of the original.

## 8 Using PRIVID

In this section, we summarize the set of decisions that both the VO and analyst need to make when interacting with PRIVID.

### 8.1 Video Owner

First, the VO must register a set of cameras with PRIVID. For *each* camera, they must supply: (1) a  $(\rho, K)$  bound (or more generally a map of masks to bounds), (2) a privacy budget  $\epsilon$ , and (3) some metadata describing the scene to analysts (e.g., a short video clip, since they cannot view the camera feed directly). All of this is public to analysts. Below we provide general suggestions for the VO, but ultimately they are responsible for choosing these values. PRIVID only handles enforcing a given policy.

**(1)  $(\rho, K)$  bounds.** In most cases, we expect the VO will record a sample of video, measure durations of objects of interest using off-the-shelf tracking algorithms, and then set the bound to the longest duration.

To provide better utility for analysts, the VO can offer a menu of static masks that remove some of the scene in exchange for tighter noise bounds than the original policy (which is itself mapped to the empty mask). Note that the VO must explicitly choose a  $(\rho, K)$  policy for each mask. A mask is only useful if it reduces the amount of time the longest objects are visible, which enables a tighter bound while protecting the same set of individuals.

The VO may draw masks manually or generate them automatically, e.g., by analyzing past trends from the camera. In general, we expect masks to be static properties of each scene, dependent only on dynamics of the scene type, rather than behaviors of any individuals. However, it is ultimately the VO’s responsibility to ensure any masks it provides do not reveal anything private, such as a person’s silhouette. PRIVID focuses on preventing the leakage of privacy when answering queries. It does not make any guarantees about the mask itself.

**(2) Budget  $\epsilon$ .** As in any deployment of DP, the choice of  $\epsilon$  is subjective. Academic papers commonly use  $\epsilon \approx 1$  [52] while recent industry deployments have used  $1 < \epsilon < 10$  [27, 36, 56]. Note that in PRIVID, this budget is *per-frame* (§5.6); two queries aggregating over disjoint time ranges of the same video draw from separate budgets. The only PRIVID-specific consideration for choosing  $\epsilon$  is that cameras with overlapping fields of view should share the same budget.

**(3) Metadata.** The VO should release a sample video clip<sup>7</sup> representative of the scene so that analysts can calibrate their executable<sup>8</sup> and query<sup>9</sup> accordingly. Any privacy loss resulting from the one-time release of this single clip is limited, and can be manually vetted by the VO. Optionally, the VO can release additional information to aid analysts, such as the camera’s GPS coordinates, make, or focal length settings.

<sup>7</sup>While a clip is not needed in principle, without it, the analyst “runs blind” and will not have confidence in the correctness of their results.

<sup>8</sup>ML models may perform better when retrained on a particular scene.

<sup>9</sup>For example, queries must specify bounds on the amount of output per chunk, which depend on the amount of activity in the scene.

### 8.2 Analyst

In order to formulate a PRIVID query the analyst must make the following decisions. For each decision, we provide an example for the query in §5.7.1 (counting cars crossing a virtual line on a highway).

**Choose a mask** (from the list provided by the VO) based on the query goal. For example, they should select a mask that covers as much of the scene as possible without covering the area near the virtual line. This would significantly reduce the bound by removing parking spots and intersections where objects linger.

**Choose a chunk size** based on the amount of context needed. A larger chunk size permits more context for each execution of the PROCESS, but results in more noise (§5.5). Thus, the analyst should choose the smallest chunk size that captures their events of interest. For example, 1 second is likely sufficient to capture cars driving past a line. If they instead wanted to calculate car speed, they would need a larger chunk size (e.g., 10 seconds) and less restrictive mask to capture more of the car’s trajectory.

**Choose upper bound on number of output rows per chunk** based on the expected (via the video sample) level of activity in each chunk. For counting cars over a short chunk, especially in less busy scenes, each chunk may see 1-2 cars and thus need 1-2 rows. For calculating speed over a larger chunk, especially in more busy scenes, each chunk will see more cars and may need 10 or 100 rows.

**Create a PROCESS executable.** This involves tuning their CV models based on the scene (via the sample video), and combining all tasks into a single executable. For example, their executable may include an object detector to find cars, an object tracker to link them to trajectories, a license plate reader to link cars across cameras or prevent double counting, and an algorithm to compute speed or determine car model.

**Choose query granularity and budget.** The query granularity and budget are directly proportional to accuracy. Given a fixed value for each, improving one requires worsening another proportionally. We elaborate upon this tradeoff in §7.3.

## 9 Ethics

In building PRIVID, we *do not* advocate for the increase of public video surveillance and analysis. Instead, we observe that it is already prevalent and seek to improve the privacy landscape. PRIVID’s accuracy and expressiveness makes it palatable to add formal privacy to existing analytics, and lowers the barrier to deployment. If privacy legislation is introduced, PRIVID could be one way to ensure compliance.

**Acknowledgements.** We thank Hari Balakrishnan, Matt Lentz, Dave Levin, Amy Ousterhout, Jennifer Rexford, Eugene Wu, the NSDI reviewers, and our shepherd, Jonathan Mace, for their helpful feedback and suggestions. This work was partially supported by a Sloan fellowship and NSF grants CNS-2153449, CNS-2152313, CNS-2140552, and CNS-2151630.

## References

- [1] Absolutely everywhere in beijing is now covered by police video surveillance. <https://qz.com/518874/>.
- [2] Are we ready for ai-powered security cameras? <https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/>.
- [3] British transport police: Cctv. [http://www.btp.police.uk/advice\\_and\\_information/safety\\_on\\_and\\_near\\_the\\_railway/cctv.aspx](http://www.btp.police.uk/advice_and_information/safety_on_and_near_the_railway/cctv.aspx).
- [4] Can 30,000 cameras help solve chicago's crime problem? <https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html>.
- [5] Data generated by new surveillance cameras to increase exponentially in the coming years. <http://www.securityinfowatch.com/news/12160483/>.
- [6] Detection leaderboard. <https://cocodataset.org/#detection-leaderboard>.
- [7] Epic domestic surveillance project. <https://epic.org/privacy/surveillance/>.
- [8] nsjail. <https://github.com/google/nsjail>.
- [9] Oakland bans use of facial recognition. <https://www.sfchronicle.com/bayarea/article/Oakland-bans-use-of-facial-recognition-14101253.php>.
- [10] Paris hospitals to get 1,500 cctv cameras to combat violence against staff. <https://bit.ly/20YiBz2>.
- [11] Powering the edge with ai in an iot world. <https://www.forbes.com/sites/forbestechcouncil/2020/04/06/powering-the-edge-with-ai-in-an-iot-world/>.
- [12] San francisco is first us city to ban facial recognition. <https://www.bbc.com/news/technology-48276660>.
- [13] Video analytics applications in retail - beyond security. [https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/](https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html).
- [14] The vision zero initiative. <http://www.visionzeroinitiative.com/>.
- [15] What's wrong with public video surveillance? <https://www.aclu.org/other/whats-wrong-public-video-surveillance>, 2002.
- [16] Abuses of surveillance cameras. <http://www.notbored.org/camera-abuses.html>, 2010.
- [17] Mission creep-y: Google is quietly becoming one of the nation's most powerful political forces while expanding its information-collection empire. <https://www.citizen.org/wp-content/uploads/google-political-spending-mission-creepy.pdf>, 2014.
- [18] Mission creep. <https://www.aclu.org/other/whats-wrong-public-video-surveillance>, 2017.
- [19] How retail stores can streamline operations with video content analytics. <https://www.briefcam.com/resources/blog/how-retail-stores-can-streamline-operations-with-video-content-analytics/>, 2020.
- [20] The mission creep of smart streetlights. <https://www.voiceofsandiego.org/topics/public-safety/the-mission-creep-of-smart-streetlights/>, 2020.
- [21] Video analytics traffic study creates baseline for change. <https://www.govtech.com/analytics/Video-Analytics-Traffic-Study-Creates-Baseline-for-Change.html>, 2020.
- [22] What is computer vision? ai for images and video. <https://www.infoworld.com/article/3572553/what-is-computer-vision-ai-for-images-and-video.html>, 2020.
- [23] P. Aditya, R. Sen, P. Druschel, S. Joon Oh, R. Benenson, M. Fritz, B. Schiele, B. Bhattacharjee, and T. T. Wu. I-pic: A platform for privacy-compliant image capture. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 235–248, New York, NY, USA, 2016. Association for Computing Machinery.
- [24] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, pages 419–434, 2020.
- [25] Amazon. Rekognition. <https://aws.amazon.com/rekognition/>.
- [26] G. Ananthanarayanan, Y. Shu, M. Kasap, A. Kewalramani, M. Gada, and V. Bahl. Live video analytics with microsoft rocket for reducing edge compute costs, July 2020.
- [27] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 1(8), 2017.
- [28] M. Azure. Computer vision api. <https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/>, 2021.
- [29] M. Azure. Face api. <https://azure.microsoft.com/en-us/services/cognitive-services/face/>, 2021.
- [30] F. Bastani, S. He, A. Balasingam, K. Gopalakrishnan, M. Alizadeh, H. Balakrishnan, M. Cafarella, T. Kraska, and S. Madden. Miris: Fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1907–1921, New York, NY, USA, 2020. Association for Computing Machinery.

- [31] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft. Simple online and realtime tracking. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 3464–3468, 2016.
- [32] Z. Cai, M. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV ’15*, pages 3361–3369, Washington, DC, USA, 2015. IEEE Computer Society.
- [33] F. Cangialosi, N. Agarwal, V. Arun, J. Jiang, S. Narayana, A. Saarwate, and R. Netravali. Privid: Practical, privacy-preserving video analytics queries (extended version). <https://arxiv.org/abs/2106.12083>.
- [34] A. Chattopadhyay and T. E. Boult. Privacycam: a privacy preserving camera using uclinux on the blackfin dsp. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.
- [35] G. Chrome. minijail0. <https://google.github.io/minijail/>.
- [36] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3571–3580. Curran Associates, Inc., 2017.
- [37] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In S. Halevi and T. Rabin, editors, *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284, Berlin, Heidelberg, Mar. 2006. Springer.
- [38] I. Ghodgaonkar, S. Chakraborty, V. Banna, S. Allcroft, M. Metwaly, F. Bordwell, K. Kimura, X. Zhao, A. Goel, C. Tung, et al. Analyzing worldwide social distancing through large-scale computer vision. *arXiv preprint arXiv:2008.12363*, 2020.
- [39] Google. Cloud vision api. <https://cloud.google.com/vision>, 2021.
- [40] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, 2018.
- [41] IBM. Maximo remote monitoring. <https://www.ibm.com/products/maximo/remote-monitoring>, 2021.
- [42] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. E. Gonzalez. Scaling Video Analytics Systems to Large Camera Deployments. In *ACM HotMobile*, 2019.
- [43] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, V. Bahl, and J. Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020.
- [44] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266. ACM, 2018.
- [45] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.
- [46] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. *IEEE Transactions on Information Theory*, 63(6):4037–4049, 2017.
- [47] D. Kang, P. Bailis, and M. Zaharia. Blazeit: optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proceedings of the VLDB Endowment*, 13(4):533–546, 2019.
- [48] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.
- [49] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, and M. Zaharia. Task-agnostic indexes for deep learning-based queries over unstructured data. *arXiv preprint arXiv:2009.04540*, 2020.
- [50] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeeepour, A. Machanavajjhala, M. Hay, and G. Miklau. Privatesql: A differentially private sql query engine. *Proc. VLDB Endow.*, 12(11):1371–1384, July 2019.
- [51] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.
- [52] Y.-H. Kuo, C.-C. Chiu, D. Kifer, M. Hay, and A. Machanavajjhala. Differentially private hierarchical count-of-counts histograms. *Proceedings of the VLDB Endowment*, 11.11:1509–1521, 2018.
- [53] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.
- [54] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. *SIGCOMM ’20*, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

- [55] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, July 2017.
- [56] A. Machanavajjhala, D. Kifer, J. M. Abowd, J. Gehrke, and L. Vilhuber. Privacy: Theory meets practice on the map. In *ICDE*, 2008.
- [57] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’09, page 19–30, New York, NY, USA, 2009. Association for Computing Machinery.
- [58] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi-passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.
- [59] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers’ track at the RSA conference*, pages 1–20. Springer, 2006.
- [60] J. R. Padilla-López, A. A. Chaaraoui, and F. Flórez-Revuelta. Visual privacy protection methods: A survey. *Expert Systems with Applications*, 42(9):4177–4195, 2015.
- [61] C. Percival. Cache missing for fun and profit, 2005.
- [62] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1039–1056, 2020.
- [63] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.
- [64] J. Stanley and A. C. L. Union. *The Dawn of Robot Surveillance: AI, Video Analytics, and Privacy*. American Civil Liberties Union, 2019.
- [65] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR ’13, pages 3476–3483, Washington, DC, USA, 2013. IEEE Computer Society.
- [66] H. Wang, Y. Hong, Y. Kong, and J. Vaidya. Publishing video data with indistinguishable objects. *Advances in database technology : proceedings. International Conference on Extending Database Technology*, 2020:323 – 334, 2020.
- [67] H. Wang, S. Xie, and Y. Hong. Videodp: A universal platform for video analytics with differential privacy. *arXiv preprint arXiv:1909.08729*, 2019.
- [68] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan. A scalable and privacy-aware iot service for live video analytics. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, pages 38–49. ACM, 2017.
- [69] N. Wojke, A. Bewley, and D. Paulus. Simple online and realtime tracking with a deep association metric. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3645–3649. IEEE, 2017.
- [70] H. Wu, X. Tian, M. Li, Y. Liu, G. Ananthanarayanan, F. Xu, and S. Zhong. Pecam: Privacy-enhanced video streaming and analytics via securely-reversible transformation. In *ACM MobiCom*, October 2021.
- [71] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick. Detectron2. <https://github.com/facebookresearch/detectron2>, 2019.
- [72] X. Yu, K. Chinomi, T. Koshimizu, N. Nitta, Y. Ito, and N. Babaguchi. Privacy protecting visual processing for secure video surveillance. In *2008 15th IEEE International Conference on Image Processing*, pages 1672–1675. IEEE, 2008.
- [73] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, volume 9, page 1, 2017.
- [74] X. Zhu, Y. Wang, J. Dai, L. Yuan, and Y. Wei. Flow-guided feature aggregation for video object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 408–417, 2017.

## A Relative Privacy Guarantees

### A.1 Proof

In this section, we provide a proof for Theorem 4.1. We begin with a lemma that will be helpful for the proof:

**Lemma A.1.** Consider an individual  $x$  whose appearance is bound by  $(\hat{\rho}, \hat{K})$  in front of a camera whose policy is  $(\rho, K, \epsilon)$ . For every query  $Q$  there exists  $\alpha, \beta \in \mathbb{R}$  such that  $\alpha K(1+\beta\rho) \leq \Delta_{(\rho,K)}(Q) \leq \alpha K(2+\beta\rho)$ .

*Proof.* Any PRIVID query must contain some aggregation  $agg$  as the outer-most relation, and thus we can write  $Q := \Pi_{agg}(R)$ .  $\Delta_{(\rho,K)}(Q)$  is defined in Figure 9 for five possible aggregation operators, which are each a function of  $\Delta_{(\rho,K)}(R)$  (the sensitivity of their inner relation  $R$ ).

First, we will prove these bounds are true for the inner relation  $\Delta_{(\rho,K)}(R)$  by induction on  $R$  (all rules for  $\Delta_{(\rho,K)}(R)$  given by Figure 9):

**Case (Base):**  $R := t$  When  $R$  is an intermediate PRIVID table  $t$ , its sensitivity is given directly by Equation 5.2, where  $\alpha = \max\_rows_t$  and  $\beta = 1/c$ . Note, the  $(1 + \dots)$  and  $(2 + \dots)$  in the lemma inequalities bound  $\lceil \frac{\rho}{c} \rceil$ .

**Case (Selection):**  $R := \sigma(R')$ . When  $R$  is a selection from  $R'$ ,  $\Delta_{(\rho,K)}(R) = \Delta_{(\rho,K)}(R')$ . If  $\Delta_{(\rho,K)}(R')$  is bound by the inequalities in the lemma statement, then  $\Delta_{(\rho,K)}(R)$  is too.

**Case (Projection):**  $R := \Pi(R')$ . Same as selection.

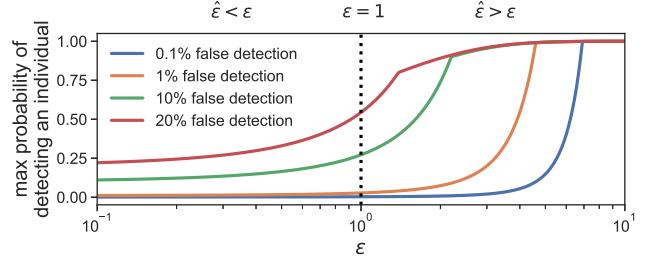
**Case (GroupBy and Join):**  $R := \gamma(R_1 \bowtie \dots \bowtie R_i)$  When  $R$  is a join of relations  $R_i$  proceeded by a GroupBy,  $\Delta_{(\rho,K)}(R) = \sum_{i=1}^N \Delta_{(\rho,K)}(R_i)$ . Let  $\Delta_{(\rho,K)}R_i$  be parameterized by  $\alpha_i$  and  $\beta_i$ . If each of  $\Delta_{(\rho,K)}(R_i)$  are bound by the inequalities in the lemma, then  $\sum_i \Delta_{(\rho,K)}(R_i)$  is as well, but with  $\alpha = \sum_{i=1}^N \alpha_i$  and  $\beta = \sum_{i=1}^N \beta_i$ .

Finally, each of the supported aggregation operators only involve multiplying  $\Delta_{(\rho,K)}(R)$  by constants (with respect to  $\rho$  and  $K$ ), and thus these constants can be subsumed into  $\alpha$ .  $\square$

We now restate Theorem 4.1 for the reader's convenience:

**Theorem A.2.** Consider a camera with a fixed policy  $(\rho, K, \epsilon)$ . If an individual  $x$ 's appearance in front of the camera is bound by some  $(\hat{\rho}, \hat{K})$ , then PRIVID effectively protects  $x$  with  $\hat{\epsilon}$ -DP, where  $\hat{\epsilon}$  is  $O(\frac{\hat{\rho}\hat{K}}{\rho K})\epsilon$ , which grows (degrades) as  $(\hat{\rho}, \hat{K})$  increase while  $(\rho, K, \epsilon)$  are fixed, and the constants do not depend on the query.

*Proof.* Recall from §5 that PRIVID uses the Laplace mechanism: it returns  $Q(V) + \eta$  to the analyst, where  $Q(V)$  is the raw query result, and  $\eta \sim \text{Laplace}(0, b)$ ,  $b = \frac{\Delta_{(\rho,K)}(Q)}{\epsilon}$  and  $\Delta_{(\rho,K)}(Q)$  is the global sensitivity of the query over any  $(\rho, K)$ -neighboring videos. Note that the sensitivity is purely a function of the query, and thus PRIVID samples noise using the same scale  $b$  regardless of how long any individual is actually visible in the video.



**Figure 8:** Plot of Equation A.4 for a few different levels of  $\alpha$ . Note that the  $x$ -axis is plotted for absolute values of  $\epsilon$  and is using a log scale. The  $y$ -axis is the maximum probability that an adversary with a given confidence level could detect whether or not  $x$  was present. If one draws a vertical line at the value of  $\epsilon$  being enforced (e.g., we mark  $\epsilon = 1$  here), the trend to the left shows how privacy is improved for individuals who are visible for less time, and the right shows how it degrades for those who are visible for more.

By Theorem B.2, this mechanism provides  $\epsilon$ -DP for all  $(\rho, K)$ -bounded events. If we rearrange the equation for  $b$  so that  $\epsilon = \frac{\Delta_{(\rho,K)}(Q)}{b}$ , we can equivalently say that PRIVID guarantees  $\frac{\Delta_{(\rho,K)}(Q)}{\epsilon}$ -DP for all  $(\rho, K)$ -bounded events. Or, more generally, that a particular instantiation of PRIVID with policy  $p = (\rho, K, \epsilon)$  guarantees  $\hat{\epsilon}$ -DP for all  $(\hat{\rho}, \hat{K})$ -bounded events in query  $Q$ , where <sup>10</sup>

$$\hat{\epsilon}_p(\hat{\rho}, \hat{K}, Q) = \frac{\Delta_{(\hat{\rho}, \hat{K})}(Q)}{b} = \frac{\Delta_{(\hat{\rho}, \hat{K})}(Q)}{\Delta_{(\rho, K)}(Q)/\epsilon} = \frac{\Delta_{(\hat{\rho}, \hat{K})}(Q)}{\Delta_{(\rho, K)}(Q)}$$

In other words, for a fixed policy,  $\hat{\epsilon}$  defines the effective level of protection provided to an event as a function of the event's (not policy's)  $(\hat{\rho}, \hat{K})$  bound.

From Lemma A.1, we can bound  $\hat{\epsilon}$  as  $\frac{\alpha\hat{K}(1+\beta\hat{\rho})}{\alpha K(2+\beta\rho)}\epsilon \leq \hat{\epsilon} \leq \frac{\alpha\hat{K}(2+\beta\hat{\rho})}{\alpha K(1+\beta\rho)}\epsilon$ . To see where this comes from, note that  $\hat{\epsilon}$  is minimized when the numerator is minimized (the lower bound from Lemma A.1) and the denominator is maximized (the upper bound from Lemma A.1). The same logic applies to the upper bound on  $\hat{\epsilon}$ .

We can simplify both bounds by first canceling  $\alpha$  and then picking units of time such that  $\beta = 1$  ( $\beta$  has dimensions of chunks per unit time). Thus,

$$\hat{\epsilon} \approx \frac{\hat{\rho}\hat{K}}{\rho K}\epsilon \quad (\text{A.1})$$

□

### A.2 Degradation of Privacy

Although  $\hat{\epsilon}$  provides a way to quantify the level of privacy provided to each individual, it can be difficult to reason about relative values of  $\epsilon$  and what they ultimately mean for privacy in practice. We can use the framework of binary hypothesis testing to develop a more intuitive understanding and ultimately visualize the degradation of privacy as a function of  $\hat{\epsilon}$  relative to  $\epsilon$ .

<sup>10</sup>Note the difference in subscript in the numerator and denominator.

Consider an adversary who wishes to determine whether or not some individual  $x$  appeared in a given video  $V$ . They submit a query  $Q$  to the system, and observe only the final result,  $A$ , which PRIVID computed as  $A = Q(V) + \eta$ , where  $\eta$  is a sample of Laplace noise as defined in the previous section. Based on this value, the adversary must distinguish between one of two hypotheses:

$$\mathcal{H}_0: x \text{ does not appear in } V$$

$$\mathcal{H}_1: x \text{ appears in } V$$

We write the false positive  $P_{FP}$  and false negative  $P_{FN}$  probabilities as:

$$P_{FP} = \mathbb{P}(x \in V | \mathcal{H}_0)$$

$$P_{FN} = \mathbb{P}(x \notin V | \mathcal{H}_1)$$

From Kairouz [46, Theorem 2.1], if an algorithm guarantees  $\epsilon$ -differential privacy ( $\delta = 0$ ), then these probabilities are related as follows:

$$P_{FP} + e^\epsilon P_{FN} \geq 1 \quad (\text{A.2})$$

$$P_{FN} + e^\epsilon P_{FP} \geq 1 \quad (\text{A.3})$$

Suppose the adversary is willing to accept a false positive threshold of  $P_{FP} \leq \alpha$ . In other words, they will only accept  $\mathcal{H}_1$  ( $x$  is present) if there is less than  $\alpha$  probability that  $x$  is not actually present.

Rearranging equations A.2 and A.3 in terms of the probability of correctly detecting  $x$  is present ( $1 - P_{FN}$ ), we have:

$$\begin{aligned} 1 - P_{FN} &\leq e^\epsilon P_{FP} \leq e^\epsilon \alpha \\ 1 - P_{FN} &\leq e^{-\epsilon} (P_{FP} - (1 - e^\epsilon)) \leq e^{-\epsilon} (\alpha - (1 - e^\epsilon)) \end{aligned}$$

Then, for a given threshold  $\alpha$ , the probability that the adversary *correctly* decides  $x$  is present is *at most* the minimum of these:

$$\mathbb{P}(x \in V | \mathcal{H}_1) \leq \min\{e^\epsilon \alpha, e^{-\epsilon} (\alpha - (1 - e^\epsilon))\} \quad (\text{A.4})$$

In Fig. 8, we visualize A.4 as a function of  $\epsilon$  for 4 different adversarial confidence levels ( $\alpha = 0.1\%, 1\%, 10\%, 20\%$ ). As an example of how to read this graph, suppose PRIVID uses a  $(\rho = 60s, K = 1, \epsilon = 1)$  policy ( $\epsilon = 1$  marked with the dotted line). An individual who appears 3 times for  $< 60s$  each is  $(\rho = 60s, K = 3)$ -bounded, and thus has an effective  $\hat{\epsilon} = 3$  relative to the actual policy for most queries (Eq. A.1). If an adversary has a  $\alpha = 1\%$  confidence level, then they would have at most a  $\sim 20\%$  chance of correctly detecting the individual appeared, even though they appeared for far more than the policy allowed. We can also see that, for sufficiently small values of  $\epsilon$  (e.g.,  $\epsilon < 1$ ), even if the adversary has a very liberal confidence level (say, 20%), a marginal increase in  $\hat{\epsilon}$  relative to  $\epsilon$  only gives the adversary a marginally larger probability of detection than they would have had otherwise.

An important takeaway is that, when an individual exceeds the  $(\rho, K)$  bound protected by PRIVID, their presence is not immediately revealed. Rather, as it exceeds the bound further,  $\hat{\epsilon}$  increases, and it becomes more likely an adversary could detect the event.

## B PRIVID Sensitivity Definition

Figure 9 provides the complete definition of sensitivity for a PRIVID query.

**Lemma B.1.** Given a relation  $R$ , the rules in Figure 9 are an upper bound on the global sensitivity of a  $(\rho, K)$ -bounded event in an intermediate table  $t$ .

*Proof.* Proof by induction on the structure of the query.

**Case:**  $t: \Delta_p(t)$  is given directly by Equation 5.2.

**Case:**  $R' := \sigma_\theta(R)$ . A selection may remove some rows from  $R$ , but it does not add any, or modify any existing ones, so in the worst case an individual can be in just as many rows in  $R'$  as in  $R$  and thus  $\Delta_p(R') \leq \Delta_p(R)$  and the constraints remain the same. If  $\theta$  includes a `LIMIT = x` condition, then  $R'$  will contain at most  $x$  rows, regardless of the number of rows in  $R$ .

**Case:**  $R' := \Pi_{a, \dots}(R)$ . A projection never changes the number of rows, nor does it allow the data in one row to influence another row, so in the worst case an individual can be in at most the same number of rows in  $R'$  as in  $R$  ( $\Delta_p(R') \leq \Delta_p(R)$ ) and the size constraint  $\tilde{C}_s(R)$  remains the same. If the projection transforms an attribute by applying a stateless function  $f$  to it, then we can no longer make assumptions about the range of values in  $a$  ( $\tilde{C}_r(R', a) = \emptyset$ ), but nothing else changes because the stateless nature of the function ensures that data in row cannot influence any others.

**Case: GroupBy.** A GROUP BY over a fixed set of  $n$  keys is equivalent to  $n$  separate queries that use the same aggregation function over a  $\sigma_{\text{WHERE} col = key}(R)$ . If the column being grouped is a user-defined column, PRIVID requires that the analyst provide the keys directly. If the column being grouped is one of the two implicit columns (chunk or region), then the set of keys is not dependent on the contents of the data (only its length) and thus are fixed regardless.

**Case: Join.** Consider a query that computes the size of the intersection between two cameras, PROCESS'd into intermediate tables  $t_1$  and  $t_2$  respectively. If  $\Delta(t_1) = x$  and  $\Delta(t_2) = y$ , it is tempting to assume  $\Delta(t_1 \cap t_2) = \min(x, y)$ , because a value needs to appear in both  $t_1$  and  $t_2$  to appear in the intersection. However, because the analyst's executable can populate the table arbitrarily, they can "prime"  $t_1$  with values that would only appear in  $t_2$ , and vice versa. As a result, a value need only appear in either  $t_1$  or  $t_2$  to show up in the intersection, and thus  $\Delta(t_1 \cap t_2) = x + y$  (the sum of the sensitivities of the tables).  $\square$

**Theorem B.2.** Consider an adaptive sequence (§2.3) of  $n$  queries  $Q_1, \dots, Q_n$ , each over the same camera  $C$ , a privacy policy  $(\rho_C, K_C)$ , and global budget  $\epsilon_C$ . PRIVID (Algorithm 1) provides  $(\rho_C, K_C, \epsilon_C)$ -privacy for all  $Q_1, \dots, Q_n$ .

*Proof.* Consider two queries  $Q_1$  (over time interval  $I_1$ , using chunk size  $c_1$  and budget  $\epsilon_1$ ) and  $Q_2$  (over  $I_2$ , using  $c_2$  and  $\epsilon_2$ ). Let  $v_1 = V[I_1]$  be the segment of video  $Q_1$  analyzes and  $v_2 = V[I_2]$  for  $Q_2$ . Let  $E$  be a  $(\rho, K)$ -bounded event.

$\mathcal{P}$	Privacy policy for each camera: $\{(\rho, K)_c \mid c \in \text{cameras}\}$
$\Delta_{\mathcal{P}}(R)$	Maximum number of rows in relation $R$ that could differ by the addition or removal of any $(\rho, K)$ -bounded event.
$\tilde{\mathcal{C}}_r(R, a)$	Range constraint: range of attribute $a$ in $R$
$\tilde{\mathcal{C}}_s(R)$	Size constraint: upper bound on total number of rows in $R$
$\emptyset$	Indicates that a relational operator leaves a constraint unbound. If this constraint is required for the aggregation, it must be bound by a predecessor. If it is not required, it can be left unbound.

NOTATION

Function	Definition	Constraints	Sensitivity( $\Delta(Q)$ )
Count	$Q := \Pi_{\text{count}}(*) (R)$	$\Delta$	$1 \cdot \Delta(R)$
Sum	$Q := \Pi_{\text{sum}}(a) (R)$	$\Delta, \tilde{\mathcal{C}}_r$	$\Delta(R) \cdot \tilde{\mathcal{C}}_r(R, a)$
Average	$Q := \Pi_{\text{avg}}(a) (R)$	$\Delta, \tilde{\mathcal{C}}_r, \tilde{\mathcal{C}}_s$	$\frac{\Delta(R) \cdot \tilde{\mathcal{C}}_r(R, a)}{\tilde{\mathcal{C}}_s(R)}$
Std. Dev	$Q := \Pi_{\text{stddev}}(a) (R)$	$\Delta, \tilde{\mathcal{C}}_r, \tilde{\mathcal{C}}_s$	$\Delta(R) \cdot \tilde{\mathcal{C}}_r(R, a) / \sqrt{\tilde{\mathcal{C}}_s(R)}$
Argmax	$Q := \Pi_{\text{argmax}}(a) (R)$	$\Delta, a \in K$	$\max_{k \in K} \Delta(\sigma_{a=k}(R))$

RELATIONAL OPERATORS

Operator	Type	Definition	$\Delta_{\mathcal{P}}(R')$	$\tilde{\mathcal{C}}_r(R', a_i)$	$\tilde{\mathcal{C}}_s(R')$
Base Case	Base Table	$R$	$m_r \cdot K \cdot (1 + \lceil \frac{\rho}{c} \rceil)$	$\emptyset$	$\emptyset$
Selection ( $\sigma$ )	Standard selection: rows from $R$ that match WHERE condition	$R' := \sigma_{\text{WHERE}(\dots)}(R)$	$\Delta_{\mathcal{P}}(R)$	$\tilde{\mathcal{C}}_r(R, a_i)$	$\tilde{\mathcal{C}}_s(R)$
	Limit: first $x$ rows from $R$	$R' := \sigma_{\text{LIMIT}=x}(R)$	$\Delta_{\mathcal{P}}(R)$	$\tilde{\mathcal{C}}_r(R, a_i)$	$\min(x, \tilde{\mathcal{C}}_s(R))$
Projection ( $\Pi$ )	Standard projection: select attributes $a_i, \dots$ from $R$	$R' := \Pi_{a_i, \dots}$	$\Delta_{\mathcal{P}}(R)$	$\tilde{\mathcal{C}}_r(R, a_i)$	$\tilde{\mathcal{C}}_s(R)$
	Apply (user-provided, but stateless) $f$ to column $a_i$	$R' := \Pi_f(a_i), \dots$	$\Delta_{\mathcal{P}}(R)$	$\emptyset$	$\tilde{\mathcal{C}}_s(R)$
	Add range constraint to column $a_i$	$R' := \Pi_{a_i \in [l_i, u_i]}, \dots$	$\Delta_{\mathcal{P}}(R)$	$[l_i, u_i]$ if $a_i \neq \emptyset$ $\tilde{\mathcal{C}}_r(R, a_i)$ otherwise	$\tilde{\mathcal{C}}_s(R)$
GroupBy ( $\gamma$ )	Group attribute(s) ( $g_i$ ) are chunk (or binned chunk) or region	$R' := g_j, \dots, \gamma_{\text{agg}}(a_i), \dots$ $g_j := \text{chunk} \text{bin}(\text{chunk})$	Equation 5.2	$\Delta(\text{agg}(a_i))$	$\frac{\tilde{\mathcal{C}}_s(R)}{(\text{bin size})}$
	Group attribute(s) ( $g_j$ ) are not chunk or region	$R' := g_j, \dots, \gamma_{\text{agg}}(a_i), \dots$	$\Delta_{\mathcal{P}}(R)$	$\emptyset$	$\emptyset$
	... discrete set of keys provided for each group (constrains size)	$R' := g_j \in K_j, \dots, \gamma_{\text{agg}}(a_i), \dots$	...	...	$\Pi_j   K_j$
	... aggregation constrains range: $a \in \text{agg}(a_i) \in [l_i, u_i]$	$R' := g_j, \dots, \gamma_{\text{agg}}(a_i) \in [l_i, u_i], \dots$	...	$[l_i, u_i]$ if $a_i \neq \emptyset$ $\tilde{\mathcal{C}}_r(R, a_i)$ otherwise	...
Joins* ( $\bowtie$ )	*When immediately preceded by GroupBy over the same key(s) ... equijoin on $g_j$ (intersection on $g_j$ ) ... outer join on $g_j$ (union on $g_j$ )	$R' := g \gamma_{\text{agg}}(a) (R_1 \bowtie g \dots \bowtie g R_n)$ $R' := g \gamma_{\text{agg}}(a) (R_1 \bowtie g \dots \bowtie g R_n)$	$\sum_{i=1}^n \Delta_{\mathcal{P}}(R_i)$	(GroupBy rules)	(GroupBy rules)

Figure 9: Full set of rules for PRIVID's sensitivity calculation.

**Case 1:  $I_1$  and  $I_2$  are not  $\rho$ -disjoint** The budget check (lines 1-3 in Algorithm 1) ensures that these two queries must draw from the same privacy budget, because their effective ranges overlap by at least one frame (but may overlap up to all frames). By Theorem 5.1, PRIVID is  $(\rho, K, \epsilon_1)$ -private for  $Q_1$  and  $(\rho, K, \epsilon_2)$ -private for  $Q_2$ . By Dwork [37, Theorem 3.14], the combination of  $Q_1$  and  $Q_2$  is  $(\rho, K, \epsilon_1 + \epsilon_2)$ -private.

**Case 2:  $I_1$  and  $I_2$  are  $\rho$ -disjoint** In other words,  $I_1 + \rho < I_2 - \rho$ , thus the budget check (lines 1-3) allows these two queries to draw from entirely separate privacy budgets. Since the intervals are  $\rho$ -disjoint, and all segments in  $E$  must have duration  $\leq \rho$ , it is not possible for the same segment to appear in even a single frame of *both* intervals.

Let  $K_1$  be the number of segments contained in  $I_1$ , each of duration  $\leq \rho$ , and  $K_2$  be the remaining segments contained in  $I_2$ , each of duration  $\leq \rho$ . In other words,  $E$  is  $(\rho, K_1)$ -bounded in  $v_1$  and  $(\rho, K_2)$ -bounded in  $v_2$ . Since  $E$  has at most  $K$  segments,  $K_1 + K_2 \leq K$ . We need to show that the probability of observing both  $A_1$  and  $A_2$  if the inputs are the actual segments  $v_1$  and  $v_2$  is close ( $e^\epsilon$ ) to the probability of observing those values if the inputs are the neighboring segments  $v'_1$  and  $v'_2$ :

$$\frac{\Pr[A_1 = Q_1(v_1), A_2 = Q_2(v_2)]}{\Pr[A_1 = Q_1(v'_1), A_2 = Q_2(v'_2)]} \leq \exp(e)$$

Since the probability of observing  $A_1$  is independent of observing  $A_2$  (randomness is purely over the noise added by PRIVID):

$$\begin{aligned} & \frac{\Pr[A_1 = Q_1(v_1), A_2 = Q_2(v_2)]}{\Pr[A_1 = Q_1(v'_1), A_2 = Q_2(v'_2)]} \\ & \leq \frac{\Pr[A_1 = Q_1(v_1)] \Pr[A_2 = Q_2(v_2)]}{\Pr[A_1 = Q_1(v'_1)] \Pr[A_2 = Q_2(v'_2)]} \\ & \leq \frac{\frac{1}{2b_1} \exp(-\frac{|A_1 - Q_1(v_1)|}{b_1}) \frac{1}{2b_2} \exp(-\frac{|A_2 - Q_2(v_2)|}{b_2})}{\frac{1}{2b_1} \exp(-\frac{|A_1 - Q_1(v_1)|}{b_1}) \frac{1}{2b_2} \exp(-\frac{|A_2 - Q_2(v'_2)|}{b_2})} \\ & \quad (\text{By Algorithm 1, Line 13}) \\ & = \exp\left(\frac{|A_1 - Q_1(v'_1)| - |A_1 - Q_1(v_1)|}{b_1} + \frac{|A_2 - Q_2(v'_2)| - |A_2 - Q_2(v_2)|}{b_2}\right) \end{aligned}$$

If  $K_1$  segments are in  $v_1$  and  $K_2$  segments are in  $v_2$ , the numerator of each fraction above is the sensitivity of a  $(\rho, K_1)$ -bounded event and a  $(\rho, K_2)$ -bounded event, respectively.  $b_1$  and  $b_2$  are the amount of noise actually added to the query, which are both based on  $K$ :

$$\begin{aligned} & \leq \exp\left(\frac{\Delta_{(\rho, K_1)}(Q_1)}{\Delta_{(\rho, K)}(Q_1)/\epsilon} + \frac{\Delta_{(\rho, K_2)}(Q_2)}{\Delta_{(\rho, K)}(Q_2)/\epsilon}\right) \\ & = \exp\left(\epsilon \cdot \left(\frac{K_1(\lceil \frac{\rho}{c_1} \rceil + 1)}{K(\lceil \frac{\rho}{c_1} \rceil + 1)} + \frac{K_2(\lceil \frac{\rho}{c_2} \rceil + 1)}{K(\lceil \frac{\rho}{c_2} \rceil + 1)}\right)\right) \\ & \quad (\text{by Equation 5.2}) \\ & = \exp\left(\epsilon \cdot \left(\frac{K_1}{K} + \frac{K_2}{K}\right)\right) \quad (\text{recall } K \geq K_1 + K_2) \\ & \leq \exp(\epsilon) \end{aligned}$$

## C Query Details

### C.1 Case 1 Query Statements

Case 1: Query 1

```
SPLIT campus
BEGIN 06-01-2019/06:00am END 06-01-2019/06:00pm
BY TIME 30sec STRIDE 0sec
BY REGION
WITH MASK C1
INTO campusChunks;
PROCESS campusChunks USING count_ppl_campus.py TIMEOUT 1sec
PRODUCING 1 ROWS
WITH SCHEMA (ppl:NUMBER=0)
INTO campusTable;
SELECT hour, sum(RANGE(ppl,0,6)) from campusTable
GROUP BY hour
CONSUMING eps=1.0;
```

Case 1: Query 2

```
SPLIT highway
BEGIN 06-01-2019/06:00am END 06-01-2019/06:00pm
BY TIME 30sec STRIDE 0sec
BY REGION
WITH MASK H2
INTO highwayChunks;
PROCESS highwayChunks USING count_cars.py TIMEOUT 1sec
PRODUCING 1 ROWS
WITH SCHEMA (cars:NUMBER=0)
INTO highwayTable;
SELECT hour, sum(RANGE(cars,0,100)) from highwayTable
GROUP BY hour
CONSUMING eps=1.0;
```

Case 1: Query 3

```
SPLIT urban
BEGIN 06-01-2019/06:00am END 06-01-2019/06:00pm
BY TIME 30sec STRIDE 0sec
BY REGION
WITH MASK U2
INTO urbanChunks;
PROCESS campusChunks USING count_ppl_urban.py TIMEOUT 1sec
PRODUCING 1 ROWS
WITH SCHEMA (ppl:NUMBER=0)
INTO campusTable;
SELECT hour, sum(RANGE(ppl,0,23)) from campusTable
GROUP BY hour
CONSUMING eps=1.0;
```

### C.2 Case 2: Complex Sensitivity Example

The code block for Case 2 contains Queries 4-6, which are computed over the same set of intermediate tables.

To demonstrate the sensitivity computation for a complex PRIVID query, we focus on Query 4. This query aims to estimate the typical working hours of taxis in the city of Porto, Portugal; it first computes the difference between the first and last time each taxi (identified by plate) was seen (by either camera 10 or 27) on a given day, then averages across all taxis and days (over a year).

In order to ensure all variables needed for the aggregation are properly constrained, we make two assumptions: most taxis will not work more than 16 hours in a day, and there are roughly 300 public taxis in Porto (based on public information). We can express this query in relational algebra as follows:

$$\Pi_{\text{Avg}(\text{hrs})}(\sigma_{\text{limit}(\text{plates})=300}(\text{plate}, \text{day} \gamma \text{range}(\text{chunks}) \in [0, 16] (t_1 \cup t_2)))$$

Case 2: Queries 4-6

```
-- Repeat for portoCam1...portoCam127:
SPLIT portoCam1
BEGIN 07-01-2013/12:00am END 07-01-2014/12:00am
BY TIME 15sec STRIDE 0sec
INTO chunks1;
-- Repeat for chunks1...chunks127:
PROCESS chunks1 USING porto.py TIMEOUT 1sec
PRODUCING 3 ROWS
WITH SCHEMA (plate:STRING="")
INTO table1;

-- Query 4: Average Taxi Working Hours
SELECT avg(avg.shift) FROM
(SELECT plate,avg(RANGE(shift, [0,16])) FROM
(SELECT plate,day,(max(chunk)-min(chunk)) as shift) FROM
table10 UNION table27 GROUP BY plate,day(chunk))
GROUP BY plate LIMIT 300
CONSUMING eps=0.33;
-- Query 5: # Taxis Traversing Both Locations On Same Day
SELECT day,count(DISTINCT plate) FROM
(SELECT day,plate FROM
table10 INNER JOIN table27 ON
(table10.plate=table27.plate AND table10.day=table27.day)
)
GROUP BY day
CONSUMING eps=0.33;
-- Query 6: Camera with highest daily traffic
SELECT argmax(arg=cam, target=avg_daily) FROM
(SELECT "cam1" as cam, avg(daily) as avg_daily FROM
(SELECT day,count(DISTINCT plate) as daily FROM
table1 GROUP BY day))
UNION
// ...
UNION
(SELECT "cam127" as cam, avg(daily) as avg_daily FROM ...)
CONSUMING eps=0.33;
```

We use the policy  $\mathcal{P} = \{(\rho = 45s, K = 1)_{c_1}, (195s, 1)_{c_2}\}$  (the max observed persistence over historical data for each camera) and an  $\epsilon$  of 1.

First, we compute the base sensitivity of each table. The SPLIT statement specifies the video will be split into 15 second chunks with 0 stride, and that each chunk will produce a maximum of 3 rows. With this we can compute:  $\Delta_{\mathcal{P}}(t_1) = \lceil \frac{(45*fps-1)}{15*fps} \rceil + 1 = 4 \cdot 3 = 12$  and  $\Delta_{\mathcal{P}}(t_2) = \lceil \frac{195*fps-1}{15*fps} \rceil + 1 = 14 \cdot 3 = 42$ . When we combine them with a union, their sensitivities add:  $\Delta_{\mathcal{P}}(t_1 \cup t_2) = 12 + 42 = 54$ . The GROUP BY creates a new table with a row per plate per day, and constrains the range of the aggregate value shift to  $[0, 16]$  (range( $a, b$ ) returns  $|b - a|$ , i.e., the time between the first and last appearance of a taxi on a given day), but we don't know how many unique plates there might be, so the size  $\tilde{C}_s(\gamma(\dots))$  is unconstrained. We add  $\sigma_{\text{limit}}$  to manually enforce a maximum of 300 plates per day, which gives us a constraint  $\tilde{C}_s(\sigma(\dots)) = 300 \text{plates} * 365 \text{days} = 109,500$ . We now have all the constraints necessary to compute the sensitivity of the average aggregation:  $\Delta_{\mathcal{P}}^{\text{AVG}}(R) = \frac{\Delta_{\mathcal{P}}(R)\tilde{C}_s(R, \text{shift})}{\tilde{C}_s(R)} = \frac{54 \cdot 16}{109,500} = 0.0079$ . Since PRIVID uses the Laplace mechanism to add noise, we can use the inverse CDF of the Laplace distribution to bound the expected error based on  $\Delta$  with a given confidence level. For example,  $\mathcal{L}^{-1}(p = 0.999, u = 0, b = \frac{\Delta}{\epsilon} = \frac{0.0079}{0.33}) \leq 0.15$  hours with 99.9% confidence.

# Spectrum: High-bandwidth Anonymous Broadcast

Zachary Newman  
*MIT CSAIL*  
zjn@mit.edu

Sacha Servan-Schreiber  
*MIT CSAIL*  
3s@mit.edu

Srinivas Devadas  
*MIT CSAIL*  
devadas@csail.mit.edu

## Abstract

We present Spectrum, a high-bandwidth, metadata-private file broadcasting system. In Spectrum, a small number of broadcasters share a file with many subscribers via two or more non-colluding broadcast servers. Subscribers generate cover traffic by sending dummy files, hiding which users are broadcasters and which users are only consumers.

Spectrum optimizes for a setting with few broadcasters and many subscribers—as is common to many real-world applications—to drastically improve throughput over prior work. Malicious clients are prevented from disrupting broadcasts using a novel blind access control technique that allows servers to reject malformed requests. Spectrum also prevents deanonymization of broadcasters by malicious servers deviating from protocol. Our techniques for providing malicious security are applicable to other systems for anonymous broadcast and may be of independent interest.

We implement and evaluate Spectrum. Compared to the state-of-the-art in cryptographic anonymous communication systems, Spectrum’s peak throughput is 4–120,000× faster (and commensurately cheaper) in a broadcast setting. Deployed on two commodity servers, Spectrum allows broadcasters to share 1 GB (two full-length 720p documentary movies) in 13h 20m with an anonymity set of 10,000 (for a total cost of about \$6.84). These costs scale roughly linearly in the size of the file and total number of users, and Spectrum parallelizes trivially with more hardware.

## 1 Introduction

An informed public often depends on whistleblowers, who expose misdeeds and corruption. Over the last century, whistleblowers have exposed financial crimes, government corruption [61, 69, 75], risks to public health [43, 52], Russian interference in the 2016 U.S. presidential election [61, 70], presidential misconduct [17, 45, 67, 79], war and human rights crimes [5, 38, 87], and digital mass surveillance by U.S. government agencies [18]. Philosophers debate whistleblowing ethics [3, 35], but agree it often has positive effects.

**Motivation for this work.** Whistleblowers take on great personal risks in bringing misdeeds to light. The luckiest enjoy legal protections [88] or financial reward [89]. But many face exile [18], incarceration [50, 70, 74], or risk their lives [87]. More recently, political activist Alexei Navalny was detained and sentenced to prison following the release of documents accusing Russian president Vladimir Putin of corruption and embezzlement [80].

To mitigate these risks, many whistleblowers turn to technology to protect themselves [47]. Secure messaging apps Signal [26] and SecureDrop [8] have proven to be an important resource to whistleblowers and journalists [44, 84]. Encryption does its job, even against the NSA [92]—but it may not be enough to protect from powerful adversaries.

Since the Snowden revelations, governments and the press have focused on *metadata*. The source, destination, and timing of encrypted data can leak information about its contents. For instance, prosecutors used SFTP metadata in the case against Chelsea Manning [96]. Newer technology is still vulnerable: a federal judge found Natalie Edwards guilty on the evidence of metadata from an encrypted messaging app [50]. To protect whistleblowers and protect against powerful adversaries (e.g., corrupted internet service providers), systems must be designed with metadata privacy in mind.

Many academic and practical metadata-hiding systems provide solutions to this problem for some applications. Tor [37] boasts a distributed network of 6,000 nodes and 2 million daily active users (the only such system with wide usage). Tor is fast enough for web browsing, but deanonymization attacks identify users with a high success rate based on observed traffic [9, 14, 42, 48, 53, 65, 68]. Moreover, the effectiveness of deanonymization attacks increases with the size of the traffic pattern. Whistleblowers using Tor to upload large files can be more easily deanonymized for this reason.

Some recent academic research systems [2, 30, 41, 54–56, 58, 86, 90] address the problem of hiding metadata in anonymous communication, providing precise security guarantees for both direct messaging and “Twitter”-like broadcast applications. However, a limitation of all existing systems is

that they are designed for low-bandwidth content, incurring impractical latencies with large messages (see Section 8).

**Contributions.** Motivated by the lack of anonymous broadcast systems suitable for high-throughput data dumps, we design and build Spectrum: a system for high-bandwidth metadata-private anonymous broadcasting. Spectrum is the first anonymous broadcast system supporting high-bandwidth, many-user settings. It optimizes for the many-subscriber and few-broadcaster setting, which reflects the real-world usage of broadcast platforms. Per-request, Spectrum’s server-side processing costs grow with the number of broadcasters rather than the total number of users, significantly improving performance over prior work when only a subset are broadcasting.

This paper contributes:

1. Design and security analysis of Spectrum, a system for high-bandwidth broadcasting with strong robustness and privacy guarantees against malicious adversaries,
2. A notion of blind access control for anonymous communication, along with a construction and a black-box transformation to efficiently support large (1 GB) messages,
3. Identification of an “audit attack” that allows malicious servers to deanonymize users, and BlameGame, a black-box blame protocol to “upgrade” Spectrum and similar systems to defend against this attack.
4. An open-source implementation of Spectrum, evaluated in comparison to other anonymous communication systems. We show that Spectrum supports high-bandwidth, latency-sensitive applications such as real-time anonymous podcasting, video streaming, and large file leaks.

**Limitations.** Like other metadata-private systems, Spectrum provides anonymity among honest online users and requires all users to contribute cover messages to a broadcast (to perfectly hide network metadata). Additionally, Spectrum achieves peak performance with exactly two servers (similarly to Riposte and Express [30, 41]). Instantiating with more than two servers requires using a less (concretely) efficient cryptographic primitive: a seed-homomorphic pseudorandom generator [12]. Finally, Spectrum requires a one-time “bootstrapping” process at setup time (similar to other systems [4, 29, 41, 58, 90, 95]); see Section 2.3.

## 2 Anonymous broadcast

In this section, we describe anonymous broadcast and its challenges, along with our system design and techniques.

**Setting and terminology.** In anonymous broadcast, one or more users/clients (*broadcasters*) share a *message* (e.g., file) while preventing network observers from learning its source. In Spectrum, passive users generate cover traffic (dummy messages) to increase the size of the *anonymity set* (the set of users who could have plausibly sent the broadcast message).

These passive users are *subscribers*, consuming broadcasts. Because the message sources are anonymous, the servers publish distinct messages to different *channels* or slots. Every broadcaster has exactly one channel, which they anonymously publish to in every iteration of the protocol. The servers cannot distinguish between a subscriber sending cover traffic and a broadcaster writing to a channel.

The primary challenge in anonymous broadcasting is preventing *disruption* by malicious clients: in simple broadcasting systems, users can clobber other users’ messages via undetectable deviations from the protocol [2, 30, 41]. We first begin by explaining the standard building-block for achieving anonymous broadcasting [2, 30]. In subsequent sections, we build off of this basic scheme to achieve disruption resistance.

### 2.1 DC-nets

Chaum [23] presents DC-nets, which enable a rudimentary form of anonymous broadcast. DC-nets use secret-sharing to obscure the source of data in the network. Like prior work [2, 30, 41, 95], we instantiate a DC-net with two or more servers and many clients. One client (the broadcaster) wishes to share a file; all other clients (subscribers) provide cover traffic. In a two-server DC-net, the  $i$ th client samples a random bit string  $r_i$  and sends  $r_i \oplus m_i$  to ServerA and  $r_i$  to ServerB. Servers can recover  $m_i$  by combining their respective shares:

$$m_i = (m_i \oplus r_i) \oplus (r_i).$$

If exactly one of  $N$  clients shares a message  $m_i = \hat{m}$  while all other clients share  $m_i = 0$ , the servers can recover  $\hat{m}$  (without learning which client sent  $m_i = \hat{m}$ ) by first locally aggregating all received shares as  $\text{agg}_A = \bigoplus_i (r_i \oplus m_i)$  and  $\text{agg}_B = \bigoplus_i r_i$  and then revealing the aggregation to the other server.

Because all subscribers send shares of zero, combining the local aggregations recovers the broadcaster’s message:

$$\hat{m} = \text{agg}_A \oplus \text{agg}_B.$$

The above scheme protects client anonymity, as each server sees a uniformly random share from each client.

**DC-net challenges.** While DC-nets allow fast anonymous broadcast, users can undetectably *disrupt* the broadcast by sending non-zero shares. Preventing such disruptions is a major challenge and primary source of latency in prior DC-net-based systems [2, 29, 30, 41, 54, 55, 95] (see related work; Section 8). Also, while DC-nets enable one broadcaster to transmit a message, many clients may wish to broadcast. Repeating the protocol in parallel is inefficient, requiring bandwidth linear in the number of broadcasters. Even prior works which overcome the linear (in the number of broadcasters) bandwidth overhead of naïve protocol repetition suffer from linear server-side work per client, regardless of whether or not all clients are broadcasters. In Spectrum, the bandwidth

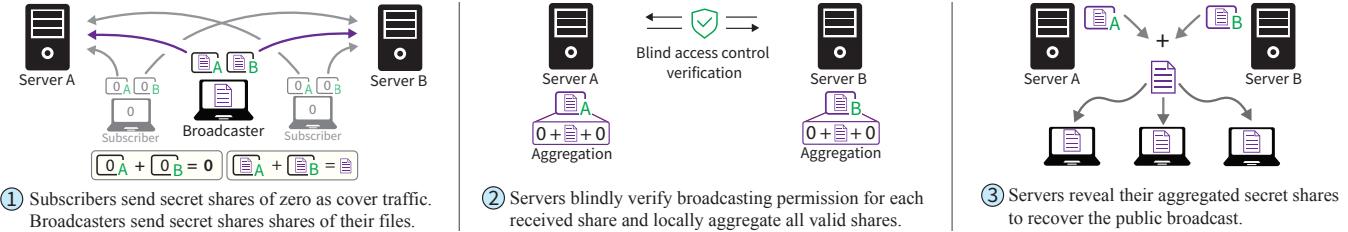


Figure 1: High-level overview of Spectrum when instantiated with two servers (and one broadcaster).

overhead grows *logarithmically* in the number of broadcasters. Additionally, the server-side work only grows linearly in the number of broadcasters, rather than the total number of clients. We compare this work and other DC-net-based anonymous broadcasting systems in Table 1.

## 2.2 Main ideas in realizing Spectrum

Spectrum builds on top of DC-nets, improving efficiency and preventing disruption by malicious clients.

**Practical efficiency.** Spectrum capitalizes on the asymmetry of real-world broadcasting: there are typically fewer broadcasters than there are subscribers. While some prior works repeat many executions of the DC-net protocol more efficiently than the naïve scheme, they still reserve space (i.e., channels) for *every* client [2, 30]. As a consequence, the per-request computation on each server is linear in the number of clients, leading to high latency and “wasted” work. Spectrum derives anonymity from *all* clients, but only the total number of *broadcasters* influences the per-request work on each server (rather than the total number of clients in the system).

**Preventing disruption.** In Spectrum, we prevent broadcast disruption by developing a new idea: *anonymous* access control (Section 3.1), which we realize from the Carter-Wegman MAC [94]. We check access to each “channel” to ensure that only a user with a “broadcast key” can write to that channel.

**Preventing “audit” attacks.** Anonymous broadcast servers can *covertly* exclude a client in order to deanonymize the corresponding user. While vanilla DC-nets do not have this problem, prior anonymous broadcast systems leave out a client’s share if they are found to be ill-formed. This is done to defend against disruption. However, it also makes it possible for a malicious server to exclude a user by framing them as malicious. In the broadcast setting, excluding a user can effectively deanonymize them. Abraham et al. [2] make the same observation and defend against the attack by requiring an honest-majority out of five or more servers. In Dissent [29], deanonymization is prevented with an expensive, after-the-fact blame protocol. Other systems [30, 41] are vulnerable to this attack (see Appendix A for details). Spectrum is the first system to efficiently defend against this attack while still preventing disruption per request (rather than assigning blame

after-the-fact). We achieve this by introducing BlameGame (Section 4.3), a lightweight blame protocol which can also be applied to other systems (e.g., Riposte [30] and Express [41]).

## 2.3 System overview

Spectrum is built using two or more broadcast servers (only one must be honest to guarantee anonymity; see Section 2.4) and many clients consisting of broadcasters and subscribers. One or more broadcaster(s) wish to share a message (as in the DC-net example). The subscribers generate cover traffic to increase the anonymity set. Each broadcaster has a private *channel*—or slot—for their message. Subscribers do not have channels. At the end of each round, Spectrum publishes the contents of each channel, hiding which client wrote to which channel (if any). Spectrum has three phases.

**Setup.** During setup, all broadcasters register with the servers. All users perform a setup-free anonymous broadcast protocol to establish a channel in Spectrum. Specifically, each broadcaster shares a public authentication key with the servers, which will be used to enforce anonymous access to write to a channel. At the end of the setup phase, the servers publish all parameters, including the number of channels and the maximum size of each broadcast message per round.

**Main protocol.** The protocol proceeds in one or more rounds (overview in Figure 1; details in Section 4.2). In each round, every client sends request shares to each server. The broadcasters send shares of their messages while the subscribers send empty shares. To enforce access control, the servers perform an efficient audit over the received shares: they obliviously check that each writer to a channel knows the secret channel broadcast key, *or* their message is zero. If the message shares pass the audit, the servers aggregate them as in a DC-net (Section 2.1). Otherwise, the servers perform a blame protocol (see BlameGame, summarized below). Finally, the servers combine aggregated shares to recover the messages.

**BlameGame.** If any client’s request fails the audit, the servers perform BlameGame, a simple blame protocol (detailed in Section 4.3). BlameGame determines whether a client failed the access control check or if a server tampered with the client request in an attempt to frame a client as malicious. If the client is blamed, the servers drop the client’s request and

proceed with the main Spectrum protocol. Otherwise, if a server is blamed, the honest server(s) abort. This protocol is much faster than fully aggregating a client’s request, so a malicious client cannot use this to cause significant delays.

## 2.4 Threat model and security guarantees

Spectrum is instantiated with two (or more) broadcast servers and many clients (broadcasters and subscribers). Clients send shares of a message to the servers for aggregation.

### Threat model

- No client is trusted by any honest server.
- Clients may deviate from the protocol, collude with other clients, or collude with a subset of malicious servers.
- At least one server must be honest to guarantee anonymity for clients (it does not matter which server is honest).
- Any subset of servers may deviate from the protocol and collude with malicious clients or the network adversary.

**Assumptions.** We make black-box use of public key infrastructure (e.g., TLS [73]) to encrypt data between clients and servers. We make the following cryptographic assumptions: (1) the hardness of the discrete logarithm problem [11], (2) the hardness of the decision Diffie-Hellman problem [10, 40] (when instantiated with more than two servers), and (3) the existence of hash functions and pseudorandom generators. We also assume a setup-free anonymous broadcast system [2, 30, 55, 95] for bootstrapping. As with prior work [2, 29, 30, 41, 55], we assume all communication between parties is observed by the network adversary.

**Guarantees.** Under the above threat model and assumptions, we obtain the following guarantees.

- *Anonymity.* An adversary controlling the network and a strict subset of servers and clients cannot distinguish between honest clients: broadcasters and subscribers are cryptographically indistinguishable in Spectrum. That is, no adversary observing the network and controlling a subset of servers and clients can distinguish between an honest subscriber and an honest broadcaster.
- *Availability.* If all servers follow the protocol, the system remains available (even if many clients are malicious). If any server halts or deviates from the protocol, then availability is not guaranteed and the protocol may abort.

**Non-goals.** We do not protect against denial-of-service attacks by a large number of clients (but we note that standard techniques, such as CAPTCHA [91], anonymous one-time-use tokens [33], or proof-of-work [39, 51] apply). Like all anonymous broadcast systems, intersection attacks on participation in the protocol can identify users, so Spectrum requires that users stay online for the duration of the protocol.

## 3 Spectrum with one channel

In this section, we introduce Spectrum with a single broadcaster (and therefore a single channel), two servers, and many subscribers. Figure 1 depicts an example. This setup mirrors the simplest DC-net protocol of Section 2.1. In Section 4, we extend Spectrum to many broadcasters and many servers.

### 3.1 Preventing disruption

We denote by  $\mathbb{F}$  any finite field of prime order (e.g., integers mod  $p$ ). We assume that all messages are elements in  $\mathbb{F}$ . (Section 5.1 shows how to efficiently support large binary messages in  $\mathbb{F}_{2^{\ell}}$ .) Each server receives secret-shares of a message  $m_i$ , where  $m_i = 0 \in \mathbb{F}$  for subscribers and  $m_i = \hat{m} \in \mathbb{F}$  for the broadcaster. To prevent disruption, we enforce the following rule: for each channel, the broadcaster (with knowledge of a pre-established broadcast key) can send a non-zero message; all subscribers (who do not have the broadcast key) can only share a zero message. We give a new technique enabling the servers to verify the rule efficiently *without* learning anything except for the validity of the provided secret-shares.

**New tool: anonymous access control.** We adapt the Carter-Wegman MAC [21, 94] to provide a secret-shared “access proof” accompanying the message shares. Each client sends a secret-shared proof that it is either: (1) sending a share of a broadcast message with knowledge of the broadcast key; or (2) sending a cover message (i.e.,  $m_i = 0$ ) that does not affect the final aggregate computed by the servers.

**Carter-Wegman MAC.** Let  $\mathbb{F}$  be any finite field of sufficiently large size for security. Sample a random authentication key  $(\alpha, \gamma) \in \mathbb{F} \times \mathbb{F}$  and define  $\text{MAC}_{(\alpha, \gamma)}(m) = \alpha \cdot m + \gamma \in \mathbb{F}$ . Observe that  $\text{MAC}_{(\alpha, \gamma)}$  is a *linear function* of the message, which makes it possible to verify a *secret-shared tag* for a *secret-shared message*. We demonstrate this with two servers ServerA and ServerB. Let  $t = \text{MAC}_{(\alpha, \gamma)}(m)$ . If  $m$  is additively secret-shared as  $m = m_A + m_B \in \mathbb{F}$ , and  $t$  is secret shared as  $t = t_A + t_B \in \mathbb{F}$ , the servers (knowing  $\alpha$  and  $\gamma$ ) can verify that the tag corresponds to the secret-shared message:

- ServerA computes  $\beta_A \leftarrow (\alpha \cdot m_A - t_A) \in \mathbb{F}$ .
- ServerB computes  $\beta_B \leftarrow (\alpha \cdot m_B - t_B) \in \mathbb{F}$ .
- Servers swap  $\beta_A$  and  $\beta_B$  and check if  $\beta_A + \beta_B = \gamma \in \mathbb{F}$ .

The final condition only holds for a valid tag. Neither server learns anything about the message  $m$  in the process (apart from the tag validity) since both the message and tag remain secret-shared between servers.

If both the servers and the broadcaster know the key  $(\alpha, \gamma)$ , the broadcaster can compute a tag  $t$  which the servers can check for correctness as above. However, there are two immediate problems to resolve. First, subscribers cannot generate valid tags on zero messages without knowledge of  $(\alpha, \gamma)$ . Second, an honest-but-curious (or compromised) server can share  $(\alpha, \gamma)$  with a malicious client who can then covertly

	<b>Request Size</b>	<b>Audit Size</b>	<b>Audit Rounds</b>	<b>Server Work</b>	<b>Malicious Security</b>	<b>Disruption Handling</b>	<b>Blame Protocol</b>	<b>Comments</b>
Blinder [2]	$ m  \cdot \sqrt{N}$	$\lambda \cdot  m $	$\log N$	$N \cdot  m $	✓	Prevent	N/A	Requires 5+ servers and MPC
Dissent [29]	$ m  \cdot L + N$	N/A	N/A	$L \cdot  m $	✓	Detect	Expensive	Blame quadratic in $N$
PriFi [6]	$ m  \cdot L + N$	N/A	N/A	$L \cdot  m $	✓	Detect	Expensive	Similar to Dissent
Riposte [30]	$ m  + \sqrt{N}$	$\sqrt{N}$	1	$N \cdot  m $	✗	Prevent	None	Requires a separate audit server
Express [41]	$ m  + \log L$	$\lambda$	1	$L \cdot  m $	✗	Prevent	None	Exactly 2 servers
Two-Server	$ m  + \log L$	$\lambda$	1	$L \cdot  m $	✓	Prevent	Lightweight	With tree-based DPF [15]
Multi-Server	$ m  + \sqrt{L}$	$\lambda$	1	$L \cdot  m $	✓	Prevent	Lightweight	With seed-homomorphic DPF [12, 30]

Table 1: Per-request asymptotic efficiency of Spectrum (highlighted) and prior anonymous broadcasting systems for  $L$  broadcasters,  $N$  total users,  $|m|$ -sized messages, and global security parameter  $\lambda$ .  $O(\cdot)$  notation suppressed for clarity. Spectrum’s advantages include: a request size that is sublinear in  $L$  (Section 5.1) and independent of  $N$  (Section 3.3), a protocol for lightweight auditing of client requests to prevent disruption (Section 3.1), and a fast blame protocol for security against malicious servers (Section 4.3).

disrupt a broadcast. (A malicious server can always *overtly* disrupt the broadcast by refusing to participate in Spectrum.)

**Allowing forgeries on zero messages.** To allow subscribers to send the zero message *without* knowing the secret MAC key, we leverage the following insight from the SPDZ [31] multi-party computation protocol. The  $\gamma$  value acts solely as a “nonce” to prevent forgeries on the message  $0 \in \mathbb{F}$  [93]. Because of this, we can eliminate  $\gamma$  while still having the desired unforgeability property of the original MAC for all *non-zero* messages. When evaluated over secret shares,  $\text{MAC}_\alpha(m) = \alpha \cdot m \in \mathbb{F}$  maintains security for all  $m \neq 0$ . This satisfies our requirement: Subscribers can send  $m = 0$  and a valid tag  $t = 0$  *without* knowing  $\alpha$  (i.e., subscribers can “forge” a valid tag but only for  $m = 0$ ).

**Preventing client-server collusion.** To prevent an honest-but-curious server from collaborating with a malicious client to disrupt a broadcast, we must prevent the servers from knowing the broadcast key  $\alpha$  while still allowing them to check the MAC tag. To achieve this, we shift the entire verification procedure “to the exponent” of a group  $\mathbb{G}$  of prime order  $p$  (so that the exponent constitutes a field  $\mathbb{F}_p$ ). For security, we also require that the discrete logarithm problem is computationally intractable in the group  $\mathbb{G}$  [85]. Then, instead of  $\alpha$ , the servers obtain a public verification key  $g^\alpha$  (here  $g$  is a generator of  $\mathbb{G}$ ) from each broadcaster. All verification proceeds as before. Each client generates secret-shares  $(t_A, t_B)$  of a tag  $t$  and shares  $(m_A, m_B)$  of the message  $m$ , which are distributed to the servers.

- ServerA computes  $g^{\beta_A} \leftarrow (g^\alpha)^{m_A} / g^{t_A}$ .
- ServerB computes  $\beta_B \leftarrow (g^\alpha)^{m_B} / g^{t_B}$ .
- Servers swap  $g^{\beta_A}$  and  $g^{\beta_B}$  and check if  $g^{\beta_A} \cdot g^{\beta_B} = g^0 = 1_{\mathbb{G}}$ .

**Security.** The unforgeability properties are inherited from the Carter-Wegman MAC. Client anonymity (i.e., secrecy of the message  $m_i$ ) follows from the additive secret-sharing.

Client-server collusion is prevented by only the broadcaster knowing the broadcast key  $\alpha$ . See Section 6 for full analysis.

## 3.2 Putting things together

In this section, we combine DC-nets for broadcast with anonymous access control to realize Spectrum with a single channel, generalizing to multiple channels in Section 4.

**Setup: broadcast key distribution.** The setup in Spectrum involves the broadcaster anonymously “registering” with the servers by giving them the authentication public key  $g^\alpha$ . The servers must not learn the identity of the broadcaster when receiving this key, which leads us to a somewhat circular problem: broadcasters need to anonymously broadcast a key in order to broadcast anonymously. We solve this one-time setup problem as follows. All clients use a slower anonymous broadcast system suitable for low-bandwidth content at system setup time [2, 30, 55, 95]. The broadcaster shares an authentication key while subscribers share nothing. Keys are small (e.g., 64 bytes) and therefore practical to share with existing anonymity systems. Moreover, once the keys for the broadcaster are established, they may be used indefinitely. This process is similar to a “bootstrapping” setup found in related work [4, 29, 41, 58, 90, 95]. Spectrum is agnostic to how this setup takes place: one possibility is to use Riposte [27, 30], which shares a similar threat model.

**Step 1: Sharing a message.** As in the DC-net scheme, the broadcaster generates secret-shares of the broadcast message  $\hat{m}$  in the field  $\mathbb{F}$ . All other clients (subscribers) generate secret-shares of the message 0. The only difference is that in Spectrum, the broadcaster knows the broadcast key  $\alpha$  while subscribers do not. Let  $y = \alpha$ , if the client is the broadcaster and  $y = 0$  otherwise. Each client proceeds as follows.

- 1.1: Sample random  $m_A, m_B \in \mathbb{F}$  such that  $m = m_A + m_B \in \mathbb{F}$ .
- 1.2: Compute  $t \leftarrow y \cdot m \in \mathbb{F}$ . // MAC tag (Section 3.1)
- 1.3: Sample random  $t_A, t_B \in \mathbb{F}$  such that  $t = t_A + t_B \in \mathbb{F}$ .

1.4: Send  $(m_A, t_A)$  to ServerA and  $(m_B, t_B)$  to ServerB.

The above amounts to secret-sharing the message and access control MAC tag between both servers.

**Step 2: Auditing shares.** Servers collectively verify access control using the shares of the message and tag.

2.1: ServerA computes  $g^{\beta_A} \leftarrow (g^\alpha)^{m_A} / g^{t_A}$ .

2.2: ServerB computes  $g^{\beta_B} \leftarrow (g^\alpha)^{m_B} / g^{t_B}$ .

2.3: The servers swap audit tokens  $g^{\beta_A}$  and  $g^{\beta_B}$  and verify that  $g^{\beta_A} \cdot g^{\beta_B} = g^0 = 1_{\mathbb{G}}$ .

The above follows the access control verification (Section 3.1). All shares that fail the audit are discarded by both servers. In Section 4, we discuss how we prevent “audit attacks” in which a server tampers with a client request so the check fails.

**Step 3: Recovering the broadcast.** Servers collectively recover the broadcast message by aggregating all received shares that pass the audit.

3.1: ServerA computes  $\text{agg}_A \leftarrow \sum_i (m_A[i]) \in \mathbb{F}$ .

3.2: ServerB computes  $\text{agg}_B \leftarrow \sum_i (m_B[i]) \in \mathbb{F}$ .

3.3: Servers swap  $\text{agg}_A$  and  $\text{agg}_B$ .

3.4: Servers compute  $\hat{m} \leftarrow \text{agg}_A + \text{agg}_B \in \mathbb{F}$ .

This recovers the broadcast message as in the vanilla DC-net scheme. The recovered message is then made public to all clients (e.g., via a public bulletin board [7, 25]).

### 3.3 Towards the full protocol

The single-channel scheme presented in Section 3.2 achieves anonymous broadcast while also preventing broadcast disruption by malicious clients. Two problems remain however. First, while the single-channel scheme is fast and robust against malicious clients, it does not efficiently extend to multiple broadcasters. Second, a malicious server can tamper with the audit to make it fail for one or more clients—and learn whether one of them was a broadcaster (see Appendix A).

**Supporting multiple channels.** To support multiple channels, we use distributed point functions (DPFs) [15, 16, 46] to “compress” secret-shares across multiple instances of the DC-net scheme. DPFs avoid the linear bandwidth overhead of repeating DC-nets for each broadcaster and have been successfully used for anonymous broadcast in other systems [2, 30, 41]. However, without access control, the DPFs must expand to a large space to prevent collisions. We show that our construction for single-channel access control extends to the multi-channel setting, where each broadcaster has a key associated with their allocated channel.

**Preventing audit attacks.** At a high level, our approach is to commit each server to the shares they receive from a client. In the case of an audit failure, each server efficiently proves that it adhered to protocol to blame the client; if it can’t, any honest server aborts Spectrum.

## 4 Many channels and malicious servers

In this section, we extend the single-channel protocol of Section 3.2 to the multi-channel setting. We first show how to use a DPF to support many broadcast channels with little increase in bandwidth overhead (compared to the one-channel setting), an idea introduced in Riposte [30]. We prevent disruption by augmenting DPFs with the anonymous access control technique from Section 3.1. Prior works [13, 16, 30, 34, 41] describe techniques to verify that a DPF is well-formed, but do not allow for access control. Spectrum does both.

### 4.1 Tool: distributed point functions

A *point function*  $P$  is a function that evaluates to a message  $m$  on a single input  $j$  in its domain  $\{1, \dots, L\}$  and evaluates to zero on all other inputs  $i \neq j$  (equivalently, a vector  $(0, 0, \dots, m, \dots, 0)$ ). We define a *distributed point function*: a point function encoded and secret-shared among  $n$  keys:

**Definition 1** (Distributed Point Function (DPF) [30, 46]). Fix integers  $L, n \geq 2$ , a security parameter  $\lambda$ , and a message space  $\mathcal{M}$ . Let  $\mathbf{e}_j \in \{0, 1\}^L$  be the  $j$ th row of the  $L \times L$  identity matrix. An  $n$ -DPF consists of (randomized) algorithms:

- $\text{Gen}(1^\lambda, m \in \mathcal{M}, j \in \{1, \dots, L\}) \rightarrow (k_1, \dots, k_n)$ ,
- $\text{Eval}(k_i) \rightarrow (m_1, m_2, \dots, m_L)$ .

These algorithms must satisfy the following properties:

- Correctness. A DPF is correct if expanding the output of Gen into the space of  $L$  messages  $\mathcal{M}^L$  and combining gives the corresponding point function:

$$\Pr \left[ \begin{array}{l} (k_1, \dots, k_n) \leftarrow \text{Gen}(1^\lambda, m, j) \\ \text{s.t. } \sum_{i=1}^n \text{Eval}(k_i) = m \cdot \mathbf{e}_j \end{array} \right] = 1,$$

where the probability is over the randomness of Gen

- Privacy. A DPF is private if any subset of evaluation keys reveals nothing about the inputs. That is, there exists an efficient simulator  $\text{Sim}$  which generates output computationally indistinguishable from strict subsets of the keys output by Gen.

We use a DPF with domain  $\{1, \dots, L\}$ , where each broadcaster/channel has an index  $j \in \{1, \dots, L\}$ . Each broadcaster must write a message  $m$  to channel  $j$ , but not elsewhere: we can think of this as a point function  $P$  with  $P(j) = m$ . Then, we can encode secret-shares of  $P$  using a DPF more efficiently than secret-sharing its vector representation (as in repeated DC-nets). Evaluated DPF shares can still be aggregated locally, and our access control protocol supports DPFs with a slight modification (Section 4.2).

**DPFs are concretely efficient.** The key size for state-of-the-art 2-DPFs [16] is  $O(\log L + |m|)$  (assuming PRGs); for the general case [15], when  $n > 2$ , the key size is  $O(\sqrt{L} + |m|)$  under the decisional Diffie-Hellman assumption [10].

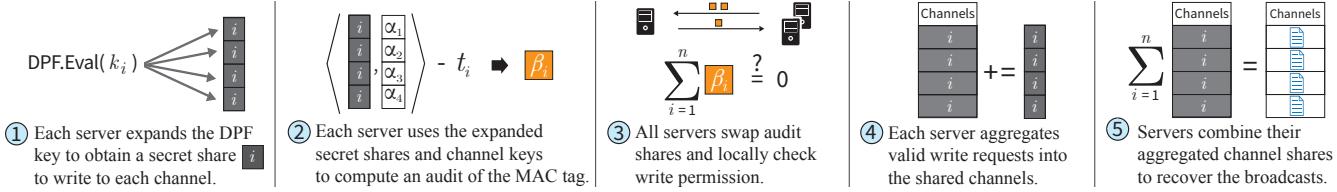


Figure 2: Overview of the server-side pipeline when processing a client’s request. Steps ①, ④ and ⑤ are computed over the field  $\mathbb{F}$ . Steps ② and ③ are computed “in the exponent” of the group  $\mathbb{G}$  when using the technique described in “Preventing client-server collusion” of Section 3.1.

Server-side work to expand each DPF uses fast symmetric-key operations in the two-server case [15, 16] and group operations in the multi-server case [30]. With  $L = 2^{20}$ , the DPF key size for the two-server construction is 325 B and for the  $n > 2$  construction 64 kB (excluding the message size).

## 4.2 Spectrum with many channels

In this section, we present the full Spectrum protocol with  $L$  channels and  $n \geq 2$  servers. Broadcasters reserve a channel in the setup phase. Clients encode their message at their channel (if any) using a DPF; the servers anonymously audit access to all channels before recovering messages.

**Setup.** The setup in this setting is similar to the setup described in Section 3.2. Each broadcaster anonymously provides a public verification key  $g^{\alpha_i}$  to the servers, to be associated with a channel. In addition to their key, any user with content to broadcast might upload a brief description or “teaser” of their content; the servers can choose which to publish, or users could perform a privacy-preserving vote [28]. We leave detailed exploration of the fair allocation of broadcast slots to users to future work. Post-setup, all servers hold a vector of  $L$  verification keys  $(g^{\alpha_1}, \dots, g^{\alpha_L})$ . Each key corresponds to one channel.

**Step 1: Sharing a message.** Let  $y = \alpha_j$  and  $j' = j$  if the client is a broadcaster for the  $j$ th channel ( $y = 0$  and  $j' = 0$  otherwise). Only broadcasters have  $m \neq 0$ . Each client runs:

- 1.1:  $(k_1, \dots, k_n) \leftarrow \text{DPF.Gen}(1^\lambda, m, j')$ . // gen DPF keys
- 1.2: Compute  $t \leftarrow m \cdot y \in \mathbb{F}$ .
- 1.3: Sample  $(t_1, \dots, t_n) \xleftarrow{R} \mathbb{F}$  such that  $\sum_{i=1}^n t_i = t \in \mathbb{F}$ .
- 1.4: Send share  $(k_i, t_i)$  to the  $i$ th server, for  $i \in \{1, \dots, n\}$ .

**Step 2: Auditing shares.** Upon receiving a request share  $(k_i, t_i)$  from a client, each server computes:

- 2.1:  $\mathbf{m}_i \leftarrow \text{DPF.Eval}(k_i) \in \mathbb{F}^L$ .
- 2.2:  $A \leftarrow \prod_{j=1}^L (g^{\alpha_j})^{\mathbf{m}_i[j]}$ . //  $A = g^{(\mathbf{m}_i, (\alpha_1, \dots, \alpha_L))}$
- 2.3:  $g^{\beta_i} \leftarrow A/g^{t_i}$ .
- 2.4: Send  $g^{\beta_i}$  to all other servers.

All servers check that  $\prod_i^n g^{\beta_i} = g^0 = 1_{\mathbb{G}}$ . If this condition does not hold, then the client’s request is dropped by all servers. In Section 4.3, we show how to detect a malicious server that tampers with a client’s request so that it fails this audit.

**Step 3: Recovering the broadcast.** Each server keeps an accumulator  $\mathbf{m}_i$  of  $L$  entries (i.e., the channels), initialized to  $\mathbf{0} \in \mathbb{F}^L$ . Let  $S = \{(k_j, t_j) \mid j \leq N\}$  be the set of all valid requests that pass the audit of Step 2. Each server:

- 3.1: Computes  $\mathbf{m}_i \leftarrow \sum_{(k,t) \in S} \text{DPF.Eval}(k) \in \mathbb{F}^L$ .
- 3.2: Publicly reveals  $\mathbf{m}_i$ . // shares of the aggregate.

Using the publicly revealed shares, anyone can recover the  $L$  broadcast messages as  $\widehat{\mathbf{m}} = \sum_i^n \mathbf{m}_i \in \mathbb{F}^L$ .

## 4.3 BlameGame: preventing audit attacks

BlameGame is a network overlay protocol that verifies who received what during protocol execution.

We use a *verifiable* encryption scheme [11, 20] where a party with a secret key can prove that a ciphertext decrypts to a certain message (DecProof makes a proof, and VerProof verifies it; see definition in Appendix C.1). Verifiable encryption reveals the plaintext request shares of a client to all servers if the client or the server is malicious (a malicious server may do this once, but will be immediately eliminated). BlameGame also uses a Byzantine broadcast protocol [19] so that all servers get the (encrypted) shares of all other servers.

**BlameGame.** BlameGame commits clients and servers to specific requests used in the audit. If the audit fails, honest servers reveal (with a publicly verifiable proof) the share they were given, which allows other servers to verify the results of the audit locally, which indict the client. Dishonest servers cannot give valid proofs for their shares.

*Setup.* All servers make a key pair  $(\mathbf{pk}_i, \mathbf{sk}_i)$  and publish  $\mathbf{pk}_i$ .

**Step 1: Generating commitments.** Let  $\tau_i$  be the client’s request secret-share for server  $i$ . The client runs:

- 1.1:  $C_i \leftarrow \text{Enc}(\mathbf{pk}_i, \tau_i)$ . // Encryption under  $\mathbf{pk}_i$ .
- 1.2: Byzantine broadcast all  $C_i$  to all servers.

Server  $i$  recovers  $\tau_i \leftarrow \text{Dec}(\mathbf{sk}_i, C_i)$ ; clients may go offline at this point. All servers are committed to the *encryption* of their secret-shares. We describe an optimization in Section 5.1 that makes the size of each  $C_i$  constant.

**Step 2: Proving innocence.** Each server publishes their share of the request  $\tau_i$  and a proof of correct decryption:

- 2.1:  $(\pi_i, \tau_i) \leftarrow \text{DecProof}(\mathbf{sk}_i, C_i)$ .
- 2.2: Send  $(\pi_i, \tau_i)$  to all servers.

**Step 3: Assigning blame.** Using the posted shares and proofs, each server assigns blame:

- 3.1: Collect  $(\pi_i, \tau_i)$  from servers  $i \in \{1, \dots, n\}$  and all  $C_i$ .
- 3.2: Check that  $\text{VerProof}(\text{pk}_i, \pi_i, C_i, \tau_i) = \text{yes}$ , for  $1 \leq i \leq n$ .
- 3.3: Check the audit using all the shares  $(\tau_1, \dots, \tau_n)$ .
- 3.4: Assign blame:
 

```
if 3.2 fails for any  $i$ : abort;           // bad server
else if 3.3 passes: abort;                // bad server
else if 3.3 fails: drop the client request. // bad client
```

## 5 Optimizations and extensions

Here, we describe extensions and optimizations to Spectrum. We show how to (1) broadcast *large* messages efficiently and (2) *privately* fetch published broadcasts as a subscriber.

### 5.1 Handling large messages efficiently

We described Spectrum in Section 4.2 with messages as elements of a field  $\mathbb{F}$ , which we check to perform access control. While a 16 B field suffices for audit security, large messages require much larger fields (or repeating the protocol many times). These approaches require proportionally greater bandwidth and computation to audit. Instead, we give a black-box transformation from a 2-server DPF over  $\mathbb{F}$  to a DPF over  $\ell$ -bit strings, *preserving security* (see Section 6.2). We use a pseudorandom generator (PRG). Clients create DPF keys encoding a short PRG seed, rather than a message. The servers efficiently audit this seed as before to enforce access control. Then, they expand it to a much longer message with the guarantee that the DPF is still non-zero at an index for which the client knows the broadcast key (if the message is non-zero).

**The transformation.** Let DPF be a DPF over the field  $\mathbb{F}$  and let  $\text{DPF}^{\text{bit}}$  be a DPF over  $\{0, 1\}$ . Let  $G : \mathbb{F} \rightarrow \{0, 1\}^\ell$  be a PRG. To write to channel  $j$ , a user computes:

1.  $\bar{s} \xleftarrow{R} \mathbb{F}$ . // random nonzero PRG seed
2.  $(k_A, k_B) \leftarrow \text{DPF.Gen}(\bar{s}, j)$ .
3.  $s_A^* \leftarrow \text{DPF.Eval}(k_A)[j], s_B^* \leftarrow \text{DPF.Eval}(k_B)[j]$ .
4.  $\bar{m} \leftarrow G(s_A^*) \oplus G(s_B^*) \oplus m$ .
5.  $(k_A^{\text{bit}}, k_B^{\text{bit}}) \leftarrow \text{DPF}^{\text{bit}}.\text{Gen}(1, j)$ .
6. Send  $(\bar{m}, k_A, k_A^{\text{bit}})$  to ServerA,  $(\bar{m}, k_B, k_B^{\text{bit}})$  to ServerB.

Every server evaluates the DPF keys to a vector  $s$ , of PRG seeds, and a vector  $b$  of bits. Each seed and bit other than the  $j$ th is *identical* on both servers (a secret-share of zero); at  $j$ , we get  $s_A^* \neq s_B^*$ . Servers evaluate the DPF by expanding each  $s[i]$  to an  $\ell$ -bit string and XORing  $\bar{m}$  only when  $b[j] = 1$ . If we define multiplication of a binary string by a bit as  $1 \cdot \bar{m} = \bar{m}$  and  $0 \cdot \bar{m} = 0$ , ServerA computes:

$$\mathbf{m}_A := (G(s_A[1]) \oplus b_A[1] \cdot \bar{m}, \dots, G(s_A[L]) \oplus b_A[L] \cdot \bar{m})$$

ServerB does the same. Then, we get that:

$$\mathbf{m}_A[i] \oplus \mathbf{m}_B[i] = \begin{cases} G(s[i]) \oplus G(s[i]) = 0^\ell & i \neq j \\ G(s_A^*) \oplus G(s_B^*) \oplus \bar{m} = m & i = j. \end{cases}$$

Servers perform the audit (in  $\mathbb{F}$ ) over the expanded PRG seeds and bits as in Section 3.2. Observe that the final output is non-zero only if: (1) some PRG seed, (2) some bit, or (3) the masked message  $\bar{m}$  is different on each server. The  $s$  and  $b$  audit checks (1) and (2); servers check (3) by comparing hashes of  $\bar{m}$ . As before, the 0 MAC tag passes the audit for an empty message, and broadcasters can provide a correct tag.

**Many servers.** The above transformation generalizes to the  $n$ -server setting. The intuition is the same: only “non-zero” PRG seeds should expand to write non-zero messages. However, we need a PRG with special properties for this to hold with  $n > 2$ . We give the full transformation in Appendix B. Applying this transformation to a square-root DPF yields the  $n$ -server DPF of Corrigan-Gibbs et al. [30], but now with access control.

**BlameGame optimization.** The masked message  $\bar{m}$ , given to all servers, constitutes the bulk of data in *each* DPF key, so clients can omit it in their request commitments (Section 4.3) when using the above transformation because servers do not need it to verify access control. (The verification performed by the servers only depends on the DPF *seeds* and checking equality of the masked message  $\bar{m}$ .)

### 5.2 Private broadcast downloads

Content published using an anonymous broadcast system is likely to be sensitive and subscribers might want to have plausible deniability when it comes to which broadcasts they are interested in. In a setting with many channels, we might allow the subscribers to download one channel while hiding *which* channel they download: the exact setting of private information retrieval (PIR) [24]. In (multi-server) PIR, a client submits *queries* to two or more servers, receiving *responses* which they combine to recover one document in a “database.” The queries hide which document was requested. In Spectrum, clients can use any PIR protocol to hide which channel they download. Modern PIR schemes based on DPFs have minimal bandwidth overhead for queries [15, 16]. However, the processing time on each server is always linear [24]. We evaluate the overhead of using PIR for subscriber anonymity in Section 7.1.

## 6 Security and efficiency analysis

In this section, we analyze the theoretical efficiency and security of Spectrum with respect to the threat model and required guarantees outlined in Section 2.4.

## 6.1 Efficiency analysis

We briefly analyze the efficiency of Spectrum (Section 4.2) and BlameGame (Section 4.3) with the above optimizations.

**Communication efficiency in Spectrum.** Spectrum can use any DPF construction with outputs in a finite field using the transformation of Section 5.1 to support  $\ell$ -bit messages with only an additive  $O(\ell)$  overhead to the DPF key size. Using optimized two-server DPF constructions [15, 16], clients send requests of size  $O(\log L + |m|)$  (for  $L$  channels). With more than two servers, the communication is  $O(\sqrt{L} + |m|)$  using the seed-homomorphic PRG based DPF construction [30]. For the audit, inter-server communication is constant.

**Computational efficiency in Spectrum.** Each server performs  $O(L \cdot |m|)$  work per client when aggregating the shares and performing the audit ( $O(N \cdot L \cdot |m|)$  total for  $N$  clients). The work on each client is  $O(\log L + |m|)$  when using two-server DPFs and  $O(\sqrt{L} + |m|)$  otherwise [15].

## 6.2 Security of Spectrum

We first describe the ideal functionality of the anonymous broadcast system which Spectrum instantiates.

**Ideal functionality.** Ideal Spectrum is defined as follows:

- Receive message  $m = 0$  from each subscriber,  $m = \hat{m}$  from the broadcaster, and no input from the servers.
- Output  $\hat{m}$  to both the clients and servers.

**Client anonymity.** We argue that Spectrum provides client anonymity by constructing a simulator for the view of a network adversary corrupting any strict subset of servers.

**Claim 1.** *If at least one server is honest, then no probabilistic polynomial time (PPT) adversary  $\mathcal{A}$  observing the entire network and corrupting any strict subset of the servers and an arbitrary subset of clients, can distinguish between an honest broadcaster and an honest subscriber.*

*Proof.* We construct a simulator  $\text{Sim}$  for the view of  $\mathcal{A}$  when interacting with an honest client. Let  $\widehat{\text{Sim}}$  be the DPF simulator (see Definition 1).  $\text{Sim}$  proceeds as follows:

1. Take as input  $(\mathbb{G}, g), (g^{\alpha_1}, \dots, g^{\alpha_L}), \mathbb{F}$ , and subset of corrupted server indices  $I \subset \{1, \dots, n\}$ .
2. Sample  $t_i \xleftarrow{R} \mathbb{F}$  for  $i \in \{1, \dots, n\}$  such that  $\sum_i t_i = 0$ .
3.  $\{k_i \mid i \in I\} \leftarrow \widehat{\text{Sim}}(I)$ . *// see Definition 1*
4. Output  $\text{View} = (\{(t'_i, k_i) \mid i \in I\}, \{g^{t_j} \mid j \in \{1, \dots, n\} \setminus I\})$ .

*Analysis.* The view includes:

- Each DPF key  $k_i$  for corrupted server  $i$ .
- Each MAC tag share  $t'_i$  for corrupted server  $i$ .
- Audit shares  $g^{t_j}$  from every honest server  $j$ .

The DPF keys are computationally indistinguishable from real DPF keys by the security of the DPF simulator. Therefore, it remains to argue that the tag and audit shares are distributed identically to the real view. Recall that during an audit, server  $i$  publishes  $g^{\beta_i} = g^{\langle m_i, (\alpha_1, \dots, \alpha_L) \rangle - t_i}$  where  $m_i$  is the output of  $\text{DPF}.\text{Eval}(k_i)$  and  $t_i$  is a secret-share of the MAC tag  $t$ . For a subscriber,  $\langle m_i, (\alpha_1, \dots, \alpha_L) \rangle$  (the inner product) gives a random secret share of 0 and  $t_i$  is a secret share of 0, so  $g^{\beta_i}$  is a random (multiplicative) secret share of  $g^0$ . For a broadcaster publishing to channel  $j$ ,  $\langle m_i, (\alpha_1, \dots, \alpha_L) \rangle$  is a random secret share of  $m \cdot \alpha_j = t$ , so  $g^{\beta_i}$  as computed by the  $i$ th server is a random multiplicative secret share of  $g^0$  as well. Therefore, the distribution of the audit and tag shares ( $g^{\beta_i}$  and  $t_i$ , respectively) is identical to the real view. Finally, because the connection between clients and servers is encrypted (and of fixed-size), we can efficiently simulate network traffic as random encrypted data.  $\square$

**Disruption resistance in Spectrum.** We prove that a client cannot disrupt a broadcast on the  $j$ th channel without knowing the channel broadcast key  $\alpha_j$ .

**Claim 2.** *Assuming the hardness of the discrete logarithm problem [11, 40] in  $\mathbb{G}$ , no probabilistic polynomial time (PPT) client can write to channel  $j$  and pass the audit performed by the servers without knowledge of  $\alpha_j$ .*

*Proof.* Assume towards contradiction that some adversarial client can generate (potentially ill-formed) DPF keys that result in a non-zero vector (WLOG, assume that index  $L$  is non-zero) and pass the audit for a given access tag with non-negligible probability. We can use the client to extract the discrete logarithm for any element of  $\mathbb{G}$  as follows. Given  $g^{\alpha^*}$ , choose random  $\alpha_i \in \mathbb{F}$  for  $i \in \{1, \dots, L-1\}$ . Give the client  $(g^{\alpha_1}, \dots, g^{\alpha_{L-1}}, g^{\alpha^*})$  and get in return DPF keys  $(k_1, \dots, k_n)$  and MAC tag  $t$ . Given these DPF keys, we can compute  $\mathbf{m} = (m_1, \dots, m_L)$  by evaluating the DPF. If the shares pass the audit, it must be that  $\langle \mathbf{m}, \alpha \rangle = t$ . However,  $\alpha$  includes  $\alpha^*$  so we can solve for  $\alpha^*$  ( $t$  and all  $\alpha_i$  except for  $\alpha^*$  are known). We conclude that the client has knowledge of  $\alpha^*$ .  $\square$

**Security of the DPF transformation.** The construction from Section 5.1 maintains security. This construction transforms a DPF DPF into a DPF DPF' over  $\ell$ -bit messages.

**Claim 3.** *If Spectrum with DPF preserves client anonymity, Spectrum with DPF' preserves client anonymity.*

*Proof.* We build a simulator  $\text{Sim}'$  for DPF' from the simulator  $\text{Sim}$  for DPF.  $\text{Sim}'$  simply runs  $\text{Sim}$  twice (once to generate the seed-DPF keys and once for the bit-DPF keys) and picks an  $\ell$ -bit message uniformly at random for  $\bar{m}$ . The simulator's  $\bar{m}$  is computationally indistinguishable from the real  $\bar{m}$  (otherwise, the PRG used to mask the message is not secure). Therefore, if there exists an efficient distinguisher, it can also

distinguish between the keys output by Sim and the real DPF keys, a contradiction.  $\square$

**Claim 4.** *If Spectrum with DPF has disruption resistance, Spectrum with DPF' has disruption resistance.*

*Proof.* Assume, towards contradiction, that there exists a computationally bounded adversary  $\mathcal{A}$  which does *not* obtain the broadcast key  $\alpha$  as input, and outputs a set of DPF' keys along with MAC tag shares. If the set of DPF' keys write to at least one channel and the tag shares output by  $\mathcal{A}$  pass the server MAC audit, then we can produce a non-zero message and tag for DPF as follows. WLOG, we fix the number of servers to  $n = 2$ . Run  $\mathcal{A}$  to get two DPF' keys  $k'_1, k'_2$  and tag  $t = (t_1, t_2)$ . By construction,  $k'_i = (k_i, k_i^{\text{bit}}, m')$ , where  $k_i$  and  $k_i^{\text{bit}}$  are DPF keys with range  $\mathbb{F}_p$  and  $\mathbb{F}_2$ , respectively, and  $m' \in \mathbb{F}_{2^\ell}$  is a masked message (identical in each DPF' key). If these keys and tags pass the audit, the masked message in each key is the same (by the collision resistance of the audit hash function). Then, because the key for DPF' writes a non-zero message, at least one of the two DPF keys (either  $k$  or  $k^{\text{bit}}$ ) must write a non-zero message (otherwise the keys would be writing zero). It follows that  $(k_1, k_2)$  or  $(k_1^{\text{bit}}, k_2^{\text{bit}})$  encode a non-zero message, which contradicts Claim 2.  $\square$

### 6.3 Security of BlameGame

We must show that in BlameGame: (1) an honest client will never be blamed, (2) a malicious client will always be blamed, (3) an honest server will never be blamed, and (4) a malicious server will always be blamed. Incorrect blame attribution indicates a failure of the verifiable encryption scheme or audit security; see Appendix C.2 for full proof.

**Overhead of BlameGame.** BlameGame requires some extra bandwidth and computation time. Clients send a shared message mask *once* to each server; DPF keys add about 100 bytes per client request (details in Section 5.1). The servers must run BlameGame for each malicious client. However, verifying decryption takes tens of *microseconds*, and running the audit is similarly quick (see Section 7.1). Because the servers delay the work of aggregating messages until *after* the audit, a malicious client often requires *fewer* cycles than an honest one (but extra network communication).

## 7 Evaluation

We build and evaluate Spectrum, comparing it to state-of-the-art anonymous broadcasting works: Riposte, Blinder, Express, and Dissent (see related work; Section 8).

**Riposte** [30] is designed for anonymous broadcasting where all users broadcast at all times. Riposte uses three servers (one trusted for audits) but generalizes to many servers (one honest). Riposte was designed for smaller messages and the source code fails to run with messages of size 5 kB or greater.

**Blinder** [2] builds on Riposte but requires an *honest majority* of at least 5 servers. Like Riposte, Blinder also assumes that all users are broadcasting. Blinder supports using a server-side GPU to increase throughput.

**Express** [41] is an anonymous communication system designed for anonymous “dropbox”-like applications. It does not support broadcast as-is, but can be easily modified to do so. We include Express in our comparison as a recent, high-performance system decoupling broadcasters and subscribers.

**Dissent** [29, 95] has a setup phase (like Spectrum’s), a DC-net phase, and a blame protocol. We give measurements both with and without the blame protocol and exclude the setup phase. Without the blame protocol, the system runs a plain DC-net without any disruption resistance and is quite fast. If *any* user sends an invalid message, Dissent runs the (expensive) blame protocol (up to once per malicious user).

We use data from the Blinder paper [2, Fig. 4] as the source did not compile. The Dissent code (last modified in 2014) ran with up to 1000 users and 10 kB messages, but hung indefinitely after increasing either (though the authors report 128 kB messages with 5000 users). Linearly scaling our measurements, we find them broadly consistent (3× faster) with the authors’ reported measurements for 128 kB messages with the same number of users in a similar setting [95, Fig. 7].

**Implementation.** We build Spectrum in ~8000 lines of open-source [1] Rust code, using AES-128 (CTR) as a PRG and BLAKE3 [66] as a hash. Because our DPF has relatively few “channels”  $L$ , a DPF with  $O(L)$ -sized keys (adapted from Corrigan-Gibbs et al. [30]) gives the best concrete performance. For the multi-server extension (Section 5.1 and Appendix B), we use a seed-homomorphic PRG [12] with the Jubjub [49] curve. We encrypt traffic with TLS 1.3 [73].

**Environment.** We run VMs on Amazon EC2 to simulate a WAN deployment. Each c5.4xlarge 8-core instance has 32 GiB RAM [76], running Ubuntu 20.04 (\$0.68 per hour in September 2021). We run clients in us-east-2 (Ohio) and servers in us-east-1 (Virginia) and us-west-1 (California). Network round trip times (RTTs) were 11 ms between Virginia and Ohio, 50 ms between Ohio and California, and 61 ms between Virginia and California. Inter-region bandwidth was 524 Mbit/s (shared between many clients simulated on the same machine).

### 7.1 Results

In our experiments, we find Spectrum is 4–7× faster than Express for 5 MB to 100 kB messages, 2× / 13–17× slower than Blinder (CPU/GPU, resp.) in *unfavorable* settings, 500–7500× / 250–520× faster than Blinder (CPU/GPU) in *favorable* settings, and 16–12,500× faster than Riposte. We run 5 trials per setting, shading the 95% confidence interval (occasionally too small to be seen).

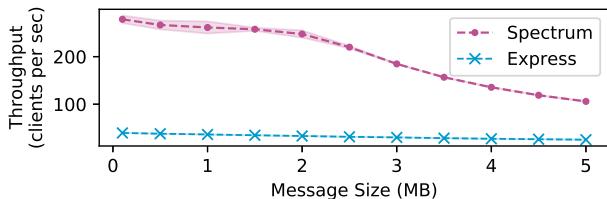


Figure 3: Throughput (client requests per second; higher is better) for a one channel deployment (one broadcaster and many subscribers).

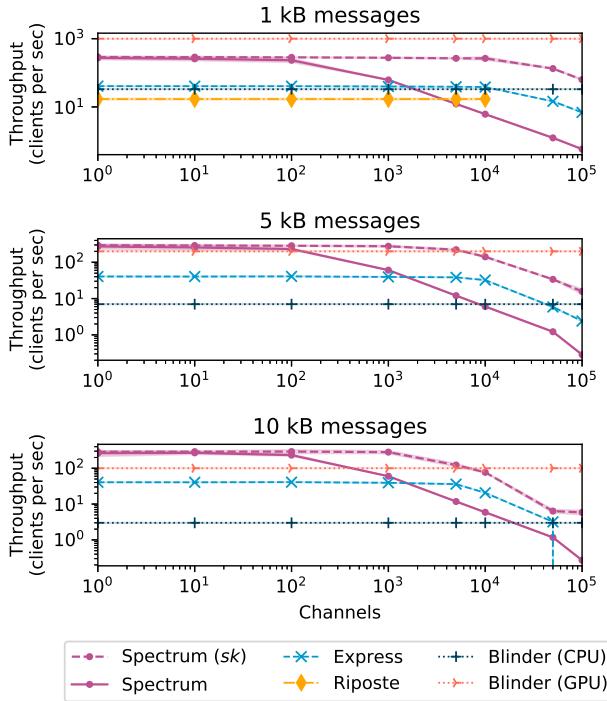


Figure 4: Throughput (requests per second; higher is better) for broadcasts with 100,000 users with varying numbers of broadcasting users (“channels”): Express and Spectrum benefit from fewer channels. (Blinder numbers as reported by the authors [2].)

**One channel.** In Figure 3, we report the throughput (client requests per second) for both Spectrum and Express in the one-channel setting. As expected, performance is worse with larger messages for both systems. However, we find that Spectrum, compared to Express, is 4–7× faster on messages between 100 kB and 5 MB. Riposte and Blinder have no analog for the single-channel setting. (Dissent does support a one-channel setting, but did not run with large messages.)

**Many channels.** Unlike Riposte and Blinder, Spectrum is faster with fewer broadcasters. To compare, we fix 100,000 users and vary the number of channels from 1 (best-case for Spectrum) to 100,000 (worst-case). We evaluate Spectrum with and without the change described in “Preventing client-server collusion” (Section 3.1). Without the change, which we call “*Spectrum (sk)*”, servers obtain the MAC secret key for each channel. This mirrors the threat model of e.g., Ex-

Request Size	Request per client	Audit per client	Aggregation once per server
	$ m  + 70$ bytes	70 bytes	$ m  + 3$ bytes
BlameGame (per failed audit)	<b>Backup Request per client</b>	<b>Audit per client</b>	<b>Decryption once per client</b>
	140 bytes	200 bytes	10 $\mu$ s

Table 2: Upper bound on request size for one channel and  $|m|$ -bit messages. BlameGame only runs if the first request audit fails.

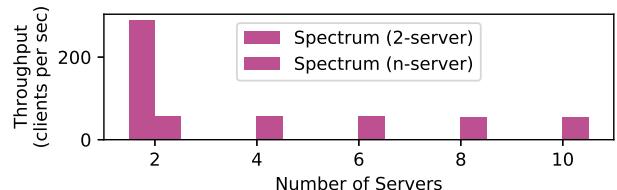


Figure 5: Spectrum can generalize to  $n > 2$  servers (shown for 10 kB messages). This uses an expensive PRG and is therefore slower, but adding more servers causes no additional slowdown.

press [41]. With the change, servers only get MAC public keys which prevents covert client-server collusion. However, there is a modest price in terms of performance due to the elliptic curve operations (see Figure 4). We find that Spectrum (both variants) outperform all other systems with 10 kB messages for relatively few channels (up to hundreds), but performs relatively worse with smaller messages or more channels. For “Twitter-like” settings, another system (e.g., Blinder or Riposte) may be appropriate.

**Overhead.** In any anonymous broadcast scheme, every client (even subscribers) must upload data corresponding to the message length  $|m|$  to ensure privacy. For DC-net based schemes, the client sends a size- $|m|$  request to each server. We measure the concrete request sizes of Spectrum and compare to this baseline in Table 2. Client request overhead is small: about 70 B, roughly 75× smaller than in Express. Moreover, in Spectrum, request audits are under 16 B, a 120× improvement over Express [41]. BlameGame imposes little overhead (both in terms of bandwidth and computation). Because BlameGame runs only when a request audit fails, these overheads occur for few requests in most settings.

**Many servers.** In Section 5.1 and Appendix B, we note that our construction of Spectrum generalizes from 2 to  $n$  servers (with one honest) in a manner similar to Riposte [30]. The  $n$ -server construction uses a seed-homomorphic pseudorandom generator (PRG) [12]. On one core of an AMD Ryzen 4650G CPU, we measured the maximum throughput of our seed-homomorphic PRG at 300 kB/s, 20,000 times slower than an AES-based PRG. For 10,000 kB messages, Spectrum was 5× slower with the seed-homomorphic PRG (Figure 5); with

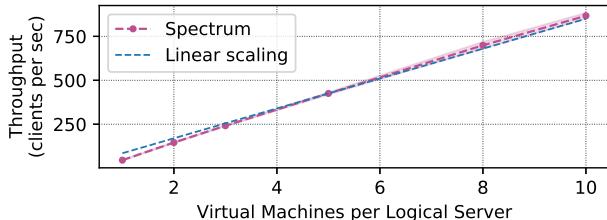


Figure 6: Spectrum is highly parallelizable: for 500 channels of 100 kB messages, 10 VMs per “server” gives a 10× speedup.

larger messages, the relative difference increases. We find *no additional* slowdown between 2 to 10 servers. An interesting direction for future work would be to evaluate Spectrum with LWE-based seed-homomorphic PRG constructions [12], as they are likely to have better concrete performance.

**Scalability.** We may trust machines administered by the same *organization* equally, viewing several worker servers as one logical server. Client requests trivially parallelize across such workers: running 10 workers per logical server leads to a 10× increase in overall throughput (Figure 6). In a cloud deployment, Spectrum handles the same workload in less time for negligible additional cost by parallelizing the servers.

**Latency.** In Figure 7, we measure the time to broadcast a single document for these systems with varying numbers of users. For Spectrum, we use a 1 MB message. For Blinder, we use numbers reported by the authors [2, Fig. 4], multiplied to the same message size (the authors explicitly state that repeating the scheme many times is the most efficient way to send large messages). We benchmark Dissent both with and without the blame protocol invoked during a round. The former (blame) is the performance of Dissent if any client misbehaves. The latter (no blame) assumes that no client misbehaves. Express doesn’t have a notion of “rounds” so we omit it here. We find that for one channel of large messages, Spectrum is much faster than other systems (except Dissent with no blame protocol; i.e., when all clients are honest).

**Client privacy.** In Section 5.2, we outlined how private information retrieval (PIR) [24] techniques provide client privacy for multiple channels. Figure 8 shows the server-side CPU capacity to process these requests for 1 kB, 10 kB, and 100 kB messages and 1–100,000 channels. We measure one core of an AMD Ryzen 4650G CPU for a simple 2-server PIR construction [24], finding good concrete performance.

## 7.2 Discussion

Our evaluations showcase the use of Spectrum for a real-world anonymous broadcasting deployment using commodity servers. Compared to the state-of-the-art in anonymous broadcasting, Spectrum achieves speedups in settings with a large ratio of passive subscribers to broadcasters. Based on our evaluation, with 10,000 users, Spectrum could publish: a

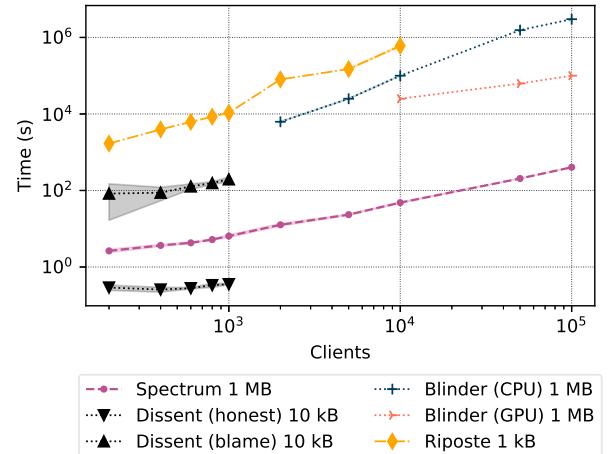


Figure 7: Latency for uploading a single document with varying numbers of users. Blinder numbers as reported by the authors [2, Fig. 4] and linearly scaled to 1 MB messages.

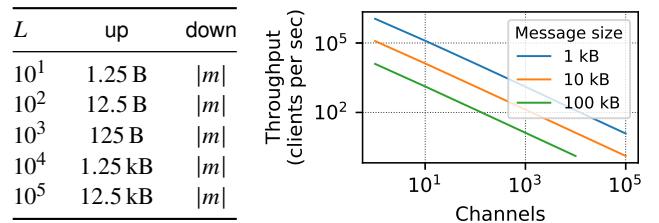


Figure 8: **Left:** Bandwidth usage of a PIR query with varying number of channels  $L$ . **Right:** Server capacity (one core) to answer PIR queries for private client downloads. For  $L$  channels, the client requests one out of  $L$  documents, where channels have size  $|m|$ .

*PDF document* (1 MB) in 50s, a *podcast* (50 MB) in 40m, or a *documentary movie* (500 MB, the size of Alexei Navalny’s documentary on Putin’s Palace at 720p [80]) in 6h40m.

**Operational costs.** We estimate costs for a cloud deployment of Spectrum using current Amazon EC2 prices, reported in US dollars. Servers upload about 100 bytes per query (in the above settings, at most 1 GB per day) and inbound traffic is free on EC2. We focus on compute costs: \$6.84 per GB published through Spectrum (with 10,000 users). Table 3 compares costs to publish 1 GB among 10,000 users.

## 8 Related work

Existing systems for anonymous broadcast are suitable for 140 B to 40 kB [2, 30, 41] broadcasts, orders of magnitude smaller than large data dumps [69, 75, 77] common today.

**Mix Networks and Onion Routing.** In a mix net [22], users send an encrypted message to a proxy server, which collects and forwards these messages to their destinations in a random order. Chaining several such hops protects users from compromised proxy servers and a passive network adversary. Mix

System	Cost (USD)
Blinder (GPU)	\$2,000,000.00
Blinder (CPU)	\$250,000.00
Riposte	\$218,000.00
Dissent (with blame protocol; one round)*	\$76,000.00
Dissent (honest clients)	\$134.00
Express	\$30.22
Spectrum	\$6.84

Table 3: Cost to upload one 1 GB document anonymously with 10,000 users, based on the *best* observed rate for each system with that many users (that is, the maximum throughput over all settings we measured; for Blinder, we use the best reported rate). We multiply the total time at the maximum throughput by hourly rate to get the cost. \*Extrapolated from 1000 users.

nets and their variations [32, 56, 57, 59, 60, 63, 64, 71, 72, 81, 82, 90] scale to many servers. However, because messages are exchanged and shuffled between many servers, mix nets are poorly suited to high-bandwidth applications. Atom [55] uses mix nets with zero-knowledge proofs to horizontally scale anonymous broadcast to millions of users (Spectrum achieves about 12,500× the throughput [55, Fig. 9]). Riffle [54] uses a *hybrid verifiable shuffle*; in the broadcast setting, it shares a 300 MB file with 500 users in 3 hours (Spectrum supports about 10,000 users in that time).

Some systems use onion routing for better performance than a mix net. In onion routing, users encrypt their messages several times (in onion-like layers) and send them to a chain of servers. Tor [37], the most popular onion routing system, has millions of daily users [83]. Tor provides security in many real-world settings, but is vulnerable to traffic analysis [53, 62, 78]. If only one user sends large volumes of data, an adversary can identify them—Tor discourages high bandwidth applications for this and other reasons [36].

**DC-nets.** Another group of anonymous communications systems use dining cryptographer networks (DC-nets) [23] (Section 2). DC-nets are vulnerable to disruption: any malicious participant can clobber a broadcast by sending a “bad” share. Dissent [29, 95] augments the DC-nets technique with a system for accountability. Like Spectrum, Dissent performs best if relatively few users are broadcasting. The core data sharing protocol is a standard DC-net, which is very fast and supports larger messages. Further, it supports many servers at little additional cost. However, Dissent is not suitable for many-user applications where disruption is a concern. If *any* user misbehaves, Dissent must undergo an expensive blame protocol (quadratic in the total number of users). This approach detects, rather than prevents, disruption. The user is evicted after this protocol, but an adversary controlling many users can cause many iterations of the blame protocol.

PriFi [6] builds on the techniques in Dissent to create indistinguishability among clients in a LAN. Outside servers

help disguise traffic using low-latency, precomputed DC-nets. Like Dissent, PriFi catches disruption after-the-fact using a blame protocol (as often as once per malicious user). The PriFi blame algorithm is much faster, but still scales with *all* users in the system (in Spectrum, each malicious user incurs constant server-side work).

Riposte [30] enables anonymous Twitter-style broadcast with many users using a DC-net based on DPFs and an auditing server to prevent disruptors. We find that Riposte is 16× slower than Spectrum with 10,000 users. Further, Riposte assumes that all users are broadcasting and therefore gets *quadratically* slower in the total number of users.

A more recent work, Blinder [2] uses multi-party computation to prevent disruption. Blinder’s threat model requires at least five servers with an honest majority. Like Spectrum, Blinder is resilient to active attacks by a malicious server. It is fast for small messages when most users have messages to share, but much slower for large messages. Blinder allows trading money for speed with a GPU.

Express [41] is a system for “mailbox” anonymous communication (writing anonymously to a designated mailbox). Express also uses DPFs for efficient write requests. However, it only runs in a two-server deployment. Express is *not* a broadcasting system, and while it is possible to adapt it to work in a broadcast setting, it is not designed to withstand active attacks by the servers and is insecure for such an application (see Appendix A for details).

## 9 Conclusions

Spectrum supports high-bandwidth, low-latency broadcasts from a small set of broadcasters to a large number of subscribers by applying new tools to the classic DC-net architecture. We prevent disruption by malicious clients with an efficient blind access control mechanism that prevents clients from writing to a channel they do not have access to.

Additionally, we introduce optimizations to decouple server-side overhead from the message size, which allows Spectrum to scale to large messages and many broadcasters. To prevent malicious servers from deanonymizing clients, we develop a lightweight blame protocol to abort Spectrum if a server deviates from the protocol. Our experimental results show that Spectrum can be used for uploading gigabyte-sized documents anonymously among 10,000 users in 14 hours.

## 10 Acknowledgments

We thank Henry Corrigan-Gibbs, Kyle Hogan, Albert Kwon, and Derek Leung, for helpful feedback and discussion on early drafts of this paper. We would also like to thank our shepherd Alan Liu and the anonymous NSDI reviewers for their insightful feedback and many suggestions that helped to significantly improve this paper.

## References

- [1] Spectrum implementation. <https://www.github.com/znewman01/spectrum-impl>, 2021.
- [2] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: Scalable, robust anonymous committed broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1233–1252, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417261. URL <https://doi.org/10.1145/3372297.3417261>.
- [3] C. Fred Alford. Whistleblowers and the narrative of ethics. *Journal of social philosophy*, 32(3):402–418, 2001.
- [4] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.
- [5] Raymond Walter Apple Jr. 25 years later; lessons from the Pentagon Papers. *The New York Times*, 23 June 1996. URL <https://www.nytimes.com/1996/06/23/weekinreview/25-years-later-lessons-from-the-pentagon-papers.html>. Accessed March 2022.
- [6] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Apostolos Pyrgelis, Bryan Ford, Joan Feigenbaum, and Jean-Pierre Hubaux. Prifi: Low-latency anonymity for organizational networks. *Proc. Priv. Enhancing Technol.*, 2020(4):24–47, 2020. doi: 10.2478/popets-2020-0061. URL <https://doi.org/10.2478/popets-2020-0061>.
- [7] Josh Daniel Cohen Benaloh. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987.
- [8] Charles Berret. Guide to SecureDrop, 2016. URL [https://www.cjr.org/tow\\_center\\_reports/guide\\_to\\_securedrop.php](https://www.cjr.org/tow_center_reports/guide_to_securedrop.php).
- [9] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-CNN: A data-efficient website fingerprinting attack based on deep learning. *Proceedings on Privacy Enhancing Technologies*, 2019(4):292–310, 2019.
- [10] Dan Boneh. The decision Diffie-Hellman problem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, pages 48–63, 1998. doi: 10.1007/BFb0054851. URL <https://doi.org/10.1007/BFb0054851>.
- [11] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Recuperado de https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup\_0\_4.pdf*, 2017.
- [12] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013.
- [13] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.
- [14] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 92–102, 2007.
- [15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer. ISBN 978-3-662-46803-6.
- [16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.
- [17] Russ Buettner, Susanne Craig, and Mike McIntire. Long-concealed records show Trump’s chronic losses and years of tax avoidance. *The New York Times*, 2020. URL <https://www.nytimes.com/interactive/2020/09/27/us/donald-trump-taxes.html>. Accessed March 2022.
- [18] Bryan Burrough, Sarah Ellison, and Suzanna Andrews. The Snowden saga: A shadowland of secrets and light. *Vanity Fair*, 2014. URL <https://www.vanityfair.com/news/politics/2014/05/edward-snowden-politics-interview>. Accessed March 2022.
- [19] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi: 10.1007/3-540-44647-8\_31. URL [https://doi.org/10.1007/3-540-44647-8\\_31](https://doi.org/10.1007/3-540-44647-8_31).
- [20] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In

- Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2003. doi: 10.1007/978-3-540-45146-4\\_8. URL [https://doi.org/10.1007/978-3-540-45146-4\\_8](https://doi.org/10.1007/978-3-540-45146-4_8).
- [21] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.
  - [22] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.
  - [23] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.
  - [24] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.
  - [25] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 719–728, 2017.
  - [26] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.
  - [27] Henry Corrigan-Gibbs. *Protecting Privacy by Splitting Trust*. PhD thesis, Stanford University, 2019.
  - [28] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, 2017.
  - [29] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 340–350. ACM, 2010.
  - [30] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.
  - [31] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.
  - [32] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III anonymous remailer protocol. In *2003 Symposium on Security and Privacy*, 2003., pages 2–15. IEEE, 2003.
  - [33] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy Pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.
  - [34] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. *Cryptology ePrint Archive*, 2021.
  - [35] Candice Delmas. The ethics of government whistleblowing. *Social Theory and Practice*, pages 77–105, 2015.
  - [36] Roger Dingledine. BitTorrent over Tor isn't a good idea, Apr 2010. URL <https://blog.torproject.org/bittorrent-over-tor-isnt-good-idea>.
  - [37] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.
  - [38] Emily Dreyfuss. Chelsea Manning walks back into a world she helped transform, 2017. URL <https://www.wired.com/2017/05/chelsea-manning-free-leaks-changed/>.
  - [39] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992. doi: 10.1007/3-540-48071-4\\_10. URL [https://doi.org/10.1007/3-540-48071-4\\_10](https://doi.org/10.1007/3-540-48071-4_10).
  - [40] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
  - [41] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association. URL <https://www.usenix.org/conference/usenixsecurity21/presentation/eskandarian>.

- [42] Nathan S Evans, Roger Dingledine, and Christian Grothoff. A practical congestion attack on Tor using long paths. In *USENIX Security Symposium*, pages 33–50, 2009.
- [43] Cassi Feldman. 60 Minutes’ most famous whistleblower. *CBS News*, 2016. URL <https://www.theguardian.com/world/2010/nov/28/how-us-embassy-cables-leaked>. Accessed March 2022.
- [44] Lorenzo Franceschi-Bicchieri. Snowden’s favorite chat app is coming to your computer. *Vice*, 2015. URL <https://www.vice.com/en/article/signalsnowdens-favorite-chat-app-is-coming-to-your-computer>. Accessed March 2022.
- [45] Anita Gates and Katharine Q. Seelye. Linda Tripp, key figure in Clinton impeachment, dies. *The New York Times*, 2020. URL <https://www.nytimes.com/2020/04/08/us/politics/linda-tripp-dead.html>. Accessed March 2022.
- [46] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer. ISBN 978-3-642-55220-5.
- [47] Robert D’A Henderson. Operation Vula against apartheid. *International Journal of Intelligence and Counter Intelligence*, 10(4):418–455, 1997.
- [48] Nicholas Hopper, Eugene Y Wasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):1–28, 2010.
- [49] Daira Hopwood. Jubjub supporting evidence. <https://github.com/daira/jubjub>, 2017. Accessed March 2022.
- [50] Bastien Inzaurrealde. The Cybersecurity 202: Leak charges against Treasury official show encrypted apps only as secure as you make them. *The Washington Post*, 2018.
- [51] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks: Communications and Multimedia Security, IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS ’99), September 20-21, 1999, Leuven, Belgium*, pages 258–272, 1999.
- [52] Laurie Kazan-Allen. In memory of Henri Pezerat. [http://ibasecretariat.org/mem\\_henri\\_pezerat.php](http://ibasecretariat.org/mem_henri_pezerat.php), 2009. Accessed March 2022.
- [53] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 287–302, 2015.
- [54] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle. *Proceedings on Privacy Enhancing Technologies*, 2016(2):115–134, 2016.
- [55] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 406–422. ACM, 2017.
- [56] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 759–776, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/kwon>.
- [57] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, 2018.
- [58] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: strong metadata security for voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 211–224, 2019.
- [59] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. *ACM SIGCOMM Computer Communication Review*, 43(4):303–314, 2013.
- [60] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 639–652, 2015.
- [61] Jason Leopold, Anthony Cormier, John Templon, Tom Warren, Jeremy Singer-Vine, Scott Pham, Richard Holmes, Azeen Ghorayshi, Michael Salallah, Tanya Kozyreva, and Emma Loop. The FinCEN Files. *BuzzFeed News*, 2020. URL <https://www.buzzfeednews.com/article/jasonleopold/fincen-files-financial-scandal-criminal-networks>. Accessed March 2022.

- [62] Shuai Li, Huajun Guo, and Nicholas Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1977–1992, 2018.
- [63] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honey-BadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.
- [64] Prateek Mittal and Nikita Borisov. ShadowWalker: Peer-to-peer anonymous communication using redundant structured topologies. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 161–172, 2009.
- [65] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 215–226, 2011.
- [66] Jack O’Connor, Samuel Neves, Jean-Philippe Aumasson, and Zooko Wilcox-O’Hearn. BLAKE3: One function, fast everywhere, 2020. URL <https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf>. Accessed March 2022.
- [67] John O’Connor. “I’m the guy they called Deep Throat”. *Vanity Fair*, 2006. URL <https://www.vanityfair.com/news/politics/2005/07/deepthroat200507>. Accessed March 2022.
- [68] Lasse Overlier and Paul Syverson. Locating hidden servers. In *2006 IEEE Symposium on Security and Privacy (S&P’06)*, pages 15–114. IEEE, 2006.
- [69] Paradise Papers reporting team. Paradise Papers: Tax haven secrets of ultra-rich exposed. *BBC News*, 2017. Accessed March 2022.
- [70] D. Phillips. Reality Winner, former NSA translator, gets more than 5 years in leak of Russian hacking report. *The New York Times*, 8, 2019.
- [71] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *26th USENIX Security Symposium USENIX Security 17*, pages 1199–1216, 2017.
- [72] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998.
- [73] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) protocol version 1.3. RFC 1654, RFC Editor, July 1995. URL <https://www.rfc-editor.org/rfc/rfc1654.txt>.
- [74] Charlie Savage. Chelsea Manning to be released early as Obama commutes sentence. *The New York Times*, 17, 2017.
- [75] Michael S Schmidt and LM Steven. Panama law firm’s leaked files detail offshore accounts tied to world leaders. *The New York Times*, 3, 2016.
- [76] Amazon Web Services. Amazon EC2 instance types. <https://aws.amazon.com/ec2/instance-types/>, 2022. Accessed March 2022.
- [77] Scott Shane. WikiLeaks leaves names of diplomatic sources in cables. *The New York Times*, 29:2011, 2011.
- [78] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1928–1943, 2018.
- [79] David Smith. Trump condemned for tweets pointing to name of Ukraine whistleblower. *The Guardian*, 2019. URL <https://www.theguardian.com/us-news/2019/dec/27/trump-ukraine-whistleblower-president>. Accessed March 2022.
- [80] The BBC. Putin critic Navalny jailed in Russia despite protests. URL <https://www.bbc.com/news/world-europe-55910974>. Accessed March 2022.
- [81] The Freenet Project. Freenet, 2020. URL <https://freenetproject.org/>.
- [82] The Invisible Internet Project. I2P anonymous network, 2020. URL <https://geti2p.net/en/>.
- [83] The Tor Project. Tor metrics, 2019. URL <https://metrics.torproject.org/>.
- [84] The Wall Street Journal. Got a tip? <https://www.wsj.com/tips>, 2020. Accessed March 2022.
- [85] Yiannis Tsiounis and Moti Yung. On the security of ElGamal based encryption. In *International Workshop on Public Key Cryptography*, pages 117–134. Springer, 1998.

- [86] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.
- [87] US Holocaust Memorial Museum. Röhm purge. *Holocaust Encyclopedia*, 2020. URL <https://encyclopedia.ushmm.org/content/en/article/roehm-purge>. Accessed March 2022.
- [88] US Occupational Safety and Health Administration. The whistleblower protection program. <https://www.whistleblowers.gov/>, 2020. Accessed March 2022.
- [89] US Securities and Exchange Commission. Office of the whistleblower. <https://www.sec.gov/whistleblower>, 2020. Accessed March 2022.
- [90] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152. ACM, 2015.
- [91] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: using hard AI problems for security. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 294–311, 2003. doi: 10.1007/3-540-39200-9\_18. URL [https://doi.org/10.1007/3-540-39200-9\\_18](https://doi.org/10.1007/3-540-39200-9_18).
- [92] Von Spiegel Staff. Inside the NSA’s war on internet security. *Der Spiegel*, 2014. URL <https://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html>. Accessed March 2022.
- [93] Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New key-recovery attacks on HMAC/NMAC-MD4 and NMAC-MD5. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 237–253. Springer, 2008.
- [94] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.
- [95] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.
- [96] Kim Zetter. Jolt in WikiLeaks case: Feds found Manning-Assange chat logs on laptop. *Wired*, 19 December 2011. URL <https://www.wired.com/2011/12/manning-assange-laptop/>. Accessed March 2022.

## A The audit attack

While many broadcast systems claim privacy with a malicious server, they trade robustness to do so. When a message is *expected*, a server can act as if a user was malicious to prevent aggregation of their request, learning whether that user was responsible for the expected message. If a system aborts in such circumstances, it no longer has the claimed disruption-resistance property. Some systems such as Atom [55] and Blinder [2] solve this by using verifiable secret-sharing in an honest-majority setting; however, this can be costly in practice; others do not prevent this attack.

**Express.** Express is designed for private readers, but it can be trivially adapted for broadcast (see Sections 7 and 8). However, a malicious server can then exploit the verification procedure [41, Section 4.1] to exclude a user, changing their request to an invalid distributed point function. This excludes the message from the final aggregation, deanonymizing a broadcaster with probability at least  $\frac{1}{(1-\epsilon)N}$  per round (where  $\epsilon$  is the fraction of corrupted clients). Even with a few rounds, this can lead to a successful deanonymization of a broadcaster *without detection* (honest servers cannot tell if a server is cheating and therefore cannot abort the protocol).

**Riposte.** The threat model of Riposte does *not* consider attacks in which servers deny a write request. As a result, a malicious server can eliminate clients undetectably by simply computing a bad input to the audit protocol which causes the request to be discarded by both servers. While this attack can be mitigated by using multiple servers and assuming an honest majority (as in Blinder [2]), this weakens the threat model and reduces performance.

**Application of BlameGame.** The BlameGame protocol applies immediately to both Riposte and Express to address this audit attack by allowing (honest) servers to assign blame to either a client or a server if an audit fails. The only cost (as in Spectrum) is a slight increase in communication overhead which, importantly, is independent of the encoded message in the request (see Section 5.1).

## B Large message optimization (multi-server)

In Section 5.1, we give a transformation from a 2-server DPF over a field  $\mathbb{F}$  to a 2-server DPF over  $\ell$ -bit bitstrings that preserves the auditability of the first DPF without increasing the bandwidth overhead proportionally. Here, we show a more

general transformation from  $n$ -server DPFs over a field  $\mathbb{F}$  to  $n$ -server DPFs over a group  $\mathbb{G}_y$  of a polynomially larger order. Our transformation uses a seed-homomorphic pseudorandom generator (PRG) [12].

**Definition 2** (Seed-homomorphic Pseudorandom Generator). *Fix groups  $\mathbb{G}_s, \mathbb{G}_y$  with respective operations  $\circ_s$  and  $\circ_y$ . A seed-homomorphic pseudorandom generator is a polynomial-time algorithm  $G : \mathbb{G}_s \rightarrow \mathbb{G}_y$  with the following properties:*

- Pseudorandom.  $G$  is a PRG:  $|\mathbb{G}_s| < |\mathbb{G}_y|$ , with output computationally indistinguishable from random.
- Seed-homomorphic. For all  $s_1, s_2 \in \mathbb{G}_x$ , we have  $G(s_1 \circ_s s_2) = G(s_1) \circ_y G(s_2)$ .

Let  $\mathbb{G}$  be a group over a field  $\mathbb{F}$  and in which the decisional Diffie-Hellman (DDH) problem [10, 11, 40] is assumed to be hard. Fix some DPF with messages in  $\mathbb{F}$ . We saw in Section 4.2 how to implement anonymous access control for such DPFs. Let  $G : \mathbb{F} \rightarrow \mathbb{G}_y$  be a seed homomorphic PRG where  $\mathbb{G}_y$  is over  $\mathbb{F}$ . Boneh et al. [12] give a construction of such a PRG for  $\mathbb{G}_y = (\mathbb{G})^L$  from the DDH assumption in  $\mathbb{G}$ .

Then, the larger DPF key for a message  $m$  is a DPF key  $k_1$  for a random value  $s \in \mathbb{F}$ , a DPF key  $k_2$  for  $1 \in \mathbb{F}$ , and a “correction message”  $\bar{m} = m \circ_y G(s)^{-1}$  (each key has the *same* correction message). For a zero message, the larger DPF key is two DPF keys  $k_1, k_2$  for  $0 \in \mathbb{F}$  and a random correction message  $\bar{m}$ .

To evaluate the DPF key, the server computes  $s \leftarrow \text{DPF.Eval}(k_1)$ ,  $b \leftarrow \text{DPF.Eval}(k_2)$ , and  $(\bar{m})^b \circ_y G(s)$ . If  $s = 0$ , then combining the DPF keys gives  $(\bar{m})^0 \circ_y G(0) = 1_{\mathbb{G}_y}$ . Otherwise, we get  $(\bar{m})^1 \circ_y G(s) = m$ .

To perform access control for the larger DPF, perform access control for  $k_1$  and  $k_2$  and then also check for the equality of the hashes of  $\bar{m}$ . We note this construction does not yield a new DPF, but does add authorization to a large class of existing DPFs.

## C BlameGame

### C.1 Verifiable Encryption

BlameGame (Section 4.3) uses a verifiable encryption scheme [20], which allows a prover to decrypt a ciphertext  $c$  and create a proof that  $c$  is an encryption of a message  $m$ . We formalize these schemes below:

**Definition 3** (Verifiable Encryption). *A verifiable public-key encryption scheme  $\mathcal{E}$  consists of (possibly randomized) algorithms Gen, Enc, Dec, DecProof, VerProof where Gen, Enc, Dec satisfy IND-CPA security and DecProof, VerProof satisfy the following properties:*

- Completeness. For all messages  $m \in \mathcal{M}$ ,

$$\Pr \left[ \begin{array}{l} (\text{pk}, \text{sk}) \leftarrow \text{Gen}(1^\lambda); \\ c \leftarrow \text{Enc}(\text{pk}, m); \\ (\pi, m) \leftarrow \text{DecProof}(\text{sk}, c); \\ \text{VerProof}(\text{pk}, \pi, c, m) = \text{yes} \end{array} \right] = 1,$$

where the probability is over the randomness of Enc.

- Soundness. For all PPT adversaries  $\mathcal{A}$  and for all messages  $m \in \mathcal{M}$ ,

$$\Pr \left[ \begin{array}{l} (\text{pk}, \text{sk}) \leftarrow \text{Gen}(1^\lambda); \\ c \leftarrow \text{Enc}(\text{pk}, m); \\ (\pi, m') \leftarrow \mathcal{A}(1^\lambda, \text{pk}, \text{sk}, c); \\ \text{VerProof}(\text{pk}, \pi, c, m') = \text{yes} \end{array} \right] \leq \text{negl}(\lambda)$$

for negligible function  $\text{negl}(\lambda)$ , where the probability is over the randomness of Enc and  $\mathcal{A}$ .

We note that many public key encryption schemes (e.g., El-Gamal [40]) satisfy Definition 3 out-of-the-box and can be used to instantiate BlameGame.

## C.2 BlameGame security

The BlameGame protocol must be *sound* and *private*.

*Soundness.* BlameGame is *sound* if no honest client or server will ever be blamed:

1. For all honest commitments  $C_i$ , no probabilistic polynomial-time (PPT) adversary can create a request share  $\tau_i$  and proof of decryption  $\pi_i$  such that the BlameGame “Assigning blame” step (Section 4.3) blames the client when run with  $(\pi_i, \tau_i, C_i)$ .
2. No PPT adversary can create commitments  $C_i$  such that an honestly-created request share  $\tau_i$  and proof of decryption  $\pi_i$  will result in blaming the server after running the BlameGame “Assigning blame” step.

*Privacy.* The privacy requirement of BlameGame is similar to that of Spectrum. Specifically, the commitments  $C_i$  must not reveal any information about the request to any subset of servers. Formally: for randomly sampled pairs of keys  $\text{pk}_i$  and  $\text{sk}_i$  (for  $i \in \{1, \dots, n\}$ ), and all proper subsets  $I \subset \{1, \dots, n\}$ , the following distributions are computationally indistinguishable:

$$\begin{aligned} \{(\text{pk}_i, \text{sk}_i) \mid i \in I, C_i \mid i \in \{1, \dots, n\}\} &\approx_c \\ \{(\text{pk}_i, \text{sk}_i) \mid i \in I, C'_i \mid i \in \{1, \dots, n\}\} \end{aligned}$$

where the  $C_i$  are created by honestly encrypting request shares corresponding to a cover request by a subscriber and the  $C'_i$  are created by honestly encrypting shares corresponding to any valid write generated by a broadcaster.

We note that BlameGame does **not** require any privacy properties during blame assignment, as it may reveal the request for the purpose of assigning blame.

We now show that BlameGame achieves these properties:

*Proof.* We prove each property in turn.

**Soundness (honest client).** Suppose, toward contradiction, that there exists a PPT adversary  $\mathcal{A}$  that generates some request shares  $\tau_i$  and proof of decryption  $\pi_i$  such that BlameGame blames the client. This means that (1) the decryption proof verification succeeds, and (2) running the audit with the request shares failed. By property (1), we can assume that  $\tau_i$  is a correct decryption of  $C_i$  and  $\pi_i$  is a valid proof of decryption; otherwise,  $\mathcal{A}$  breaks the soundness property of the verifiable encryption scheme. However, we know that running the audit with the given request shares will pass, because (by assumption) they were created honestly by the client. This is a contradiction.

**Soundness (honest server).** Let  $\tau_i$  be a set of request tokens such that the Spectrum audit fails when run with  $\tau_i$ . Suppose, toward contradiction, that some client creates commitments  $C_1, \dots, C_n$  for  $\tau_1, \dots, \tau_n$  such that the BlameGame “Assigning blame” step blames some server (instead of the client, as required). Then, it holds that either (1) the proof of decryption failed, or (2) the audit performed by the servers over the decrypted requests passes. However, if (1) is true (the proof of decryption failed), then the completeness property of the verifiable encryption scheme does not hold (because the request share and proof of decryption are generated honestly by the server). Therefore, we are left with (2); the audit performed by the servers over the decrypted request shares

passes. However, this isn’t true (by assumption) if the client is malicious. Hence, we have a contradiction.

**Privacy.** For all honest broadcasters, privacy is guaranteed with probability  $\frac{L}{N \cdot (1-\epsilon)}$  where  $\epsilon$  is the fraction of corrupted clients. If the first audit fails but the second audit (generated from the decrypted requests) passes, then privacy follows from the analysis of Spectrum and privacy of the audit therein. If the second audit fails, then the request is revealed to both servers for inspection (in order to adequately assign blame). However, predicated on the revealed request being generated correctly (since we are interested in when an *honest* broadcaster gets deanonymized), the protocol aborts if the second audit fails (an honest broadcaster would have encrypted the request correctly). In this case, all servers see the request which deanonymizes the client. Thus, for a fraction of corrupted clients  $\epsilon$ , the probability that the malicious server chooses the correct request to tamper with before being aborted is  $\frac{L}{N \cdot (1-\epsilon)}$ .  $\square$

Spectrum (with BlameGame) achieves our desired security properties: a malicious client cannot cause disruption, and a malicious server cannot deanonymize a broadcaster. Because BlameGame is sound, if all servers are honest then Spectrum does not abort (because either the audit passes, or BlameGame blames the client); this prevents disruption due to audit failure. The second property follows from the privacy of BlameGame.

# Donar: Anonymous VoIP over Tor

Yérom-David Bromberg, Quentin Dufour, Davide Frey  
*Univ. Rennes - Inria - CNRS - IRISA, France*

Etienne Rivière  
*UCLouvain, Belgium*

## Abstract

We present DONAR, a system enabling anonymous VoIP with good quality-of-experience (QoE) over Tor. No individual Tor link can match VoIP networking requirements. DONAR bridges this gap by spreading VoIP traffic over *several* links. It combines active performance monitoring, dynamic link selection, adaptive traffic scheduling, and redundancy at no extra bandwidth cost. DONAR enables high QoE: latency remains under 360 ms for 99% of VoIP packets during most (86%) 5-minute and 90-minute calls.

## 1 Introduction

Tor [20] is by far the largest anonymization network with over 6,000 relay nodes distributed worldwide. Tor has been very successful for applications such as web browsing with, e.g., TorBrowser, but is generally considered inadequate for latency-sensitive applications [31,66]. Voice-over-IP (VoIP) is one such application that has become the *de facto* solution for global voice calls. Being able to deploy VoIP over Tor would immediately benefit privacy-conscious users by enabling simple, efficient, and safe voice communication answering two objectives: (i) protecting the content of the communication from adversaries, i.e., using end-to-end encryption, and (ii) hiding metadata and in particular the identity of communicating partners. Metadata may, indeed, be used to infer private information, e.g., uncovering a journalist’s sources [27] or illegally gathering information about employees [60].

Providing good-quality interaction between VoIP users, i.e., a good Quality-of-Experience (QoE), requires good network Quality-of-Service (QoS) and in particular low and stable latency [28, 35, 68], as we detail in **Section 2**. This comes in tension with the way Tor is designed [31,66]: Tor links<sup>1</sup> implement multi-hop communication for TCP traffic using *onion routing* over pre-established circuits formed of several relays, which leads to high and unstable latencies. Surprisingly, Sharma *et al.* [70] recently posited that using Tor *as is* would

be sufficient to obtain the stable and low latencies required by high-QoE VoIP. This statement is, unfortunately, incorrectly grounded. Three biases in their analysis led to this conclusion: (1) they only consider average latencies, while VoIP QoE is primarily determined by tail latency (99th percentile with a standard codec) [57], (2) they measure performance for only 30 seconds, a much shorter duration than an average call [33], and (3) they only consider the case of one-way anonymity, i.e., when the callee is not anonymous. We present in **Section 3** our analysis of Tor links’ performance considering these elements and conclude that the use of a Tor link *as is* does not, in fact, allow VoIP with sufficient QoE.

TorFone [24] attempts to overcome Tor latency issues by duplicating VoIP traffic over two statically chosen links. However, even if going in the right direction, TorFone’s strategy turns out to be ineffective due to the large variability of link performance over time, as we demonstrate in **Section 6**.

Alternative anonymization networks targeting voice communication were also proposed recently, e.g., Herd [50] and Yodel [49]. These systems, however, are yet to be deployed and need to reach a sufficient scale to be efficient, i.e., to provide sufficient bandwidth for a large number of geographically-distributed users.

**Motivations.** We are interested in providing VoIP support over a *readily-available* anonymization network. More specifically, we target a deployment using (1) legacy VoIP applications and (2) the existing, unmodified Tor network. We do not wish to propose design changes to Tor, or a novel anonymity network [49, 50, 64, 73, 76], and neither do we want to overcome Tor’s existing security flaws. We believe that these lines of work are, in fact, orthogonal to our own.

While our observation of the performance of Tor (presented in **Section 3**) confirms that a *single* Tor link cannot provide the stable and low latency required by high-QoE VoIP, it also allows us to make a case for dispatching traffic over *multiple* links. Unlike TorFone, our strategy multiplexes traffic over a *dynamically selected* set of Tor links using a smart scheduling mechanism. Our motivation is that the use of multiple dynamically and adequately chosen links, together with controlled

<sup>1</sup>We use in this paper the generic term *link* to denote the unidirectional TCP channel that is exposed to applications by the Tor client.

and smart content redundancy, can mask the transient faults and latency spikes experienced by individual links.

**Contributions.** We present the design and implementation of DONAR, a user-side proxy interfacing a legacy VoIP application to the existing Tor network ([Section 4](#)).

DONAR enforces *diversity* in the paths used for transmitting VoIP packets, i.e., using distinct Tor links. In addition, it leverages *redundancy* by sending the same VoIP packet several times using different links. This redundancy does not, in fact, add additional bandwidth costs for the Tor network beyond those incurred by the setup and maintenance of these multiple links. We leverage, indeed, the fact that Tor only transmits 514-Byte cells over the network to protect users against traffic analysis [[53, 59](#)]. DONAR takes advantage of the available padding space to re-transmit previous VoIP packets. Diversity and redundancy mask the impact of the head-of-line blocking implied by the TCP semantics of Tor links, whereby an entire stream of packets may get delayed by a single belated one.

DONAR builds on the following key technical components:

- The *piggybacking* of VoIP packets in the padding space of Tor cells enables redundancy without incurring additional bandwidth costs on the Tor network.
- A *link monitoring* mechanism observes and selects appropriate links, switching rapidly between them when detecting performance degradation.
- Two *scheduling* strategies for selecting links when transmitting VoIP packets enable different tradeoffs between cost and robustness.

We further analyze in [Section 5](#) how attacks on Tor can affect the security properties of DONAR. In particular, we discuss how different DONAR configurations implement different tradeoffs between Quality-of-Experience and security.

We evaluate DONAR over the Tor network and present our findings in [Section 6](#). We use VoIP-traffic emulation as well as the off-the-shelf `gstreamer` [[26](#)] VoIP client using the OPUS [[14](#)] audio codec. We assess the performance of DONAR against the VoIP requirements detailed in [Section 2](#) and compare it with the approach followed by TorFone [[24](#)]. Our results show that DONAR, using alternatively 6 out of 12 carefully monitored and dynamically selected onion links, achieves high QoE with latency under 360 ms and less than 1% of VoIP frame loss for the entire duration of a large number (86%+) of 5-minute and 90-minute calls, with no bandwidth overhead for its optimized configuration (i.e., alternate sending over different links) and an overhead similar to that of TorFone for its default configuration.

We detail related work and conclude in [Sections 7 and 8](#).

## 2 VoIP networking requirements

DONAR aims at Providing good Quality-of-Experience (QoE) for anonymous VoIP while limiting the costs imposed on the

Metric	Objective
Dropped calls rate	$\leq 2\%$ for 90-minute calls
Packet loss rate	$\leq 1\%$
Bandwidth	$\geq 32 \text{ kbps} (4.3 \text{ kB/s})$
One way delay (99th perc. <i>ideal</i> )	$\leq 150 \text{ ms} - T_{\text{frame}} - T_{\text{buffer}}$
One way delay (99th perc. <i>max</i> )	$\leq 400 \text{ ms} - T_{\text{frame}} - T_{\text{buffer}}$

Table 1: VoIP network performance requirements, following the recommendations of the International Telecommunication Union [[35](#)] and applying them to the OPUS codec [[74, 75](#)].

Tor infrastructure. We base our analysis of QoE requirements on recommendations by the International Telecommunication Union (ITU) [[35–37](#)]. The ITU defines good QoE as the combination of the following guarantees: (1) uninterrupted calls, (2) good voice quality, and (3) support for interactive conversations. We analyze these requirements and derive our network QoS objectives, summarized in [Table 1](#).

**VoIP protocols.** VoIP requires two types of protocols. A signaling protocol such as the Session Initiation Protocol (SIP) [[67](#)] makes it possible to locate a correspondent and negotiate parameters for the communication. The signaling protocol only impacts QoE with delays upon the establishment of the call. When the call is established, a protocol such as the UDP-based Real-time Transport Protocol (RTP) [[69](#)] is used to transmit VoIP audio frames encoded using a codec, whose configuration is negotiated by the signaling protocol. QoE is primarily impacted by this codec and its ability to deal with hazards in network QoS, as we detail next.

**Impact and choice of the audio codec.** Bandwidth, latency, or maximum packet loss requirements depend on the audio codec used by the VoIP application. We base our analysis on the state-of-the-art open audio codec OPUS, which we also use in our evaluations. OPUS is a widely-used, loss-tolerant audio codec developed by the Xiph.Org Foundation and standardized by the IETF [[74, 75](#)]. It targets interactive, low-delay communication and computational efficiency. OPUS has been consistently ranked in comparative studies as the highest-quality audio format for low and medium bit rates [[32, 41](#)]. We emphasize that our analysis would be similar for other open codecs, e.g., the Internet Low Bit Rate Codec (iLBC) [[4](#)] or Xiph.Org Foundation’s former codecs Vorbis [[7](#)] and Speex [[30](#)].

**First guarantee: no call interruption.** A call interruption is the most impacting event on user-perceived QoE. The ITU does not provide a recommendation for general networks but recommends at most 2% dropped calls for VoIP over 4G [[37](#)]. We adopt the same goal but need to define a time span on which to evaluate this metric. Holub *et al.* [[33](#)] provided us with a dataset of more than 4M call durations ([Figure 1](#)). Its analysis confirms that call duration follows a log-normal distribution considered as standard for voice calls. We observe an average call duration slightly above 3 minutes, with 90% of

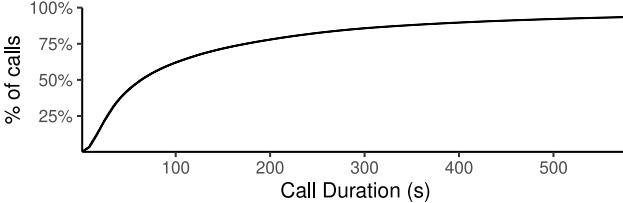


Figure 1: Call Duration ECDF on a 4M calls dataset provided by Holub *et al.* [33] zoomed on the first 10 minutes.

calls lasting less than 10 minutes. However, we still observe 1,040 calls lasting for 90 minutes or more which is characteristic of the long tail of the distribution. As this value matches the limitation set by major representative carriers [34, 77], we evaluate reliability for calls over this duration.

**Second guarantee: good voice quality.** Users want to clearly hear their communication partners. Voice quality depends both on the bitrate being used and the amount of packet loss: (1) Listening tests with OPUS [14, 32] concluded that a bitrate of 32 kbps is sufficient to offer a sound quality that test users cannot distinguish from a reference unencoded version of the recording. We set, therefore, this bitrate as the minimum required bandwidth that we must offer to the VoIP application. (2) OPUS provides two mechanisms to mask the impact of lost packets: a domain-specific one, named Packet Loss Concealment (PLC) and a generic one, via redundancy, named Forward Erasure Coding (FEC)<sup>2</sup> [72]. Han *et al.* [28] studied the perceived quality of a call on various packet rates. This study shows that while PLC compensates for packet loss, the perceived voice quality nonetheless decreases quickly: a 1% packet loss is essentially unnoticed, while 10% packet loss results in usable but degraded call conditions. Based on these results, we set as a requirement a packet loss of at most 1%.

**Third guarantee: interactive conversations.** In addition to an uninterrupted and good-quality voice signal, users of voice calls expect to be able to exchange information interactively, e.g., be able to seamlessly synchronize on when to stop and start talking in a conversation.

Interactivity primarily depends on latency [68]. The ITU published recommendation G.114 [35] on mouth-to-ear latency in voice calls. This recommendation indicates that a delay below 150 ms is unnoticeable for users, compared to a direct voice conversation. We set, therefore, this value as our ideal latency. On the other hand, the recommendation stipulates that delays must remain below 400 ms to make an interactive call possible under good conditions. Higher latencies result in synchronization difficulties and significantly reduce user-perceived QoE. We set this threshold of 400 ms as our maximum acceptable mouth-to-ear latency.

We emphasize that the actual network latency for transmitting VoIP frames is only a subset of mouth-to-ear latency.

<sup>2</sup>We configure OPUS to use only the former, as DONAR already enables redundancy mechanisms that are specific to the Tor network.

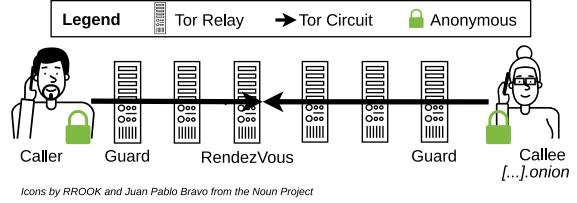


Figure 2: Structure of a Tor link with onion services.

Additional latency is introduced by (1) audio capture and playing, (2) packetization, and (3) buffering. Once digitized, audio is encapsulated, every  $T_{\text{frame}}$  ms, into frames that will form a packet. OPUS enables configurable values for  $T_{\text{frame}}$  from 2.5 to 60 ms.

We consider an ideal jitter-buffer model similar to the one by Moon *et al.* [57]. This model delays all frames by the maximum or  $n$ th percentile of the observed latency and allows frame drops. Moon *et al.* [57] and others [44, 52] have proposed jitter-buffer implementations performing close to this theoretical optimum. Therefore, we consider  $T_{\text{buffer}}$ , the unnecessary delay added by a wrong jitter-buffer configuration as negligible. Finally, as we allow a 1% frame drop, we consider the 99th latency for our mouth-to-ear delay constraints.

### 3 VoIP over Tor: How bad is it?

Tor [20] is a large-scale network that enables users to access remote resources anonymously. Tor relies on *onion routing*: it relays traffic through *circuits* consisting of at least two relays (three by default) chosen from more than 6,000 dedicated nodes. The first relay in a circuit is known as the *Guard*. The Tor client chooses a small set of  $n$  (by default<sup>3</sup>,  $n = 2$ ) possible guards. Thereafter, it builds circuits by using one guard from this set, choosing the remaining relays randomly from the list of all available relay nodes.

Tor enables both connections to the regular Internet (referred to as *Exit*) and to other Tor users (referred to as *Onion Services*). In contrast to the *Exit* mode, *Onion Services* provide two-way anonymity by default. The Tor client on the caller's side connects to an anonymous onion service (set up by the callee's Tor client). In doing so, it creates a Tor route, i.e., the concatenation of two Tor circuits, one from the caller to a rendezvous relay, and another from the callee to the same rendezvous relay. In this paper, we use the term *link* to refer to the TCP connection over this route that the Tor client exposes to the application. Figure 2 illustrates a Tor link based on an onion service used for transmitting VoIP frames.

Tor seeks to prevent adversaries from inferring communicating parties. To this end, at least one relay in the route should lie in an administrative domain that the adversary cannot observe. Furthermore, to prevent traffic analysis attacks, Tor only sends fixed-sized messages between relays, in the

<sup>3</sup>While Tor advertises using  $n = 1$  by default, it effectively uses  $n = 2$  [62].

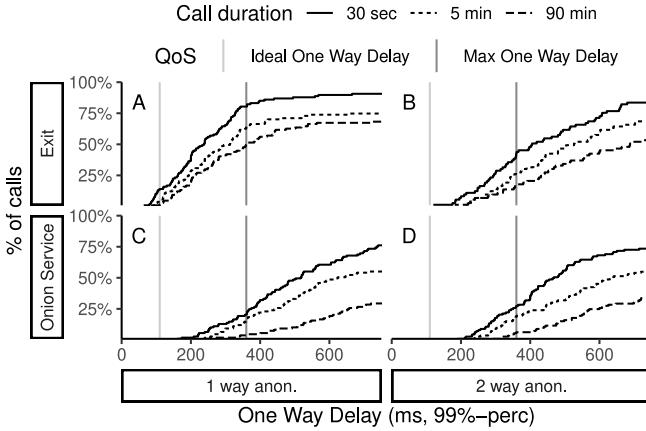


Figure 3: VoIP over a single Tor link: Evolution of the one-way delay’s 99th percentile according to connection-link type and call duration.

form of 514-Byte *cells* [53, 59]. When a packet being transmitted over a Tor link is less than 514 Bytes in size, the Tor client pads it with random data.

### 3.1 Evaluation of Tor links’ QoS

Tor is often described as a *low-latency* anonymization network. Its TCP streams over pre-established circuits enable, indeed, lower latency than anonymization networks where the relays for each message in a stream are chosen independently [12, 73, 76]. The latency of links in Tor, and in particular their stability, is however known to be unpredictable, which made several authors doubt Tor’s ability to support low-latency applications such as VoIP [31, 66].

In this section, we report on our own experimental evaluation of the network QoS of Tor links. We confirm the observation made by other authors that a single Tor link is unsuitable for VoIP networking requirements as defined in the previous section. However, these measurements allow us to make the case for the foundational design choice in DONAR: using several, dynamically selected links.

We consider the following metrics: the connection stability, the variability of one-way latency, and the predictability of high latency from prior measurements. We use a load injector with varying packet-sending rates and, in order to measure one-way latency, a stub communication endpoint located on the same machine. The injector and the stub use two separate instances of the Tor client in its default configuration and create circuits independently. All reported experiments were conducted in January 2021.

**Connection links.** We start by analyzing how the two Tor modes, *Exit* and *Onion Services*, perform in terms of tail latency. Each of these modes can be declined in links providing either one-way or two-way anonymity. *Exit* links provide one-way anonymity by default but we can mimic two-way anonymity by making both caller and callee access the same

public VoIP server. *Onion-Service* links provide two-way anonymity by default but we can reduce the number of relays and keep only one-way anonymity. We use the `HIDDENSERVICESINGLEHOPMODE` feature in the Tor daemon to achieve one-way anonymity over Onion Services.

Considering these 4 configurations, we simulated VoIP calls lasting 30 seconds, 5 minutes, and 90 minutes. The simulation strictly follows the requirements presented in Section 2. For each combination of configuration and call duration, we made 64 calls and present the results in Figure 3.

We start our analysis by focusing on Figure 3.A as it features the configuration on which Sharma *et al.* [70] base their claim that Tor links are suitable *as is* to support VoIP. With 37% of unacceptable calls (resp. 50%) for 5-minute (resp. 90-minute) calls, we argue the opposite. We identified three reasons explaining why our analysis differs. (1) They do not account for  $T_{frame}$  in their analysis. Since we use  $T_{frame} = 40$  ms, our max acceptable latency is 360 ms. (2) They consider average latencies instead of the 99th percentile of their distribution. While we obtain similar average latencies, considering tail latency shows that 20% of calls suffer from unacceptable delays, even for short 30-second calls. (3) They consider only such 30-second calls when the average call duration is 3 minutes and when a significant share of calls last up to 90 minutes. Measuring links over a longer timespan shows, in fact, that latencies tend to increase with call duration.

Comparing the different configurations we observe that, in fact, no link type offers acceptable delays. We note (Figure 3.{B,D}) that the latency benefits from using the *Exit* mode mostly vanish when considering 2-way anonymity. Using one-way anonymity with the *Onion Service* mode (Figure 3.C) does not seem to improve tail latency; we presume this is because this feature is still experimental.

Moreover, not all link types are equal: using *Exit* links has two drawbacks. First, it requires the last relay of the circuit to hold the *Exit* tag. As *Exit* links can send data on the regular Internet, the last relay is particularly sensitive: only 25% of the relays accept to have this position. From the user’s perspective, this situation eases de-anonymization attacks and, by limiting the scalability of the network, also harms performances. Moreover, using *Exit* links requires relaying traffic through an ad-hoc public server that must be trusted (e.g., Sharma *et al.* [70] use Mumble and Freeswitch PBX).

Considering that (i) no link type over Tor enables VoIP, and (ii) the *Exit* mode has severe limitations, we choose to focus solely on leveraging *Onion-Service* links to provide anonymous voice calls in the rest of this paper.

**Connection stability.** We evaluate the reliability of each Tor link type over our longest considered call duration (90 minutes). Figure 4 reports the cumulative rate of failed links (i.e., for which packets are no longer transmitted) as a function of time. After 10 minutes, all link types exhibit failure rates of at least 4%. The rate rises to between 7% and 16% after one hour. The failure difference between link types seems

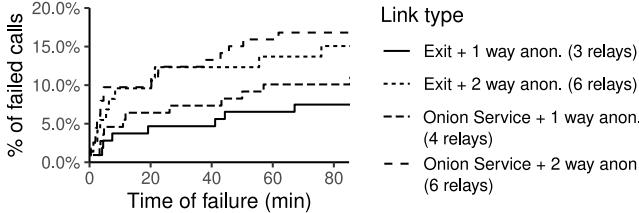


Figure 4: Failed Tor links over time.

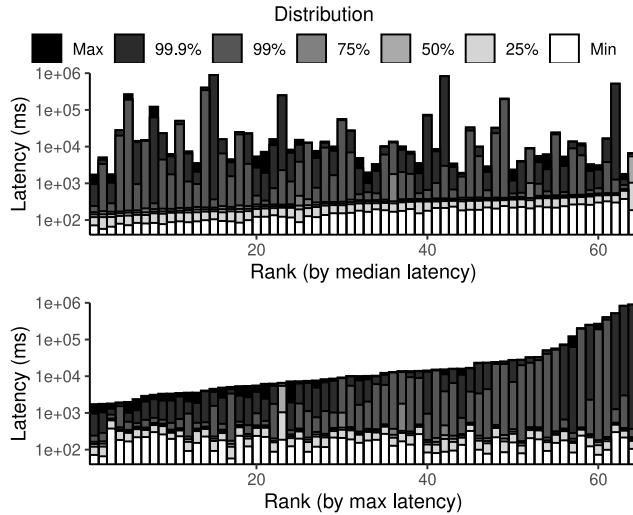


Figure 5: Tor links’ latency distribution at 25 packet/sec ordered by median (top) and max (bottom) latency.

to be correlated with their number of relays: the more there are relays, the more likely failures are. None of the available links satisfies our QoS requirements: we need a solution that overcomes link breakage and allows calls to continue seamlessly.

**Predictability of high latencies.** The previous experiment shows that the distribution of latency across multiple links is highly skewed. We now evaluate if this skew results from a large number of poorly performing links with a few, identifiable, good links, or if any link can experience periodic latency bursts. Figure 5 presents the one-way-latency distribution for each of the 64 links, ranked by median latency (top) or max latency (bottom). There is no clear relationship between the general performance of a link and the occurrence of latency spikes. The maximal latency does not seem to depend much on the rest of the distribution and can reach very high values in all cases (often 3 times higher than the 75th percentile)<sup>4</sup>. We refer to these high latency periods as *latency spikes* in the rest of this paper.

**Discussion.** Our experiments confirm the general unpredictability of the performance of Tor links. Due to Tor’s ex-

<sup>4</sup>This unpredictable performance is confirmed, in fact, by a blog post by the Tor project [63]. We quote: “While adding more relays to the network will increase average-case Tor performance, it will not solve Tor’s core performance problem, which is actually performance variance.”

clusive support for TCP<sup>5</sup>, latency spikes for a single packet result in high latency for all following packets, delayed to be delivered in order—a phenomenon referred to as *head-of-line blocking*.

We observe, however, that the number of relays correlates with the probability of networking problems: larger numbers of relays are associated with higher failure rates or with latency spikes. We also note that most links provide good performance for a fraction of their use time, and failures across links do not seem to be correlated. As a result, we make the case for using multiple links, benefiting from periods of good performance, and quickly switching links when experiencing latency spikes.

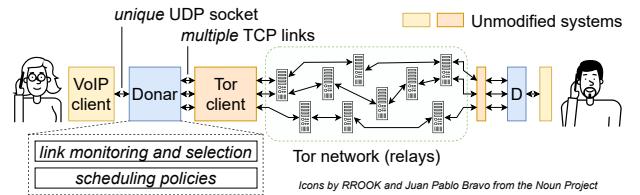


Figure 6: DONAR plays the role of a proxy between an unmodified VoIP application and the unmodified Tor client.

## 4 Donar: Enabling VoIP over Tor

DONAR operates as a proxy between a VoIP application and the Tor client, as illustrated in Figure 6. It does not require modifying either of the two systems. DONAR runs without any specific privileges; it only offers a UDP socket to the VoIP application’s RTP protocol and opens TCP sockets (links) with the local Tor client. In conformance with our objective to make anonymous VoIP available with *readily-available* systems, we do not require the deployment of external support services. In particular, DONAR does not rely on the SIP signaling protocol but leverages instead Tor onion addresses to establish communication without leaking metadata about communicating parties.

**Redundancy by piggybacking.** DONAR leverages the fact that Tor only transmits data in the form of fixed-sized cells. Setting OPUS to the target bitrate of 32 kbps and using a sending period of 40 ms results in 172-Byte frames. The Tor client pads the remaining space with random data to reach a cell size of 514 Bytes. DONAR leverages, instead, this space to re-send the previous frame without changing the necessary bandwidth requirements<sup>6</sup>. Naturally, a redundant frame must

<sup>5</sup>TCP maps well to an efficient implementation of onion routing, i.e., it makes it possible to know when to create and dispose of circuits and it avoids the presence of packets that are untied to an existing circuit. UDP would also pose security challenges, e.g., enable DDoS attacks. The designers of Tor have clearly dismissed any support of UDP in Tor in the future [56].

<sup>6</sup>We are not limited to this configuration, and only require that the size of the frames emitted by the codec be less than half the available space minus the Tor headers (8 Bytes) and DONAR metadata (38 Bytes in the default configuration), i.e., less than 233 Bytes.

be sent on a different link than the first copy, to avoid head-of-line blocking between replicas. While redundant frames are subject to an additional  $T_{\text{frame}}$  latency (40 ms in the presented configuration), our rationale is that this latency combined with that of the link itself will still be lower than that of a link experiencing a latency spike. We detail next how we effectively enable link diversity.

**Link Diversity.** DONAR leverages multiple Tor Onion-Service links to multiplex traffic in two complementary ways. First, it spreads frame copies onto different links. This prevents packets containing subsequent frames from being subject to the same latency spike thereby arriving too late in a burst at the destination. This also lowers the load on each individual link (resulting, as shown in Section 3, in better availability). Second, DONAR ensures that the first and the second (redundant) copy of a given frame always travel on different links.

Enabling diversity requires (1) maintaining a set of open links and monitoring their performance; and (2) implementing a scheduling policy for selecting appropriate links for new packets. In the following, we detail these two aspects (§4.1 and §4.2) and complete the description of DONAR by detailing how calls are established (§4.3).

## 4.1 Link monitoring and selection

DONAR opens and monitors a set of Tor links and associates them with scores reflecting their *relative* latency performance. We start by detailing how latency scores are collected at the local client’s side, and why they must also be collected from the remote client. We motivate our choice to classify links in performance groups, and how we dynamically select links in these groups throughout a call.

**Measuring latency.** Measuring transmission delays for packets sent over Tor is not straightforward. The RTP protocol uses UDP and does not send acknowledgments. We do not wish to add additional acknowledgment packets over Tor to measure round-trip times, as their padding in 514-Byte cells would double bandwidth consumption.

Rather than attempting to measure the *absolute* latencies of links, we leverage the use of multiple links to approximate their *relative* latency performance. Measures of performance are continuously collected on both sides of the communication, which we denote as node A and node B in the following. Local *aggregate* measures are then computed over a time window of duration  $w$ . We explore the impact of durations ranging from 0.2 to 32 seconds in our evaluation.

We base our measurements on an *out-of-order* metric for VoIP frames. This metric denotes, for an incoming frame  $f$  with sequence number  $i$ , the number of frames received *before*  $f$  with a higher sequence number than  $i$ . As TCP delivers packets in order, these frames necessarily travel on different links. For instance, if node A receives frame  $f$  with sequence number  $i$  from node B on link  $l$  after receiving frames with

sequence numbers  $i + 1$ ,  $i + 2$ , and  $i + 3$  on other links, we associate an out-of-order metric of 3 to frame  $f$ .

The local calculation of the out-of-order metric also applies to *missing* frames. Node A is aware of any *missing* frame  $f_m$  with a sequence number  $i_m < i_c$  where  $i_c$  is the largest sequence number among all the frames received from node B. However, since the decision on which link a packet is sent is made by node B, it is not possible for node A to assign  $f_m$ ’s measurement to a specific link. To solve this problem, we include, in the DONAR headers in each packet, the list of links used for sending the latest  $n$  frames, where  $n$  is the maximum number of used links.

Nodes A and B must share their local aggregate measures to enable fast detection of latency spikes. Node A’s local information about a link  $l$  approximates, indeed, the one-way latency from B to A, but not from A to B. Our experimental evaluation has shown that one-way latencies are highly consistent in both directions of a link, making node A’s local estimation a good approximation also for the latency from A to B. However, this local approximation may be missing if the link has not been used recently by B to send packets to A. We alleviate this problem by embedding, in the DONAR metadata sent with each packet, the local aggregate measures for links that have been measured recently. Node A computes a final array of measures that include, for each link, either (1) the local aggregate measure only, if no remote aggregate was received; (2) the remote aggregate only, if the link was not recently used by B to send data to A; or (3) the average of these two measures if the link was used in both directions.

**Link selection.** Every  $w$  seconds, DONAR sorts links in decreasing order of aggregated scores over the last period and assigns links to three groups. The  $L_{1\text{ST}}$  (first-class) group contains the  $n_{1\text{ST}}$  *fastest* links. The  $L_{2\text{ND}}$  (second-class) group contains the  $n_{2\text{ND}}$  following links. Typically, we use the same number of links in the two groups, i.e.,  $n_{1\text{ST}} = n_{2\text{ND}}$ . Finally, the remaining  $n_{\text{INACTIVE}} = n_{\text{LINKS}} - n_{1\text{ST}} - n_{2\text{ND}}$  slowest links are assigned to the  $L_{\text{INACTIVE}}$  group.

The rationale for this classification is as follows. Links in the  $L_{\text{INACTIVE}}$  group experience sub-par performance and must remain idle. Links in the  $L_{1\text{ST}}$  group have good performance and are invaluable in allowing fast delivery of VoIP packets. However, the number of good-performing links is limited at a given point in time, and using them systematically bears the risk of overloading them, resulting in lower performance and reliability (§3). Links in the  $L_{2\text{ND}}$  group are less performant, but remain usable, and can reduce this risk of overload.

**Links opening and maintenance.** DONAR uses standard operations of the Tor client to open links. It lets the client select relays according to Tor rules. The client allows users to parameterize the number of used guard relays, as well as the length of the links (number of relays). DONAR leverages these parameters to enable different security/performance tradeoffs. We defer the discussion of strategies for setting these values and their security implications, to Section 5.

When starting a call, DONAR opens  $n_{\text{LINKS}} = n_{1\text{ST}} + n_{2\text{ND}} + n_{\text{INACTIVE}}$  links and assigns them randomly to the three groups. When the Tor client notifies a link failure, DONAR simply requests a new link and assigns it to the  $L_{\text{INACTIVE}}$  group.

Links in the  $L_{\text{INACTIVE}}$  group will not be monitored locally. Some of these links may be associated with a remote score, but others will not be monitored on either side of the call. To enable *all* links to be monitored periodically, we implement a promotion and demotion mechanism between the  $L_{2\text{ND}}$  and  $L_{\text{INACTIVE}}$  groups. When assigning links to groups at the end of a period of  $w$  seconds, DONAR picks the worst-performing link from the  $L_{2\text{ND}}$  group and demotes it to the  $L_{\text{INACTIVE}}$  group. In return, it promotes to  $L_{2\text{ND}}$  the link from the  $L_{\text{INACTIVE}}$  group that has been unused for the longest time.

## 4.2 Scheduling policies

The DONAR scheduler receives UDP RTP packets containing a single frame from the VoIP application. It first implements redundancy by using the pad space to piggyback packets that were previously sent on different links, then adds the necessary metadata, and finally creates a TCP packet to be sent onto one or two links from the  $L_{1\text{ST}}$  and/or  $L_{2\text{ND}}$  groups.

DONAR’s default scheduling policy is named ALTERNATE. It sends each new packet to a *single* link. In doing so, it *alternates* between links from the  $L_{1\text{ST}}$  and  $L_{2\text{ND}}$  groups. This complies with the requirement to send the first and redundant copies of a frame on different links. DONAR picks the links from each group using a round-robin policy, thereby complying with the requirement of maximizing diversity.

We implement a second policy named DOUBLE-SEND. As the name implies, this policy selects *two* links—one from  $L_{1\text{ST}}$  and one from  $L_{2\text{ND}}$ —for sending each new packet. Each frame is received four times: two as a primary copy, and two as a duplicate. This policy doubles the required bandwidth but has a higher chance to select a fast link for the primary copy of a frame, thereby reducing the risk of delivering the frame with an additional delay of  $T_{\text{frame}}$ . We note that the resulting bandwidth is the same as for TorFone [24]’s Duplication mode, which systematically sends VoIP packets onto the same two links.

## 4.3 Establishing communication

DONAR leverages Tor’s mechanisms to allow callers and callees to establish a connection anonymously. Following our design goal of using only readily-available systems, we do not require the deployment of an existing or novel signaling protocol and, in particular, we do not use a SIP deployment. SIP requires, in fact, the use of trusted proxies and has been documented as leaking metadata to network observers [21, 43]. Furthermore, with the exception of the audio codec negotiation, SIP functionalities largely overlap mechanisms already offered by Tor [21, 43].

A caller can discover a callee by looking up a specific onion service identifier using the Tor DHT. This onion service identifier is obtained by other means, e.g., by using an anonymous chat service. The identifier can also be public while still preserving anonymity, as Tor prevents an external observer from determining that a specific client opens a circuit to a specific onion service. For instance, journalists could advertise an anonymous onion service for whistleblowers to use. We note that client-side authorization, as defined in the Tor rendezvous specification [54], could enable a callee to only allow calls from a whitelist of callers, but we leave the integration of this functionality to future work.

In the current DONAR implementation, the codec and its configuration are hardcoded. Codec and configuration negotiation require, unlike discovery, only communication between the two parties, and could employ a protocol similar to the subset of SIP dedicated to this task. We also leave this implementation to future work.

## 5 Security

DONAR leverages Tor without deploying additional infrastructure or modifying Tor itself. As a result, DONAR inherits the security assumptions and shortcomings of Tor. For instance, like Tor, DONAR does not provide protection from adversaries that can control the *entire* network [20, 59] to perform traffic-correlation attacks [40, 82]. Nevertheless, in terms of guarantees, it is reasonable to wonder whether DONAR worsens the security properties of Tor and to what extent.

In the definition of the so-called predecessor attack, Wright [82] observed that repeatedly creating new circuits causes clients to continuously degrade their security while increasing the probability that they will eventually choose a malicious relay as the first node of a circuit. Wright [81] proposed to address this problem by using what is now known as guards. Specifically, each Tor client chooses a small number of guards and uses them for all the circuits it ever creates. This suggests that DONAR’s impact on security depends mainly on the fact that it can use a larger number of guards than the standard Tor implementation. We evaluate this impact from the perspective of three threats: (1) one endpoint deanonymizing the other, (2) an attacker that controls some relays or ASes and that tries to identify DONAR users, and (3) the same attacker deanonymizing both endpoints of a call and finally breaking anonymity.

**Deanonymizing the other endpoint.** According to the AnoA classification [6], sender/recipient anonymity refers to the ability to hide one endpoint’s identity from the other. As discussed by Wright *et al.* [81], in a system with  $c$  corrupted relay nodes out of  $n$  and 1 guard per user, the probability of an endpoint’s de-anonymizing the other is  $\frac{c}{n}$ . If we increase the number of guards to  $g$ , this probability becomes  $1 - (1 - \frac{c}{n})^g$ , which, for small values of  $\frac{c}{n}$ , can be approximated from above

by its first-order Taylor/Maclaurin expansion  $g_n^{\frac{c}{n}}$ . Like most previous work, this analysis focuses on a random distribution of compromised guards. Adversaries can also leverage path selection algorithms to strategically place malicious guards and increase their probability of being selected although countermeasures exist [78].

**Identifying DONAR users.** Identifying a DONAR endpoint is equivalent to de-anonymizing any onion service, i.e., identifying which client node is reachable through this service. An adversary controlling a guard relay and knowing the onion address of a callee may observe traffic and employ traffic fingerprinting techniques [10, 45, 55, 61, 65] or use a fake DONAR client and perform timing attacks [58] to identify that a specific client is accepting DONAR calls. The use of several ( $g$ ) guards in DONAR also increases the probability of this attack to  $1 - (1 - \frac{c}{n})^g$ , and thus by a factor of  $g$  for small values of  $\frac{c}{n}$ , like for the de-anonymization of one endpoint. This attack can however be mitigated by using the client-authorization feature offered by V3 Onion Services [54]. Finally, while several authors have shown that an adversary could locate onion service endpoints even when they were not publicly advertised [9, 45, 55, 61], they have also proposed solutions to the Tor community.

**De-anonymizing an ongoing call.** To de-anonymize an ongoing call, an attacker needs to control guard nodes at both endpoints and employ traffic-correlation techniques [40]. As a result, like for the first two threats, the choice of the number of guards used by DONAR identifies a tradeoff between the likelihood of this attack and the performance of a call. In particular, since the attacker needs to control at least one guard on each side of the call, the associated probability grows from  $(\frac{c}{n})^2$  with one guard to  $(1 - (1 - \frac{c}{n})^g)^2$  with  $g$  guards. This implies that it grows even more slowly for small values of  $\frac{c}{n}$  than the two previous probabilities.

Finally, we also observe that passive traffic correlation attacks turn out to be more difficult to perform when multiple calls are ongoing as DONAR’s traffic patterns do not vary between different calls. In this case, a passive attack must observe the start and/or the end of a call to be effective.

### DONAR security configurations.

As discussed above, increasing the number of guards improves performance but it also increases the attack surface. For this reason, DONAR implements three security configurations that strike different tradeoffs between privacy and performance, as illustrated in Figure 7. We emphasize that each configuration sets up the unmodified Tor client via its legacy API. DONAR systematically uses 12 links, but link settings are different in each configuration. The *Default* configuration provides a security strength similar to the legacy Tor client with default Tor link settings, i.e., each link has 6 relays, and each client employs only 2 guards.<sup>7</sup> The *2-hop* configuration

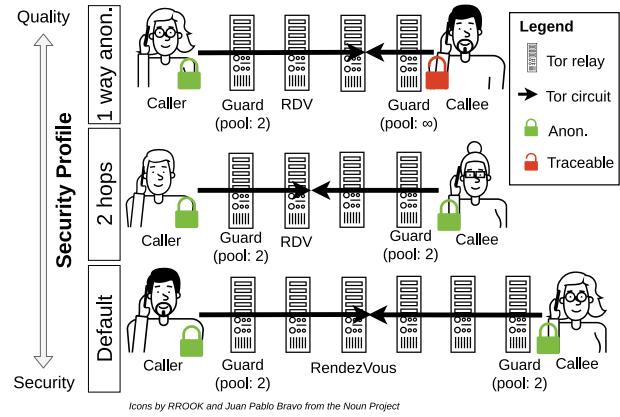


Figure 7: Security configurations.

sets up the Tor client so that each created link has two fewer Tor relays compared to Tor’s default link settings. Finally, the *1-way-anonymity* configuration totally removes the anonymity of the callee using a *single* Tor relay (without the guard pool constraint) between the callee and the rendezvous point.

**Security Discussion.** Each of the threats we identified above relies on the control of at least one guard relay per affected endpoint. As discussed above, DONAR does not modify the guard configuration when providing anonymity for a user. Moreover, the use of guards decorrelates the number of links and the de-anonymization probability: using 12 links at once does not expose a user more than using only one. Additionally, compared to the *Default* configuration, the *2-hop* one reduces the number of relays in links by two. Decreasing the number of relays in links has been long debated in the Tor community. The main rationale for using 3-relay circuits (and thus 6-relay links) is that it makes it more difficult for an adversary that controls the last relay to identify the entry guard. On the other hand, an adversary can overcome this protection with relatively low investment in additional relays, and 3-relay circuits are more vulnerable to attacks based on denial of service [8]. These observations motivate our choice of 2-relay circuits with better latency in our *2-hop* configuration. Finally, the *1-way-anonymity* configuration does not provide anonymity to the callee but does not hamper the anonymity of the caller. Moreover, this mode is a standard feature of the Tor daemon that is used in production (e.g., by Facebook [1]).

Finally, we emphasize that DONAR users may also explore entirely different security configurations, by changing the number of Tor guards and/or relays for links, according to their own expected tradeoffs between performance and security.

<sup>7</sup>Even though Tor’s documentation discusses using only one guard, the default client uses two.

## 6 Evaluation

DONAR is available open-source at <https://github.com/CloudLargeScale-UCLouvain/Donar>. The DONAR proxy interfaces a VoIP application with the Tor client.<sup>8</sup> We use two applications: (1) a configurable RTP emulator allowing a fine-grained control on the frames sent between parties, and running multiple occurrences of an experiment to study statistical variations; and (2) the actual gstreamer VoIP application using the OPUS codec. We deploy two isolated instances of either application on the same machine to accurately measure one-way delays for packets sent over Tor.

Tor's performance varies over time, with failures, disconnections, and latency spikes as identified in Section 3. Unless mentioned otherwise, we run each experiment a total of 64 times and present the distribution of results. We run the same configuration over a long time span, typically 5 hours, and also compare different configurations running in parallel.

### 6.1 Performance & comparison to SOTA

We start with the evaluation of the global performance of DONAR and its ability to meet the requirements summarized in Table 1. We use an audio stream of 32 kbps with a rate of 25 frames per second. We configure DONAR as follows: The window duration is  $w = 2s$  and we open a total of  $n_{\text{LINKS}} = 12$  links including  $n_{\text{1ST}} = 3$  links,  $n_{\text{2ND}} = 3$  links, and  $n_{\text{INACTIVE}} = 6$  links. We present a comprehensive analysis of the influence of these parameters in Section 6.2.

We consider the six possible variants of DONAR using either of the two scheduling policies ALTERNATE and DOUBLE-SEND combined with one of the three security configurations (*Default*, *2 hops*, or *1-way anonymity*). In addition, we implement two approaches representing the state of the art. SIMPLE is the direct use of a single Tor link to transfer VoIP data. It represents our reference in terms of bandwidth usage for the ALTERNATE policy. TORFONE implements the duplication strategy used in TorFone [24]: It sends each new packet on two links, representing a reference for bandwidth usage for the DOUBLE-SEND policy.

**No call interruption.** We start by studying the percentage of dropped calls for all configurations. We run 96 instances of a 90-minute call for each combination and count the percentage of dropped calls. For SIMPLE, a broken Tor link invariably results in a dropped call. The DONAR variants and TORFONE, instead, re-establish broken links and thus consider their calls dropped whenever they miss 25 consecutive frames. Figure 8 presents the results. All DONAR variants perform better than the previous approaches and meet the goal of less than 2% of dropped calls. We only record, in fact, dropped calls for the most conservative of our setups, i.e., combining the ALTERNATE policy with the *default* configuration, and even then not

<sup>8</sup>The Tor software is evolving quickly, especially considering v3 onions. To benefit from latest bug fixes, we compiled Tor from branch `maint-0.4.4` (commit 09a1a34ad1) and patch #256.

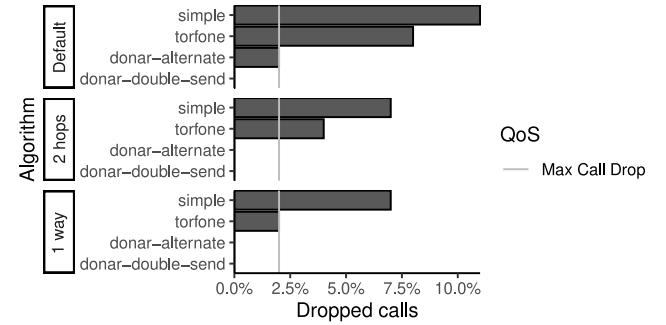


Figure 8: Dropped calls after 90 minutes for SIMPLE, TORFONE, and DONAR setups.

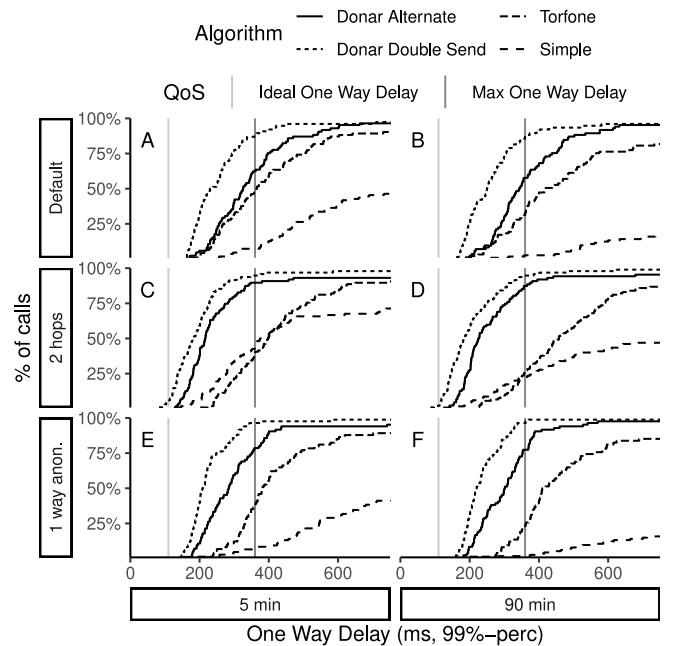


Figure 9: Latency comparison between SIMPLE, TORFONE, DONAR ALTERNATE and DONAR DOUBLE-SEND.

exceeding 2%. TORFONE only meets the goal in the *1-way anonymity* configuration.

**Interactive conversations & good voice quality.** These objectives require a sufficient bitrate—met by using a 32 kbps bitrate in our experiments—and receiving at least 99% of VoIP frames within the maximum acceptable latency. The OPUS codec can, indeed, mask the loss of 1% of the frames with no perceptible quality degradation.

We present the distributions of frame delivery latencies in Figure 9. Our mouth-to-ear latency objective is 150 ms, and our limit is 400 ms. As  $T_{\text{frame}}=40$  ms, we wish network delays for delivering frames to be of 110 to 360 ms. We use two vertical lines to denote these boundaries.

For all security policies and call durations, the DONAR DOUBLE-SEND algorithm provides at least 87% (*Default*, 90 minutes) of successful calls. Considering only our optimized security policies, the ratio of successful calls is even higher at

95%. These results must be compared to TORFONE, as both approaches send the same amount of data on the wire. TORFONE enables as low as 23% (*1-way anon.*, 90 minutes) and at most 47% (*Default*, 5 minutes) of successful calls. Compared to DONAR DOUBLE-SEND’s worst performance (*Default*, 90-minute configuration), there is a 55-point difference with TORFONE in favor of DONAR.

Conversely, we observe that DONAR ALTERNATE does not fit all configurations: for its *Default* security policy, it enables only 62% (resp. 57%) of successful calls for 5 minutes (resp. 90 minutes). Results are better with *1-way anon.*: 78% (resp. 77%) for 5-minute (resp. 90-minute) calls. However, only the *2-hop* configuration seems to offer acceptable quality, enabling at least 87% of successful calls. Compared to the SIMPLE mode that sends the same amount of data, this is a 55-point gain points compared to DONAR’s worst performance. With the *2-hop* configuration, it is a 43 points (resp. 65 points) for 5-minute (resp. 90-minute) calls improvement on SIMPLE.

To conclude, DONAR DOUBLE-SEND is able to offer a high ratio of successful calls in most situations (87%+ compared to 23%+ for TORFONE); it is a versatile solution at the cost of added redundancy on the wire. In comparison, DONAR ALTERNATE has no overhead but is way more sensitive to the configuration: it only works well with the *2-hop* security policy (87%+ compared to 46%+ for SIMPLE). With a difference of at most 4% between the 5-minute and 90-minute measurements, DONAR adds a new interesting property: latency stability over time. We argue that our two sending policies represent a significant improvement in terms of delay compared to the state of the art.

**Using the `gstreamer` VoIP client.** We experiment with the replay of an audio file using the `gstreamer` VoIP application. We collect statistics about its jitter buffer. `gstreamer` only allows a static-size jitter buffer. We configure this buffer based on our previous experiments, so as to absorb latencies between the minimum observed latency and the 99th-perc. latency, and count the number of calls that systematically meet latency requirements out of the 64 experiments done for each configuration. Our results confirm that DONAR DOUBLE-SEND is able to meet the 360 ms latency threshold for most experiments in all configurations, while the ALTERNATE policy works best under the *2-hop* configuration. We also confirmed empirically the results obtained under the *2-hop* configuration and the two scheduling policies by performing actual calls between two laptops: we could not detect any degradation in sound quality throughout any of the calls.

## 6.2 Microbenchmarks

In the following, we present an analysis of the influence of each of DONAR’s parameters, and of the complementarity of its mechanisms. We focus on the six possible DONAR variants and, to factor out the impact of security configurations, we also consider a version of DONAR using 4 relays per link and an unlimited number of guards.

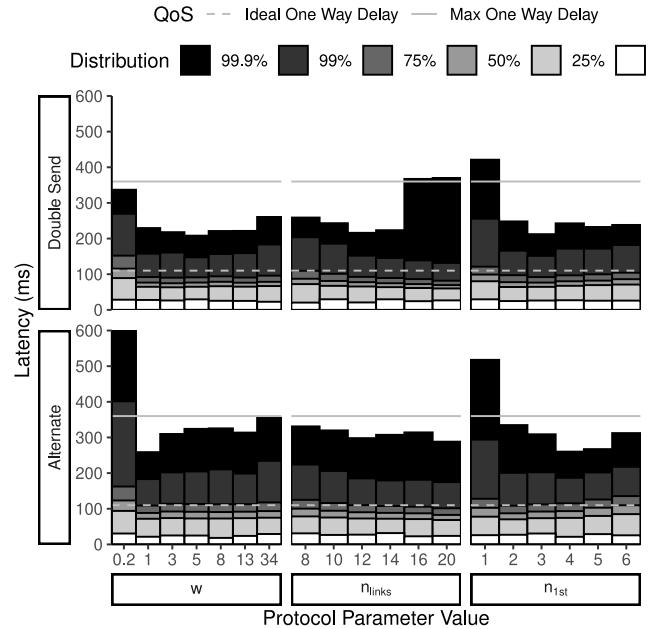


Figure 10: Impact of protocol parameters ( $w$ ,  $n_{\text{LINKS}}$  and  $n_{1\text{ST}} = n_{2\text{ND}}$ ) on frame delivery latencies.

**Protocol parameters.** DONAR has 3 main parameters:  $w$ ,  $n_{\text{LINKS}}$ ,  $n_{1\text{ST}}$  (we use  $n_{1\text{ST}} = n_{2\text{ND}}$ ). In the experiments reported in the previous section, we employed the default values of  $w = 2s$ ,  $n_{\text{LINKS}} = 12$ , and  $n_{1\text{ST}} = n_{2\text{ND}} = 3$ . We detail in the following how we selected this default configuration.

We present, in Figure 10, an analysis of the influence of each parameter on the distribution of frame delivery latencies. Parameter  $w$  determines how far in the past we consider out-of-order metrics when computing link scores. It also determines how many times we need to probe a link before deciding to stop using it. A lower value of  $w$  enables a fast reaction at the risk of switching too many links with unreliable scores, while a larger value promotes links that are stable over time. We can observe on the left side of Figure 10 that the best value of  $w$  for the DOUBLE-SEND policy is 5s, while the best for ALTERNATE appears to be 2s. Additional benchmarks on the [1, 8] range with a smaller step led us to select the latter value as the default.

The  $n_{\text{LINKS}}$  parameter controls the total number of open links and, therefore, both the level of achievable diversity and the load of route maintenance on the Tor network. We evaluate  $n_{\text{LINKS}}$  values from 8 to 20. The ALTERNATE policy performs best with 20 links, while the DOUBLE-SEND policy performs best with 12 links. To limit the load on Tor, we select this latter value as the default.

Finally, parameter  $n_{1\text{ST}} = n_{2\text{ND}}$  directly controls the number of links that are actively used to send packets. On the one hand, for a given value of  $n_{\text{LINKS}}$ , a small value of  $n_{1\text{ST}}$  increases the likelihood of selecting only good-performing links. On the other hand, a large value increases diversity and the frame rate on each link, resulting in higher stability as we have shown

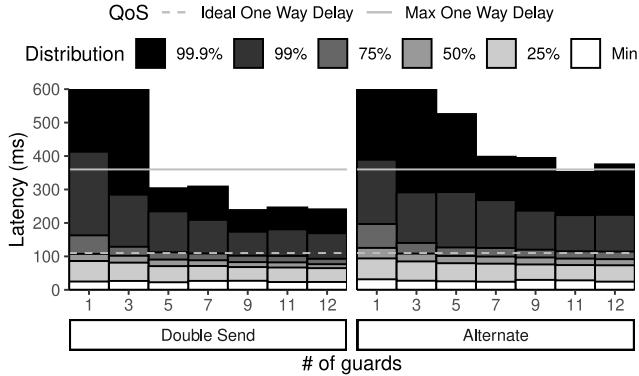


Figure 11: Impact of Tor guards number on latencies.

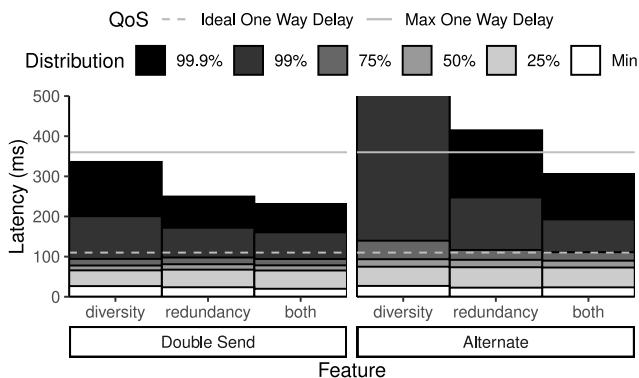


Figure 12: Diversity & redundancy complementarity.

in Section 3. Using  $n_{1\text{ST}} = 1$  yields high latencies with either variant, while  $n_{1\text{ST}} = 3$  or  $n_{1\text{ST}} = 4$  offers a good compromise. We choose  $n_{1\text{ST}} = 3$  as our default value.

**Impact of the size of the guard pool.** We considered using different sizes for the guard pool for the different security configurations detailed in Section 5. We further explore the impact of this parameter on DONAR’s performance. Our results, shown in Figure 11, confirm that, in order to achieve the best latency, it is preferable to have as many guards as the number of links, in our case  $n_{\text{LINKS}} = 12$ . This number is, however, the result of a compromise with the attack surface. In our performance evaluation, we chose to stay conservative by not modifying the number of guards but we demonstrate here this choice has a cost in terms of performance.

**Complementarity of diversity and redundancy.** We analyze to which extent the two enabling mechanisms of DONAR, diversity and redundancy, contribute to its performance. We present in Figure 12 latency when using only link selection (diversity), using only redundancy by piggybacking, and using both. Activating both features is clearly beneficial for both scheduling policies, but, unsurprisingly, the impact of redundancy by piggybacking on high percentiles of the distribution is larger for the ALTERNATE strategy than for the DOUBLE-SEND strategy, as the latter enables redundancy by sending packets twice.

We further wish to understand how diversity and redundancy interact when used simultaneously, by analyzing, for each frame, which group of links delivers it for the first time, and whether this first delivery concerns a primary or a duplicate copy. The first delivery of a frame, indeed, results from a *race* between two send operations (with the ALTERNATE policy) and four send operations (with DOUBLE-SEND).

When using the ALTERNATE policy, 94% of the primary frame copies sent on a link of the  $L_{1\text{ST}}$  group arrive first. In 6% of the cases, the first copy that is received is the duplicate sent 40 ms later over an  $L_{2\text{ND}}$  link. When primary frame copies are sent over  $L_{2\text{ND}}$  links, however, only 48% arrive before the duplicate copy sent over an  $L_{1\text{ST}}$  link; 52% of the frames arrive first as the duplicate copy, despite the latter being sent 40 ms later. When using the DOUBLE-SEND policy, 73% of the frames are received first as a primary copy on the  $L_{1\text{ST}}$  link, 14% are received as a primary copy on an  $L_{1\text{ST}}$  link, and only 13% are received as a duplicate copy. Using  $L_{2\text{ND}}$  links remains useful. It provides more diversity, while still leveraging the reliability of the best links. Moreover, it decreases the load on each individual link, reducing the risk of performance degradation on each of them.

**Link monitoring effectiveness.** Appendix A presents a supplementary study of the effectiveness of link monitoring, where we analyze a trace of link classification and selection.

## 7 Related Work

VoIP over anonymization networks poses significant challenges as it combines the need for strong security with low and stable latency requirements. Three main families of anonymization networks have emerged: onion-route-, mix-net- and DC-net-based networks. The former do not protect from global adversaries that control the entire network whereas the latter two do.

The objective of DONAR is to leverage a readily-available system. Only two anonymity networks satisfy this requirement, Tor and Vuvuzela [76]. We discuss, nonetheless, the practicability of VoIP over a larger set of existing approaches, even if they are not effectively deployed.

**Onion-route-based networks.** Sharma *et al.* [70] called to re-think the feasibility of voice calling over Tor and claim that VoIP is feasible over Tor. However their analysis suffers from several shortcomings: they consider average latency instead of tail latency, they do their measurements only for 30 seconds, they do not evaluate dropped calls, they only provide one-way anonymity, etc. Karopoulos *et al.* [21, 43] explore the porting of SIP infrastructures on Tor. The main principle of their work is to preserve privacy in the SIP signalling protocol, in contrast with DONAR that leverages Tor’s built-in mechanisms for establishing calls. The RTP stream is transmitted using a single Tor onion link. This approach behaves like SIMPLE from our experimental evaluation in this respect. TorFone [24] tries to improve latency by duplicating traffic over only two onion

links without any scheduling and monitoring mechanisms. As demonstrated in Section 6, the TORFONE policy is not sufficient to meet VoIP requirements.

**Mix-net-based networks.** Mix networks [13] batch and shuffle packets via *mix nodes* to prevent attackers from performing global traffic analysis. However, in doing so, they inherently incur high latency, which makes them unusable in latency-sensitive applications. A key solution to reduce packet delivery times consists in using cover traffic to prevent the mixes from having to wait too long before having enough packets to send a batch. Accordingly, the challenge faced by the latest research on mix-nets, such as Karaoke [48], Vuvuzela [76], Riffle [46], Loopix [64], Aqua [51], and Stadium [73], consists in designing an adequate mix-net with the best tradeoff between minimizing the necessary cover traffic while guaranteeing good resilience to traffic analysis. In their best-case usage scenario, these approaches drastically reduce latency from several hundred seconds to a few seconds, but this remains very far from VoIP requirements.

**DC-net-based networks.** Latency can be reduced by avoiding batching. Instead of using mix nodes, Dining-Cryptographer Networks (DC-nets) rely on anonymous broadcast among all network participants [13]. DC-nets have two inherent shortcomings: (i) they incur a high bandwidth overhead, i.e., the number of messages exchanged to send one message anonymously grows quadratically with the number of network participants, and (ii) they are vulnerable to denial of service attacks from malicious participants that can jam the whole network. Being resistant to such attacks requires, for instance, the use of zero-knowledge proofs to detect misbehavior but this is very costly in computation and results in increased delivery latency [25]. Consequently, a number of research works on DC-nets have emerged in recent years. Dissent [16, 80], Riposte [15], and Verdict [17] resist jamming attacks while trying to provide the best tradeoff between reducing the number of exchanged messages (e.g. by splitting the network into smaller parts) and the impact of computational cost on latency. However, despite their efforts, their latency remains far too high for VoIP and increases with the number of users.

**Anonymization networks designed for VoIP.** Herd [50] is based on the mix-net principle. It was specifically designed for VoIP. Its hybrid approach uses mix nodes along with super peers organized in trust zones. Herd can provide VoIP on its anonymity network with good resistance against global adversaries. Its evaluation shows expected latency values of 400 ms. The recent work on Yodel [49] removes the concept of trust zones and supports higher percentages of dishonest nodes than Herd. However, this comes at the cost of latency increasing with the probability of having dishonest mix nodes. For instance, in a Tor-like environment (i.e.,  $\sim 20\%$  of malicious servers) latency already reaches  $\sim 900$  ms. To counterbalance this latency, Yodel uses a codec with poorer quality than OPUS. Even if both Herd and Yodel are promising de-

signs, neither is currently deployed. Today's whistleblowers are, therefore, unable to communicate using these systems. Moreover, we point out that the evaluation of both systems has been performed in optimal conditions, and their performance in settings comparable to Tor's deployment remains unstudied. For instance, Yodel is evaluated on 100 powerful Amazon EC2 servers with no external interference. DONAR, on the other hand, satisfies VoIP latency requirements, even if Tor constantly relays traffic generated by over 2 million daily users. To summarize, Tor and Vuvuzela represent the only anonymization networks that are readily available and widely deployed today. Since Vuvuzela cannot support VoIP due to its high latency, DONAR over Tor represents the only solution that enables privacy-conscious users to communicate anonymously using VoIP and with a good QoE.

**Latency improvements on Tor.** We also reviewed existing proposals to improve latency in Tor. This latency depends on two main factors: (i) queuing delays (time spent in a relay), and (ii) transmission delays (time spent on the "wire", between two Tor relays). Ting [11] and LASTor [2] both reduce transmission delays by modifying the path selection algorithm. However, latency spikes are due to queuing delays [19], particularly because Tor does not perform any centralized load balancing of traffic. To reduce queuing delays, improved traffic scheduling policies have been integrated in the latest versions of Tor [38, 39], but we still observe latency spikes. Alternative path selection algorithms use historical data on relay performance [71, 79] or probe circuits upon their creation [5]. They are inefficient for VoIP as latency spikes are ephemeral; predictions are outdated after a few seconds.

**Multipath.** We are not the first to advocate for multipath. MORE [47] proposes to route independently each cell, but it is not designed to be used with circuits like in Tor. MPTCP [22, 23, 29] aggregates TCP links over multiple network interfaces. However, it makes assumptions (e.g., latency is independent of traffic) that do not hold over Tor. In response, dedicated multipath protocols specially tailored for onion routing networks [3, 18, 42, 83] emerged. Nevertheless, compared to DONAR, none of these approaches optimize tail latency as required by all real-time protocols, including VoIP.

## 8 Conclusion

We presented DONAR, a solution for readily-available, anonymous, and high-quality VoIP calls using the challenging but existing Tor network. DONAR circumvents Tor's inability to support the networking requirements of VoIP by sending audio frames over a diversity of links and using redundancy. It offers different tradeoffs between performance and security and successfully enables high-quality VoIP calls, e.g., with latency below 360ms during an entire 90-minute call.

**Acknowledgments:** We are thankful to the anonymous reviewers and to our shepherd, Harsha V. Madhyastha, for their constructive feedback. This work was partially funded by the O'Browser ANR grant (ANR-16-CE25-0005-03).

## References

- [1] Tor project issue trackers: Facebook’s onion site is a single hop onion, but clicking on the Tor onion icon shows that it is a 6 hop circuit (issue #23875). <https://gitlab.torproject.org/legacy/trac/-/issues/23875>.
- [2] Masoud Akhoondi, Curtis Yu, and Harsha V Madhyastha. LASTor: A low-latency AS-aware Tor client. In *IEEE Symposium on Security and Privacy*, S&P, 2012.
- [3] Mashael AlSabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. The path less travelled: Overcoming Tor’s bottlenecks with traffic splitting. In *International Symposium on Privacy Enhancing Technologies*, PETs. Springer, 2013.
- [4] Soren Vang Andersen, Alan Duric, Henrik Astrom, Roar Hagen, W. Bastiaan Kleijn, and Jan Linden. Internet Low Bit Rate Codec (iLBC). Request for Comments (RFC) 3951, Internet Engineering Task Force (IETF), December 2004.
- [5] Robert Annessi and Martin Schmiedecker. Navigator: Finding faster paths to anonymity. In *European Symposium on Security and Privacy*, EuroS&P. IEEE, 2016.
- [6] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. AnoA: A framework for analyzing anonymous communication protocols. In *26th Computer Security Foundations Symposium*, CSF. IEEE, 2013.
- [7] Luca Barbato. RTP payload format for Vorbis encoded audio. Request for Comments (RFC) 5215, Internet Engineering Task Force (IETF), August 2008.
- [8] Kevin Bauer, Joshua Juen, Nikita Borisov, Dirk Grunwald, Douglas Sicker, and Damon McCoy. On the optimal path length for Tor. In *3rd Hot Topics in Privacy Enhancing Technologies*, HotPETS, 2010.
- [9] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Trawling for tor hidden services: Detection, measurement, deanonymization. In *Symposium on Security and Privacy*, S&P. IEEE, 2013.
- [10] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM conference on Computer and communications security*, CCS, 2012.
- [11] Frank Cangialosi, Dave Levin, and Neil Spring. Ting: Measuring and exploiting latencies between all tor nodes. In *Internet Measurement Conference*, IMC, 2015.
- [12] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), 1981.
- [13] David L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1), 1988.
- [14] Opus Codec. Codec landscape. <https://opus-codec.org/comparison/>, 2020.
- [15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Symposium on Security and Privacy*, S&P. IEEE, 2015.
- [16] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *17th ACM conference on Computer and communications security*, CCS, 2010.
- [17] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in Verdict. In *22nd USENIX Security Symposium*, 2013.
- [18] Vladimir De la Cadena, Daniel Kaiser, Asya Mitseva, Andriy Panchenko, and Thomas Engel. Analysis of multi-path onion routing-based anonymization networks. In *IFIP Annual Conference on Data and Applications Security and Privacy*, DBSec. Springer, 2019.
- [19] Prithula Dhungel, Moritz Steiner, Ivinko Rimac, Volker Hilt, and Keith W Ross. Waiting for anonymity: Understanding delays in the Tor overlay. In *10th International Conference on Peer-to-Peer Computing*, P2P. IEEE, 2010.
- [20] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.
- [21] Alexandros Fakis, Georgios Karopoulos, and Georgios Kambourakis. OnionSIP: Preserving privacy in SIP with onion routing. *J. Univers. Comput. Sci.*, 23(10), 2017.
- [22] Alexander Froemgen, Jens Heuschkel, and Boris Koldehofe. Multipath TCP scheduling for thin streams: Active probing and one-way delay-awareness. In *International Conference on Communications*, ICC. IEEE, 2018.
- [23] Alexander Frommgen, Tobias Erbshäuser, Alejandro Buchmann, Torsten Zimmermann, and Klaus Wehrle. ReMP TCP: Low latency multipath TCP. In *International Conference on Communications*, ICC. IEEE, 2016.

- [24] Van Gezel. TORFone: secure VoIP tool. <http://torfone.org/>, 2013.
- [25] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004.
- [26] GStreamer. Gstreamer: open source multimedia framework. <https://gstreamer.freedesktop.org/>, 2020.
- [27] Ben Doherty (The Guardian). Vodafone australia admits hacking fairfax journalist's phone. <https://www.theguardian.com/business/2015/sep/13/vodafone-australia-admits-hacking-fairfax-journalists-phone>, 2015.
- [28] Yi Han, Damien Magoni, Patrick Mcdonagh, and Liam Murphy. Determination of bit-rate adaptation thresholds for the opus codec for voip services. In *Symposium on Computers and Communications*, ISCC. IEEE, 2014.
- [29] Mark Handley, Olivier Bonaventure, Costin Raiciu, and Alan Ford. TCP extensions for multipath operation with multiple addresses. Request for Comments (RFC) 6824, Internet Engineering Task Force (IETF), January 2013.
- [30] G. Herlein, J. Valin, A. Heggestad, and A. Moizard. RTP payload format for the Speex codec. Request for Comments (RFC) 5574, Internet Engineering Task Force (IETF), June 2009.
- [31] Stephan Heuser, Bradley Reaves, Praveen Kumar Pendyala, Henry Carter, Alexandra Dmitrienko, William Enck, Negar Kiyavash, Ahmad-Reza Sadeghi, and Patrick Traynor. Phonion: Practical protection of metadata in telephony networks. *Proceedings on Privacy Enhancing Technologies*, 2017(1), 2017.
- [32] Christian Hoene, Jean-Marc Valin, Koen Vos, and Jan Skoglund. Summary of Opus listening test results. <https://tools.ietf.org/html/draft-ietf-codec-results-03>, 2013.
- [33] Jan Holub, Michael Wallbaum, Noah Smith, and Hakob Avetisyan. Analysis of the dependency of call duration on the quality of VoIP calls. *IEEE Wireless Communications Letters*, 7(4):638–641, 2018.
- [34] Monty Icenogle. T-mobile does have a hard 4 hour single call duration limit. <https://kd6cae.livejournal.com/271120.html>, 2015.
- [35] ITU. ITU-T recommendation G.114, "one way transmission time". <https://www.itu.int/rec/T-REC-G.114>, 2003.
- [36] ITU. E.800 : Definitions of terms related to quality of service, 2008.
- [37] ITU. G.1028: End-to-end quality of service for voice over 4G mobile networks. <https://www.itu.int/rec/T-REC-G.1028>, 2019.
- [38] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport. In *23rd USENIX Security Symposium*, 2014.
- [39] Rob Jansen, Matthew Traudt, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. Kist: Kernel-informed socket transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1–37, 2018.
- [40] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *ACM SIGSAC conference on Computer & communications security*, CCS, 2013.
- [41] kamedo2. Results of the public multiformat listening test. <https://listening-test.coresv.net/results.htm>, 2014.
- [42] Hasan T Karaoglu, Mehmet Burak Akgun, Mehmet Hadi Gunes, and Murat Yuksel. Multi path considerations for anonymized routing: Challenges and opportunities. In *5th International Conference on New Technologies, Mobility and Security*, NTMS. IEEE, 2012.
- [43] Georgios Karopoulos, Alexandros Fakis, and Georgios Kambourakis. Complete SIP message obfuscation: Priv-aSIP over Tor. In *9th International Conference on Availability, Reliability and Security*. IEEE, 2014.
- [44] Byeong Hoon Kim, Hyoung-Gook Kim, Jichai Jeong, and Jin Young Kim. VoIP receiver-based adaptive play-out scheduling and packet loss concealment technique. *IEEE Transactions on consumer Electronics*, 59(1):250–258, 2013.
- [45] Albert Kwon, Mashaal AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *24th USENIX Security Symposium*, 2015.
- [46] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proceedings on Privacy Enhancing Technologies*, 2016(2):115–134, 2016.
- [47] Olaf Landsiedel, Alexis Pimenidis, Klaus Wehrle, Heiko Niedermayer, and Georg Carle. Dynamic multipath

- onion routing in anonymous peer-to-peer overlay networks. In *IEEE Global Telecommunications Conference, GlobeCom*, 2007.
- [48] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.
- [49] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: strong metadata security for voice calls. In *27th ACM Symposium on Operating Systems Principles, SOSP*, 2019.
- [50] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for voip systems. In *ACM Conference on Special Interest Group on Data Communication, SIGCOMM*, 2015.
- [51] Stevens Le Blond, David Choffnes, Wenzuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. *ACM SIGCOMM Computer Communication Review*, 43(4):303–314, 2013.
- [52] Yi J Liang, Nikolaus Farber, and Bernd Girod. Adaptive playout scheduling and loss concealment for voice communication over IP networks. *IEEE Transactions on Multimedia*, 5(4):532–543, 2003.
- [53] Zhen Ling, Junzhou Luo, Wei Yu, and Xinwen Fu. Equal-sized cells mean equal-sized packets in Tor? In *International Conference on Communications, ICC*. IEEE, 2011.
- [54] Nick Mathewson, George Kadianakis, David Goulet, Tim Wilson-Brown, Hans-Christoph Steiner, Filipo Valsorda, and Roger Dingledine. Tor rendezvous specification - version 3, 2017.
- [55] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *18th ACM conference on Computer and communications security, CCS*, 2011.
- [56] Nick Montfort, Arthur Edelstein, Robert Ransom, and Yawning Angel. Tor project feature tracker: Closed enhancement, “UDP over Tor”. <https://trac.torproject.org/projects/tor/ticket/7830>, 2013.
- [57] Sue B Moon, Jim Kurose, and Don Towsley. Packet audio playout delay adjustment: performance bounds and algorithms. *Multimedia systems*, 6(1):17–28, 1998.
- [58] Steven J Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *Symposium on Security and Privacy, S&P*. IEEE, 2005.
- [59] Steven J Murdoch, Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router (2013 DRAFT v1). <https://gitweb.torproject.org/tor-design-2012.git/>, 2014.
- [60] David Kaplan (Newsweek). Suspicious and spies in silicon valley. <https://www.newsweek.com/suspicious-and-spies-silicon-valley-109827>, 2006.
- [61] Andriy Panchenko, Asya Mitseva, Martin Henze, Fabian Lanze, Klaus Wehrle, and Thomas Engel. Analysis of fingerprinting techniques for Tor hidden services. In *Workshop on Privacy in the Electronic Society*, 2017.
- [62] Mike Perry. The move to two guard nodes. <https://gitweb.torproject.org/user/mikeperry/torspec.git/tree/proposals/xxx-two-guard-nodes.txt?h=twoguards>, 2018.
- [63] Mike Perry. Tor’s open research topics: 2018 edition | tor blog. <https://blog.torproject.org/tors-open-research-topics-2018-edition>, 2018.
- [64] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *26th USENIX Security Symposium*, 2017.
- [65] Tobias Pulls and Rasmus Dahlberg. Website fingerprinting with website oracles. *Proceedings on Privacy Enhancing Technologies*, 2020(1):235–255, 2020.
- [66] Maimun Rizal. *A Study of VoIP performance in anonymous network-The onion routing (Tor)*. PhD thesis, Niedersächsische Staats-und Universitätsbibliothek Göttingen, 2014.
- [67] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. Request for Comments (RFC) 3261, Internet Engineering Task Force (IETF), June 2002.
- [68] Katrin Schoenenberg, Alexander Raake, Sebastian Egger, and Raimund Schatz. On interaction behaviour in telephone conversations under transmission delay. *Speech Communication*, 63:1–14, 2014.
- [69] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Request for Comments (RFC) 3550, Internet Engineering Task Force (IETF), July 2003.

- [70] Piyush Kumar Sharma, Shashwat Chaudhary, Nikhil Hassija, Mukulika Maity, and Sambuddho Chakravarty. The road not taken: re-thinking the feasibility of voice calling over Tor. *Proceedings on Privacy Enhancing Technologies*, 2020(4):69–88, 2020.
- [71] Robin Snader and Nikita Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *16th Annual Network & Distributed System Security Symposium*, NDSS, 2008.
- [72] Tim Terriberry and Koen Vos. Definition of the Opus audio codec, 2012.
- [73] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *26th Symposium on Operating Systems Principles*, SOSP. ACM, 2017.
- [74] JM. Valin and K. Vos. Updates to the Opus Audio Codec. RFC 8251, October 2017.
- [75] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec. Request for Comments (RFC) 6716, September 2012.
- [76] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *25th Symposium on Operating Systems Principles*, SOSP, 2015.
- [77] Voyced. Is there a maximum call length or duration. <https://www.voyced.eu/clients/index.php/knowledgebase/397/Is-there-a-maximum-Call-length-or-duration.html>, 2019.
- [78] Gerry Wan, Aaron Johnson, Ryan Wails, Sameer Wagh, and Prateek Mittal. Guard placement attacks on path selection algorithms for Tor. *Proceedings on Privacy Enhancing Technologies*, 2019(4):272–291, 2019.
- [79] Tao Wang, Kevin Bauer, Clara Forero, and Ian Goldberg. Congestion-aware path selection for tor. In *International Conference on Financial Cryptography and Data Security*, FC. Springer, 2012.
- [80] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2012.
- [81] Matthew Wright, Micah Adler, Brian N. Levine, and Clay Shields. Defending anonymous communications against passive logging attacks. In *IEEE Symposium on Security and Privacy*, S&P, page 28, USA, 2003. IEEE Computer Society.
- [82] Matthew K Wright, Micah Adler, Brian Neil Levine, and Clay Shields. The predecessor attack: An analysis of a threat to anonymous communications systems. *ACM Transactions on Information and System Security*, 7(4):489–522, 2004.
- [83] Lei Yang and Fengjun Li. mtor: A multipath Tor routing beyond bandwidth throttling. In *2015 IEEE Conference on Communications and Network Security*, CNS. IEEE, 2015.

## A Appendix: Link monitoring effectiveness

We provide in this appendix a supplementary study of the effectiveness of link monitoring, dynamic link classification, and link selection. In particular, we assess whether link classification and selection reflect the behaviors discussed in Section 3.

We start by observing the distribution, over 64 calls, of the number of links that were classified as  $L_{1\text{ST}}$  at least once through the duration of a 90-minute call. This distribution is depicted in Figure 13. Note that we do not consider the first 40 seconds of each call, as DONAR has to bootstrap the process with random scores, and poorly-performing links could be assigned to the  $L_{1\text{ST}}$  group during this bootstrap. Between 6 and 12 links per call have been considered at least once in the  $L_{1\text{ST}}$  group in every call, with a majority of 8 to 10 links selected. This confirms our analysis that there is no single link that is consistently performing well in Tor, and that link performance varies significantly over time: Links that are poorly performing at a given time may be the best ones a few minutes later.

We study, in finer detail, the stability of links over time, focusing on a single call using the ALTERNATE policy with the Default configuration. We represent the latency of the first delivery of each frame in the first plot of Figure 14. This is the latency that is observed by the VoIP application. Latency remains low throughout the call. In the second plot, we decompose the latency of frames received on the  $L_{1\text{ST}}$  and  $L_{2\text{ND}}$  groups, including the first and second receptions. We can clearly see that the latency of the links in the  $L_{1\text{ST}}$  group is generally lower, and that outlier values are compensated by lower latency on a link in the  $L_{2\text{ND}}$  group. The third plot represents the assignment of the 12 links to link groups over

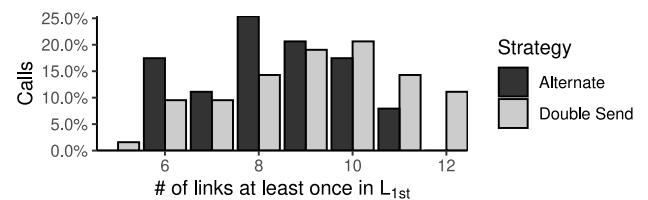


Figure 13: How many links were  $L_{1\text{ST}}$  at least once?

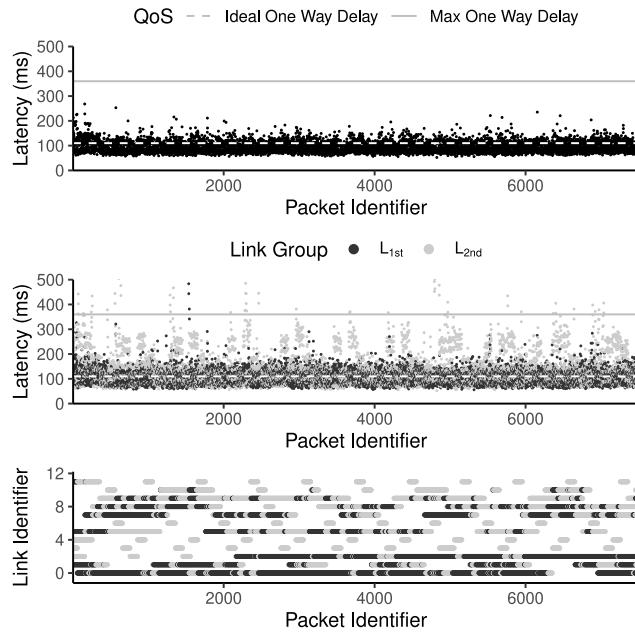


Figure 14: Stability over time.

time. We note that there was no link failure (and therefore no link replacement) in this experiment. Link 0 is, for instance, classified in  $L_{1\text{ST}}$  for a large part of the call, but suffers a latency spike around frame 6,500 and is rapidly classified in the  $L_{\text{INACTIVE}}$  group. Link 2, initially in  $L_{\text{INACTIVE}}$ , is promoted 3 times with no effect to the  $L_{2\text{ND}}$  group, before being selected as  $L_{1\text{ST}}$  after its fourth promotion. Links 1, 5, 7 and 8 have highly heterogeneous behaviors, while links 3, 4, 6, 11 and 12 have consistently bad behaviors, and only appear in the  $L_{2\text{ND}}$  group upon their promotion before being quickly deactivated. While these links could be proactively replaced by opening new links, we do not deem it necessary and choose not to impose further link setup load on the Tor network.



# Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon

Weitao Wang, Xinyu Crystal Wu, Praveen Tammana<sup>†</sup>, Ang Chen, T. S. Eugene Ng  
Rice University <sup>†</sup>Indian Institute of Technology Hyderabad

## Abstract

Performance monitoring and diagnosis are essential for data centers. The emergence of programmable switches has led to the development of a slew of monitoring systems, but most of them do not explicitly target posterior diagnosis. On one hand, “query-driven” monitoring systems must be pre-configured with a static query, but it is difficult to achieve high coverage because the right query for posterior diagnosis may not be known in advance. On the other hand, “blanket” monitoring systems have high coverage as they always collect telemetry data from all switches, but they collect excessive data. SpiderMon is a system that co-designs monitoring and posterior diagnosis in a closed loop to achieve low overhead and high coverage simultaneously, by leveraging “wait-for” relations to guide its operations. We evaluate SpiderMon in both Tofino hardware and BMv2 software switches and show that SpiderMon diagnoses performance problems accurately and quickly with low overhead.

## 1 Introduction

An efficient network monitoring and diagnosis system are essential to meeting the performance requirements of modern applications. Since production clouds have stringent SLAs, even a small network performance degradation may lead to significant application slowdown [13, 30]. Many network performance problems, such as high end-to-end latency, low throughput, and packet drops [38], can be attributed to traffic contention of some kind [4], although across scenarios, the root causes for the contention are diverse (e.g., bursty UDP traffic, ECMP load imbalance, and routing loops).

The emergence of programmable switches has led to a slew of monitoring systems being developed [12, 16, 32, 33, 39, 44, 48], but most of them do not explicitly target posterior diagnosis. For instance, “query-driven” monitoring systems [16, 32] need to be pre-configured with a static query. Since root causes for performance degradation could vary, and there may be a wide variety of reasons for performance problems, it is challenging to select the right query in advance. In principle, one could adaptively change the monitoring query based on the observed symptom; but in practice, many transient problems happen at fine timescales and their sporadic nature

requires always-on monitoring. On the other hand, “blanket” monitoring systems always monitor and collect telemetry data from the switches to achieve high coverage [10, 14, 22, 26, 27]. However, this would result in excessive data that may not be needed by the diagnosis in the first place.

Therefore, having a monitoring and diagnosis system that achieves either low overhead or high coverage is not hard, but achieving both simultaneously is challenging. The key question we explore is whether it is possible to design a streamlined system that performs efficient monitoring but achieves high coverage, achieving the “best of both worlds”. We present SpiderMon, a system where the monitoring and diagnosis operations are explicitly designed to work with each other in a closed loop. It enables a suitable tradeoff between accuracy and overhead when debugging network-wide performance problems. To achieve efficient and accurate monitoring, SpiderMon leverages a concept called “wait-for” [46] relations. Since many performance problems stem from in-network contention, “wait-for” relations target such behaviors in the telemetry collection in a precise manner. Moreover, such information is also exactly what is needed in diagnosis. For instance, a victim flow with high latency may have “waited for” many interfering events across multiple hops. By capturing and analyzing such relations, SpiderMon can achieve an effective diagnosis, with precise, targeted, but also high-coverage operations.

Since the symptom of “wait-for” events is usually high latency, SpiderMon uses timing information to trigger reactive telemetry collection. Precisely, SpiderMon detects performance problems when it encounters flows with excessively high queuing delay. After a problem is detected, SpiderMon uses the wait-for relations to track and collect other relevant information in the data plane across the network. For diagnosis, SpiderMon also identifies the root causes of the performance problem by summarizing the most significant wait-for relations from the collected telemetry data. It does so by jointly analyzing wait-for patterns together with other types of network knowledge (e.g., topology) and telemetry data (e.g., flow-level results). In this way, SpiderMon collects telemetry data only when the diagnosis process needs to analyze a problem, and it performs targeted collection based on what

the diagnosis process would require.

To realize this idea, SpiderMon addresses three technical challenges. The first challenge is to detect performance degradation without interfering with actual packet processing. SpiderMon leverages programmable switches to record telemetry data about network traffic. It piggybacks telemetry data in packet headers and checks for performance anomalies. The second challenge is to precisely collect the relevant telemetry information across the network. Relying on wait-for relations, SpiderMon notifies relevant switches and activates telemetry data collection from these locations. Finally, SpiderMon identifies the root causes of the performance problem using the telemetry information and the knowledge of the network configuration. The wait-for relation again is critical for identifying abnormal network behaviors, and for matching those behaviors to the signatures of root causes.

**Contributions.** Overall, SpiderMon is a *closed-loop* system for monitoring and diagnosing performance problems in the network. We have implemented a prototype of SpiderMon, and our results show that SpiderMon can diagnose performance problems accurately and quickly with low overhead.

## 2 Motivation

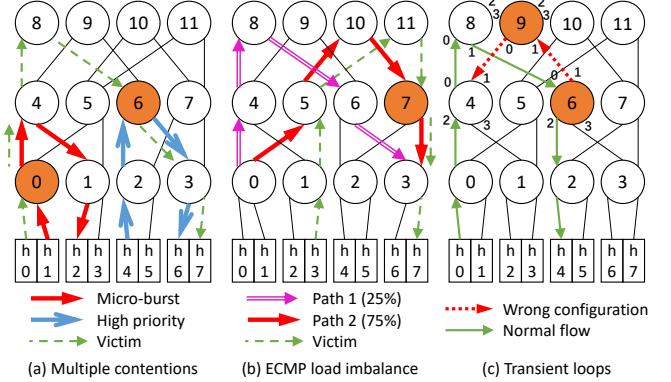
SpiderMon focuses on network performance problems that arise due to contention, which are challenging for at least three reasons. First, network contention may occur due to many root causes, so its diagnosis requires a general mechanism. Second, the root cause can be unpredictable both spatially and temporally, requiring agile solutions that can capture transient problems. A third practical challenge is that the solution must have a sufficiently low overhead on the network. SpiderMon does not target problems that happen because of silent packet drops, packet corruptions, control plane misconfigurations, slow servers, or other causes unrelated to network contention, although it can be used in combination with other techniques for these scenarios.

### 2.1 Root Causes Are Diverse

To illustrate the diversity of root causes of network performance problems, consider some examples in a 3-layer Clos network as shown in Figure 1.

**Micro-bursts.** Recent studies [10, 22, 45] found micro-bursts—i.e. momentary surges in traffic volume—to be a common root cause for sporadic excessive delays and packet losses. Detecting and diagnosing a micro-burst requires switch queuing delays to be monitored and the main contributor to queuing delays to be identified before the micro-burst disappears.

**Multiple flow contentions.** A victim flow encounters multiple contentions at different switches—flow 1 (e.g., a bursty UDP flow) and flow 2 (e.g., a high-priority flow) contend with the victim flow at switch 0 and switch 6, respectively (Figure 1(a)). The end-to-end latency for the victim flow becomes very high. For detection, we need to monitor per-flow



**Figure 1: Several performance degradation problems**

latency; for diagnosis, information about all contending flows is needed to identify the root causes.

**ECMP load imbalance.** Due to the skewed nature of flow distributions or imperfect hash mechanisms, ECMP load imbalance is a common problem in data centers [3]. Consider the network in Figure 1(b), where all links are 40Gbps. Switch 0 assigns 25% of the total traffic (32Gbps) to path 1 and 75% to path 2. The victim flow contends with the flows on path 2, which leads to high congestion at switch 7. This could be avoided if switch 0 assigns the traffic for the two paths equally. The root cause for this problem is the imbalanced assignment at switch 0, but the performance degradation occurs at switch 7, which is 3 hops away from switch 0. Once high latency is detected at switch 7, the previous hops’ information of the flows involved in the congestion is required for debugging.

**Transient/persistent loops.** During network updates, the configurations of different switches may not be synchronized. Some switches may fail to execute the reconfiguration commands silently. Under those circumstances, a forwarding loop may form [28]. An example is shown in Figure 1(c), where switches 6 and 9 are wrongly configured, which causes some flows to be stuck in a loop, leading to congestion and packet drops. The incompatible switch configurations should be blamed for the loop in the network. However, to identify the switches that need to be reconfigured, information from all the switches along the loop, namely, switches 6, 9, 4, and 8, needs to be collected for analysis.

### 2.2 Root Causes Are Unpredictable

There are three key features that make network performance problems challenging to detect or diagnose.

**Sporadic.** Performance degradation is usually sporadic, occurring occasionally at different places and at an unpredictable time [1]. Any flow may be affected, so detection algorithms need to monitor every flow all the time.

**Network-wide.** The root causes may be network-wide, e.g., contention at different hops. The interfering flows may even have normal performance [38], despite the fact that they cause performance degradation to other flows. Thus root cause diagnosis requires network-wide monitoring.

**Transient.** Traffic contentions sometimes are transient and disappear quickly [21]. For instance, transient loops may only form for a short time during network updates, but the performance problem introduced by packet drops may need a much longer time to fully recover. This feature requires the debugging system to maintain fine-grained information about recent events, in case the problems disappear quickly but happen in the network frequently.

### 2.3 Existing Solutions Fall Short

Existing solutions all fall short in monitoring and diagnosing network performance problems due to the above challenges.

**Host-based solutions.** Solutions like Trumpet [31] and Dapper [14] rely on end hosts to store telemetry data for diagnosis. But they all use inference algorithms to reconstruct what may have happened in the network from the collected data, which may not be accurate. Instead, SpiderMon collects data from the switches to achieve a better in-network view for diagnosis.

**In-network solutions.** Some existing solutions also collect telemetry data from the switches. (i) **Blanket telemetry systems** like NetSight [17] and PINT [8] collect information network-wide indiscriminately, even on network nodes unrelated to the problem. Those systems usually have high overheads, and much of the collected data is unnecessary for diagnosis. (ii) **Query-based systems** deploy queries into switches for data collection, such as Sonata [16], Marple [32], FlowRadar [26], and NetSeer [47]. They require that the operators know the nature and location of the problems, but problems could arise from sporadic congestion at random locations. Although in principle, queries can be changed based on the monitoring results, this happens at coarse timescales and cannot capture transient problems. SpiderMon can cover problems that cannot be succinctly defined using static queries and only capture events relevant to the problems.

## 3 SpiderMon Design

SpiderMon monitors and diagnoses performance problems caused by in-network contention in three steps: 1) SpiderMon encodes every packet’s accumulated latency in header fields, and triggers telemetry collection once excessive latency is detected (§3.1); 2) the switch that detects high latency initiates “spider” packets and rapidly delivers them to relevant switches using the wait-for relations; relevant switches receiving spider packets report their telemetry data (§3.2); 3) the root cause analyzer constructs wait-for relations from the evidence for root cause analysis (§3.3).

### 3.1 Problem Monitoring

**Goal: Detect excessive cumulative queuing delays.** Rather than wait for the occurrence of harmful events (e.g., packet loss, TCP congestion window back-off), SpiderMon detects the performance problems based on a much earlier sign—abnormal cumulative queuing delays experienced by packets. It reacts quickly to performance degradation.

**Design:** 1) **Use cumulative latency for detection.** Instead of storing per-hop latency information in the header, SpiderMon uses cumulative latency to guarantee that the header length stays constant regardless of hop count. The cumulative latency  $L$  is updated at every hop based on the current queuing delay and the cumulative latency experienced by the packet so far,  $L = L + \text{queuing\_delay}$ . Every switch on the path checks whether the cumulative delay exceeds the latency threshold. To further reduce overhead, SpiderMon can compress the additional fields to less than 2 bytes by extracting the most significant bits (more in §C.2). 2) **Assign different latency thresholds for different traffic types.** Given that the tolerable latency varies for different applications, SpiderMon allows network operators to customize the latency thresholds for different applications. 3) **Detect problems and trigger telemetry in the switch data plane.** Unlike some monitoring systems using a central controller to monitor network problems [6, 31, 48], SpiderMon triggers fast reactions in the data plane. The communication delay within the data plane (tens of ns) is much lower than that between the data plane and the control plane (hundreds of  $\mu\text{s}$ ). 4) **Monitor every packet at every hop for target flows.** Compared to sampling-based detection [2, 34], SpiderMon achieves full coverage without losing any important information. Also, rather than detecting problems at the end hosts [9, 24], SpiderMon detects performance problems inside the network and reacts more quickly to the problem. 5) **Be transparent to end-hosts.** The latency threshold and cumulative latency are added at the edge switches when packets enter the network and removed when packets leave the network. Hosts remain unchanged.

Consider Figure 1(a) as an example. The victim flow suffers from queuing delay at switches 0 and 6, but the cumulative latency exceeds the threshold only at switch 6. Thus the problem is detected at switch 6, and switch 6 triggers the telemetry collection procedure.

### 3.2 Telemetry Collection

**Goal: Only collect evidence relevant to root cause analysis.** SpiderMon maintains a small amount of telemetry information as evidence on the switches to facilitate subsequent diagnosis; this information is not collected from the switches unless needed. First, to minimize the amount of telemetry data collected to the analyzer while maintaining the diagnosis accuracy, SpiderMon only targets switches relevant to the observed performance problem as detailed in §3.2.1. Second, SpiderMon collects the relevant telemetry data within a short time such that each switch only needs to keep a small amount of historical telemetry data as detailed in §3.2.2.

#### 3.2.1 Relevant Switches Notification

**#1: Only collect data after problem detection.** Compared to other systems which collect data to a centralized collector all the time [6, 16, 32, 48], SpiderMon uses a default-off

collection strategy to minimize overhead. After the problem is detected, a special ‘spider’ packet is generated to notify relevant switches and start the telemetry collection on those switches. A ‘spider’ packet carries: 1) an event\_ID, which concatenates the switch ID and the event index to uniquely identify the problem, and 2) the 5-tuple of the victim flow. Spider packets are generated by mirroring the packet that triggered the diagnosis and recirculating it for transmission, while the original packet transmits as normal. To prevent possible packet drops during the transmission, all ‘spider’ packets are prioritized in the network for lossless transfer.

**#2: Only collect data from relevant switches.** Instead of collecting telemetry from all switches, SpiderMon identifies the switches that are relevant to the detected problem by tracking packet-level provenance; it only retrieves data from these switches to minimize overhead. Packet-level provenance is modeled as  $G := (V, E)$  for a detected event and the corresponding causality relations.  $G$  is a directed acyclic graph, where each node  $v$  represents an event, and each directed edge  $e = (v_1 \rightarrow v_2)$  represents that  $v_1$  leads to the event  $v_2$ . For latency problems in a network, all wait-for contentions in the switch queues are considered events in the provenance data. Since events at the upstream switches affect the events at the downstream switch, such upstream events are also incorporated into the provenance model. In this way, we can construct a provenance graph for a performance problem. By analyzing the locations of events, SpiderMon can select switches relevant to the specific problem.

**#3: Track the provenance graph in the data plane.** Unlike the central controller that Trumpet uses to inform relevant nodes, SpiderMon performs this procedure entirely in the data plane to reduce the latency of notifying relevant switches. It only requires switches to maintain telemetry data for a shorter time for the recent interval without losing necessary data. To achieve this, SpiderMon repeats the following two steps on each switch that receives the ‘spider’ packet: 1) sends a trace-back ‘spider’ packet along the historical path of the victim flow, where the path is obtained using a bloom filter, 2) sends branch-search ‘spider’ packets to ports that sent traffic and contended with the victim flow, where the ports are identified by a per-port traffic meter. Switches drop spider packets with duplicate IDs to avoid unnecessary processing (§C.1).

**Timeout bloom filter.** SpiderMon uses a timeout bloom filter (TBF) to track the victim flow’s historical path. Regular bloom filters allow the insertion and the membership test of a flow ID. However, they can only support insertions, and the false positive rates increase with the number of inserted flows. A rotating bloom filter, on the other hand, can instantiate one instance per epoch, so that older data can be safely discarded; however, this is very coarse-grained as it only supports per-epoch deletion. To address those problems, SpiderMon adds a timeout feature to remove unneeded data from the bloom filter; this method provides a ‘sliding window’ of historical flow information. For a switch with  $N$  ports, each egress

---

### Algorithm 1: Timeout bloom filter data structure

---

```

Input:  $B$ : Timeout bloom filter,  $inPort$ : Incoming port index,
      5-tuple: 5-tuple,  $curr\_TS$ : Current timestamp,  $epoch$ :
      Timeout epoch
Function  $updateBF(inPort, 5\text{-tuple})$ :
  1 |  $hashValues = HASH(5\text{-tuple})$ 
  2 | for  $hashValue \in hashValues$  do
  3 |    $B[hashValue][inPort] \leftarrow curr\_TS$ 
  4 |
  5 return
Function  $checkBF(inPort, 5\text{-tuple})$ :
  6 |  $hashValues \leftarrow HASH(5\text{-tuple})$ 
  7 | for  $hashValue \in hashValues$  do
  8 |    $stamps \leftarrow B[hashValue][inPort]$ 
  9 |   if  $curr\_TS - stamp > epoch$  then
 10 |     return False
 11 |
 12 return True

```

---

pipeline maintains a bloom filter group with  $M$  rows and  $N$  cells per row, and each column represents a bloom filter for the corresponding port. The TBF replaces the bit record with a short timestamp, which can be used to recognize the outdated records when querying the TBF. The details about maintaining and querying the TBF are shown in Algorithm 1, Figure 2(a) and Figure 2(b). The memory footprint of TBF can be reduced by shrinking the size of stored timestamps (§C.2).

**Most recent, per-port traffic meter.** SpiderMon identifies the relevant ports that contribute to high latency. To distinguish an ingress port with low throughput, SpiderMon maintains a traffic meter for each ingress port’s traffic volume in the most recent time. Normal traffic meters in the switch are reset to 0 periodically, leading to information loss. Therefore, SpiderMon divides the time window into several small windows and associates those meters’ values to realize a sliding window of the traffic amount within the most recent time window (details in §B).

**#4: Reduce the collected telemetry data by pruning the provenance graph.** Some causality relations are more important than others. SpiderMon leverages this to reduce overhead without sacrificing diagnosis accuracy. Specifically, if the traffic volume from some ingress ports is significantly lower than others, it is excluded from the possible root causes; so switches that contribute minimally to the problems are ignored. SpiderMon provides a tunable threshold and only sends spiders to the ports with high traffic rates. The robustness of this threshold is shown in §4.3.

To illustrate the relevant switch notification procedure, we use Figure 3 as an example of a multiple contention scenario. The high latency is detected at switch 0. Then the traceback ‘spider’ is sent to the reverse path of the victim flow, namely, switches 1, 2, and 3. At the same time, the branch-search ‘spider’ is sent to switches 4 and 6, with switch 5 being ignored due to the small traffic volume. If the traffic from switch 4 came from two other switches has sufficient volume, the branch-search ‘spider’ packets will also be sent to those ports.

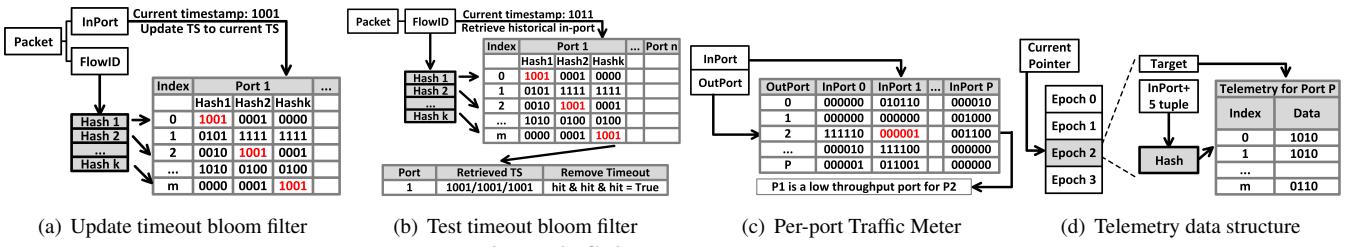


Figure 2: SpiderMon data structures

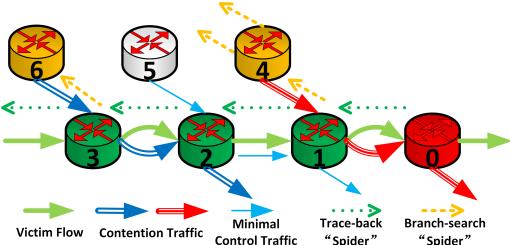


Figure 3: “Spider” packets propagation

### 3.2.2 Telemetry Data Collection

**#1: Collect per-epoch per-flow information.** Per-packet telemetry incurs a very high overhead and usually is unnecessarily fine-grained for diagnosis. SpiderMon records the history with a per-epoch flow-level log, which is stored in the switches’ egress pipeline and each egress port has its own log. Dividing into epochs this way allows SpiderMon to observe changes among epochs. Each switch keeps a fixed number of epochs on the data plane and keeps the most recent ones in a circular buffer. When reporting the telemetry data, information of all epochs will be sent to the analyzer.

SpiderMon collects 36 bytes of data per flow, including the flow’s 5-tuple, sequence number range, total traffic volume, total packet count, total queuing depth, the priority of the flow, and the incoming port. The network operators can add extra flow-level information in the telemetry data structure for diagnosing other network problems. The total amount of telemetry data varies with the flow arrival rate. To update, SpiderMon first identifies the right telemetry table based on the outgoing port, then hashes the flow ID to assign a slot in the telemetry data structure for that flow. By doing a bit-wise XOR between the packet’s 5-tuple and the 5-tuple in the slot, we can determine whether this packet belongs to the existing flow by checking whether the result is 0. If so, this packet will be used to update this entry; otherwise, it will be considered as a new flow and replace the old one. The old entry will be packed and sent to the control plane for storage.

SpiderMon must maintain telemetry data for a minimum duration to ensure that the needed evidence for diagnosis is available, and this duration can be estimated as follows. Denote the threshold for detecting an unacceptable cumulative delay as  $T$  and the maximum round-trip propagation delay across the network as  $RTT$ . The time it takes to propagate spider packets from the initiator to relevant switches—recall that spider packets have high transmission priority and do not

wait for normal traffic—is half  $RTT$  in the worst case. Since the problem is detected after accumulated delay exceeds  $T$ , the time duration a switch must maintain telemetry data to diagnose this problem is, therefore,  $T + \frac{RTT}{2}$ . The common  $RTT$  and  $T$  in the data center network is 0.5–2 ms and 10–15 ms respectively [15], so it would be more than enough for SpiderMon to preserve history for 20 ms.

**#2: Provide synchronization among switches using flows’ sequence number.** The host-based solution cannot replay accurately, one of the reasons is the various network delay for packets, namely, the order of packets is not preserved at switches. SpiderMon has a similar problem when choosing the most relevant epoch on different switches for analysis. The correct epoch for the switch that triggered the problem is no doubt the most recent epoch, but for other switches on the historical path, the delay from the queuing and propagation may have caused the most relevant epoch to become a historical epoch. To solve this, SpiderMon keeps track of the  $[min\_seq, max\_seq]$  for each flow, and uses the victim flow’s sequence numbers to find the correct epoch with the maximum overlap with this sequence number interval for the relevant switches.

**#3: Trigger telemetry packet generation in the data plane.** Unlike NetSight that uses mirroring for collection, SpiderMon uses the packet generator to report the per-epoch per-flow log to the root causes analyzer. The packet generator can be directly triggered in the data plane to minimize latency. Compared to retrieving the data via the switch control plane as in several previous works [27], SpiderMon is much more agile because it bypasses the low bandwidth and high latency connection between the data plane and the control plane.

The telemetry packet header contains 1) an event ID for identifying the performance problem; 2) a switch ID; 3) a partition index of the telemetry data; 4) a part of the telemetry data. The telemetry packets are generated by the packet generator on a programmable switch. The generated packets only have Ethernet and IPv4 headers without the payload for bandwidth savings. The IPv4 destination address of telemetry packets is set to the root cause analyzer so that the network will forward the packets to the analyzer. There is a maximum amount of telemetry data that can be inserted into a single packet, which is around 200 bytes due to the limitation of the PHV fields for the programmable switches. So the packet generator will generate a fixed number of telemetry packets according to the size of the telemetry tables.

---

**Algorithm 2:** Replay the queue condition

---

**Input:**  $T$ : the epoch period;  $N$ : flow packets count,  $s$ : time for the last packet  
**Output:**  $time\_list$ : time list for the packets

```

1 for  $t \in N$  do
2    $t \leftarrow s + \frac{T}{N}$ 
3    $time\_list \leftarrow time\_list + t$ 
4 return  $time\_list$ 
```

---

**#4: Only collect the telemetry data from relevant ports to reduce overhead.** When a switch receives a spider packet from a certain port, usually only the telemetry data for that port will be reported to the analyzer, which reduces the amount of data collected.

### 3.3 Root Cause Analysis

SpiderMon develops a diagnosis strategy that is generalizable to diverse root causes with high precision and recall.

Efficiently localizing network problems and accurately identifying the root causes can be difficult, especially when the network conditions are dynamic and complex. Firstly, a good diagnosis algorithm needs to understand flow interactions and find the corresponding flows that occupied the queue. Secondly, once the problem has been localized, the diagnostic algorithm needs to further identify each problematic scenario with one or more root causes, such as micro-bursts or transient loops. However, most existing diagnostic algorithms do not have a clear boundary between those two steps. The identifications of the root causes are based on the matching of the problem patterns and observations, leading to slow diagnosis time and reduced diagnosis accuracy.

SpiderMon addresses these challenges with a two-step diagnostic algorithm: 1) efficiently analyze the queuing information at both flow level and aggregate level to recall all the problematic flows using wait-for graphs (WFG), as discussed in §3.3.1; 2) apply signature matching between the problematic flows and the root cause type, as described in §3.3.2.

#### 3.3.1 Find the Possible Root Causes

To find all possible root causes with a high recall rate, SpiderMon uses WFG at both flow-level and aggregate-level to identify the abnormal behaviors from the telemetry data.

**Wait-for relation.** *If a packet from flow A enters a queue where the packets from flow B already exist in the queue, then flow A waits for flow B at this queue.*

**Flow-level wait-for graph (WFG).** *Each vertex represents a flow, and a directed edge from vertex A to vertex B represents that flow A waits for flow B.*

**Wait-for weight.** *Each directed edge's weight is calculated as follows: for a packet  $p_k$  from flow A, if  $x_k$  packets from flow B exist in the queue when  $p_k$  enters, then flow B blocks flow A with weight  $x_k$ . For all  $n$  packets from flow A during a certain period, the average weight  $\frac{1}{n} \cdot \sum_{k \in [1, n]} x_k$  is the wait-for weight for the directed edge from vertex A to vertex B.*

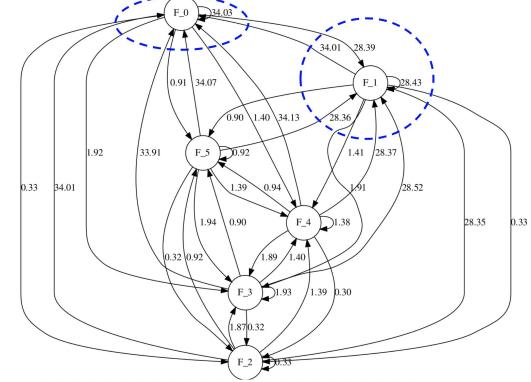


Figure 4: Identify the main contributors in WFG

---

**Algorithm 3:** Wait-for Graph Construction

---

**Input:**  $Seq$ : A sequence of packet,  $level$ : flow or port  
**Output:**  $G$ : Wait-for graph for the given sequence

```

1 for  $i \in [0, Seq.length]$  do
2   if  $level=flow$  then
3      $Seq[i].vertex \leftarrow Seq[i].flow$ 
4   else if  $level=port$  then
5      $Seq[i].vertex \leftarrow Seq[i].port$ 
6   if  $Seq[i].vertex \notin G$  then
7      $G.AddVertex(Seq[i].vertex)$ 
8 for  $i \in [0, Seq.length]$  do
9   for  $j \in [0, pkt.qdepth]$  do
10    edge  $\leftarrow (Seq[i].vertex \Rightarrow Seq[i-j].vertex)$ 
11     $G.AddEdgeWeight(edge, 1)$ 
12 return  $G$ 
```

---

**Aggregated wait-for graph.** SpiderMon also aggregates the flow according to the source IP, incoming port, or other keys to construct aggregated-level WFGs to find root causes other than flows' misbehavior. One typical example used in SpiderMon is the port-level WFG.

After receiving all the telemetry data from the switches, SpiderMon uses the gap-based sampling strategy [25] to replay the queuing condition on the switch (Algorithm 2). The actual sequence of the packets is not important since we only need the generated wait-for graph to be similar.

To find the main contributors for the queuing, we rely on the wait-for graphs to show the provenance relations between contending flows. For each queue, SpiderMon will construct flow-level WFGs and port-level WFGs as in Algorithm 3, which will be used to determine the main contributors. Basically, to identify the main contributors of the queue is to divide the flows in the queue into victims (suffer from queuing) and main contributors (contribute to queuing) and maximize the wait-for relations between those two groups. SpiderMon is able to show that this division can be easily derived by the following Theorem 1, and identify the main contributors as in Algorithm 4. We prove Theorem 1 in Appendix §A.

**Degree of the vertex.** *Sum of all incoming edge weights subtracts the outgoing edge weights.*

---

**Algorithm 4:** FindContributor

```
Input:  $G$ : Wait-for graph for the given sequence
Output:  $ctrs$ : A set of main contributors
1 for  $X \in G$  do
2    $D(X) = \sum_{e \in \{<i,j> | j=A\}} w_e - \sum_{e \in \{<i,j> | i=A\}} w_e$ 
3   if  $D(X) > 0$  then
4      $ctrs \leftarrow ctrs + X$ 
5 return  $ctrs$ 
```

---

**Theorem 1.** *The wait-for relation between two groups, divided by one cut, is maximum, if and only if one group only contains positive degree vertices while the other contains only negative degree vertices.*

Figure 4 is an example scenario of micro-burst with flows 0 and 1 as the burst flows, and both of them have been identified by the algorithm as the main contributors.

### 3.3.2 Precisely Identify Root Causes

To precisely identify the reason behind the main contributors determined in the first step, SpiderMon relies on signature matching to recognize different root causes. We give four signatures for four common root causes in Algorithm 5, using both telemetry and network configuration information. The signatures can be extended if more root causes are added. For better illustration, we consider the scenarios in Figure 1 and show the signatures in Figure 5. A detailed signature definition can be found at §G.

**Micro-bursts.** SpiderMon can identify all the main flow-level contributors at different hops along the victim flow's historical path. As shown in Figure 5(a), the micro-burst flow has many wait-for edges with large weights pointing to itself due to a large amount of traffic during the problematic time.

**Different priorities.** For contention between flows with different priorities, SpiderMon checks the priority of the victim flow and the main flow-level contributors. The contributor flows with higher priority compared to the victim flow can be identified as the root causes, as shown in Figure 5(b).

**ECMP load imbalance.** For the load imbalance problem displayed in Figure 1(b), SpiderMon will find the flow-level main contributors and check if they are routed by ECMP. Then SpiderMon calculates the ECMP imbalance ratio with the throughput of all flows routed by ECMP rules, using the traffic volume provided by per-flow telemetry data. The problematic ECMP groups can be identified when the calculated ratio is largely imbalanced as in Figure 5(c).

**Transient/persistent loops.** For the latency problem caused by transient or persistent loops as shown in Figure 1(c), SpiderMon searches the port-level contributors along the contributor traffic's path. If the same port is observed twice during the search procedure, all those ports have a high possibility to form a loop for specific traffic. Furthermore, the flow ID will be checked to further confirm the transient/persistent loop.

---

**Algorithm 5:** Root Causes Diagnostic Algorithm

```
Input:  $f\_WFG$ : flow-level WFG,  $p\_WFG$ : port-level WFG,  $T$ : Telemetry information,  $K$ : Network topology and configuration
1 /* Diagnose flow-level problems */ *
2 for  $sw \in \text{Switches on victim's path}$  do
3    $f\_CTR_{sw} \leftarrow \text{FindContributor}(f\_WFG_{sw})$ 
4   for  $f \in f\_CTR_{sw}$  do
5     // Is micro-burst?
6     check flow  $f$  throughput
7     // Is priority problem?
8     check flow  $f$  priority
9     // Is routed by ECMP rules?
10    check aggregated throughput for ECMP switches
11   /* Diagnose port-level problems */ *
12  for  $sw \in \text{Switches on victim flow's path}$  do
13     $p \leftarrow \text{victim flow's outgoing port}$ 
14     $\text{CheckPort}(p, \{\})$ 
15   /* Recursive function for port-level */ *
16  Function  $\text{CheckPort}(p, p\_set)$ :
17    // Does routing contain loop?
18    check whether there is a loop
19    // Search dominant port contributors
20     $p\_CTR_{sw} \leftarrow \text{FindContributor}(p\_WFG_{sw})$ 
21    for  $p' \in p\_CTR_{sw}$  do
22      // Check the related port
23       $src\_p \leftarrow \text{the port connect to port } p'$ 
24       $\text{CheckPort}(src\_p, p\_set + p)$ 
```

---

## 4 Evaluation

Next, we evaluate SpiderMon along several dimensions: diagnosis effectiveness, overheads, and robustness.

**Setup.** Our hardware testbed deploys SpiderMon to a Barefoot Tofino switch, written in 1147 lines of P4-Tofino code, to evaluate the switch-level performance. The switch is logically partitioned to emulate a topology with multiple logical switches; logical links are emulated by port-to-port connections using direct attach cables. The switch is also physically connected to eight servers through 25 Gbps links. The switch has  $32 \times 100$ Gbps ports, and each can be configured as four 25Gbps ports with a breakout cable; each server has two six-core 3.4GHz CPUs, 128GB RAM, and one 25Gbps NIC. In addition, we have set up a simulation environment that uses the BMv2 software switches in the NS3 simulator with 945 lines of P4 code running on CloudLab servers, evaluating the network-level performance. Each server has an eight-core 2.0GHz CPU and 32GB RAM. A K=4 standard fat-tree topology with 20 switches and 32 hosts is simulated with 1 Gbps link bandwidth. We also implement the root causes analyzer with 843 lines of Python code.

**Workloads.** We simulate empirical workloads from production networks for our evaluation. The flow size distribution is taken from three different traces: web search [5], cache [35], and Hadoop [35]. The arrival time of different flows is based on a Poisson process and the flow arrival rate is varied to obtain different load utilizations in the network. The source and destination for each flow are chosen uniformly at random.

All flows are TCP.

**Baseline systems.** We compare SpiderMon against five baseline solutions. 1) **Trumpet** [31]: a trigger-based reactive host system. When it detects a problem requiring network-wide information on one host, the controller will collect data from related servers upon a trigger. This incurs a latency of at least an RTT. 2) **NetSight** [17]: an in-network system that proactively collects ‘postcards’ for each packet from the switches. 3) **Marple** [32]: a query-based in-network system, which is deployed to all switches using monitoring queries that a) detect high latency, b) query packet counts, and c) perform ‘EWMA over latencies’. 4) **Pathdump** [37] and **SwitchPointer** [38]: two proactive, network+host solutions. Pathdump tracks paths and performs diagnosis on end-hosts, and SwitchPointer further tracks packet epochs in the network.

#### 4.1 Diagnosis Effectiveness

We evaluate the diagnostic effectiveness of SpiderMon using multiple scenarios.

**1. Micro-bursts** are created by injecting 5 short-lived (10-100  $\mu$ s) UDP flows from SW0 to SW1 and from SW2 to SW3 as in Figure 1(a). The throughput of micro-burst flows is set to 90%  $\times$  line-rate. **Diagnosis:** Fig. 5(a) shows the combined wait-for graph at two switch ports generated by SpiderMon, which shows that the two micro-burst flows E and H dominate the queues and are the only two main contributors with positive degrees. The other 3 UDP flows are not included in the WFG since they end before the victim flow starts or start later than the 2 contending UDP flows.

**2. Priority contentions** inject 5 high-priority TCP flows with priority queuing from SW0 to SW1 and from SW2 to SW3 as in Figure 1(a). **Diagnosis:** As Figure 5(b) shows, flow C and D are the main contributors to the congestion with higher priority and larger degrees. Other priority flows have no interference with the victim flow so the WFG excludes them.

**3. ECMP imbalance** scenarios randomly pick a switch (except core switches) and split traffic to two uplink ports with 4:1 imbalanced load. The ECMP group imbalance lasted for hundreds of microseconds. **Diagnosis:** When we find the main contributors to the queuing, SpiderMon will check whether they are routed by ECMP policy. In Figure 5(c), both main contributors (flow C and D) are routed by ECMP rules on switch 0, so SpiderMon uses the telemetry information for switch 0 and computes the number of flows and traffic amount sent to each ECMP port. If the number of flows or traffic amount within that epoch is largely imbalanced, then there is an issue with the ECMP rules or hash functions.

**4. Loops** create a 4-hop routing loop with 2 aggregation switches and 2 core switches as in Figure 1(c). The routing loop only affects a small group of flows and the problem only lasts for 100  $\mu$ s. **Diagnosis:** Port-level WFGs identify a loop as the root cause: the victim flow is reported on switch 8 port 1 so that the WFG leads us to the main contributor, port 0. Since SW8-P0 receives traffic from SW4-P0, we further construct

a WFG for SW4-P0 and determine another main contributor. With this recursive searching procedure, SpiderMon finds that the port-level contributors form a loop and the traffic belongs to the same group of flows.

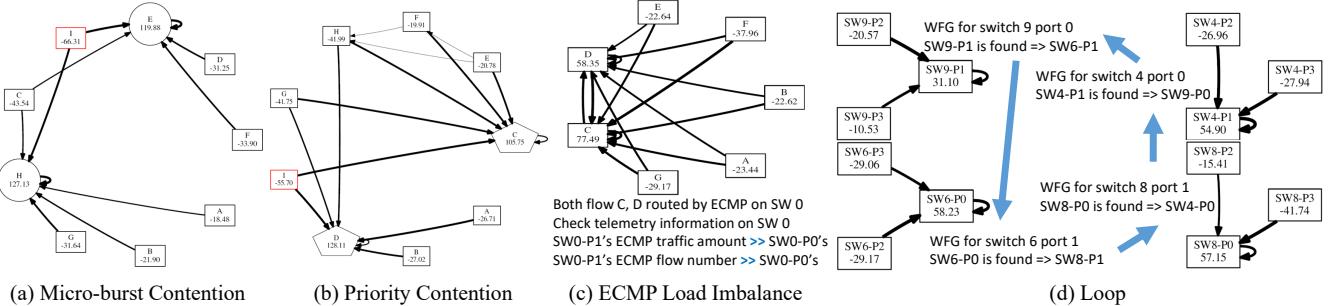
**5. Complex problem diagnosis.** Next, we test a diagnostic scenario with multiple problems. In Figure 6, the victim flow contends with a micro-burst flow at switch 1, a high priority flow at switch 7, and high-volume traffic caused by ECMP imbalance at switch 5. First, SpiderMon constructs the WFG with the collected information for the problem and identifies 5 flows (flow C, E, F, J, and L) with positive degrees. Next, SpiderMon checks the property of each such flow and identifies flow C as a micro-burst flow without any congestion control, while flow J is a flow with higher priority than any other flows crossing those switches. Then it checks the amount of the transmitted traffic in the same epoch and identifies flows E and F to be related to an ECMP imbalance. However, flow L is removed from the root causes; it is a normal TCP flow since its degree is small and there is no further evidence from the telemetry information to show that this flow is problematic.

**6. Sporadic & transient problem diagnosis.** We also evaluate multiple diagnostic situations with sporadic and transient problems. The traffic workloads are generated from random sources and destinations, and the problems could happen at different locations in the network randomly with short-lived root causes. Take the micro-burst experiment as an example. A high throughput UDP flow is introduced between a random source and destination at a random time, lasting for 100  $\mu$ s. The experimental results shown in Section 4.2 are generated with sporadic problems for each scenario.

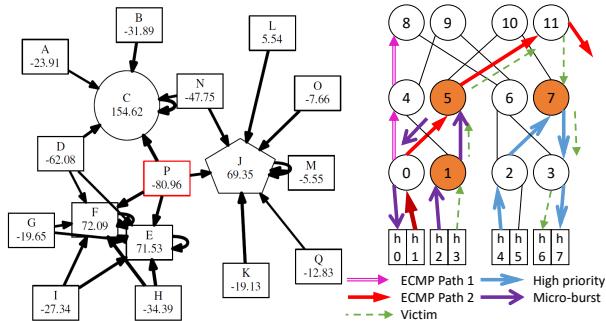
#### 4.2 Comparison with Baseline Systems

**Precision and recall.** We first show the precision and recall rate for different solutions, by tuning the parameters of each system so that it can achieve the best performance for each scenario. Those include the maximum tolerable link load imbalance ratio, link utilization, per-flow throughput, and so on. Details about each scenario’s parameters are in §F. Here we show the results for web trace only, the results for cache and Hadoop traces are included in §E.2. For the web trace, 30% of the flows are 1–30MB, so that multiple large flows can be concurrently active from/to one switch port.

As shown in Figure 7, Trumpet cannot achieve both high recall and accuracy at the same time for the transient congestion since it can only infer the in-network condition based on the calculated link utilization and end-to-end delay. Due to the different network delays and packet loss, the evidence for the transient problems may be inaccurate and unreliable on the host. Trumpet also fails to diagnose the ECMP imbalance problem because it does not have path information for every flow to identify the traffic split at the ECMP switches. Trumpet also fails to diagnose the loop problem because packets involved in loops do not reach the hosts, leaving no evidence for Trumpet to find out the root cause.



**Figure 5: Example wait-for graphs of several root causes.** Each box (TCP flow/port), circle (UDP flow), and pentagon (High priority flows) represent one flow or port, and the port name is described according to Figure 1(c). Bolder edges represent heavier wait-for relations, edges with small weights are tailored. The number under the flow/port name shows the node degree, and positive degrees will be identified as main contributors.

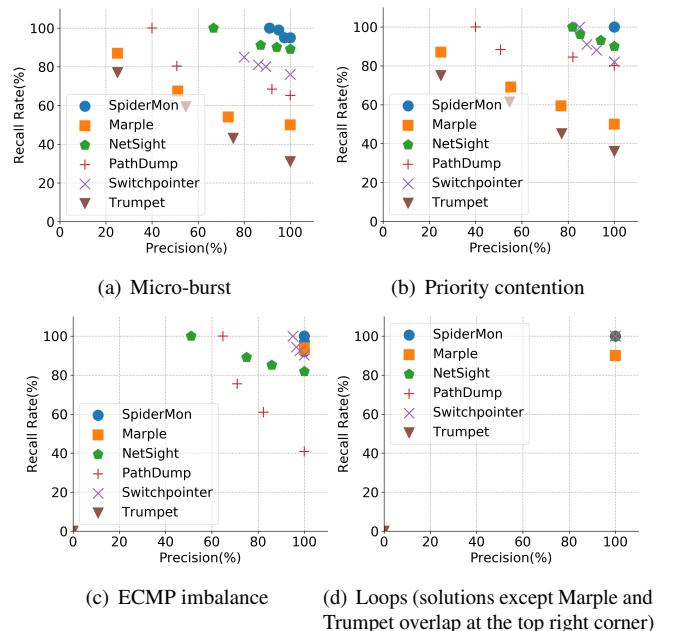


**Figure 6: The WFG for victim flow P, with a micro-burst, a priority-related contention, and an ECMP imbalance at different hops.**

Marple falls short in diagnosing transient contention like micro-bursts. This is because Marple enables queries only when needed, so it collects data reactively, which incurs an additional latency. The per-hop queuing information is only collected when the accumulated queuing latency exceeds the threshold. This control loop delay leads to information loss for transient problems—when the system begins collecting data from a switch near the destination, the transient bursty flow at a previous hop may have already ended. Only Marple and Trumpet are reactive systems in our evaluation.

PathDump and SwitchPointer both achieve relatively good performance. PathDump carries path information along with the packets, and SwitchPointer upgrades PathDump with switch data that records the flows that travel the same switch in the same epoch, which outperforms PathDump. However, both of them failed to identify transient problems since they lack queuing information—they instead recompute link utilization using packets received at end hosts. If a large amount of packets are dropped in the network due to congestion loss or TTL expiration, it would be very hard to reconstruct the transient network condition. Another interesting fact is that both solutions add extra in-network mechanisms (path tracking [37]) to detect the routing loop, so they both achieve great performance in detecting and diagnosing loops.

NetSight achieves the second-best performance since it

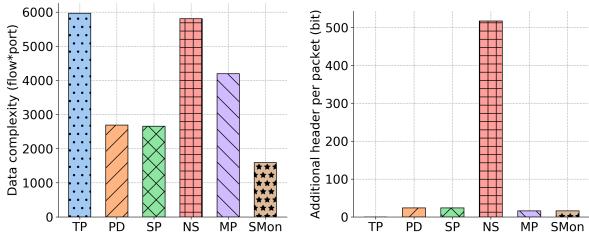


**Figure 7: Diagnostic effectiveness for different solutions**

collects per-packet postcards. One drawback is that to keep overhead down, NetSight omits important data like packet priority or precise timestamps. Instead, it uses topology information to place the postcards in order. However, information that describes how flows interact cannot be obtained, which is essential for diagnosing transient problems.

SpiderMon is able to achieve nearly 100% recall and precision for all tested scenarios. The reason is that SpiderMon collects accurate packet-level information within a time interval. For micro-burst and priority flow contention, each flow's throughput within the same epoch where congestion happens will be recorded and reported in the telemetry data; for the ECMP imbalance problem, the flow ID and output port will be recorded, so that the ECMP imbalance ratio can be calculated; for the loop problem, the loop can be easily detected in the procedure of WFG construction.

To summarize, host-based solutions (Trumpet, PathDump



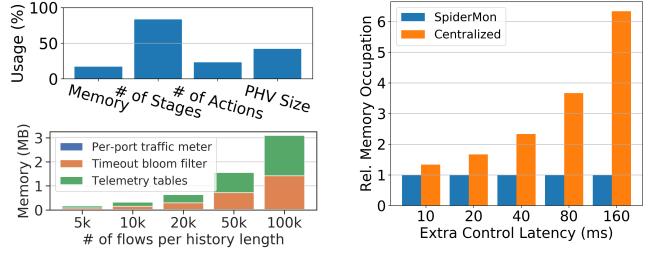
(a) Diagnostic data complexity (b) Additional bandwidth overhead

**Figure 8: Diagnostic data complexity for different systems; the additional per-packet header shows the bandwidth overheads for Trumpet (TP), PathDump (PD), SwitchPointer (SP), NetSight (NS), Marple (MP), and SpiderMon (SMon).**

and SwitchPointer) all lack accurate in-network information, like accurate queuing information and the packet loss for traffic other than TCP (they can only observe packet loss at the sender with the help of TCP’s congestion control). As for the proactive in-network approach in NetSight, it sacrifices the telemetry data granularity to keep overhead low. Only the packet header, switch ID, output port, and a version number are included. It uses topology information to assemble out-of-order postcards since the fine-grained timestamps and queuing information are not included in the postcards. The reactive in-network Marple system can potentially collect the information at very fine granularity but it can only start this reactive network-wide query after a half-RTT delay after the problem has been detected. The experiments over Cache and Hadoop traces have qualitatively similar results with the web search trace; more details can be found in §E.2.

**Diagnostic overhead.** To evaluate the diagnosis complexity and resource usage of different solutions, we measure the amount of collected data and the extra bandwidth requirements. We measure the diagnosis complexity using the amount of telemetry data stored and used in the diagnostic procedure, using  $(\text{flow} \times \text{port})$  as the unit to denote the complexity of flow information collected at switch ports. Since the host-based solutions collect information from the end hosts, and they reconstruct the utilization of different links [37], we multiply the average path length with the  $\text{flow} \times \text{host}$  as the overall complexity. Both switches and hosts have limited storage spaces and may restrict the scalability of the solutions. Under the same scenario for diagnosing micro-bursts, we show the amount of telemetry data for different systems in Figure 8(a). Reducing the diagnosis complexity not only relieves the burden to process the collected information for the central controller but also saves the storage space to store the diagnostic data for future usages.

Trumpet processes packets and match triggers in real time during the monitoring phase, so no packet is stored. But in the reactive data gather-report phase, data from multiple hosts will be reported. In order to construct every link utilization, the throughput of all flows will be reported and stored for



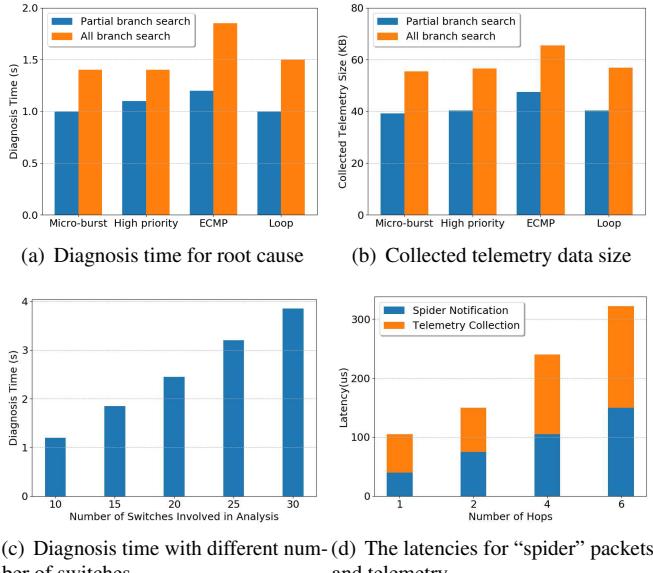
(a) The resource usage on Tofino switch is low. Per-port traffic meter is too small to be visible in the figure. (b) Relative memory usage under different controller latency with SpiderMon as the baseline.

**Figure 9: Switch memory occupation**

further analysis. Pathdump and SwitchPointer need to store per-packet history, since the problem may be detected after analysis. But both systems rely on the path information to find out the flows that travel the same link with the victim flow so that the data complexity can be reduced by filtering out irrelevant flows. Marple stores the query results from every switch to reproduce the scenarios, so such data will be transmitted as well as stored on the hosts. But Marple starts the collection after problem detection and stops after the problem disappears, collecting less but potentially incomplete data. NetSight stores all packet postcards and processes them in real time. All flows from all the switch ports are collected and stored, leading to a similar data complexity as Trumpet. SpiderMon only collects data after a problem is detected and only from relevant switches. Thus, the overhead for collecting telemetry data is much lower than the other systems.

**Monitoring bandwidth overhead.** Next, we measure the amount of extra bandwidth usage during monitoring. Trumpet never collects in-network data; it only uses the network to communicate with other servers, so it has a low overhead. PathDump and SwitchPointer both use two VLAN tags of 24 bits for path and switch epoch information. NetSight always collects per-packet postcards to the host for analysis, and the per-packet additional bandwidth occupation is 15 bytes/packet  $\times$  average hop count because NetSight will generate a postcard for the packet at every hop. Marple introduces a 16-bit header to carry the per-packet end-to-end latency, and during the monitoring phase, it will group the packets with their per-hop queuing latency and sent them to the controller. SpiderMon adds a 16-bit monitor header to every packet when it enters the network, and removes it before forwarding the packet to the end-host as mentioned in §3.1.

**Switch resource overhead.** Figure 9(a) shows the switch resource usage of SpiderMon, which fits comfortably in a Tofino pipeline. It also shows how SpiderMon scales with the number of flows seen during a collection period. Modern data centers have millions of concurrent flows per switch, but since SpiderMon only keeps tens of milliseconds of history, the number of flows per epoch is much smaller. Switch memory size increases steadily over time [29], so SpiderMon can scale to even more flows with more recent hardware.



**Figure 10: Branch-search metrics for SpiderMon**

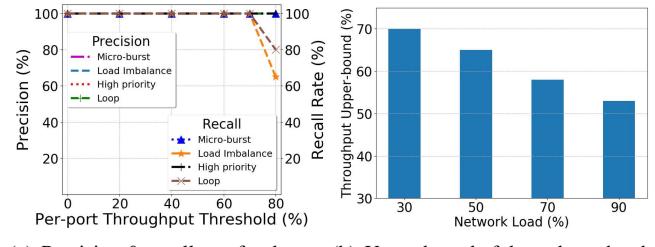
To show the benefit of informing related switches in the data plane in a distributed manner, we compare SpiderMon with a centralized reactive strawman system, which uses a centralized node to receive the detected problems, identifies the related switches, and retrieves data from them. We vary the additional latency that this centralized controller introduces. Figure 9(b) shows that this solution requires more memory to store a larger amount of historical data to avoid the loss of relevant evidence for diagnosis. In comparison, SpiderMon only needs to preserve the history within the maximum queuing latency + half RTT (§3.2.2).

### 4.3 Diagnostic Robustness

We finally evaluate the diagnostic robustness of SpiderMon using different metrics related to branch-search coverage, epoch length, and cumulative latency. Within a range of adjustments, SpiderMon can diagnose the performance problems with ideal precision and recall. Network operators are allowed to adjust the parameters of SpiderMon according to their requirements.

**Overall methodology.** SpiderMon empirically adjusts the parameters under different network loads. Given a particular network traffic load, operators could systematically test the precision and recall rates of SpiderMon with different metric choices. Suitable choices should strike a good balance between the recall rate and the size of collected telemetry data for throughput metrics, switch memory consumption for epoch metrics, and the sensitivity of problem detection for latency metrics. The optimal parameters vary under different network loads. We provide the results of parameter adjustments using our experimental settings in the following, while network operators could follow the same methodology to obtain their preferred parameters.

**Branch-search threshold.** SpiderMon provides different options for spider packet propagation in terms of its reach (e.g.,

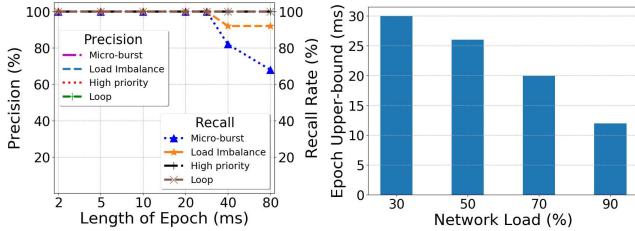


**Figure 11: Throughput metrics for SpiderMon**

all or some branches). Figure 10(a) and Figure 10(b) provide comparisons with different options on both the diagnosis time of root cause analysis and the size of collected telemetry data. Note that the number of relevant switches in SpiderMon is generally much smaller than the total network size since SpiderMon uses the wait-for relation and provenance model to precisely target only those relevant switches that contribute to the observed performance problem. Therefore, even with all-branches spider packets propagation (search all ports with  $> 0$  throughputs), SpiderMon is efficient compared to more rudimentary diagnosis strategies that must comb through all data from all switches. Even for relatively widespread performance problems involving up to 30 relevant switches, it takes under 4 seconds to run the root cause diagnosis algorithm (Algorithm 5) on a 4.3GHz CPU, as shown in Figure 10(c). In addition, we evaluate the latency for spider packets propagation and the subsequent retrieval of the telemetry data, using 50 Gbps link bandwidth and  $20\mu\text{s}$  link delay. From the results shown in Figure 10(d), we can see that a few microseconds are enough to perform the entire retrieval operation with arbitrary fat-tree topologies, no matter the choices of branch-search options. This is because SpiderMon’s mechanisms run in the data plane. As a result, network operators can send “spider” packets without setting the branch-search threshold if the overhead can be tolerated based on their requirements.

We further evaluate the precision and recall rates under different branch-search coverage with different network loads. Figure 11(a) shows the results under 30% network load, indicating that the precision can always achieve 100% while the recall rates decrease if the threshold is too high. To trade-off the branch-search overhead and the recall rates, we suggest using 70% as the threshold in this case since it strikes a good balance. Following the same strategy, we summarize the upper bound of branch-search thresholds for operators to adjust under different network loads, as shown in Figure 11(b).

**Epoch length.** SpiderMon can change the length of the telemetry epoch to save memory but trade-off telemetry granularity. Network operators can adjust the telemetry epoch according to their requirements. Under different network loads, we provide the upper bound of the epoch length. For example, Figure 12(a) shows the results with the network load at 30%. We evaluate the precision and recall rates under different epoch lengths. The precision is always 100%, while



(a) Precision & recall rate for the root causes with 30% load (b) Upper-bound of epoch length causes with 30% load

**Figure 12: Epoch metrics for SpiderMon**

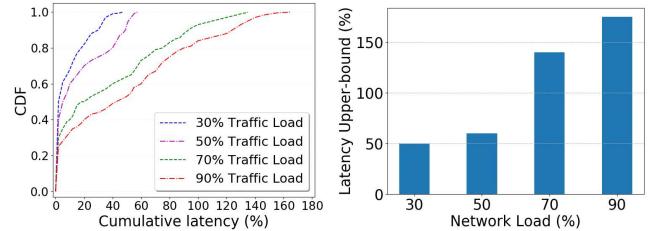
the recall rate decreases in some scenarios when the length of epoch exceeds 30 ms. We further measure the precision and recall rates under different network loads, and identify the upper-bounds of epoch length, as shown in Figure 12(b). The upper-bound epoch length used for telemetry collection decreases with increasing network load.

**Cumulative latency threshold.** SpiderMon provides a tunable cumulative latency threshold for problem detection, allowing network operators to customize problem trigger frequency for different applications. Figure 13(a) shows the CDF of different cumulative latency under different network loads in the absence of problems, where the cumulative latency is normalized by the maximum queuing latency of a single switch. Under different loads, the choice of cumulative latency threshold varies according to the trade-off between overhead and recall rate. The higher the sensitivity of the network to problem detection, the more switches are visited, and thus higher overhead. We further evaluate the recall rates of SpiderMon under different loads and summarize the upper bound of cumulative latency thresholds for reaching 100% recall in all scenarios in Figure 13(b).

## 5 Related Work

**Switch-based telemetry.** Telemetry systems such as Sonata [16], Marple [32], FlowRadar [26], \*Flow [36], NetSeer [47] and Dapper [14] leverage programmable switches for fine-grained data collection. However, query-driven systems [16, 32] cannot dynamically change the targeted events at small timescales, and blanket monitoring systems [17, 36] incur high collection overhead. SpiderMon aims to achieve lightweight yet accurate telemetry information collection. Two recent works, NetSeer [47] and PINT [8], share our high-level goal of reducing telemetry overhead. NetSeer detects per-flow performance events for compression, and PINT aggregates telemetry information across hops or flows to save bandwidth. Compared to these works, SpiderMon co-designs monitoring and posterior diagnosis based on wait-for relations for closed-loop diagnosis.

**Diagnosis systems.** SwitchPointer [38] and PathDump [37] collect both in-network and host data for diagnosis. Trumpet [31] monitors every packet at hosts and reports triggered events. SNAP [43] diagnoses network problems using logs (e.g., TCP statistics, socket calls) collected at hosts. How-



(a) Cumulative latency under different network loads (b) Upper-bound of cumulative latency threshold

**Figure 13: Latency metrics for SpiderMon**

ever, these systems rely on a central controller and perform software-based monitoring. NetMedic [23], 007 [6], NetPoirot [7] use statistical methods and/or machine learning to identify root causes. Network provenance [42] tracks how packets flow through a network and apply formal reasoning to identify root causes. Deter [25] can process and replay a TCP trace to diagnose performance degradation. Compared to these works, SpiderMon leverages the telemetry information from programmable switches, and it uses wait-for relations to reason about performance contention in-network. Our recent workshop paper sketches a similar roadmap [41], but it does not contain a concrete design, implementation, or evaluation.

**Monitoring.** Another line of recent work focuses on designing compact data structures [11, 18, 19, 27, 44] with tradeoffs between accuracy and resource footprints. OmniMon [19] divides flow-level monitoring across different network entities to satisfy resource constraints. BeauCoup [11] supports multiple distinct counting queries simultaneously while requiring a small number of memory accesses. These data structures complement SpiderMon by reducing switch resource usage.

## 6 Conclusion

SpiderMon is a system that achieves high coverage and low overhead in monitoring and diagnosing network performance problems. It monitors every flow in the data plane and triggers diagnostic events upon problem detection. It precisely collects diagnostic information in an as-needed fashion. We prototype SpiderMon on Tofino hardware and BMv2 software switches and show that it can leverage wait-for relations to accurately pinpoint root causes for complex problems. SpiderMon also has low overheads for telemetry collection, switch resources, and network bandwidths.

## Acknowledgment

We thank our shepherd Theophilus A. Benson and the anonymous reviewers for their valuable feedback. This research is sponsored by the NSF under CNS-1718980, CNS-1801884, and CNS-1815525.

## References

- [1] Network Congestion Management: Considerations and Techniques. <https://www.sandvine.com/hubfs/downloads/archive/whitepaper-network-congestion-management.pdf>.
- [2] sFlow. <http://www.sflow.org/>.
- [3] Solving the mystery of link imbalance: A metastable failure state at scale. <https://engineering.fb.com/production-engineering/solving-the-mystery-of-link-imbalance-a-metastable-failure-state-at-scale/>.
- [4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, et al. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.
- [5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.
- [6] 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.
- [7] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with NetPoirot. In *ACM SIGCOMM*, 2016.
- [8] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, 2020.
- [9] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SOSR*, 2016.
- [10] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich. Catching the microburst culprits with Snappy. In *SelfDN*, 2018.
- [11] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, 2020.
- [12] J. Cho, H. Chang, S. Mukherjee, T. Lakshman, and J. Van der Merwe. Typhoon: An SDN enhanced real-time big data streaming framework. In *ACM CoNEXT*, 2017.
- [13] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX NSDI*, 2018.
- [14] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of TCP. In *ACM SOSR*, 2017.
- [15] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, 2015.
- [16] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network monitoring as a streaming analytics problem. In *ACM HotNets*, 2016.
- [17] 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.
- [18] 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.
- [19] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao. OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy. In *ACM SIGCOMM*, 2020.
- [20] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4->NetFPGA workflow for line-rate packet processing. In *ACM FPGA*, 2019.
- [21] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally. Network congestion avoidance through speculative reservation. In *IEEE HPCA*, 2012.
- [22] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *ACM APSys*, 2018.
- [23] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM*, 2010.
- [24] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX NSDI*, 2019.
- [25] Y. Li, R. Miao, M. Alizadeh, and M. Yu. Deter: Deterministic TCP replay for performance diagnosis. In *USENIX NSDI*, 2019.
- [26] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI*, 2016.
- [27] 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.

- [28] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It’s good to relax! In *ACM PODC*, 2015.
- [29] 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 *ACM SIGCOMM*, 2017.
- [30] 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*, 2015.
- [31] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trum-pet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.
- [32] 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.
- [33] Y. Ran, X. Wu, P. Li, C. Xu, Y. Luo, and L.-M. Wang. EQuery: Enable event-driven declarative queries in programmable network measurement. In *IEEE NOMS*, 2018.
- [34] 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.
- [35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network’s (datacenter) network. In *ACM SIGCOMM*, 2015.
- [36] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with \*flow. In *USENIX ATC*, 2018.
- [37] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with PathDump. In *USENIX OSDI*, 2016.
- [38] P. Tammana, R. Agarwal, and M. Lee. Distributed network monitoring and debugging with SwitchPointer. In *USENIX NSDI*, 2018.
- [39] Y. Tang, Y. Wu, G. Cheng, and Z. Xu. Intelligence enabled SDN fault localization via programmable in-band network telemetry. In *IEEE HPSR*, 2019.
- [40] 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 *ACM SOSR*, 2017.
- [41] W. Wang, P. Tammana, A. Chen, and T. S. E. Ng. Grasp the root causes in the data plane: Diagnosing latency problems with SpiderMon. In *ACM SOSR*, 2020.
- [42] Y. Wu, A. Chen, and L. T. X. Phan. Zeno: Diagnosing performance problems with temporal provenance. In *USENIX NSDI*, 2019.
- [43] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*, 2011.
- [44] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.
- [45] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM IMC*, 2017.
- [46] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: Generic off-CPU analysis to identify critical waiting events. In *USENIX OSDI*, 2018.
- [47] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, 2020.
- [48] 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.

## A Proof for Contributors Identification Algorithm

**Definition 6.** *Degree of vertex. In a WFG, the degree of vertex A is the sum of all the adjacent edges' weights  $w_e$ :*

$$D(A) = \sum_{\{e = <i,j> | i=A \text{ or } j=A\}}^e \alpha_e \cdot w_e \quad (1)$$

where  $\alpha_e$  is 1 when A is the sink of edge e and -1 when vertex A is the source.

**Lemma 1.** *For a WFG, the sum of all the vertex's degree is 0:*

$$\sum_{X \in V}^X D(X) = 0 \quad (2)$$

Proof: the WFG is a directed graph where every edge is pointing from a vertex to another vertex in the graph, so each edge will add weight  $w$  to the sink vertex and weight  $-w$  to the source vertex.

**Definition 7.** *Flux of cut. For a cut in a WFG, the vertex will be divided into two sets, S1 and S2. Given all edges in the WFG has a positive weight according to the definition, we denote the flux of this cut as:*

$$\text{Flux}(cut) = \left| \sum_{i \in S1, j \notin S1}^{e = <i,j>} w_e + \sum_{i \notin S1, j \in S1}^{e = <i,j>} -w_e \right| \quad (3)$$

where  $e$  represents the edge from vertex  $i$  to vertex  $j$

Though the sum of all vertex's degree is 0, we can always find a cut whose flux is maximum, representing the provenance relation between vertexes from those two groups is the strongest. The set with a positive degree considers as the main contributor to the queue, while the other set contains victims of the queue, like normal flows or small flows. To find this cut efficiently, we have shown the hints by the following lemmas and theorems.

**Lemma 2.** *The flux of one cut is just the absolute value of the sum of all vertices' degrees in either set.*

Proof: The absolute value of the sum of all vertices' degrees in one set (ASD) can be written as:

$$\begin{aligned} \text{ASD} &= \left| \sum_{i \in S1, j \in S1}^{e = <i,j>} \alpha_e \cdot w_e \right| \\ &= \left| \sum_{i \in S1 \& j \in S1}^{e = <i,j>} \alpha_e w_e + \sum_{i \in S1 \& j \notin S1}^{e = <i,j>} \alpha_e w_e + \sum_{i \notin S1 \& j \in S1}^{e = <i,j>} \alpha_e w_e \right| \\ &= \left| 0 + \sum_{i \in S1 \& j \notin S1}^{e = <i,j>} -w_e + \sum_{i \notin S1 \& j \in S1}^{e = <i,j>} w_e \right| = \text{Flux}(cut) \end{aligned} \quad (4)$$

**Theorem 1\*.** *The WFG cut with maximum flux will divide the vertices with positive degrees into one set and negative degrees into the other set.*

Given the sum of all vertices' degrees are 0, for any cut:  $\sum_{X \in S1} D(X) = -\sum_{Y \in S2} D(Y)$ , namely, the absolute sum of degree for two sets are the same. Thus, for the cut that divide all vertices with positive degrees into one set, by contradiction, we can easily prove this is the cut with maximum flux.

The flux represents the wait-for relation between two groups from a cut of the wait-for graph, and the degree represents the value of incoming edges weights subtracting outgoing edges weights so that Theorem 1 is proofed.

## B Fine-grained Sliding Window

During the telemetry collection process, SpiderMon maintains bloom filter and per-port per-epoch data structures to trace back all the relevant switches. However, part of these structures (e.g. traffic meter) needs to be reset to 0 at the beginning of an epoch due to the limited resources of the switch data plane. Therefore, there will be some information loss at the beginning of an epoch, leading to the diagnosis algorithm being inaccurate. SpiderMon employs a fine-grained sliding window on the data plane to achieve high accuracy for the used data structures.

The sliding window strategy slices each epoch into multiple pieces, and it proceeds in two actions: an update action and a decrease action. To explain simply, we take the traffic meter in the per-port data structure as an example. Assume one epoch  $T$  is divided into  $n$  small time slots. There will be  $n$  sub-traffic meters and each of them aims at a single time slot. When a switch receives a new packet during the update phase, the switch will update the corresponding sub-traffic meter based on the current time slot, as well as the total traffic meter. For decrease action, when the oldest sub-traffic meter no longer exists in the sliding window, the value of the corresponding sub-traffic will be subtracted from the total traffic meter and that sub-traffic meter will be reset to 0. Network operators are able to tune the fine-grained sliding window according to their demands. Basically, the more time slots an epoch is divided into, the higher the accuracy that the system can achieve. On the other hand, the overhead of telemetry data structures can be reduced with fewer time slots.

## C Resource Usage Optimization

### C.1 Avoid Duplicate Detection

In the scenario of the performance problem, there are lots of packets from the victim flow suffering from high latency problems, but not all of them will generate a diagnostic event independently. SpiderMon sets a limitation on the interval between two diagnostic events generated by the same flow, meaning that during one congestion, only the first packet suffering from high accumulated latency will trigger the diagnostic event. To avoid receiving multiple audit requests for the same diagnosis event, the switches will drop the duplicate "spider" packets with the same event ID as well.

## C.2 Data Field Compression

For the applications like SpiderMon built on top of the programmable switches, keeping track of some data fields in the packet header or on the switch memory is always required. Compressing those data fields in order to reduce the extra header size or switch memory occupation is critical to the application performance. SpiderMon provides a method to compress the size of the data by extracting the most significant bits. This idea can be widely applied to many recorded data in such systems, and here are two typical examples that use this strategy:

The timeout bloomfilter in SpiderMon requires storing a large number of timestamps for each slot in the bloom filter, which is very resource consuming and inefficient. The timestamp is usually stored with 48 bits on the switch and SpiderMon uses the timestamp to perform the timeout operation. Given that the only operation on the timestamp is the subtraction of two timestamps and compare the difference with the timeout period, we can easily observe that the only significant bits in the timestamp are the bits around the period. Take the timeout period as 1 ms as an example, the most significant bits in the timestamp are the 10th, 11th, and 12th bits from the right, representing 0.512 ms, 1.024 ms, and 2.048 ms respectively. By extracting these three bits from the original timestamps and comparing the difference with bit array 010, we can get an approximation of the exact value that is calculated with the original timestamp. Adding more bits on the left (*e.g.* 13th and 14th) can prevent us from the danger of overflow while adding more bits on the right (*e.g.* 9th and 8th) can help us obtain a more precise result of the subtraction. With this method, SpiderMon only needs to store 6 bits for each timestamp and reduce the memory usage of the timeout bloomfilter by 87.5%.

Another example is the queuing information carried by the packets in SpiderMon, which is used to detect the performance problem by comparing the accumulated delay with the maximum delay threshold. For a certain application, the maximum delay threshold may be 1 ms. Then when we calculate the accumulated delay, the most significant bits are 8th, 9th, and 10th bits from the right, representing 0.128 ms, 0.256 ms, and 0.512 ms respectively. If any bit on the left of the 10th bit is not 0, SpiderMon will trigger the problem immediately, since it exceeds the threshold with this single-hop delay. In this way, SpiderMon only needs to add an extra header with 4 bits to carry each delay field instead of 19 bits, shrinking the overhead from the extra header by 78.95%. Note that in evaluation, we use 8 bits for each data field to provide better accuracy.

## D Implementation

We have implemented SpiderMon on a Barefoot Tofino switch with 1147 lines of P4-Tofino code and also a BMv2 version for NS3 and MiniNet environments with 945 lines of P4 code. We also implement the root causes analyzer on the end-host

with 843 lines of Python code.

Figure 14 depicts different components in a switch and the workflow for different packet types. The event record is used for checking duplicate “spider” packets, and the telemetry counter for guiding telemetry packet generation. Those two data structures are placed in the ingress because they need to make decisions on whether to mirror packets in the traffic management unit. The per-port meter and timeout bloom filter provide provenance data to guide the propagation of the “spider” packets, and the telemetry data structure stores historical flow information for diagnosis. Those two data structures, along with the problem detection component, are placed in the egress pipeline because they may require queuing information, which is only available in the egress pipeline. Note that the per-port telemetry information is stored separately on the switch, but not necessarily one table per stage. One stage in SpiderMon can store multiple egress ports’ telemetry information.

To implement SpiderMon, the egress pipeline is required to detect the problems, store telemetry information, and provide temporary provenance hints for “spider” packet propagation. For switch architectures like SimpleSumSwitch [20] (NetFPGA), P4FPGA [40], and SmartNICs, SpiderMon can also be implemented by taking the next switch’s pipeline as the “egress pipeline” of former switches to detect congestion and collect telemetry information. This design requires more communication among switches, so both the latency for diagnosing the problem and the link bandwidth used by SpiderMon would also increase.

As for the hardware switch resource, modern switches have increasing memory sizes [29], and more ports usually represent more on-chip memory, which, we shall demonstrate in §4, is more than sufficient to support SpiderMon.

## E Additional Experiment Results

### E.1 Header Bandwidth Usage

Packet Size (B)	1480	1000	500	100
SpiderMon (Gbps)	23.51	23.5	22.84	20.51
Baseline (Gbps)	23.65	23.5	22.84	21.87

**Table 1: SpiderMon’s maximum throughput is quite close to the baseline switch with only forwarding rule.**

As the monitor header added by SpiderMon is removed before forwarding the packet to the end-host, the corresponding overhead of the additional header is very trivial. We use iPerf to show the maximum throughput of traffic with different average packet sizes on the Tofino switch equipped with SpiderMon in Table 1 and compare it with a baseline switch program with only basic forwarding rules. As expected, SpiderMon’s end-to-end throughput is nearly identical to the baseline, meaning that the bandwidth overhead of the monitoring phase could be neglected.

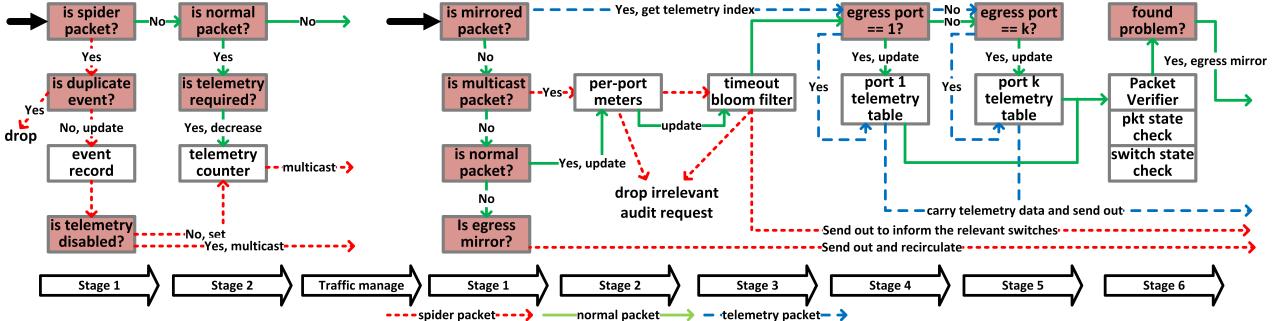


Figure 14: The placement of SpiderMon components on the switch stages

## E.2 Cache & Hadoop Workloads

Besides the Web search trace, we also run the same experiments on the Cache trace and Hadoop Trace.

For the Cache trace, most of its flow sizes fall into 1KB to 100KB. Thus, to reach the same link utilization, we have to insert more number flows during the simulation. The results for Cache trace are similar to the Web search trace. The only difference is that all algorithms have improved performance. This is because the flow sizes are very small so that the root-cause traffic (e.g. micro-burst) flow can be easily distinguished from the normal flows; false positive and false negative are reduced.

For the Hadoop trace, most of the flows have less than 10 KB flow size. Similar to the Cache trace, we also increase the number of flows to keep the same link utilization. The overall results for the Hadoop trace are also similar to the Cache trace.

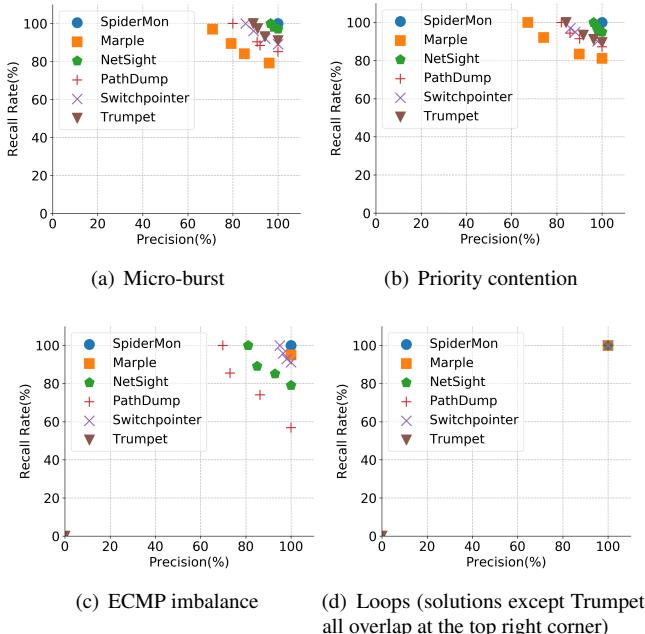


Figure 15: Diagnostic effectiveness with Cache trace

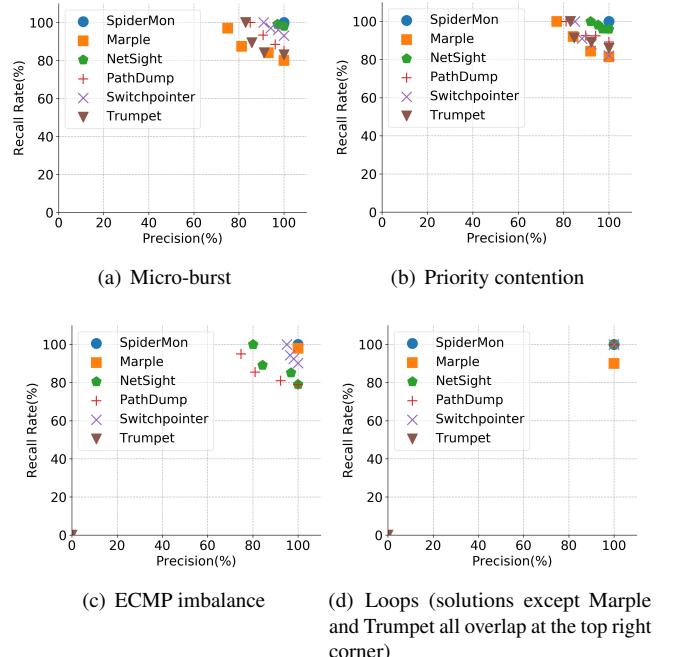


Figure 16: Diagnostic effectiveness with Hadoop trace

## F Tunable Parameters for Different Solutions

We vary the following parameters when using those systems to diagnose problems of the four scenarios. The goal is to find the parameter sets with the best precision and recall rate. We do nested iterations over different parameters by fixing some parameters and iterate the other parameters. The parameters are different across systems, and for the same system, the parameters vary according to the scenarios that we are trying to diagnose. The details are shown in Table 2 and Table 3.

## G Constructing Signatures for Root Causes

SpiderMon uses both the collected telemetry information and the static network configuration information to recognize the root causes. The telemetry information is collected by SpiderMon, and the configuration information is simply provided

	Micro-burst-related Contention	Priority-related contention
Trumpet	Tolerable per-flow throughput, tolerable end-to-end latency difference, tolerable TCP packet loss	Tolerable per-flow throughput, tolerable end-to-end latency differences, tolerable TCP packet loss
PathDump	Tolerable per-flow throughput, tolerable link utilization	Tolerable per-flow throughput, tolerable link utilization
SwitchPointer	Tolerable per-flow throughput, tolerable link utilization	Tolerable per-flow throughput, tolerable link utilization
NetSight	Related time interval length, tolerable link utilization	Related time interval length, tolerable link utilization, postcard arrival sequences
Marple	Network-wide query lasting time, tolerable per-flow throughput	Network-wide query lasting time, tolerable per-flow throughput
SpiderMon	Maximum allowed flow throughput	/

**Table 2: Parameters for micro-burst and priority**

	ECMP load imbalance	Loop
Trumpet	/	/
PathDump	Tolerable link utilization, tolerable link utilization imbalance ratio	Maximum header size
SwitchPointer	Tolerable link utilization, tolerable link utilization imbalance ratio	Maximum header size
NetSight	Related time interval length, tolerable link utilization, tolerable link utilization imbalance ratio	/
Marple	Network-wide query lasting time, tolerable link utilization imbalance ratio	Network-wide query lasting time
SpiderMon	Tolerable link utilization imbalance ratio	/

**Table 3: Parameters for load imbalance and Loop**

by the topology information and routing information, which is known by the operator in advance.

To add a new signature for a new root cause, network operators could simply use the above information to construct their own signatures. Here we provide some telemetry information and static configuration information used in the 4 example signatures in Table 4. This is not an exhaustive list and more information could be added when new signatures are introduced. To construct new signatures, we should know that any signature consists of two parts: 1) the root cause's pattern, like a flow with large throughput for the micro-burst root cause; 2) the relation between the problematic flow and the victim flow, namely, the problematic flow should be one of the main contributors to the victim flow's poor performance. Here we also provide 4 different signatures as examples.

Telemetry Info	Edge weight from flow i to flow j: $E(flow_i, flow_j)$
	Main contributors for a queue: $Contributors(Switch_iPort_j)$
	Flows traveling a switch port: $Flows(Switch_iPort_j)$
	Priority: $P(flow)$
	Data volume: $V(flow)$
Config Info	Port mapping in Topology: $Topo(Switch_iPort_j)=Switch_xPort_y$
	Flows belonging to an ECMP group: $Flows(group)$

**Table 4: Selected telemetry information and static configuration information**

**Micro-bursts.** SpiderMon can identify all the main flow-level contributors at different hops along the victim flow's historical path. As shown in Figure 5(a), the micro-burst flows have many wait-for edges with large weights pointing to them-

selves due to a large amount of traffic during the problematic time. For example, for the micro-burst problem, there must exist one micro-burst node *root* which satisfies:

The root cause flow has the same priority as the victim flow:

$$P(victim) = P(root) \quad (5)$$

The root cause flow has similar edge weight to itself as to other flows:

$$E(root, root) \approx E(victim, root) \quad (6)$$

The victim flow contends with the root cause flow:

$\exists m, n$ , where

$$victim \in Flows(Switch_mPort_n) \quad (7)$$

$$root \in Contributors(Switch_mPort_n)$$

The larger the weights of  $E(root, root)$  and  $E(victim, root)$ , the more confidence SpiderMon has on determining the micro-burst flow.

**Different priorities.** For contention between flows with different priorities, SpiderMon checks the priority of the victim flow and the main flow-level contributors. The contributor flows with higher priority compared to the victim flow can be identified as the root causes, as shown in Figure 5(b). The high priority flow *root* should satisfy:

The root cause flow has higher priority than the victim flow:

$$P(victim) < P(root) \quad (8)$$

The root cause flow has smaller edge weight for the edge pointing to itself than the edge pointing to the victim:

$$E(root, root) < E(victim, root) \quad (9)$$

The victim flow contends with the high priority flow:

$\exists m, n$ , where

$$victim \in Flows(Switch_mPort_n) \quad (10)$$

$$root \in Contributors(Switch_mPort_n)$$

**ECMP load imbalance.** For the load imbalance problem displayed in Figure 1(b), SpiderMon will find the flow-level main contributors and check if they are routed by ECMP. Then SpiderMon calculates the ECMP imbalance ratio with the throughput of all flows routed by ECMP rules, using the traffic volume provided by per-flow telemetry data. The problematic ECMP groups can be identified when the calculated ratio is highly imbalanced as in Figure 5(c). Within the problematic ECMP group *ecmp* on Switch *Switch<sub>x</sub>*, there must exist one or more flows *root*, which satisfies:

The ECMP traffic split on some switches is not balanced:

$$\text{Throughput}(\text{Switch}_x \text{Port}_y) = \sum V(\text{flow}_i), \quad (11)$$

where  $\text{flow}_i \in \text{Flows}(\text{Switch}_x \text{Port}_y)$

$$\begin{aligned} & \exists x, y, \forall i \neq y, \\ & \text{Throughput}(\text{Switch}_x \text{Port}_y) \\ & > \text{Throughput}(\text{Switch}_x \text{Port}_i) \end{aligned} \quad (12)$$

The root cause flow is one of the flows from the ECMP port that has larger throughput.

$$\text{root} \in \text{Flows}(ecmp) \cap \text{Flows}(\text{Switch}_x \text{Port}_y) \quad (13)$$

On another switch, the victim flow contends with the root cause flow:

$$\begin{aligned} & \exists m, n, \text{where} \\ & \text{victim} \in \text{Flows}(\text{Switch}_m \text{Port}_n) \\ & \text{root} \in \text{Contributors}(\text{Switch}_m \text{Port}_n) \end{aligned} \quad (14)$$

**Transient/persistent loops.** For the latency problem caused by transient or persistent loops as shown in Figure 1(c), Spider-Mon searches the port-level contributors along the contributor

traffic's path. If the same port is observed twice during the search procedure, all those ports are highly likely to have formed a loop for specific traffic. Furthermore, the flow ID will be checked to further confirm the transient/persistent loop. The formal signature for a flow *root* with a transient/persistent loop can be written as:

$$\text{Exist a port list: } [\text{Switch}_{m_0} \text{Port}_{n_0}, \dots, \text{Switch}_{m_k} \text{Port}_{n_k}] \quad (15)$$

The port list forms a ring in the topology and the root cause flow routed in a loop on that ring:

$$\begin{aligned} & \forall i, \\ & \text{Topo}(\text{Switch}_{m_i} \text{Port}_{n_i}) == \text{Switch}_{m_{i+1}} \text{Port}_{n_{i+1}} \\ & \text{root} \in \text{Flows}(\text{Switch}_{m_i} \text{Port}_{n_i}) \end{aligned} \quad (16)$$

The victim flow contends with the loop traffic on one of the switches on that ring:

$$\begin{aligned} & \exists j, \text{where } j \in [0, 1, \dots, k] \\ & \text{victim} \in \text{Flows}(\text{Switch}_{m_j} \text{Port}_{n_j}) \\ & \text{root} \in \text{Contributors}(\text{Switch}_{m_j} \text{Port}_{n_j}) \end{aligned} \quad (17)$$



# Collie: Finding Performance Anomalies in RDMA Subsystems

Xinhao Kong<sup>1,2</sup> Yibo Zhu<sup>2</sup> Huaping Zhou<sup>2</sup> Zhuo Jiang<sup>2</sup>  
Jianxi Ye<sup>2</sup> Chuanxiong Guo<sup>2</sup> Danyang Zhuo<sup>1</sup>

<sup>1</sup>Duke University <sup>2</sup>ByteDance Inc.

## Abstract

High-speed RDMA networks are getting rapidly adopted in the industry for their low latency and reduced CPU overheads. To verify that RDMA can be used in production, system administrators need to understand the set of application workloads that can potentially trigger abnormal performance behaviors (e.g., unexpected low throughput, PFC pause frame storm). We design and implement Collie, a tool for users to systematically uncover performance anomalies in RDMA subsystems without the need to access hardware internal designs. Instead of individually testing each hardware device (e.g., NIC, memory, PCIe), Collie is holistic, constructing a comprehensive search space for application workloads. Collie then uses simulated annealing to drive RDMA-related performance and diagnostic counters to extreme value regions to find workloads that can trigger performance anomalies. We evaluate Collie on combinations of various RDMA NIC, CPU, and other hardware components. Collie found 15 new performance anomalies. All of them are acknowledged by the hardware vendors. 7 of them are already fixed after we reported them. We also present our experience in using Collie to avoid performance anomalies for an RDMA RPC library and an RDMA distributed machine learning framework.

## 1 Introduction

Data center applications relentlessly demand low packet latency and high CPU efficiency. That makes Remote Direct Memory Access (RDMA) an appealing solution for cloud providers and other data center operators. Today, many top companies have already adopted RDMA in their data centers [11, 20, 46]. RDMA has been integrated into many application domains, such as graph processing [2, 41], data stores [4, 16], and deep learning [14, 44].

To deploy RDMA in production, i.e., using RoCEv2 for Ethernet-based data center network, we need to make sure that the RDMA network performance can meet our expectations, free of performance anomalies like low throughput and pause frame storm [11, 13, 32, 46]. This is important because applications require high-performance RDMA networks to de-

liver their service-level objectives (SLO). Furthermore, some abnormal behaviors, like pause frame storms, can cause catastrophic consequences including deadlocking the entire data center network [8, 11, 13, 37].

We have encountered the following anomalies in our RoCEv2 production environment:

- A particular application workload’s performance of the same RDMA NIC (RNIC) varies substantially on servers with only a slight difference in their PCIe specifications.
- A specific application workload only triggers pause frame storms with certain NUMA settings on a particular RNIC combined with particular server hardware.
- A particular application workload triggers pause frame storms with only a single connection on a particular RNIC from a particular vendor.

Although we collaborate with the most reliable vendors and they have conducted extensive tests on individual devices, the entire RDMA subsystem still has anomalies. The RDMA subsystem consists of RNICs and other server hardware that interacts with the RNICs. Our observation is that most of the anomalies are highly related to the interactions between RNICs and rest of the server hardware. Additional integration tests are thus critical, and we usually conduct these tests on our own because of two reasons. First, vendors cannot access our highly customized hardware, system configurations, and applications. Second, anomalies are too critical for the reliability and performance of the entire data center network, and we cannot completely rely on third parties for testing.

Currently, there are two approaches to conduct tests over the entire subsystem. The first approach is to run simple test benchmarks (e.g., PerfTest [34]) to conduct basic throughput and latency tests. The second approach is to run a set of representative RDMA applications. Unfortunately, these two approaches are not able to comprehensively uncover RDMA subsystem anomalies. The fundamental problem is that these approaches only test simple or existing workloads. They therefore fail to capture anomalies comprehensively because real

application workloads change over time. In addition, even if an anomaly is found with an application workload, application developers do not know how to modify the workload to avoid the anomaly.

Our goal for this paper is to explore the possibility of **systematic search** for application workloads that can trigger performance anomalies in RDMA subsystems. Finding these anomalies for the vendors can help them improve their hardware and thus improve the reliability and the performance of the entire data center network. Besides, the systematic approach can help developers understand the conditions to trigger such anomalies and how to avoid them by changing application workloads.

To realize this goal, the first question is *how to formally define an anomaly?* Having such a definition is difficult because application performance highly depends on the workload and the hardware. In this paper, we focus on two types of performance anomalies that can be precisely defined: no PFC pause frames if the network is not congested and throughput should be bottlenecked either by bits/second or packets/second as in RNIC specification.

Given this definition, we still need to address three challenges. The first challenge is how to build a comprehensive workload search space. An ideal approach for testing with the entire RDMA subsystem is to exactly modeling each component and then construct the search space. However, this is extremely hard for us, given the black-box nature of RNIC and other hardware components. The second challenge is even after we successfully construct a comprehensive enough search space, how can we search efficiently? The search space is inherently very large because RDMA subsystems are complicated. For example, traffics within an RDMA subsystem can be from/to different memory devices (e.g., main memory and GPU memory) and the transportation setting for a given workload is various (e.g., number and type of connections). Conducting tests blindly in such a large space is inefficient. The third challenge is how to find the complicated triggering conditions of such anomalies? This is important both during the search and after the search. During the search, we need the triggering condition to avoid testing similar application workloads for the same anomaly to speed up the search. After the search, we need to use these conditions to help developers avoid anomalies.

To this end, we design and implement Collie, the first tool to systematically uncover RDMA subsystem performance anomalies, with the following three ideas.

Our first idea is to construct the search space from a developer’s perspective. Though the underlying hardware is various and opaque to us, the narrow-waist RDMA programming abstractions (i.e., *verbs*) are clearly defined and stable. All application workloads can be interpreted as a combination of *verbs* operations. We carefully analyze the standard *verbs* library and the design decisions developers are allowed to make (the request pattern, how RDMA buffers are allo-

cated, etc.). Moreover, to cover the entire RDMA subsystem, we analyze all the potential data flows within a given server configuration. In this way, Collie constructs a comprehensive search space for application workloads in the domain of RDMA subsystem, including the host of the network traffic (e.g., GPU connected to a different PCIe bridge from the RNIC, DRAM from a different CPU socket), message sizes, number of connections, and memory region configurations.

Our second idea is that we can use two sets of counters to guide the search. The first set is the performance counters (e.g., bits per second), which are provided by all commodity RNICs and other hardware components. In addition, modern commodity RNICs and other hardware components provide diagnostic counters (e.g., PCIe backpressure). Diagnostic counters are mapped to particular unexpected events that happen to the hardware components. These counters are currently only used for debugging and monitoring purposes. Collie uses search algorithms based on simulated annealing to maximize/minimize counter values to uncover anomalies.

Our third idea is to find the minimal area in the search space that covers the found anomalies. We call this area (i.e., the conditions to trigger the anomaly) the minimal feature set (MFS). Collie includes a MFS algorithm to test each feature that an anomaly has (e.g., number of connections) and generate the necessary conditions set. With the MFS algorithm, Collie can further improve search efficiency by avoiding redundant tests of the same area. Also, finding the triggering conditions of an anomaly allows developers to avoid the anomaly by breaking one of the provided conditions.

We evaluate Collie on 8 different RDMA subsystems, including 6 types of RNICs from NVIDIA Mellanox and Broadcom, with speeds between 25 Gbps and 200 Gbps. Before we build Collie, we already know 3 existing performance anomalies by testing with existing RDMA applications. Collie successfully reproduces all of them and has found 15 new anomalies. We report these anomalies to the vendors, and all of them are acknowledged. 7 of them are already fixed by firmware upgrade or detailed configuration following our vendors’ instructions. We also describe our experience in using Collie to guide an RDMA RPC library and an RDMA distributed machine learning framework to avoid these anomalies. These experiences show Collie can help data center operators to uncover anomalies and assist RDMA application developers to implement better applications.

This work makes the following contributions:

- We design a developer-oriented approach to systematically construct a search space of application workloads to find performance anomalies in RDMA subsystems.
- We propose the first work to leverage hardware counters to guide the search for performance anomalies. These counters do not have proprietary hardware knowledge. This makes Collie general and useful for all types of RDMA subsystems.

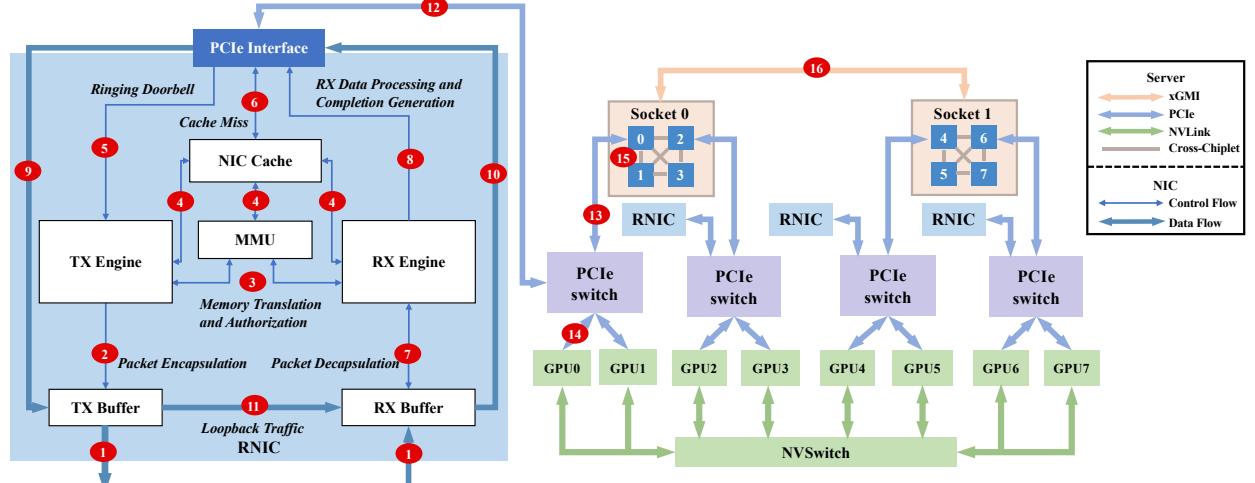


Figure 1: An example of an RDMA subsystem (RNIC internal design and its deployment environment in a server). Red circles mean potential performance bottlenecks that can trigger performance anomalies.

- We develop a simulated annealing based search algorithm and MFS algorithm. These algorithms speed up search and help developers avoid anomalies.
- We implement Collie, the first tool to help data center operators to uncover and avoid RDMA subsystem performance anomalies. Collie has found 18 anomalies (3 known ones and 15 new ones). We present these anomalies, their mitigation strategies, and their implications.

## 2 Background

### 2.1 RDMA Subsystem Performance Anomalies

RDMA is increasingly deployed in data centers for applications to achieve high throughput and high CPU efficiency. An application process can directly communicate through an RNIC with a remote process without involving either side's CPUs. RDMA requires a lossless network to achieve high performance. The default technology to deploy RDMA for Ethernet-based data centers is RoCEv2 [11, 46]. It relies on Priority-based Flow Control (PFC) [35] mechanism to guarantee a lossless network: once an ingress queue length exceeds a threshold, the switch/NIC sends out a PFC pause frame to the upstream egress queue, asking the egress queue to pause for a duration to avoid ingress queue overflow.

RDMA subsystem performance does not always meet user expectations and can have severe performance anomalies. According to our production experience, specific application workloads can trigger hardware bottlenecks of a particular type of RDMA subsystem and cause the entire subsystem performance to drop drastically. Applications of the same subsystem will be affected (e.g., throughput drop) and miss the service level agreement. Worse still, an anomalous RDMA subsystem can send out a large amount of PFC pause frames, which pauses the priority queue of the corresponding switch

port and may threaten the entire data center network, such as causing head-of-line blocking and PFC deadlocks [11, 13, 28]

Though the vendors of RNICs and other hardware components (e.g., GPU, motherboard) have conducted extensive tests on their products, we still find many anomalies in our RoCEv2 production environment. The fundamental reason is that RDMA performance is highly related to the entire RDMA subsystem, consisting of both RNIC internals and other hardware components. **This figure is based on public resources [24, 32, 42] and does not expose proprietary information. Our conversation with Mellanox indicates that a real RNIC is much more complex than our figure shows.** To the best of our understanding, an RNIC has at least 6 components: (1) a *TX engine* that receives doorbells (a signal mechanism for the server to notify RNIC to send a request), fetches and processes requests, and initiates transmission; (2) an *MMU* that translates the virtual address to physical address for RDMA memory regions; (3) an SRAM-based *NIC cache* that caches per-connection metadata and memory translation table; (4) a *RX engine* that processes incoming data and generates completion to notify server; (5)(6) *buffers* that hold packets to transmit and received packets. An RNIC is connected to a server via PCIe. The server has two CPU sockets and each CPU socket has four CPU chiplets (Only AMD CPUs and new-generation Intel CPUs have cross-chiplet communication, otherwise all the cores inside a CPU socket share the last-level cache.) RNICs and GPUs are all connected to PCIe switches.

There are many potential performance bottlenecks inside the RNIC and between the RNIC and other hardware components within the RDMA subsystem. We use red circles to show such potential bottlenecks (in Figure 1). When these bottlenecks are triggered, the network performance may drop and the RNIC can even send out pause frames to reduce

the amount of traffic going through the RNIC. We find that many anomalies only occur when multiple bottlenecks or the bottlenecks between different components are triggered. For example, when the RNIC receives a packet, it will store the packet in RX buffer, process the packet (circle 7), and finally DMA the content to main memory or GPU memory (circles 10, 12, 13 or circles 10, 12, 14). Normally, the RX buffer won't accumulate much because the PCIe bandwidth is larger than RNIC's line rate (circle 1). However, once there exists loopback traffic (e.g., the client and server are collocated on the same host), the loopback traffic (circle 11) may drain the PCIe bandwidth and cause RX buffer accumulation. It depends on both the RNIC and the PCIe slot. The worst consequence is that the RNIC keeps sending a large amount of PFC pause frame and threatens the entire data center network. Vendors' individual tests are not able to uncover this anomaly because it depends on the combination of circles 1, 11, 12 (even more) from different components. Further, data center operators like us may use highly customized hardware or specific system configurations that are not accessible to vendors. This makes it necessary and crucial for us to conduct our own independent tests before deploying RDMA hardware in production, especially for anomalies that can potentially generate pause frame storms.

## 2.2 Existing Approaches

Data center operators' tests are integration tests: instead of testing individual hardware components, these tests focus on the performance of the entire RDMA subsystems. There are two existing approaches. The first approach is to run a set of test traffic, such as `PerfTest` [34] and `OSU` micro-benchmarks [33]. The second approach is to run a representative set of real applications. However, these two approaches can not uncover RDMA subsystem performance anomalies comprehensively. For example, we deploy 200 Gbps RNICs in our clusters to support a performance-critical distributed machine learning framework. We test the machine learning framework on the cluster of these RNICs, and there is no performance anomaly found. We also have done extensive testing both with synthetic testing workloads and other real applications before deployment. However, months after deployment, our developers find that the performance of the framework has reduced significantly, even worse than just using 100 Gbps RNICs. At the same time, a substantial amount of pause frames are generated from these 200 Gbps RNICs. This is strange because pause frames usually appear with hundreds of connections that trigger congestion, but our machine learning framework only creates a few connections between each server pair. We stopped the machine learning framework and ran our performance tests again, and everything is normal. After several weeks of careful debugging, we finally realize that the case only happens when the application (1) use one-sided RDMA operations with Reliable Connection, (2) has bidirectional traffic, (3)

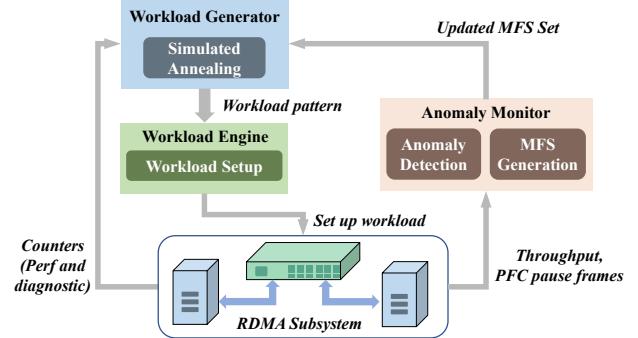


Figure 2: System Overview. The workload engine sets up RDMA traffic. The anomaly monitor detects performance anomalies and their minimal feature sets. The workload generator fetches hardware counters and decides the workload pattern to test.

uses a particular workload including a mixture of small and large messages, (4) with 200 Gbps RNIC on particular AMD servers. We find that the developers for our machine learning framework slightly modified their code after passing our application tests. The new workload contains messages of mixed lengths (i.e., a large message followed by a small message followed by a large message in bidirectional traffic), which triggers a performance bottleneck between the RNIC and the PCIe switch. This is not a problem with 100 Gbps RNICs from the same vendor or on other types of servers.

**The fundamental reason** why current approaches fail to uncover such anomalies is that they only test existing workloads and inherently are not able to capture anomalies triggered by unknown workloads. However, real application workloads are various and will change over time. Besides, even current approaches have found such anomalies, it is hard and time-consuming to locate the triggering conditions. Capturing the triggering conditions of performance anomalies allows data center operators to work with vendors to fix potential hardware/firmware bugs, and improve the reliability and performance of the data center network. When fixes to the anomalies are not immediately available (e.g., firmware upgrade, hardware replacement), application developers can build high-performance RDMA applications by avoiding workload that can trigger anomalies.

## 3 Overview

We build Collie, the first tool to help data center operators systematically search for application workloads that can trigger performance anomalies.

The first question we need to answer is *how to define an anomaly?* Unfortunately, today there does not exist such a definition. Having such a definition is fundamentally hard because application performance (e.g., latency) can be highly dependent on the workload and the hardware. Instead of trying to capture the entire set of anomalous behaviors, we focus on two types of performance anomalies that are of great importance in production environment and can be precisely defined: when applications keep injecting RDMA traffic into

the network, (1) no PFC pause frames should be generated if the network is not congested; (2) throughput should be bottlenecked either by total bits/second or total packets/second as in RNIC specifications. The first definition ensures that an RDMA subsystem will not threaten the entire data center network and the second ensures that an RDMA subsystem’s capability matches user expectation.<sup>1</sup> We discussed this definition with several hardware vendors, and they all agree with our definition. Even though some anomalies may be due to system limitations rather than bugs, it is also important for both vendors and us to be aware of them. We report all the anomalies we found using this definition to the hardware vendors, and they acknowledged all the reported anomalies. We believe that this definition naturally matches the application developer’s mental model of RDMA and thus allows developers to roughly estimate the network performance.

Given this definition of anomaly, we still need to overcome three major research challenges.

**Challenge #1:** How to design a comprehensive workloads search space for a given RDMA subsystem? An ideal solution is to carefully analyze and modeling the entire RDMA subsystem, and then construct the search space from the perspective of hardware. This complete white-box approach allows us to test all bottlenecks and the combinations of them given an RDMA subsystem. However, it is impractical for data center operators like us due to the black-box nature of RNICs and other hardware components. Our key observation is that though the components of RDMA subsystems are black boxes and there are diverse RDMA applications, the abstractions between the hardware and applications are clearly defined and stable. All application workloads are essentially composed of a series of basic *verbs* operations, a *narrow waist* of the RDMA programming. With this observation, we carefully analyze this RDMA programming abstraction and design a general search space ([§4](#)).

**Challenge #2:** How to search efficiently? Due to the complexity of RDMA subsystems and the variety of workloads, the size of search space is very large. Unfortunately, none of existing heuristic search algorithms can be directly applied due to the lack of a search signal (e.g., direction for the next workload to test). We observe that there are two sets of counters commodity RDMA subsystem provide can be leveraged to guide the search. The first set is known as performance counters. For example, all modern RNIC provide the counter of bits sent per second for monitoring purpose. The second is known as diagnostic counters. Modern RNICs and other hardware components expose diagnostic counters for debugging purpose (e.g., the counter indicates PCIe backpressure and NIC internal cache miss) [[22](#), [23](#)]. Diagnostic counters

<sup>1</sup>We do not use latency as a metric to define anomalies. The only latency specification for RNICs is the latency under zero load. We did not observe any anomaly in this way, probably because the RNIC is not stressed. However, when RNIC is stressed, it is hard to accurately define the correctness of latency or tail latency due to queuing delay.

are more informative. For example, when some bottlenecks of the RDMA subsystem are triggered, the performance may not drop while the corresponding diagnostic counter has increased. However, using diagnostic counters typically requires vendor’s assistance, and the number of diagnostic counters customers can access depends on vendors. For Collie to be general, we use both performance counters and optionally diagnostic counters as search signals. We conduct the efficient search by using a simulated annealing based algorithm to drive these counters to extreme value regions ([§5.1](#)).

**Challenge #3:** How to find the set of conditions to trigger anomalies? Some anomalies are complicated and only occur when many features co-exist, such as a certain type of transportation, particular message pattern, lots of connections, and specific batching operations. We invent a minimal feature set (MFS) algorithm to detect each factor’s contribution to the uncovered anomaly and construct the necessary conditions set. To search efficiently, we use MFS to avoid testing similar workloads that map to the same anomalies. After the search, developers use the MFS to understand the triggering conditions for each anomaly and bypass them accordingly when the fixes are temporarily unavailable ([§5.2](#)).

[Figure 2](#) shows our system design. Collie consists of three core components: (1) a workload engine that conducts experiments on RDMA subsystems by setting up RDMA traffic; (2) an anomaly monitor that detects performance anomaly and MFS to reproduce the observed anomaly; and (3) a workload generator that decides the next workload pattern to experiment based on the counters collected in the RDMA subsystem and the current search space. All the experiments Collie does are on the RDMA subsystem with two servers with RNICs, connected with a commodity switch.

## 4 Search Space and Workload Engine

There are two types of factors that can affect an RDMA subsystem performance in deployment. First, we need to consider the application workloads. These include host topology (i.e., where does traffic come from inside a server), how many memory regions the application registered, what transport applications choose to use, and the message patterns. Second, we need to consider the network behavior, for example, congestion on switch and packet loss rate. Currently, our paper focuses on constructing a comprehensive search space for the first factor. For the network behavior, we consider a simplified environment: two RNICs connected by a single switch, and there is no packet drop on the switch. Collie can be easily generalized to test more complicated environments.

We take the bottom-up approach to construct a comprehensive search space for various application workloads. We decompose application workloads into combinations of basic RDMA operations and construct the search space based on these combinations. [Figure 3](#) shows the key abstractions and operations of RDMA programming. These are only high-level software abstractions exposed by standard *verbs* API and we

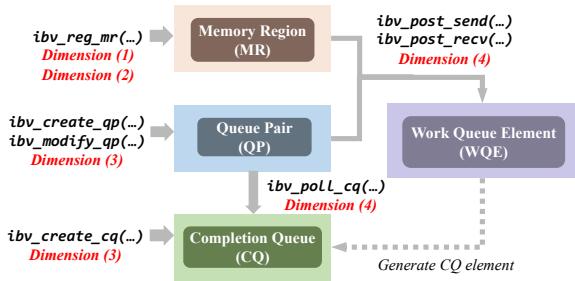


Figure 3: RDMA programming abstractions

do not need to know how these high-level abstractions are implemented in the RNIC. In this way, the search space is more comprehensive and general because it does not rely on either extra proprietary RDMA subsystem hardware knowledge or specific application features. In addition, the combinations of *verbs* operations are inherently able to describe workloads of both single application and co-existing scenarios.

We examine the RDMA programming model at first and extract four search dimensions that jointly describe the application workloads of the entire subsystem. To send a message through RDMA networks, applications first need to register a set of memory regions (MRs), using `ibv_reg_mr`. Once registered, an RNIC has the right to directly access these MRs without CPU involvement. Second, applications create a set of queue pairs (i.e., connections in traditional networking terminology), using `ibv_create_qp` and `ibv_modify_qp`. Applications need to choose a transport type for each queue-pair (QP). There are three standard types of QPs: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). Applications can use `ibv_post_send` or `ibv_post_recv` to post a list of Work Queue Element (WQE). Each WQE represents a work request and has a scatter-gather (SG) list. Each SG list contains a list of entries and each entry designates a contiguous memory region that is within the registered memory regions. A WQE can notify the RNIC to READ/WRITE remote memory (1-sided operation) or SEND/RECV local memory to/from a remote server (2-sided operation). To know that a WQE is complete, the application can register a completion queue (CQ) using `ibv_create_cq`, and the application can call `ibv_poll_cq` to poll on the CQ to get completion queue elements (CQE). Given this RDMA programming model, we extract the following search dimensions.

**Dimension 1. Host Topology.** Host topology describes how traffic flows to/from an RNIC to/from other server hardware components. Individual component tests are hard to cover this dimension while the topology has a huge impact on RDMA subsystem performance. For example, traffic can be from NUMA-affinitive DRAM or from a GPU that needs to traverse both PCIe and SMP interconnect between NUMA nodes. The latter will have a longer data path and therefore higher average DMA latency. This will trigger PCIe backpressure to the RNIC and may induce performance anomalies under some

specific application workloads. We list all accessible memory devices for this dimension.

**Dimension 2. Memory Allocation Settings.** Traditional RDMA testing is not comprehensive for this dimension, while the memory allocation settings are crucial for RDMA subsystem performance testing. First, the number of MRs affects RDMA subsystem performance because RNIC has an MMU that translates virtual addresses of memory regions to DMA-capable physical addresses and handles memory protection (e.g., authorization). RNIC only caches a fixed size of entries of the memory address translation table. If too many MRs are registered, it is then likely that the RNIC encounters cache miss and needs to access memory address translation tables on server DRAM via extra PCIe operations. These interactions have an impact on the performance. Second, MRs can have different sizes. This also affects RDMA performance because the size also affects the number of translation table entries. Moreover, many RNICs use Intel Data Direct IO (DDIO) to directly access the CPU's last-level cache. If the access range of an MR is large, it can cause severe cache misses in the CPU's last-level cache [3]. This dimension is bounded because we can set a reasonable upper bound on the number of MRs (200K), and the MR size is bounded by the total amount of memory that can be registered (pinned) in the physical server.

**Dimension 3. Transport Setting.** Transportation setting is crucial for RNIC performance, and this is well known in the research community [15, 17, 27]. We use the following factors to compose the transport setting: (1) QP type (RC, UC, UD), (2) the number of QPs, (3) the opcode type (SEND/RECV, WRITE, READ), and (4) the usage of SG and WQE. Different QP type with different opcode creates different pressure for the RNIC. For example, UD does not require ACK for each packet, which lessens the RNIC packet processing pressure. However, the SEND/RECV requires pre-posted receive buffers, which puts more pressure on the RNIC cache. The number of QPs also has a great impact on RNIC performance because of the limited RNIC cache. This is known as the scalability problem [3, 15, 32]. How SG list and WQE affect RNIC performance is a bit tricky. RNICs have to consume extra PCIe bandwidth to fetch WQE from the host DRAM [17]. The PCIe bandwidth consumed by WQE becomes substantial under some particular application workloads and can even be the performance bottleneck. We enumerate all the transport types and the opcodes (e.g., RC WRITE, UD SEND). It is practical and reasonable to set an upper bound (e.g., 20K) for the number of QPs because data center operators will hardly set up more connections. The SG list and WQE can be parameterized by this formula:  $\sum_{i=1}^n m_i = k$ , where  $k$  denotes the number of messages to send,  $n$  denotes the number of WQE and  $m_i$  denotes the number of SG elements within the  $i^{th}$  WQE.

**Dimension 4. Message Pattern.** Existing RDMA testing approaches lack flexibility and comprehensiveness, es-

pecially for this dimension. Perftest [34] only repeatedly send messages of a fixed size and other collective communication benchmarks (e.g., OSU benchmark [33]) test RDMA similarly. These simple benchmark traffic are inadequate for RDMA subsystem testing because they ignore the interaction among different requests (i.e., WQE) in a sequence.

Our ideal goal is to construct this dimension that can represent any application message pattern. However, it is impractical because application traffic can be very different and the interaction among different requests is unknown given the black-box nature of RNIC. We therefore construct this dimension in the following way. We build a request vector with  $n$  elements, where each element describe the request attribute (e.g., size of the message to send). We assume that the 1<sup>st</sup> request affects the 2<sup>nd</sup>, the 3<sup>rd</sup>, ..., the  $n^{\text{th}}$  requests but won't affect the request after the  $n^{\text{th}}$ . The larger  $n$  we set the larger search space we can cover, but we also need to consume more time. This kind of trade-off is similar to the approach when testing file systems [21], where researchers test fixed-length file system operation sequences. Modern RNIC has limited Processing Units (PU) and pipeline stages [39], restricting the number of outstanding requests an RNIC can process. We thus set  $n$  to be the product of the number of PUs and the pipeline stages. We discretize request size into multiple discrete value regions based on MTU and the burst size of the RNIC. The RNIC splits a long request into multiple bursts and processes each burst at one time to avoid Head-of-Line (HoL) blocking. The granularity can be easily modified. With more search time, we can discretize request size in a more fine-grained way. Message inter-arrival time is usually considered as a parameter for application workloads. However, adding the inter-arrival time will substantially extend our search space, so we temporarily only consider the pattern without such inter-arrival time.

**Workload engine.** We build a flexible workload engine to conduct tests in our search space. Compared to traditional traffic generation tools, e.g., Perftest<sup>2</sup>, our workload engine is more flexible and has a holistic view. It can send and receive traffic with particular pre-defined patterns (e.g., a large WRITE request followed by a small SEND request). Besides, it supports various memory and transport settings, which can test the entire subsystem holistically. To test with a point in our search space, Collie first translates a test point's settings into a set of input parameters of the workload engine. For example, the setting of dimension 1 and 2 are translated into memory allocation parameters (i.e., which GPU or NUMA DRAM to use and how many MR to register) of the engine. Then, the workload engine will take these input parameters to set up connections and generate traffic.

<sup>2</sup>Existing tools, e.g., Perftest, are arguably not designed for this type of testing. They are performance benchmark tools. However, we are not aware of any other tools can that test RDMA subsystems.

---

### Algorithm 1 Search for Performance Anomalies

---

**Input:**  $S$ : initial anomaly set;  $T_0$ : a high enough initial temperature;  $T_{\min}$ : the lowest limit of temperature;  $n$ : the number of times SA runs for a certain temperature;

**Output:**  $S$ : An updated anomaly set;

```

1:  $P_{old}, M_{old} = \text{MeasureRandomPoint}()$ ; pick a random point,
   setup traffic and collect metrics as  $M$ 
2: while  $T > T_{\min}$  do
3:   for  $i = 0$ ;  $i < n$ ;  $i++$  do
4:     mutate  $P_{old}$  for a new application workload  $P_{new}$ ;
5:     if  $\text{MatchMFS}(S, P_{new})$  then continue;
6:      $M_{new} = \text{MeasurePoint}(P_{new})$ ;
7:      $\Delta E = \text{CompareMetric}(M_{new}, M_{old})$ ;
8:     if  $\Delta E < 0$  then
9:        $P_{old} = P_{new}$ 
10:    else
11:      the probability  $prob = \exp(-\Delta E / T_{(i)})$ ;
12:      if  $\text{rand}(0, 1) < prob$  then  $P_{old} = P_{new}$ ;
13:    end if
14:    if  $\text{IsAnomaly}(M_{new})$  then
15:       $new\_mfs = \text{ConstructMFS}(P_{new})$ ;
16:      Put  $new\_mfs$  into  $S$ ;
17:       $P_{old}, M_{old} = \text{MeasureRandomPoint}()$ ; pick another
       random point when a new anomaly is found
18:    end if
19:  end for
20:   $T = T * \alpha$ ; where  $\alpha$  is decay factor
21: end while
22: return  $S$ 
```

---

## 5 Search for Performance Anomalies

The total size of our search space (i.e., the combination of parameters) is on the order of  $10^{36}$ . Each experiment we do requires 20-60 seconds, mostly depending on the number of QPs to create and the number of MRs to register. This means we cannot exhaust the search space. One naive approach is to generate random input in the search space. This approach is already much better than existing tests because the design of our search space is more comprehensive than that in existing tools (§7). However, similar to typical black-box fuzz testing on software, random inputs can only find few anomalies and cannot efficiently uncover complicated anomalies that require multiple conditions to hold simultaneously.

### 5.1 Workloads Generation

We leverage two types of counters to guide the search. The high-level approach is to use an optimization algorithm to drive counters to extreme value regions by keeping mutating the test workloads. For performance counters, we drive the counters to low-value regions. For diagnostics counters (which map to unexpected events), we drive the counters to high-value regions.

Our algorithm is based on simulated annealing (SA). SA is a probabilistic algorithm to find the global minimum of a given function. The idea is to keep mutating the input in the direction of minimizing a given function. SA calls the func-

tion value energy. To avoid getting stuck at a local minimum, SA maintains a temperature value. At the beginning of the algorithm, the temperature is high and SA allows mutating input in the direction of increasing the energy. As temperature decreases during the search, SA is less likely to move the input in the direction of increasing the energy. Finally, when the temperature is below a certain threshold, every mutation of the input must decrease energy. SA finishes when there is no way to mutate the input to make the energy lower.

[Algorithm 1](#) shows our algorithm that is based on SA. We maintain a list of performance anomalies. Each anomaly is an MFS (e.g., an area in the search space) that contains workloads to reproduce the performance anomaly. The search starts from a random workload in the search space, and our algorithm measures the counter values. In each iteration of SA, we mutate the workload in one of our search dimensions (line 4). We test whether the new workload causes a performance anomaly with our anomaly monitor. If so, we run our MFS algorithm to determine the entire area in the search space that belongs to this anomaly. We add the new anomaly to the set and change the current workload to a random one. If the new workload does not trigger a performance anomaly, we measure the point by comparing counter values and decide whether to move the current workload to the new one. We always skip workloads that belong to an existing performance anomaly for efficient search.

Our algorithm extends the standard SA algorithm in several important ways to adapt it for our context. First, we compute the energy in the following way: assuming the counter value changes from A to B, we set the different in energy ( $\Delta E$ ) to be  $\frac{B-A}{A}$  for performance counters and  $\frac{A-B}{B}$  for diagnostic counters because we are minimizing performance counters and maximizing diagnostic counters to trigger potential anomalies. This also allows us to avoid value region problem (e.g., the value regions of diagnostic counters are sometimes opaque). Second, we do not require SA algorithm to find the actual global optimum because we care about all potential anomalies. We therefore always set a more relaxed temperature and  $\alpha$  that enable the algorithm to jump out of a certain stage even when it has already run lots of iterations. In addition, we maintain a set of performance anomalies (i.e., MFS). When mutating the point, we compare the mutated point with our existing MFS (line 5). Each MFS contains a list of parameters ranges. If the mutated point matches all parameters ranges of an MFS (i.e., the parameter value of this point is in the MFS's range), we claim this point belongs to the MFS and skip testing it. This ensures that the future search does not redundantly test workload already covered by the existing set of anomalies.

## 5.2 Anomaly Monitor

Our anomaly monitor detects performance anomalies and computes the MFS of them.

**Anomaly Detection Condition.** We use two conditions to

detect anomalies. First, if any pause frame is generated. Here we use a metric called *pause duration ratio*. If the pause duration ratio is 1%, this means for every second, transmission is paused by 10 ms. We set our threshold to be 0.1%. The reason is our experiment platform only has two servers and our switch that connects the servers support line rate traffic, so there is no network congestion to begin with. We set the threshold to be above 0, because RNIC may generate a few pause frames when the memory bus or PCIe bus is busy temporarily, especially when connections are just set up. Second, each RNIC has its maximum bits per second and maximum packets per second in its specification that can be easily verified by running simple benchmarks. Without performance anomalies, network traffic should be restricted by either one of these upper bounds. If a workload's throughput (in terms of both metrics) is 20% lower than the upper bounds, it means that the performance is likely to be restricted by some other bottlenecks of the RDMA subsystem. Collie reports this and runs the MFS algorithm below.

**Minimal Feature Set (MFS).** After we detect an anomalous workload, we need to know what features of this workload actually trigger the anomaly. For example, if we currently find a new anomaly that has 5 features. It may be the case that 3 features are already sufficient to reproduce this anomaly. One approach is to use machine learning based algorithms to generate decision trees or deep neural networks to locate the area in the search space for the anomaly. However, machine learning approaches usually require much more training data and thus many more hardware experiments.

We instead use a heuristic approach. Since we only have 4 search dimensions with few factors, we just do a few tests on each dimension to determine whether a factor belongs to the MFS. For example, if our search algorithm finds a certain workload using UD can cause a performance anomaly. We test whether the same workload with RC and UC can cause performance anomalies. If not, UD belongs to the MFS because it is necessary to reproduce the anomaly. To determine the MFS of a dimension that is continuous (e.g., number of connections), we discretize them manually into a set of value regions and test each of them. Finer-granularity discretization is acceptable because MFS algorithm only runs when uncovering a new anomaly and the number of anomalies is relatively small compared to the entire search space.

We report all the anomalies to RNIC vendors and we can wait for their fixes. Unfortunately, the solutions to these anomalies are case by case. Some anomalies require vendors to spend a substantial amount of time on coming up with solutions and the solutions may not be applicable for data center operators immediately, such as hardware replacement. Hence, developers need to avoid such anomalies instead of waiting for a fix. Collie provides MFS to help developers avoid such anomalies by changing application workload to break the conditions in the MFS.

MFS helps developers to avoid anomalies in two areas.

Type	RNIC	Speed	CPU	PCIe	NPS	Memory	GPU	BIOS	Kernel
A	CX-5 DX	25 Gbps	Intel(R) Xeon(R) CPU 1	3.0 x 16	1	128 GB	-	INSYDE	4.19
B	CX-5 DX	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	768 GB	-	AMI	4.14
C	CX-5 DX	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	384 GB	V100	AMI	5.4
D	CX-6 DX	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	768 GB	-	AMI	4.14
E	CX-6 DX	200 Gbps	AMD EPYC CPU 1	4.0 x 16	1	2 TB	A100	AMI	5.4
F	CX-6 DX	200 Gbps	Intel(R) Xeon(R) CPU 3	4.0 x 16	1	2 TB	A100	AMI	5.4
G	CX-6 VPI	200 Gbps	AMD EPYC CPU 1	4.0 x 16	2	2 TB	-	AMI	5.4
H	P2100G	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	384 GB	-	AMI	5.4

Table 1: Testbed RDMA subsystems configurations. We use numbers in the name of concrete CPU types for confidentiality.

The first one is anomaly prevention. Before an application is implemented, Collie lets developers restrict the search space using their knowledge of their applications to represent all the possible workloads. Then, Collie outputs whether the restricted search space contains performance anomalies. If not, assuming the developers’ understanding of their applications is correct, the application won’t encounter any performance anomaly found by Collie. The second one is debugging. When an existing application unfortunately encounters anomalies, we can run Collie on the RDMA subsystem and generate all the MFS. Comparing the application with the generated MFS, Collie provides several suggestions that help to break the triggering conditions. We present two real cases to show how MFS helps developers in §7.3.

One caveat of our approach is that we are not able to know the root causes of these anomalies given the black-box nature of the RNICs and other hardware components in the RDMA subsystem. This means it may be the case that multiple MFS are actually due to the same anomaly (i.e., the same hardware bug). This is acceptable because the goal of MFS is to accelerate the search algorithm by eliminating redundant test cases and help developers understand what features of the workloads can trigger the anomaly. We anyway need to report all the anomalies (i.e., all the MFS) we found to the vendors and that is also the best we can do given the RNIC black-box hardware nature.

## 6 Implementation

The workload generator and the anomaly monitor are written in ~2100 lines of Python. The workload engine is implemented with ~2000 lines of C/C++. We directly use monitor tools from vendors to collect hardware counters (both performance and diagnostic counters) from the RDMA subsystem.

The workload engine is implemented with the *verbs* API and *rdma-core-34.0* libraries [38]. In deployment, the Mellanox RNIC uses mlx5 driver (OFED 5.2-1.0.4.0) and the Broadcom RNIC uses bnxt driver (1.10.1.216.2.89.0). The workload engine set up connections by TCP out-of-band transmission. When all connections are set up, the engine starts to generate workload.

The anomaly monitor collects primary metrics, such as throughput and pause frame duration, four times per iteration. It first decides whether the traffic is stable and then compares the primary metrics (e.g., bits per second, packets per second)

with the pre-defined thresholds.

The workload generator collects counters using monitors provided by vendors. These monitors provide counters every second. Collie fetches these counters four times per iteration and uses the average results.

## 7 Evaluation and Experience

We evaluate Collie on 8 different RDMA subsystems. Table 1 shows the hardware and related configurations. We use the same anomaly detect conditions as described in §5.2

### 7.1 Performance Anomalies Found

Before we build Collie, we already know 3 existing anomalies. Collie can find all the existing ones and find 15 new anomalies. All of them are reported to our vendors and are acknowledged by them. Table 2 shows the 18 anomalies. We only present those found on subsystem F and H because anomalies found on other subsystems are subsets of those found on F. Appendix A provides details about these anomalies, including the exact workload, as well as the explanations and solutions from vendors. Here we choose two tricky anomalies to show the importance of Collie’s systematic search.

**Anomaly #4:** Bidirectional RC READ with large WQE batch size, long SG list, and a few connections causes PFC pause frames. Our vendors have successfully reproduced this anomaly in their environment using Collie’s traffic generator and acknowledged it, but currently there is no fix. This anomaly cannot be found by existing approaches such as using *PerfTest* to generate workloads, because *PerfTest* does not support flexible WQE and SG list batching strategies. Though *PerfTest* is not designed for this purpose, it is the prevalent tool to uncover performance anomalies. To the best of our knowledge, we don’t see any other state-of-the-art work address this problem, which also shows that Collie is the first work to fill this vacancy.

**Anomaly #10:** Bidirectional RC WRITE with large WQE batch size, particular message pattern, and a few connections causes PFC pause frames. This anomaly is not captured by existing approaches (e.g., running current applications) but we successfully reproduce it by slightly modifying our production RDMA RPC library: when users call the library to send a message, it will try to send as many messages as possible in a WQE batch. The batch size is highly dependent on the timeout value. If the application is throughput sensitive rather than

	RNIC	Direc.	Transport	MTU	WQE	SGE	WQ depth	Message Pattern	# of QPs	Symptom
#1	CX-6	-	UD SEND	-	$\geq 64$	-	$\geq 256$	-	-	pause frame
#2	CX-6	-	UD SEND	-	$\leq 8$	-	$\geq 1024$	$\leq 1KB$	$\geq \approx 16$	low throup.
#3	CX-6	-	RC READ	1K	-	-	-	$\geq 16KB$	-	pause frame
#4	CX-6	Bi-	RC READ	-	$\geq 32$	$\geq 4$	-	-	$\geq \approx 160$	pause frame
#5	CX-6	-	RC SEND	1K	$\geq 64$	-	$\geq 1024$	$\geq 2KB$ and $\leq 8KB$	-	pause frame
#6	CX-6	-	RC SEND	1K	$\leq 16$	$\geq 2$	$\geq 1024$	$\leq 1KB$	$\geq \approx 32$	low throup.
#7	CX-6	-	RC WRITE	-	No	-	-	$\leq 1KB$ and $\geq \approx 12K$ MRs	-	low throup.
#8	CX-6	-	RC WRITE	-	No	-	$\leq 16$	$\leq 1KB$	$\geq \approx 500$	low throup.
#9	CX-6	Bi-	-	-	-	$\geq 3$	-	mix of $\leq 1KB$ & $\geq 64KB$	-	pause frame
#10	CX-6	Bi-	RC WRITE	-	$\geq 64$	-	-	mix of $\leq 1KB$ & $\geq 64KB$	$\geq \approx 320$	pause frame
#11	CX-6	Bidirectional cross-socket traffic on particular AMD servers								pause frame
#12	CX-6	Particular GPU-Direct RDMA traffic on particular servers								pause frame
#13	CX-6	Co-existence of loop traffic and receiving traffic								pause frame
#14	P2100	Bi-	RC	4K	-	$\geq 4$	-	-	$\geq \approx 1300$	low throup.
#15	P2100	-	UD SEND	-	-	-	$\geq 64$	-	$\geq \approx 32$	pause frame
#16	P2100	-	RC READ	1K	$\geq 8$	-	-	-	$\geq \approx 500$	pause frame
#17	P2100	-	RC SEND	-	$\leq 16$	-	$\geq 128$	$\leq 1KB$	$\geq \approx 64$	pause frame
#18	P2100	Bi-	RC	1K	$\geq 32$	-	-	$\leq 64KB$	$\geq \approx 30$	pause frame

Table 2: Performance anomalies found on subsystem F and H with the necessary conditions to trigger them. Anomalies marked with green color are new anomalies found by Collie. Rest are the anomalies we know before building Collie.

latency sensitive, the timeout value can be set high, which allows a larger batch size. Currently the timeout value is set small because most applications supported by this library are latency sensitive. However, by changing this value we successfully enlarge the WQE batch size and the conditions of #10 are all met. This shows the importance of the anomalies found by Collie, as well as how Collie can capture those anomalies missed by existing solutions.

We try our best to reproduce the anomalies found by Collie using existing workload generators (e.g., Perftest), only 4 of them (#3, #8, #13, #15) can be reproduced with very careful parameters tuning. Rest anomalies are all outside the search space of existing approaches.

## 7.2 Running Time for Anomaly Search

To evaluate the efficiency of performance anomaly search, we compare Collie with two baselines: (1) random input generation in our search space and (2) Bayesian Optimization (BO), a widely used method in search problem [31]. We implement the BO approach based on [31]. We set the counter values as BO’s optimization target. Our vendors provide us with 9 diagnostic counters. For Collie and BO, we first generate 10 random points. We then compute the standard deviation over the mean of the counter values collected in the first 10 run and use the result to rank these diagnostic counters in decreasing order. Both Collie and BO optimize each diagnostic counter in this order. For a fair comparison, we use MFS to enhance BO as well. In this section, we use subsystem F as an example. We run each search for 10 hours.

Figure 4 shows the running time to find performance anomalies. Random input (i.e., fuzzing) can already find 7 anomalies that only require simple conditions to trigger. BO does improve efficiency but to a very limited extent. BO can speed up the search process but only find 8 anomalies with

the given time. We analyze the optimization process of BO and find that it is not able to optimize the corresponding counters. Our guess is that BO works well when counter values are smooth in the search space. However, the counter values in our search space can have sudden changes, because some discrete dimensions have a huge impact on the counter values (e.g., QP type). Collie uses a simulated annealing based algorithm to optimize the counter values and successfully speed up the search process. Given limited time, it can find all the performance anomalies of this RDMA subsystem. We believe this improvement comes from the optimization process: driving counters to extreme regions is more likely to trigger performance anomalies. It is possible that a more efficient search algorithm (e.g., a fine-tuned BO, reinforcement learning) can perform better, and it is worth future exploration. However, our goal here is to demonstrate that existing simple optimization algorithms, such as simulated annealing, can search efficiently with these hardware counters.

Collie uses diagnostic counters and MFS to further speed up the search. Now we break down their contribution to our overall search speed. Figure 5 shows the result.

**The value of diagnostic counters.** Figure 5 shows that with performance counters, Collie (Perf) has already found 11 of the 13 anomalies, including the 3 existing ones. This proves that the performance counters are informative and can be used to improve search efficiency. It shows the generality of Collie because performance counters are general and provided by all commodity RDMA subsystems. Figure 5 also shows that using diagnostic counters can further improve the speed. Given limited time, Collie (Diag) can uncover more anomalies and is faster. For example, Anomalies #7 and #8 are not captured by Collie (Perf) because there is no performance change during the search, but Collie (Diag) can observe the

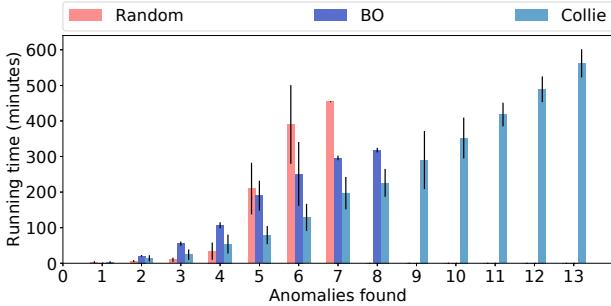


Figure 4: Mean time to find anomalies with random input generation, BO, and Collie. Error bars denote standard deviations. There is no red bar starting from 8, and no purple bar starting from 9, because random input generation and BO can only find 7 and 8 anomalies, respectively.

increase of RNIC internal cache miss and uncover them.

**The value of minimal feature set (MFS)** The main difference between SA and Collie is whether MFS is applied. With MFS, the efficiency of all approaches (both using diagnostic counters and using performance counters) is significantly improved. For example, Collie (Diag) only uses about half of the time to uncover all the anomalies found by SA(Diag). MFS improves efficiency by eliminating redundant tests from the search space. Otherwise, approaches without MFS may be stuck in the area of an uncovered anomaly.

To understand why increasing diagnostic counter values can help to find anomalies and how MFS works, here we use *Receive WQE Cache Miss* counter as an example. We do not rely on the meaning of these diagnostic counters during the search. To the best of our knowledge, the counter means the number of times that RNICs need to issue extra DMA operations to fetch receive WQE from host DRAM.

Figure 6 shows the diagnostic counter values during the search. The random input generation approach (the orange line) does not increase the diagnostic counter value and thus cannot find many performance anomalies. Collie w/o MFS (the green line) can drive the diagnostic counter value very high, but it cannot find many distinct performance anomalies because further increasing the counter value in the neighboring regions of existing performance anomalies wastes time. Collie (the blue line) is effective in finding performance anomalies, because it can both increase the diagnostic counter value to find application workloads that cause anomalies and also do not need to test application workloads that belong to the same anomaly. Figure 6 shows that most anomalies are found when the diagnostic counter value is high. This also supports the intuition that it is likely to trigger performance anomalies when the diagnostic counter value is driven to extreme regions, which indicates the RDMA subsystem is under pressure. Some anomalies in Figure 6 do not show a high value of this counter. This is mainly due to that they are anomalies that can be easily triggered. They are usually triggered at the beginning of the search process (left corner of Figure 6) and another corresponding diagnostic counter value

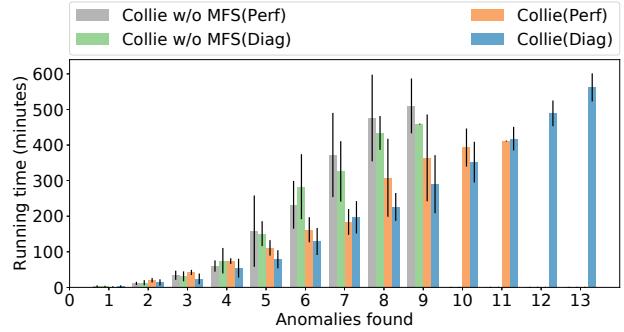


Figure 5: Mean time to find anomalies. (Diag) means diagnostic counters, and (Perf) means performance counters. Error bars denote standard deviations.

is high. For example, Anomaly #13 has simple triggering conditions and is usually found very soon. It does not increase the *Receive WQE Cache Miss* counter but will increase another counter, the counter of *PCIe Internal Back Pressure*.

### 7.3 Using Collie for Application Design

We use Collie in the development and performance debugging of two key RDMA applications.

First, Collie provides design suggestions for our self-developed efficient RDMA RPC library during its design and implementation. The library needs to be CPU-efficient, and we thus only consider RC as the transport because it is the only transport that supports all one-sided RDMA operations (i.e., READ, WRITE) and ensures reliable messages. In addition, major services that use this RPC library will mainly be deployed on subsystem B and C. Given the search space, Collie provides two suggestions to the developers. (1) Anomaly #4 is in the restricted search space if the RDMA RPC library uses READ, large WQE batch size, and a long SG list to improve throughput and shape the message format. (2) The library needs to use SEND/RECV to deliver small control messages and generally keeps a large receive queue in case of receive-not-ready error. This can potentially trigger Anomaly #5. Unfortunately, both #4 and #5 temporarily have no fix, so Collie suggest developers (1) use RDMA WRITE to transmit data in a batch and (2) configure receive queue depth carefully in SEND/RECV for small control messages transmission. This RDMA based RPC library achieves expected performance and is currently supporting three major services in production.

Second, Collie helps an distributed machine learning (DML) application based on BytePS [14] bypass anomalies during its further development in our production environment. Our DML application encountered anomaly #9 when deploying on our new subsystem E. We worked with multiple vendors (RNIC, server, CPU), but for several weeks we didn't find the root cause or the fix for this anomaly. During this time, we ran Collie and compared the anomalous application with the MFS we got. We found that the application's behaviors matched one of the MFS: (1) use a long SG list to send tensors with several meta data and (2) the message pattern of

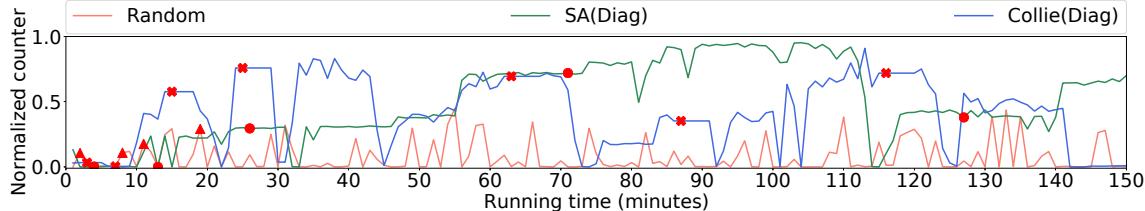


Figure 6: Diagnostic counter values (*Receive WQE Cache Miss*) during the search. Counter values are normalized based on the maximum value we observed in the search. Red crossings denote the performance anomalies found by Collie. Red triangles denote the performance anomalies found by random input generation. Red squares denote the performance anomalies found by Collie without MFS. Collie (the blue line) is flat for a few minutes after finding a new performance anomaly. This is to represent the time needed for extracting the MFS.

tensors and meta data is a typical pattern that contains mix of short and long messages. Collie suggested the developers to avoid these conditions. The developers hence bypassed this anomaly before vendors’ fix is ready.

#### 7.4 Implications of the Performance Anomalies Found

After careful analysis of the anomalies found by Collie, we have several interesting and important observations.

**Holistic performance testing/tuning over entire RDMA subsystems is important.** With our vendors’ help, we try our best effort to present the root causes of these anomalies in Appendix A. The root causes can be bottlenecks from RNIC internals, PCIe controllers, and host topologies (cross socket communication). This is because the RDMA network performance is highly related to the entire subsystem and the holistic test is thus important. Besides, we need to configure systems carefully (MTU, PCIe, NUMA, IOMMU, etc.) to fully leverage RDMA’s performance [17, 30]. Collie shows that it is sometimes difficult to choose what configuration to use. For example, comparing the Anomaly #14 with other cases related to the MTU setting (e.g., #6), we observe there is no optimal MTU setting for all types of RDMA subsystems. This also indicates that data center operators have to test various RDMA subsystem configurations and tune the system carefully before deploying them.

**Opaque resource limitation of the RDMA subsystems.** RDMA virtualization, especially performance isolation is important for deploying RDMA to the public cloud environment. Researchers have spent a lot of effort and proposed several solutions [12, 19, 36, 43, 45]. However, anomalies found by Collie suggest that there are new challenges. Existing approaches mainly focus on the isolation of visible resources like *verbs* structures (e.g., QP, MR, CQ), pinned memory, and bandwidth. However, there exist resources that are opaque for developers and data center operators. For example, the RNIC has limited caches that store many data structures, including connection context (well known as QPC) and receive WQE. Anomalies #1, #3, #4, #5 show that severe WQE cache miss can have a huge impact on performance. Hence, it is possible that a connection with a specific message pattern affects another connection by triggering cache misses, even when the bandwidth and other resources are well isolated. We therefore believe it is necessary to take these invisible resources into

consideration when enforcing RDMA performance isolation, especially in public clouds.

**Does Ethernet-based RDMA need end-to-end flow control?** Currently there is no end-to-end flow control mechanism (e.g., the sliding window for TCP) for production Ethernet-based RDMA deployment (i.e., RoCEv2). Collie shows that this is a major barrier for RDMA subsystems to achieve high-performance and reliability. For example, many anomalies (e.g., #9 and #12) show that the host limitation can slow down RNIC’s outbound rate (dispatching received data to host memory). This makes the receiver cannot consume packets as fast as the sender sends. Without end-to-end flow control, the RoCEv2 now can only rely on PFC, the hop-by-hop flow control mechanism. PFC helps to avoid such overflow packet drop but can cause catastrophic consequences [11, 13]. Note that RDMA congestion control [20, 28, 46] mainly targets in-network congestion, so it is orthogonal. A similar observation has been shown in IRN [29], but they mainly focus on in-network behaviors. Collie shows that, in addition to switches, the hosts can also generate PFC pause frames, which requires attention when deploying RDMA in production.

## 8 Discussion and Future Work

**Search space.** Collie mainly focuses on how specific application workloads can stress the RDMA subsystems and trigger performance anomalies. We therefore focus on a simple setting of two RNICs and assume the network is free of anomaly. In addition, we temporarily ignore control path behaviors and the inter-arrival time between requests of a connection. The main reason is that adding these factors substantially enlarge the size of our search space. How to efficiently expand Collie’s search space is an interesting direction for exploration.

**Search algorithm.** Collie uses simulated annealing based algorithm with minimal feature set (MFS) to search efficiently. Though powerful data centers can run Collie on multiple machines for a longer time, the search algorithm is also important. According to the MFS found by Collie, the expected time for a random approach is tens of days to find some anomalies that require complicated triggering conditions. There are many other search algorithms alternatives that can be leveraged, such as reinforcement learning. Integrating more search algorithms into Collie is another interesting direction to explore.

**Generality of Collie.** We believe that Collie can be used for

any type of RDMA subsystem or even subsystems with other types of NICs. For example, though the link/transport protocols are different for Infiniband and RoCEv2, the NIC internal structures should be similar (e.g., both can use Mellanox CX-6 VPI RNIC). Collie only relies on non-proprietary counters that expose NIC internal status. Therefore, this methodology should be generalizable to any NIC in any deployment environment if similar counters are available.

**Analysis of Performance Anomalies.** Collie is designed to uncover anomalies and help to bypass them from the perspective of data center operators, so it assumes minimal hardware knowledge of RDMA subsystems for generality and does not directly analyze the underlying causes. However, since the anomalies found by Collie can be severe (e.g., triggering PFC pause storms), we believe to fully understand them is also an important direction to explore. For example, as mentioned in §7.4, many anomalies are due to bottlenecks on some opaque resources. Both RNIC vendors and data center operators hence need to understand what extra resources should be considered if they want to provide performance isolation for RDMA in a public cloud.

## 9 Related Work

**Hardware bottlenecks in host networking.** With the fast growth in NIC performance, researchers have noticed several potential hardware bottlenecks in host networking. Neugebauer et al. [30] study the implication of PCIe performance in host networking. Farshin et al. [6] examine when and when not Intel Data Direct I/O technology can speed up host networking by allowing NIC to access CPU’s last-level cache directly. Kalia et al. [15] observe the scalability bottlenecks of caching per-connection metadata in RNIC. Stanko et al. [32] study how the number of connections and memory regions affect performance. These works have raised our attention to RNIC hardware behaviors. Our work is on a different angle: we systematically uncover the performance anomalies that can be triggered by specific application workload due to hardware bottlenecks.

**Fuzz testing.** Our techniques are in the broader category of fuzz testing. There are three types of fuzz testing: black-box [25, 26], white-box [7, 9, 10], and gray-box fuzzing [1, 40]. Black-box fuzzing is to generate random inputs to test a program, and usually black-box fuzzing can only uncover shallow bugs. In our context, this is also true that using randomly generated application workload can only uncover a small set of anomalies (§7). White-box fuzzing is to use symbolic execution on source code to guide the fuzzer to generate inputs that can have high coverage. We do not have the internal designs of the various components within an RDMA subsystem, so we cannot use white-box approaches. Gray-box fuzzing in the software context is to use the coverage in the control flow graph to guide the fuzzer to incrementally generate inputs that can lead to larger coverage. Our approach is similar to gray-box fuzzing that we both use simulated annealing

and mutation-based test case generation. However, the key difference is that we use hardware counters in the RDMA subsystem to guide the search rather than the coverage on the control flow graphs of the source code.

**Application design on top of RDMA.** Many RDMA application designs leverage specific RDMA performance characteristics, and some already try to circumvent certain RNIC performance anomalies. HERD [16] uses UD SEND and UC Write to implement an RPC library for reduced RNIC packet processing overheads and better scalability. FASST [18] and eRPC [15] uses UD to further mitigate RNIC scalability bottlenecks in RPC libraries. Kalia et al. [17] provide guidelines to optimize HERD’s transport by considering PCIe bottlenecks. FaRM [4, 5] uses RC to access remote in-memory key-value stores, so that it can use RDMA 1-sided READ/WRITE operation for reduced CPU overheads. Our goal is complementary: we systematically uncover the set of performance anomalies of RDMA subsystems that application developers need to be aware of. We show that for RDMA developers, in reality, there is no optimal choice for a particular design decision (e.g., all transport types have certain performance anomalies). Developers therefore need to have a holistic view of all the design decisions and the entire RDMA subsystem before designing and implementing RDMA applications.

## 10 Conclusion

RDMA has been increasingly used in the industry for its low latency and reduced CPU overheads. Performance anomalies hurt application performance and can lead to catastrophic consequences (e.g., deadlocking the data center network). We build Collie, a tool to help RDMA users to find performance anomalies of the entire RDMA subsystems, without the need for access to any hardware internals design. Collie constructs a comprehensive search space for RDMA application workloads and finds performance anomalies by using simulated annealing to optimize two types of vendor-provided counters. We evaluate Collie on 8 commodity RDMA subsystems and Collie found 15 new performance anomalies that are all acknowledged by the vendor. 7 of them are already fixed under vendors’ guidance. We also present our experience in using Collie to guide our development of an RDMA RPC library and help our distributed machine learning applications bypass performance anomalies before vendor fix is ready. Collie is available at <https://github.com/bytedance/Collie>.

## Acknowledgement

We thank Alvin R. Lebeck, Xiaowei Yang, Xi Wang, Wei Bai, Mahmoud Elhaddad, Jitu Padhye, and Shachar Raindel for their helpful comments and discussion. We thank NVIDIA, Broadcom, and AMD for their strong technical support. We thank our shepherd Costin Raiciu and other anonymous reviewers for their insightful feedback. Our work is partially supported by an Amazon Research Award, a Meta Research Award, and an IBM Academic Award.

## References

- [1] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *CCS*, 2017.
- [2] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. A1: A Distributed In-Memory Graph Database. In *SIGMOD*, 2020.
- [3] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *EuroSys*, 2019.
- [4] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *NSDI*, 2014.
- [5] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*, 2015.
- [6] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX ATC*, 2020.
- [7] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-Based Directed Whitebox Fuzzing. In *ICSE*, 2009.
- [8] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *NSDI*, 2021.
- [9] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-Based Whitebox Fuzzing. In *PLDI*, 2008.
- [10] Patrice Godefroid, Michael Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.
- [11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*, 2016.
- [12] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *SIGCOMM*, 2020.
- [13] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *HotNets*, 2016.
- [14] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.
- [15] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.
- [16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014.
- [17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*, 2016.
- [18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, 2016.
- [19] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*, 2019.
- [20] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *SIGCOMM*, 2019.
- [21] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Automatically Test File-System Crash Consistency. In *HotStorage*, 2017.
- [22] Mellanox. Device Proprietary Counters. <https://docs.nvidia.com/networking/display/WINOFv55052000/Device+Proprietary+Counters>.
- [23] Mellanox. NEO-Host. <https://support.mellanox.com/s/productdetails/a2v5000000N201AAK/mellanox-neohost>.
- [24] Mellanox Adapters Programmer’s Reference Manual (PRM). [https://www.mellanox.com/related-docs/user\\_manuals/Ethernet\\_Adapters\\_Programming\\_Manual.pdf](https://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf), 2021.
- [25] Barton Miller, Mengxiao Zhang, and Elisa Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, page 1–1, 2020.

- [26] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.
- [27] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, 2013.
- [28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.
- [29] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *SIGCOMM*, 2018.
- [30] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *SIGCOMM*, 2018.
- [31] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python. <https://github.com/fmfn/BayesianOptimization>, 2014.
- [32] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pis'menny, Liran Liss, Michael Wei, Dan Tsafir, and Marcos Aguilera. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *SYSTOR*, 2019.
- [33] OSU benchmarks. <https://mvapich.cse.ohio-state.edu/benchmarks/>, 2021.
- [34] OFED perfest. <https://github.com/linux-rdma/perf-test>, 2021.
- [35] IEEE DCB. 802.1Qbb - Priority-based Flow Control. <https://1.ieee802.org/dcb/802-1qbb/>, 2021.
- [36] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces. In *VEE*, 2015.
- [37] Kun Qian, Wenzhe Cheng, Tong Zhang, and Fengyuan Ren. Gentle Flow Control: Avoiding Deadlock in Lossless Networks. In *SIGCOMM*, 2019.
- [38] Linux rdma-core. <https://github.com/linux-rdma/rdma-core>, 2021.
- [39] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet!, 2021.
- [40] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security*, 2017.
- [41] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *OSDI*, 2016.
- [42] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *SIGCOMM*, 2020.
- [43] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.
- [44] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lin-tao Zhang, and Lidong Zhou. Fast Distributed Deep Learning over RDMA. In *EuroSys*, 2019.
- [45] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI*, 2022.
- [46] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

## A Performance Anomalies Found

More details of these anomalies and the lesson we learn are included in this section. We present a concrete example of each anomaly and try our best to simplify each anomaly so that they can be reproduced easier. It is possible to find milder or stricter conditions that trigger the anomaly. We, to the best of our knowledge, also categorize these performance anomalies to their root causes based on our observation and conversations with our vendors.

### A.1 Subsystem F with Mellanox 200 Gbps CX-6 VPI

#### Root cause #1: Receive WQE cache misses bottleneck RNIC receiving rate.

(New) *Anomaly #1: UD with large WQE batch size and long WQ causes PFC pause frames and drastic throughput drop.* Collie observes that the pause duration ratio can be up to  $\approx 20.0\%$  with only a single UD QP. The pause duration ratio means that RNIC is asking the corresponding switch port to pause for  $\approx 200$  milliseconds within one second on average. We share the NIC vendor with our traffic engine tool and the running command. They have reproduced the anomaly in their environments, but the root cause is still not clear yet. Therefore, we claim this anomaly not fixed yet. To the best of our knowledge, it is likely due to the cache miss triggered by the pre-fetch mechanism for the receive WQE. This bottlenecks the receiver from receiving traffic.

Here is a simplified concrete trigger setting of Anomaly #1: There is 1 connection of UD QP using SEND/RECV Opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 256 (i.e.,  $\text{max\_send/recv\_wr} = 256$ ). The MTU is 2KB. The sender keeps sending 64 requests in a batch. Each request only has one SG element and a fixed size of 2KB.

(New) *Anomaly #2: UD with small WQE batch size, long WQ, small messages, and a few connections causes throughput to drop without pause frames.*

This anomaly is similar to #1 but more tricky and has a different end-to-end symptom. Unlike #1, Collie does not observe PFC pause frames when the setting is slightly different from #1: if the sender does not post sending requests in batch or the batch size is small (e.g., less than 8) and the messages are relatively small (e.g., 512B, 1KB), the throughput will drop by more than 20% without any PFC pause frame triggered when the receiver has an extremely long work queue. If we set a smaller work queue for the receiver, the throughput returns to the line rate. This anomaly is also reproduced and acknowledged by NIC vendor. We conjecture that it has a similar root cause to #1, but due to unknown RNIC bottlenecks, it behaves differently that the throughput drops without pause frame.

Here is a simplified concrete trigger setting of Anomaly #2: There are 16 connections of UD QP using SEND/RECV Opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 1024. The

MTU is 1KB. The sender keeps sending 4 requests in a batch. Each request only has one SG element of 1KB.

(New) *Anomaly #3: RC READ with large messages causes PFC pause frames when MTU is under 1500 (the default MTU for Ethernet).*

We observe the throughput drops drastically once we use RDMA READ opcode with 1500 MTU (1024 for RDMA), the default value for our data centers. The pause duration can be up to 10% and throughput drops to less than half. We report this to our NIC vendor and they tell us the low MTU may trigger the RNIC internal packet processing bottleneck for this 200 Gbps NIC. We carefully survey the potential effect of MTU modification in our deployment and modify the MTU from 1500 to 4200, which supports 4096 as RDMA MTU. This anomaly is successfully fixed in this way.

Here is a simplified concrete trigger setting of Anomaly #3: There are 8 connections of RC QP using Read opcode. Each QP has 1 sending MR of 4MB and 1 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending RDMA READ requests. Each request only has one SG element and a fixed size of 4MB.

(New) *Anomaly #4: Bidirectional RC READ with large WQE batch size, long SG list, and a few connections causes PFC pause frames, even when MTU is set to 4200 (4096 for RDMA).*

This anomaly is tricky but severe. Even with 4200 MTU (Anomaly #3 is solved), Collie observes about 30% PFC pause duration ratio that when bidirectional RDMA READ happens and both sides post a large number of requests in a batch (e.g., 32), each request consists of multiple scatter gather element (e.g., 4) and there are a few connections (e.g.,  $\approx 160$ ). As usual, this newly found anomaly is reported to the vendor and they have reproduced and confirmed the anomaly. For now, the root cause of this anomaly is still unknown. Therefore, we claim this anomaly not fixed yet.

Here is a simplified concrete trigger setting of Anomaly #4: There are 80 connections of RC QP using Read opcode for each direction. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 128 requests in a batch. Each request has 4 SG elements and a fixed size of 128B.

(New) *Anomaly #5: RC SEND with small MTU, large WQE batch, long WQ, and long messages causes PFC pause frames and drastic throughput drop.*

(New) *Anomaly #6: RC SEND with small MTU, small WQE batch, large SG list batch, long WQ, small messages, and a few connections causes reduced throughput without any pause frame.*

They are similar to UD ones (Anomaly #1 and #2) but have a more complex and stricter trigger. For example, Collie observes such anomaly only when MTU is small (e.g., 1024 for RDMA), work depth exceeds 1K for each QP as well as

post multiple receive WQE in a batch. These anomalies are different because they have different QP types and stricter trigger conditions. For example, those anomalous application workloads in #1 and #2 won't trigger anomalies if we only switch the type of QP from UD to RC. Several discussion with our vendors tells us that the *Reliable Connection* type contains some subtle variance inside the RNIC that result in such difference. These two are currently not fixed yet.

Here is a simplified concrete trigger setting of Anomaly #5: There is 1 connection of RC QP using SEND/RECV opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 1024. The MTU is 1KB. The sender keeps sending 64 requests in a batch. Each request has 2 SG elements and a fixed size of 2KB.

Here is a simplified concrete trigger setting of Anomaly #6: There are 32 connections of RC QP using SEND/RECV opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 1024. The MTU is 1KB. The sender keeps sending 8 requests in a batch. Each request has 2 SG elements and a fixed size of 1KB.

#### **Root cause #2: Interconnect Context Memory cache misses reduce RNIC sending rates.**

(New) Anomaly #7: *RC WRITE with many QPs, small messages, small WQ depth, and small WQE batch size causes reduced throughput.*

(New) Anomaly #8: *RC WRITE with many MRs, small messages, and small WQE batch size causes reduced throughput.*

Though these two anomalies are well-known as the RDMA scalability problem, our real applications do not meet them even when the number of QPs exceeds 10K and the number of MRs exceeds 100K. However, Collie uncovers these two so we classified them into *New* anomalies. We take a deep look into how Collie discovers them and have many discussions with our vendors. We find our experience interesting and worthy of sharing: RNIC caches many necessary structures on its cache (e.g., memory translation table and connection context). When a request triggers cache miss, the RNIC has to issue extra PCIe operation to fetch them from the host DRAM. This will certainly induce extra PCIe latency for processing this request (victim request). However, RNIC is highly pipelined, so even when the victim request has finished the PCIe operation, it may still have to wait for the other pipeline stages to get ready (e.g., a previous long egress request blocks this short egress request). Therefore, if the request size is relatively large enough, the cache miss will not have a large effect on end-to-end performance because the overhead is hidden due to the pipeline.

Here is a simplified concrete trigger setting of Anomaly #7: There are 480 connections of RC QP using RDMA WRITE opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 16. The MTU is 1KB. The sender keeps sending requests without WQE batch. Each request has 1 SG element and a fixed size of 512B.

Here is a simplified concrete trigger setting of Anomaly #8: There are 24 connections of RC QP using RDMA WRITE opcode. Each QP has 1024 sending MR of 64KB and 1024 receiving MR of 64KB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending requests without WQE batching. Each request has 1 SG element and a fixed size of 512B.

#### **Root cause #3: PCIe controller blocks RNIC from reading host memory.**

(Old) Anomaly #9: *Bidirectional traffic with a mixture of small and large messages in an SG list on particular AMD servers causes PFC pause frames and drastic throughput drop.*

This anomaly is found by one of our production applications that keeps sending such message patterns (described in 2). The root cause of this anomaly is due to PCIe ordering issue. If the RNIC on the AMD server is not configured as PCIe relaxed ordering device, a DMA request may be blocked by the previous one. Therefore, when bidirectional traffic with a mix of short and long requests. The ingress short requests, together with the completion of egress traffic, blocks the ingress long requests. This results in RNIC buffer accumulation and triggers a large amount of PFC pause frames. The throughput can only achieve 60 Gbps with 25% pause frame duration ratio on average. With much effort from our appreciative vendors, we finally fix this by configuring RNIC as a forced relaxed ordering PCIe device.

Here is a simplified concrete trigger setting of Anomaly #9: There are 8 connections of RC QP using RDMA WRITE opcode for each direction. Each QP has 1 sending MR of 4MB and 1 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 8 requests in a batch. Each request has 3 SG elements and the pattern is [128B, 64KB, 1KB].

#### **Root cause #4: RNIC packet processing bottleneck.**

(New) Anomaly #10: *Bidirectional RC Write with large WQE batch size, a mixture of long messages and lots of short messages, and a few connections causes PFC pause frames.*

Collie finds that when several RC QPs keep posting multiple short requests (e.g., 64B, 128B) in batch and a few long requests for both directions, a large amount of pause duration is triggered. This RNIC of the RDMA subsystem has already been configured as forced relaxed ordering PCIe device (Anomaly #8 is solved). Our vendors have confirmed this anomaly and announce it fixed in their upcoming firmware release. The lengthy discussion with our vendor shows us the rough root cause: some component for packet processing inside the RNIC is not fully bidirectional, and our bidirectional reliable traffic (requires packet-level ACK) pattern with a huge amount of short requests, trigger that component's bottleneck. This results in long requests blocked and then many PFC pause frames are generated.

Here is a simplified concrete trigger setting of Anomaly #10: There are 320 connections of RC QP using RDMA

WRITE opcode for each direction. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending 64 requests in a batch. Each request has 1 SG element and the pattern is [64KB, 128B, 128B, 128B].

#### **Root cause #5: Host topology causes PCIe latency to increase, and this bottlenecks RNIC receiving rate.**

(New) *Anomaly #11: On specific types of AMD servers, Bidirectional cross-socket traffic causes pause frame storm and drastic throughput drop.*

Collie outputs the minimal feature set with only source/destination NUMA set and bidirectional traffic, indicating these two are the dominant factors. With this bidirectional (A to B and B to A) cross-socket NUMA setting (e.g., NUMA 0 from socket 0 for A and NUMA 2 from socket 1 for B, where socket 0 is the affinitive node for RNIC), even mild traffic with only a single connection can trigger up to 15.7% pause frame duration ratio. After several conversations with our RNIC and server vendors, we conjecture the root cause lies in these particular servers' cross-socket performance because we run the same traffic with the same NIC on different servers but do not observe the same phenomenon. We consider this anomaly as fixed because the vendor helps us roughly understand the root cause and suggest we use 2x100 Gbps NIC (each for a socket) to reduce cross-socket traffic, and we follow this guidance.

Here is a simplified concrete trigger setting of Anomaly #11: There is 1 connection of RC QP using RDMA WRITE opcode for each direction. Each QP has 32 sending MR of 4MB and 32 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 16 requests in a batch. Each request has 1 SG element with a fixed size of 256KB. The QP on host A is using the memory of socket 0 and the QP on host B is using the memory of socket 1.

(Old) *Anomaly #12: GPU-direct RDMA causes pause frame storm and drastic throughput drop on particular AMD servers.*

We observe a huge amount of pause frames and drastic throughput drop only on some servers in our clusters. The pause duration ratio can be up to 15% and throughput can drop to less than 20% (i.e., 40 Gbps) in this scenario. After careful debugging with our NIC vendor's strong support, we find out that there is a slight difference in PCIe bridge configuration (PCIe ACSCtl) between the anomalous server and normal ones. The anomalous configuration will forward GPU traffic to the root complex rather than directly to the RNIC. We fix this anomaly by adopting the correct configuration.

Here is a simplified concrete trigger setting of Anomaly #12: There are 8 connections of RC QP using RDMA WRITE opcode for each direction. Each QP has 1 sending MR of 4MB and 1 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 8 requests in a batch. Each request has 3 SG elements and the

pattern is [128B, 64KB, 1KB]. All MRs are allocated from GPU memory and we use the GPU under the same PCIe bridge (i.e., shown as PIX/PXB in *nvidia-smi* result).

#### **Root cause #6: RDMA NIC has potential in-NIC incast/congestion.**

(Old) *Anomaly #13: Co-existence of receiving traffic and loopback traffic causes PFC pause frames.*

This anomaly is found in our real applications and can also be uncovered by Collie. Our machine learning system runs workers and servers, and they use RDMA to accelerate the communication. However, once a worker and a server are scheduled on the same physical machine, there will be loopback traffic: the worker will send RDMA traffic to the server on the same host. Meanwhile, the server is receiving traffic from workers on other physical machines. The combination of receiving and loopback traffic triggers congestion/incast inside the NIC. And this RNIC lacks a mechanism to limit the loopback traffic rate, which makes the problem worse. After several discussions with our vendor, we bypass this anomaly by identifying the loopback communication and using other IPC mechanisms (e.g., shared memory). We do not consider this anomaly fixed because we cannot fully rely on other IPC mechanisms, especially for the virtualization environment. This anomaly exposes that a proper design of RNIC needs to consider NIC incast and we are glad to see that some latest RNIC have done so.

Here is a simplified concrete trigger setting of Anomaly #13: There are 16 connections of RC QP using RDMA WRITE opcode. 16 receivers are 8 senders are on the same host A and the other 8 senders are on the host B. Each QP has 32 sending MR of 4MB and 32 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 16 requests in a batch. Each request has 1 SG element with a fixed size of 256KB.

#### **A.2 Subsystem H with Broadcom 100 Gbps P2100G**

(New) *Anomaly #14: Bidirectional RC traffic with lots of connections and the large MTU causes reduced throughput without PFC pause frame.*

Collie observes that a large MTU is necessary to trigger this anomaly. Once we switch the MTU from 4096 (for RDMA) to 1024, both directions can achieve the line rate. This is unusual because most cases show that large MTU improves the performance and small MTU triggers performance anomalies. We don't observe the same phenomenon on any other type of RNICs.

Here is a simplified concrete trigger setting of Anomaly #14: There are 1024 connections of RC QP using RDMA WRITE opcode for each direction. Each QP has 81 sending MR of 256KB and 83 receiving MR of 256KB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 1 request in a batch. Each request has 4 SG elements with a fixed size of 64KB.

(New) *Anomaly #15: UD with long WQ and lots of connec-*

tions causes PFC pause frames.

This anomaly is similar to the Mellanox anomaly #1 but has a slightly different trigger. Collie successfully trigger #1 with only a single connection, but for P2100 RNIC our multiple runs show that a few connections are necessary.

Here is a simplified concrete trigger setting of Anomaly #15: There are 32 connections of UD QP using SEND/RECV opcode. Each QP has 1 sending MR of 4KB and 1 receiving MR of 4KB. Each QP has a work queue of length 64. The MTU is 2KB. The sender keeps sending 1 request in a batch. Each request has 1 SG element. The message pattern is like [256B, 1KB, 64B, 1KB].

(New) *Anomaly #16: RC READ with lots of connections, large WQE batch size, and small MTU causes PFC pause frames.*

This anomaly is similar to the Mellanox anomaly #4 and it shows that for the same RNIC and other hardware components, the best MTU choice can be different when workloads change.

Here is a simplified concrete trigger setting of Anomaly #16: There are 500 connections of RC QP using RDMA READ opcode. Each QP has 1 sending MR of 256KB and 1 receiving MR of 256KB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending 8 requests in a batch. Each request has 1 SG element with a fixed size of 64KB.

(New) *Anomaly #17: RC SEND with lots of connections, small WQE batch size, small MTU, short messages, and long WQ causes PFC pause frames.*

We have reported this anomaly to our vendor. To the best of our knowledge, we conjecture this anomaly is related to some corresponding WQE cache component inside RNIC.

Here is a simplified concrete trigger setting of Anomaly #17: There are 80 connections of RC QP using SEND/RECV opcode. Each QP has 1 sending MR of 1MB and 1 receiving MR of 1MB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending 1 request per batch. Each request has 1 SG element of fixed size 1KB.

(New) *Anomaly #18: Bidirectional RC WRITE with a few connections, large WQE batch, and small messages causes PFC pause frames.*

Our vendor has confirmed anomalies #17 and #18. They have reproduced these two anomalies and help us fix them. The solution is to configure some specific registers of the RNIC, and these two anomalies disappear.

Here is a simplified concrete trigger setting of Anomaly #18: There are 16 connections of RC QP using RDMA WRITE for each direction. Each QP has 1 sending MR of 12KB and 1 receiving MR of 12KB. Each QP has a work queue of length 64. The MTU is 1KB. The sender keeps sending 16 requests in a batch. Each request has 1 SG element of fixed size 64KB.



# SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers

Siva Kesava Reddy Kakarla<sup>1</sup>    Ryan Beckett<sup>2</sup>

<sup>1</sup>*University of California, Los Angeles*

Todd Millstein<sup>1,3</sup>    George Varghese<sup>1</sup>

<sup>2</sup>*Microsoft*    <sup>3</sup>*Intentionet*

## Abstract

The Domain Name System (DNS) has intricate features that interact in subtle ways. Bugs in DNS implementations can lead to incorrect or implementation-dependent behavior, security vulnerabilities, and more. We introduce the first approach for finding RFC compliance errors in DNS nameserver implementations, via automatic test generation. Our SCALE (Small-scope Constraint-driven Automated Logical Execution) approach *jointly* generates zone files and corresponding queries to cover RFC behaviors specified by an executable model of DNS resolution. We have built a tool called FERRET based on this approach and applied it to test 8 open-source DNS implementations, including popular implementations such as BIND, POWERDNS, KNOT, and NSD. FERRET generated over 13.5K test cases, of which 62% resulted in some difference among implementations. We identified and reported 30 new unique bugs from these failed test cases, including at least one bug in every implementation, of which 20 have already been fixed. Many of these bugs existed in even the most popular DNS implementations, including a critical vulnerability in BIND that attackers could easily exploit to crash DNS resolvers and nameservers remotely.

## 1 Introduction

The Domain Name System (DNS) plays a central role in today’s Internet, as it allows users to connect to online services through user-friendly domain names in place of machine-friendly IP addresses. Organizations across the Internet run DNS *nameservers*, which use DNS configurations called zone files to determine how to handle each query, either returning an IP address, rewriting the query to another one, or delegating the responsibility to another nameserver. There are many popular nameserver implementations of the DNS protocol in the wild, both open-source [21, 23, 25, 76] and in public or private clouds [2, 39, 85, 97].

Over time DNS has evolved into a complex and intricate protocol, spread across numerous RFCs [41, 80, 86, 96]. It is difficult to write an efficient, high-throughput, multithreaded implementation that is also bug-free and compliant with these RFC specifications. As a result, nameserver implementations frequently suffer from incorrect or implementation-specific behavior that causes outages [34, 103, 106], security vulnerabilities [74, 94], and more [15, 19, 22].

This paper presents the first approach for identifying RFC compliance errors in DNS nameserver implementations, by automatically generating test cases that cover a wide range of RFC behaviors. The key technical challenge is the fact that a DNS test case consists of both a query and a zone file, which is a collection of *resource records* that specify how queries should be handled. Zone files are highly structured objects with various syntactic and semantic well-formedness requirements, and the query must be related to the zone file for the test even to reach the core query resolution logic.

Existing standard automated test generation approaches are not suitable for our needs, as illustrated in the top of Figure 1. Fuzz testing is scalable but has well-known challenges in navigating complex semantic requirements and dependencies [13, 36], which are necessary to generate behavioral tests for DNS. As a result, fuzzers for DNS only generate queries and hence are used only to find parsing errors [10, 32, 89, 99]. Symbolic execution [72] can, in principle, generate DNS tests that achieve high code coverage but, in practice, suffers from the well-known problem of “path explosion” [9, 13, 36] that limits scalability and coverage. As a result, symbolic execution has only been used to identify generic errors like memory leaks in individual functions within nameserver implementations, again avoiding the need to generate zone files [93].

Our approach to automated testing for DNS nameservers, which we call **SCALE** (Small-scope Constraint-driven Automated Logical Execution), *jointly* generates zone files and the corresponding queries, does so in a way that is targeted toward covering many different RFC behaviors, and is applicable to black-box DNS nameserver implementations. The key insight underlying SCALE is that we can use the existing RFCs to define a model of the logical behaviors of the DNS resolution process and then use this model to guide test generation. Specifically, we have created an *executable* version of a recent formal semantics of DNS [71], which we then symbolically execute to generate tests for black-box DNS nameservers — each test consisting of a well-formed zone file and a query that together cause execution to explore a particular RFC behavior. Intuitively, tests that cover a wide variety of behaviors in our executable model will also cover a wide variety of behaviors in DNS nameservers since they have the same goal, namely to implement the RFCs.

Symbolic execution of our logical model is still fundamentally unscalable — there are an unbounded number of possible

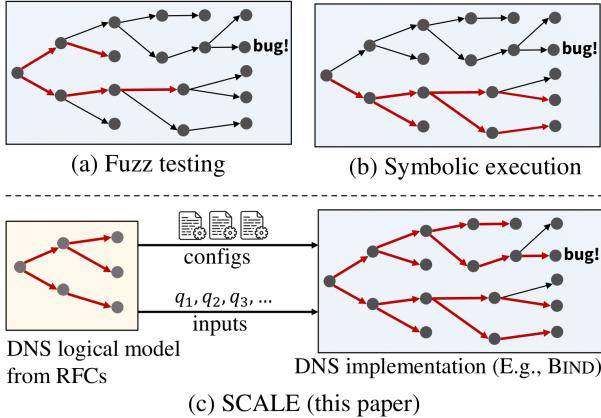


Figure 1: Overview of different automated testing approaches. Tested implementation paths are shown in red. (a) Fuzz testing is scalable but is often unable to navigate complex input requirements. (b) Symbolic execution can solve for input conditions but suffers from path explosion and has difficulty with complex data structures and program logic, and will thus only typically explore a small subset of possible program paths. (c) SCALE uses a logical model of the DNS RFCs to guide symbolic search toward *many* different *logical behaviors*.

execution paths, they grow exponentially in the size of the zone file, and expensive constraint solvers must be used to generate a test case for each path. We therefore bound the generated zone files to contain a very small number of resource records and short domain names — a maximum of 4 for each of these in our experiments, which is much smaller than real-world zone files. However, we provide experimental evidence of the existence of a *small-scope* property [43], meaning that many interesting behaviors can be covered with small tests. First, each return point in our logical model can be reached with a test where the length of domain names and the number of records in the zone file is at most 3. Each return point represents a distinct RFC-specified scenario for DNS resolution (e.g., a particular flavor of query rewrite). Second, while increasing this constant from 2 through 4 increased the number of errors that our tool identified, no new errors were found in a sample of paths that required size 5. This finding makes sense because, while zone files can contain a large number of records, the number of records that are relevant to any particular query tends to be small.

We have used the SCALE approach as the basis for a tool called FERRET<sup>1</sup> for automated testing of DNS nameserver implementations (Figure 2). FERRET generates tests using our logical model, which we have implemented in a modeling language called Zen [4] that has built-in support for symbolic execution. FERRET then performs *differential* testing by running these tests on multiple DNS nameserver implementations and comparing their results to one another. In this way FERRET can identify RFC violations, crashes, as well as situations where the RFCs may be ambiguous or underspecified, leading

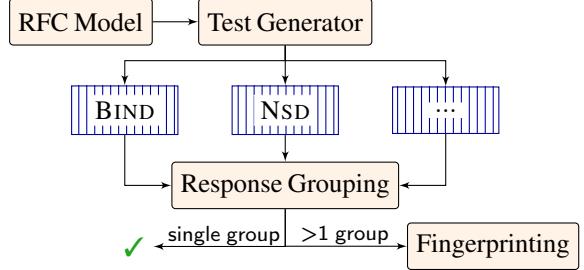


Figure 2: FERRET system architecture.

to implementation-dependent behavior. Because DNS implementers strive for behavioral consistency among their implementations [92], any test that produces divergent results among the implementations represents a likely error. However, there can be orders-of-magnitude fewer root causes than divergent tests, so as a final step we provide a simple but effective technique to help users with bug deduplication. We create a *hybrid fingerprint* for each test, which combines information from the test’s path in the Zen model with the results of differential testing, and then group tests by fingerprint for user inspection.

Using FERRET, in just a few hours we generated over 12.5K valid test cases<sup>2</sup> with a maximum zone-file size of 4 records. Running these tests on 8 different open-source DNS nameserver implementations, we found that the implementations’ behaviors only completely agreed on 35% of the tests. Our fingerprinting technique reduced the remaining cases to roughly 75 groups. Because our executable model includes a specification of the well-formedness conditions for zone files, we also leveraged Zen to systematically generate zone files that violate one of these conditions. We generated 900 invalid zone files of which 184 resulted in some difference among implementations. Inspecting tests from each fingerprinted group resulted in the discovery of 30 unique bugs across the different implementations. Developers have confirmed all of them as actual bugs and fixed 20 of them, at the time of writing. The most severe bug FERRET found was a subtle combination of zone file and query that an attacker could easily use to crash both BIND nameservers *and* resolvers remotely. We engaged in a secure disclosure process, after which the developers fixed the issue and then publicly disclosed the vulnerability, through a CVE (CVE-2021-25215) [26, 38] rated with high-severity.

**Contributions:** This paper’s contributions are:

- The first automated approach to identify RFC violations in black-box DNS nameservers. A unique feature of our approach, SCALE, is the joint generation of zone files and queries to produce high-coverage behavioral tests.
- An implementation of our approach in FERRET that combines SCALE with differential testing.
- A novel fingerprinting approach for bug deduplication that takes advantage of our RFC model to help triage bugs.
- An evaluation from testing 8 different open-source DNS nameserver implementations with tests generated by FER-

<sup>1</sup>FERRET: <https://github.com/dns-groot/Ferret>

<sup>2</sup>Test cases: <https://github.com/dns-groot/FerretDataset>

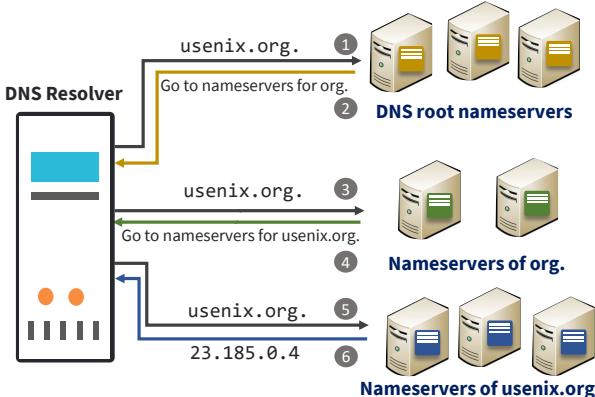


Figure 3: The resolution process for the domain name usenix.org (with no caching).

RET consisting of over 13.5K zone files, which resulted in the discovery of 30 new unique bugs and no false positives.

## 2 Background And Motivation

In this section, we first give a brief overview of DNS and then motivate FERRET through two previously unknown errors that it found in the popular BIND software for DNS [23].

### 2.1 Overview of DNS

The Domain Name System (DNS) is the phone book of the Internet. Its primary role is to translate domain names (like usenix.org) into various pieces of information, IP addresses being the most common. A domain name is represented as a sequence of labels joined by the . character. These labels form a tree-like hierarchy with the root as . and org as a child of it and so on. Each label at any level in the hierarchy can contain information, and the user obtains that information by querying the domain name formed by joining the labels from that node to the root. Data is stored as DNS *resource records* where each record has a domain (owner) name, a type for its information, and the content, among other things.

The namespace database tree is divided into a large number of *zones*. A zone is a collection of records that share a common end domain name. For example, the usenix.org zone has only records ending with usenix.org. All the resource records of a zone are available to the user through a set of authoritative nameservers, which are in turn identified by a domain name. For example, the usenix.org zone is available from servers like dns1.easydns.com, dns2.easydns.net and dns3.easydns.ca. The same zone is served by multiple servers to ensure redundancy and availability.

To resolve a domain name like usenix.org to its IP address, a client will traverse the tree from one of the root nameservers. The root nameserver checks its local zone file and either provides the IP record or returns a set of authoritative nameservers to ask instead. The client continues

Example Record	Description
a.exm.org. A 1.2.3.1	IPv4 record
*.exm.org. AAAA 1:db8::2:1	Wildcard IPv6 record
s.exm.org. NS ns.dns.com.	Delegation record
c.exm.org. DNAME cs.org.	Domain redirection
w.exm.org. CNAME a.exm.org.	Canonical name

Table 1: Examples of common DNS record types.

by querying the new set of nameservers either until the query is resolved or gets a non-existent domain name error. The process or the software that performs this traversal on the client side is called a *resolver*. The resolution process for the domain usenix.org is shown in Figure 3.

A nameserver can serve multiple zones. When a query comes to the nameserver, it first checks whether the query ends with any of the zone domains; otherwise, it sends a refusal message to the resolver. After picking a zone, the nameserver will look up the query name’s closest matching records. It then creates a response based on the query type and the records selected. DNS supports many record types, including records for IP addresses, pointers to other records, domain aliases, delegation records, and more. Table 1 shows a few example records.

### 2.2 Finding DNS Errors with FERRET

The goal of FERRET is to automatically generate high-coverage query and zone file inputs to find behavioral errors in DNS nameserver implementations. In this subsection we illustrate both the challenges in doing so and FERRET’s capabilities through two example errors that it automatically found in BIND.

**Bug #1: BIND sibling glue records bug.** FERRET generated the following test case, which identified a previously unknown performance bug in BIND [47].<sup>3</sup>

campus.edu. SOA ...
foo.campus.edu. NS ns1.campus.edu.
ns1.campus.edu. A 1.1.1.1

**Query:** <anything.foo.campus.edu., A>

In this test case, the query matches the NS record in the zone file, which delegates the query to another nameserver, ns1.campus.edu. However, that nameserver happens to be a sibling of foo.campus.edu (as they are both directly under campus.edu), and the zone file contains an A record, called a *glue record* [41], for the nameserver’s IP address. NSD, KNOT, and POWERDNS correctly return the NS record along with the glue record, avoiding extra round-trips to determine the nameserver’s IP address, while BIND returns only the NS record. Returning the sibling glue record is not compulsory, but our test case exposed two unrelated errors that can negatively affect the performance of many queries.

<sup>3</sup>Note that we have renamed the labels for all the example bugs for clarity.

After we filed the issue the BIND developers confirmed the bug saying, “This report turns out to be very interesting...” Briefly, BIND uses a “glue cache” that had two bugs. First, if the cache lookup fails, then glue records are supposed to be searched for in the zone file, but this was not happening. Second, glue records for siblings domain nameservers were accidentally never searched for at all.

This example illustrates the challenges of identifying nameserver behavior errors. Even though the zone file has only a few records, they have complex dependencies. First, there must be a delegation of the query to another nameserver. Second, that nameserver must be in the same zone. Third, that nameserver must be a sibling domain. Fourth, there must be a glue record for that domain in the zone. Given these dependencies, it is understandable that prior testing techniques did not uncover these bugs. Further, by comparing the outputs from multiple implementations, FERRET is able to identify this test case as potentially buggy behavior despite receiving a valid response from BIND.

**Bug #2: BIND crash.** As another, more dire example, consider the following zone file that FERRET generated. The zone file is invalid due to having two identical records, but BIND, NSD, and KNOT accept the zone file and make it valid by ignoring the duplicate record.

attack.com.	SOA	...
attack.com.	NS	ns1.outside.com.
attack.com.	NS	ns1.outside.com.
host.attack.com.	DNAME	com.
<b>Query:</b> <host.attack.host.attack.com., DNAME>		

FERRET generated multiple queries for this zone file (§ 3.6) and the one showed above caused BIND to crash.

In this test case, the DNAME record is applied to rewrite any queries ending with host.attack.com to end with just com, so the query that FERRET generated is rewritten to the new query host.attack.com. The nameservers add the DNAME record and rewritten query to the response before resolving the new query. The new query exactly matches the same DNAME record, so implementations are expected to return the current response. All implementations except BIND behaved as expected. BIND did not respond, and the query timed out. Inspecting the logs, we found that the server crashed with an assertion failure due to an attempt to add the same DNAME record to the response twice.

This error constitutes a critical security vulnerability. We next describe two scenarios to show how this failed assertion check can be exploited remotely by an attacker.

**Scenario 1 - Attack on a DNS hosting service that uses Bind:** DNS hosting services using BIND’s authoritative nameserver implementation (e.g., Dyn [42]) are vulnerable to this attack. An attacker can upload the above zone file to the authoritative server instances through the hosting service. Then, when the above query is requested, the server instances will crash as shown in Figure 4(a). Since a server instance

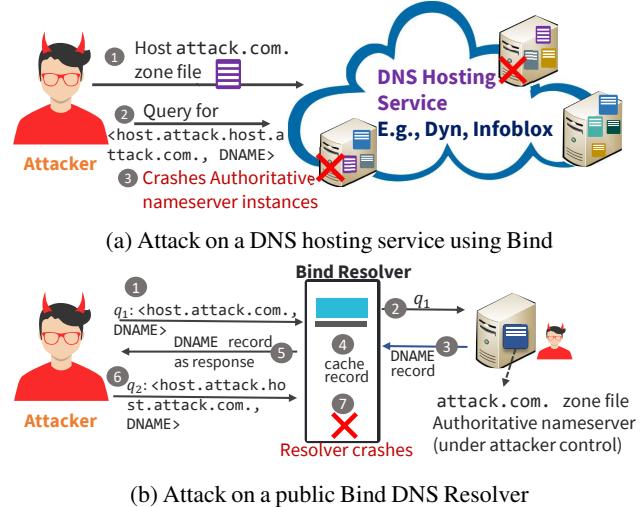


Figure 4: DNAME attack targeting the DNS hosting services (a) and the public BIND based recursive resolvers (b).

will generally be serving zone files from multiple customers, such a crash will take down the zones for all customers hosted at that nameserver. This provides a method for attackers to trivially and remotely initiate a denial of service attack against customers hosted by such a service.

#### Scenario 2 - Attack on a public Bind DNS resolver:

In this second scenario, the attacker can crash any public DNS resolver based on BIND, thereby constituting, as stated by the BIND security team, an “easily-weaponized denial-of-service vector.” As illustrated in Figure 4(b), the attacker purchases, registers, and controls the attack.com zone and its authoritative servers. The attacker then simply requests the DNAME record from a public recursive resolver running BIND, which attempts to fetch the result from the attacker’s authoritative server. This record is cached, and then the test query is sent to the resolver. The resolver uses the cached DNAME record and ultimately crashes as described earlier. In some estimates, BIND accounts for over half of all DNS resolvers in use [75], which means that attackers could effectively initiate a simple distributed denial of service (DDoS) attack against the numerous ISPs and public resolvers available to end users.

**Disclosure:** After discovering the DNAME attack, we initiated a responsible disclosure procedure with the BIND maintainers. Understanding the attack severity, they requested that we keep the issue confidential until they worked through their process to patch and then disclose the bug to the relevant parties in a controlled manner. BIND released a Common Vulnerabilities and Exposure (CVE-2021-25215) [26, 38], with a “high severity” rating and asked developers and users to upgrade to the patched version. The attack affected all maintained BIND versions, which in turn affected RHEL, Slackware, Ubuntu, and Infoblox.

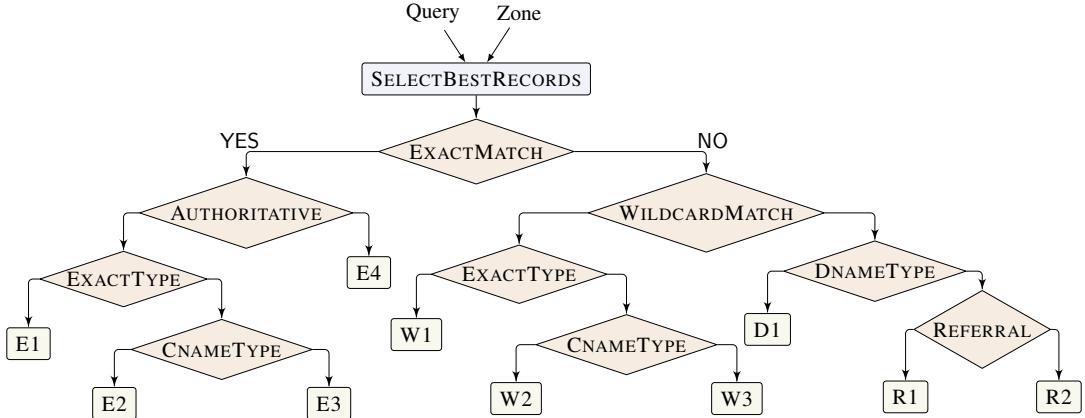


Figure 5: Abstract representation of the Authoritative DNS decision tree used to respond to a user query.

### 3 Methodology

In this section we overview our methodology for generating high-coverage tests for DNS nameserver implementations and discuss how we address several technical challenges.

#### 3.1 SCALE Approach

As illustrated by the examples in the previous section, the inputs to a DNS nameserver — a query and a zone file containing a set of records — are highly structured. Further, records can be of many different types and have many different kinds of dependencies among them. Therefore, an effective approach to automatically identifying RFC violations must be able to generate *valid* inputs that meet the required structural and semantic constraints of the domain, and it must also be able to explore different combinations of record types and features in a systematic way. To solve this joint generation problem, our approach, SCALE (Small-scope Constraint-driven Automated Logical Execution) leverages a specification of the DNS nameserver logic to drive test generation. Specifically, we have created an *executable* version of an existing DNS specification [71] and generate tests through *symbolic execution* [72] on this executable specification. Symbolic execution is a static analysis technique that enumerates execution paths in a program and uses automated constraint solvers to produce an input that will take each enumerated path, thereby generating tests that cover many different program behaviors.

While the end-to-end behavior of a DNS query lookup can require contacting many nameservers, we employ a *compositional* approach that only generates tests for a single nameserver in isolation. Because our formal model considers the space of all inputs to the nameserver that could be produced by the rest of the system, and because the “next step” delegation of the resolution process is captured in the output at a single nameserver, this approach still allows us to generate tests for all behaviors of the end-to-end DNS. In other words, any implementation bug that exists in a DNS nameserver

implementation can be found using our approach. In general, a downside of compositional testing is that it can lead to false positives if the tester considers input states that are, in reality, unreachable with respect to the rest of the system. However, in the case of DNS, nameservers keep no internal state — the response they provide is based only on the supplied query and configuration. This stateless nature implies that compositional testing will not incur any false positives.

Hence our formal semantics focuses on query lookup at a single nameserver, which we model as a stateless function that takes a user query and a zone file and produces a DNS response. Figure 5 shows an abstract view of this function. Given the input query and zone, DNS will first select the closest matching records in the zone for the query using the SELECTBESTRECORDS function and then follow the decision logic laid out in the figure using these records. Each leaf node represents a unique case in the DNS. For example, the tree shows four different cases of exact matches, labelled E1 through E4. Symbolic execution of our query-lookup function generates inputs that drive the function down different execution paths, thereby enabling us to systematically explore the space of DNS behaviors and feature interactions.

*Example:* Consider the path in Figure 5 to the leaf labelled R1. In order to reach that leaf, the selected records must not contain one with either an exact match or a wildcard match on the query domain name. Further, there should not be a DNAME match but should be one of type NS (REFERRAL). Finally, while not shown in the figure, when preparing a response to the query the function will also search for a glue record if the NS target is in the same zone. Solving all of these constraints caused symbolic execution to automatically generate the first test case shown in § 2.2, which identified two errors in BIND.

#### 3.2 An Executable Model of DNS

We have created an implementation of the formal semantics of query lookup [71] as a program in a modeling language called Zen [4], a domain-specific language (DSL) embedded in C#.

```

1  Zen<Response> QueryLookup(
2    Zen<Query> q,
3    Zen<Zone> z)
4  {
5    var records = SelectBestRecords(q, z);
6    var rname = records.At(0).Value().Name();
7    var types = records.Select(r => r.Type());
8
9    return If(
10      rname == q.Name(),
11      ExactMatch(records, q, z),
12      If(
13        IsWildcardMatch(q.Name(), rname),
14        WildcardMatch(records, q, z),
15        If(
16          types.Any(t => t == RType.DNAME),
17          Rewrite(records, q),
18          If(
19            And(types.Any(t => t == RType.NS),
20                Not(types.Any(t => t == RType.SOA))),
21            Response(Tag.R1,
22              Delegation(records, z), Null<Query>()),
23            Response(Tag.R2, empty, Null<Query>())
24          )));
25    }

```

Figure 6: Record lookup model in C# using Zen.

To illustrate this approach, we show several components of our model. Figure 6 shows the model’s main query-lookup function, as depicted in Figure 5. The function first selects the best records (Line 5) and then tests if the query domain name is equal to the records’ domain name (Line 10). If so, then this is an exact match and the model calls out to a helper function to specifically handle the ExactMatch subcase (Line 11). Similarly, if the query domain name is a wildcard match for the record domain name (Line 13), then we invoke the WildcardMatch subcase (Line 14). We show the implementation of wildcard matching in Figure 7. This function implements the case where the best matching record is a wildcard, properly handles interactions with CNAME records, and synthesizes the correct records for use in the resolver cache.

Our complete executable model consists of 520 lines of C# code. The model can also easily extend to new DNS RFCs that would be added in the future. Similarly, if an organization has a particular way of resolving RFC ambiguities or purposely deviates from the RFCs in specific ways, the organization can modify the logical model to reflect that intent.

We chose to implement our formal model in Zen because it has built-in support for symbolic execution. In Zen, certain inputs can be marked as *symbolic*, and the tool will then leverage SMT solvers [27] to produce concrete values for these inputs that drive the program down different execution paths. In our code examples, the `Zen<T>` type for inputs has the effect of marking them as symbolic. The tests produced by symbolic execution can then be used to test any DNS nameserver implementation. However, making symbolic execution effective required us to address several challenges, which we describe in the rest of this section.

```

26  Zen<Response> WildcardMatch(
27    Zen<IList<ResourceRecord>> rrs,
28    Zen<Query> q,
29    Zen<Zone> z)
30  {
31    var exact = rrs.Where(r => r.Type() == q.Type());
32    var record = rrs.At(0).Value();
33    var newQuery = Query(record.RData(), q.Type());
34    var exactSyn = RecordSynthesis(exact, q.Name());
35    var cnameSyn = RecordSynthesis(rrs, q.Name());
36
37    return If(
38      exact.Length() > 0,
39      Response(Tag.W1, exactSyn, Null<Query>()),
40      If(
41        rrs.Any(r => r.Type() == RType.CNAME),
42        Response(Tag.W2, cnameSyn, Some(newQuery)),
43        Response(Tag.W3, empty, Null<Query>())
44      );
45  }

```

Figure 7: Wildcard match model in C# using Zen.

### 3.3 Generating Valid Zone Files

The first challenge that we encountered is that zone files must satisfy several constraints in order to be considered well-formed. For instance, if there is a DNAME record in a zone file for `math.uni.edu`, then no other records below this domain name may exist, for any record type (e.g., an A record for `fun.math.uni.edu` is not allowed). The DNS RFCs define many such constraints as a way to eliminate ambiguous or useless zones, as shown in Table 2. Naively performing symbolic execution will produce many zone files that are not well formed. Further, DNS implementations typically preprocess zone files to reject ill-formed zones, thereby failing to test the intended execution path of the query lookup logic.

Fortunately, our SCALE approach admits a natural solution to this problem. We have formalized all of the DNS zone validity conditions as predicates in Zen. Whenever Zen’s symbolic execution engine produces a constraint representing the conditions under which the query lookup function takes a particular execution path, we conjoin these predicates to that constraint before Zen passes it off to an automated constraint solver. In this way we ensure that all test cases will have well-formed zone files by construction.

### 3.4 Data Representation

In our Zen model, we represent zone files as a list of resource records, where each resource record contains a domain name, record type, and data fields. We represent user queries similarly as consisting of a domain name and a query type. Record and query types are represented using enums, which Zen translates to integer values.

One challenging decision we ran into was how best to represent and model domain names, for both zone records and record data, in a manner that permits fully automatic and scalable analysis. For instance, a natural way to encode domain names

Validity Condition	RFC Document
i. All records should be unique (there should be no duplicates).	2181 [28]
ii. A zone file should contain exactly one SOA record.	1035 [87]
iii. The zone domain should be prefix to all the resource records domain name.	1034 [86]
iv. If there is a CNAME type then no other type can exist and only one CNAME can exist for a domain name.	1034 [86]
v. There can be only one DNAME record for a domain name.	6672 [96]
vi. A domain name cannot have both DNAME and NS records unless there is an SOA record as well.	6672 [96]
vii. No DNAME record domain name can be a prefix of another record’s domain name.	6672 [96]
viii. No NS record can have a non-SOA domain name that is a prefix of another NS record.	1034 [86]
ix. Glue records must exist for all NS records in a zone.	1035 [87]

Table 2: Summary of DNS zone file validity conditions specified in various RFCs.

would be as string values (a domain name is just a ‘.’ separated string). Indeed, modern SMT solvers like Z3 [27] support the logical theory of strings, so this is a natural approach to consider. However, the theory of strings is in general undecidable [14, 35]. Moreover, this encoding would require us to define complex predicates for manipulating domain names, including extracting each of the labels of a domain name and checking whether one domain name is a prefix of another.

Therefore, rather than model domain names as strings, we take advantage of the observation that the particular character values in a domain name label string do not matter for DNS lookup. Instead, all that matters is whether two labels are equivalent to one another and whether a label represents a wildcard. As such, we encode a domain name in Zen as a list of integers and use a specific integer value to represent the wildcard character ‘\*’. This allows us to use simple, efficient integer operations and constraints to manipulate domain names according to our formal model.

### 3.5 Handling Unbounded Data

A final challenge associated with symbolic execution for our formal model is the fact that there are several sources of *unboundedness*. For example, a zone file can contain an unbounded number of records, and a domain name can contain an unbounded number of labels. Our Zen model contains an unbounded number of paths, since the number of resource records in a zone file is unbounded and the function to select the best records must examine all of them and compare them to one another. SMT constraint solvers have limited support for unbounded data structures such as lists, and in general, reasoning about such constraints requires quantifiers, which lead to undecidability [95]. Therefore, in our Zen implementation we only consider inputs that have a bounded size, e.g., at most  $N$  records in a zone file, and hence only produce test cases that respect these bounds. The size of inputs is a parameter that is configurable by the user. While the SCALE approach can therefore fail to detect some errors, we provide experimental evidence of the existence of a *small-scope* property [43], meaning that many interesting behaviors, and behavioral errors, can be exercised with small tests (§ 5.1).

### 3.6 Generating Tests for Invalid Zone Files

While it’s critical to be able to generate well-formed zone files for testing, bugs can also lurk in implementations’ handling of ill-formed zones. Many DNS implementations use zone-file preprocessors to perform syntactic and semantic checks. For example, BIND uses named-checkzone [24], KNOT uses kzonecheck [18], and POWERDNS uses pdnsutil [20]. The implementations either reject an ill-formed zone or accept it but convert it to a valid one by ignoring certain records that cause it to be semantically ill-formed.

Many security vulnerabilities for software lie in the incorrect handling of unexpected inputs (e.g., in parsers [1]), and DNS software should be no different. Since our executable model includes a formulation of the validity conditions for zone files, we leverage Zen to systematically generate zone files that violate one of these conditions. For example, we ask Zen to generate a zone file in which all but the 7<sup>th</sup> condition in Table 2 is violated and the rest are satisfied.

If an invalid zone is rejected, then there is no issue, but if it is accepted, then there can be errors in how the zone is used for DNS lookups. To test for such errors we must also be able to generate queries for these zones. However, our formal model is only well defined for valid zone files so we cannot use it to generate queries. Instead, we use a technique from our prior work on zone-file verification [71] to partition queries into equivalence classes (ECs) relative to a given zone file. An equivalence class is a set of queries with the same resolution behavior, assuming a correct underlying DNS implementation, and the ECs are generated through a simple syntactic pass over a zone file. FERRET generates these ECs and then uses one representative query from each EC as a test. Though the number of ECs can vary widely, depending on the records in a zone file, in practice a zone containing four records will typically induce tens of ECs.

## 4 System Overview

FERRET is divided into several components, which are depicted in Figure 2. First it uses our Zen model described above to generate test inputs. Because domain names are encoded in

Zen using lists of integer labels (see § 3.4), FERRET includes a shim layer that translates the generated zone files and queries into meaningful domain names by mapping these labels to a collection of predefined strings (e.g., com). FERRET uses the equivalence-class (EC) generation algorithm of GROOT [71] to generate test queries for invalid zone files (§ 3.6).

FERRET uses Docker [83] to construct a working container image of each implementation. We cloned the implementations’ code as of October 1st, 2020 [16, 21, 23, 25, 29, 33, 76, 102], from their open-source repositories on GitHub [84] and GitLab [100]. FERRET starts a container for each image, and each container serves one zone file at a time as an authoritative zone. FERRET uses a Python library `dnspython` [17] to construct queries and send them to each implementation’s container. For each test case, the Python script prepares the container by stopping the running DNS nameserver, copying the new zone file and the necessary implementation-dependent configuration files to the container, and then restarting the DNS nameserver.

Finally, FERRET performs response grouping followed by fingerprinting to deduplicate errors that are likely to have the same root cause. For each test case, two DNS responses are considered equivalent, and hence in the same group, if they have the same response flags, return code, answer, and additional sections. FERRET only compares the authority section in two responses when their answer sections are empty. We do this because implementations are free to add additional records like a zone’s SOA or NS records along with the requested records. We then fingerprint tests that result in more than one group and thereby represent a likely error. The fingerprint for a valid test is a tuple consisting of (1) the case in the formal model (the leaf label in the decision tree from Figure 5) as well as (2) the response groupings. An example fingerprint is  $\langle R1, \{\{NSD, KNOT, POWERDNS, YADIFA\}, \{BIND, COREDNS\}, \{TRUSTDNS, MARADNS\}\} \rangle$ . The fingerprint for an ill-formed test is similar but we use the validity condition being violated instead of the model case.

## 5 Results

### 5.1 Testing Using Valid Zone Files

Using FERRET, we generated thousands of tests and used them to compare the behavior of 8 popular open-source authoritative implementations of DNS. Table 3 shows the 8 implementations, the languages they are implemented in, and a brief description of their focus or how they are used. We constrained FERRET to generate tests where the length of each domain name and the number of records in the zone was at most 4. We ran FERRET on a 3.6GHz 72 core machine with 200 GB of RAM and it generated a total of 12,673 valid test cases, one per path in our Zen model that is consistent with the length constraints, in approximately 6 hours. Users can run the tests in parallel, so the runtime depends heavily on the

Implementation	Language	Description
BIND [23]	C	<i>de facto</i> standard
POWERDNS [21]	C++	popular in N. Europe
NSD [76]	C	hosts several TLDs
KNOT [25]	C	hosts several TLDs
COREDNS [16]	Go	used in Kubernetes
YADIFA [29]	C	created by EURid (.eu)
TRUSTDNS [33]	Rust	security, safety focused
MARADNS [102]	C	lightweight server

Table 3: The eight open-source DNS nameserver implementations tested by FERRET. FERRET can test implementations implemented in any language.

Model Case	#Tests	#Tests Failing	#Fingerprints
E1	3180	239	7
E2	12	10	5
E3	96	12	3
E4	6036	5312	11
W1	60	33	8
W2	24	21	9
W3	18	16	1
D1	230	65	4
R1	2980	2529	27
R2	37	3	1

Table 4: Test generation statistics for  $n = 4$ . The model case refers to the leaves in Figure 5. Even though the number of failed tests is higher, the number of fingerprints is small.

user resources for parallelization. Each test takes around 10 seconds to run on average, and most of the time is spent setting up the zone file and necessary configuration files.

As described in § 4, FERRET runs each test against all 8 implementations and groups their responses. Out of 12,673 tests, FERRET found more than one group in the majority (8,240) of tests. Table 4 shows the number of tests generated for each case in the model (Figure 5), the number of tests where there was more than one group, and the number of unique fingerprints formed for each model case.

In total the 8,240 tests with more than one group were partitioned into 76 unique fingerprints, for a reduction of more than two orders of magnitude. For 24 of these fingerprints there exists only a single test case, while one fingerprint has 1892 corresponding tests. These 76 fingerprints can over-count the number of bugs since a single implementation issue can cause errors on multiple model paths. For example, YADIFA, TRUSTDNS, and MARADNS do not support DNAME records; so any generated test containing this feature will cause them to give the wrong answer or fail to respond. However, two tests can also have the same fingerprint despite different implementation root causes; so the number of fingerprints can

also under-count the number of bugs.

For these reasons, we manually examined the test cases matching each fingerprint, examining them all when the fingerprint has 4 or fewer tests and otherwise examining a small random sample. By doing this we identified 24 unique bugs, as summarized in [Table 6](#) (all except the ones marked with  $\ddagger$ ). All of these have been confirmed as actual bugs (no false positives) and developers have fixed 14 of them at the time of writing.

## 5.2 Testing Using Invalid Zone Files

FERRET generated 900 ill-formed zone files, 100 violating each of the validity conditions in [Table 2](#), in 2.5 hours. We used these zone files to test the four most widely used DNS implementations — BIND, NSD, KNOT, POWERDNS — as these have a mature zone-file preprocessor available.

There is no practical limit on the number of invalid zone files the tool can generate. We limited it to 100 for each violation in our experiments, but one could use FERRET to generate many more such tests if desired. Similarly, though we only explored violations of single well-formedness rules, it is straightforward to use FERRET to generate tests that violate a combination of rules. As a first step, FERRET checked all of the zone files with each implementation’s preprocessor: named-checkzone [24] for BIND, kzzonecheck [18] for KNOT, nsd-checkzone [77] for NSD, and pdnsutil [20] for POWERDNS. Each implementation can either reject or accept the invalid zone file and [Table 5](#) shows the statistics of how different implementations treat the zone files.

All together there are 573 invalid zone files (the first five rows in the table) that are accepted by more than one DNS implementation and so are amenable to differential testing. Our formal model relies on zones to be well-formed: so we cannot use it to generate queries for these zones. Instead we leverage GROOT [71], which generates query equivalence classes (ECs) of the form  $\langle \text{example.com}, t \rangle$  for a given zone file, one for each DNS record type  $t$ , and does not require the zone to be semantically well-formed. We used 7 query types: A, NS, CNAME, DNAME, SOA, TXT, AAAA. We excluded 19 zone files as GROOT generated over 200 ECs for each of them due to multiple interacting DNAME loops. For the remaining 554 zone files, the average number of ECs is  $21 * 7$  i.e., 21 domains names and each domain name is paired with the 7 types, and we chose one representative query from each EC.

The last column in [Table 5](#) shows the results of differential testing. For example, 106 out of the 201 zone files in the first row exhibited differences among the three implementations during testing. We manually inspected all differences for the zone files that violated conditions of i, ii, iii, vi, and ix, as there were 12 or fewer such differences in each category, and we inspected a random sample for the others. By doing this we identified 6 new errors as shown in [Table 6](#) with the  $\ddagger$  symbol and all of them are fixed. Some of the errors identified earlier were also present here but are not double-counted.

B I N D	N S D	K N O T	P D N S	#Zones	Condition violated	#Zones with a difference
A	A	A	R	100 + 100 + 1	i or viii or ix	11 + 94 + 1
A	A	R	R	100 + 61	vi or ix	8 + 3
A	R	A	R	17 + 100	ii or iii	1 + 6
A	R	R	A	60	vii	53
R	A	R	A	34	ix	7
A	R	R	R	39	vii	-
R	A	R	R	4	ix	-
R	R	R	A	95 + 1	v or vii	-
R	R	R	R	83 + 100 + 5	ii or iv or v	-

[Table 5](#): Invalid zone file statistics. The second row shows that 100 (61) zone files that violate condition vi (ix) are accepted by only BIND and NSD, and 8 (3) of them resulted in some difference between the two implementations.

## 5.3 Example Bugs

We now provide a detailed description of some of the bugs from [Table 6](#). Two of them were already described in § 2.2.

**Bug #3: COREDNS Crash.** FERRET generated the following test that causes COREDNS, the recommended nameserver for Kubernetes, to crash. It was subsequently confirmed and fixed by the COREDNS developers.

```
example. SOA ...
*.example. CNAME foo.example.
```

**Query:** `<baz.bar.example., CNAME>`

In this example the zone file has a wildcard CNAME record that rewrites any query ending with the label example to foo.example. This rewritten query will then match the wildcard record again and so on, causing COREDNS to loop and consume resources until, eventually, the server crashes with the following message:

```
runtime: goroutine stack exceeds 1000000000-byte limit
runtime: sp=0xc03c6c0378 stack=[0xc03c6c0000, ...]
fatal error: stack overflow
```

Interestingly, COREDNS correctly guards against CNAME loops that do not involve wildcard; so only a test that combines CNAME and wildcards will trigger the bug. After our bug report, the developers fixed the issue by adding a loop counter and breaking the loop if the depth exceeds nine. They commented: “Note the answer we’re returning will be incomplete (more cnames to be followed) or illegal (wildcard cname with multiple identical records). For now it’s more important to protect ourselves than to give the client a valid answer.”

Crashes like this represent serious security vulnerabilities, particularly in multi-tenant settings such as the attack described earlier in [Figure 4\(a\)](#).

**Bug #4: Wrong RCODE for synthesized CNAME.** FERRET generated a zone that violates condition vii in [Table 2](#):

Implementation	Bugs Found	Bug Type	Status
BIND	Sibling glue records not returned [47]	Wrong Additional	✓
	Zone origin glue records not returned [45]	Wrong Additional	✓
	DNAME recursion denial-of-service <sup>◊</sup> [44]	Server Crash	✓
	Wrong RCODE for synthesized record <sup>◊</sup> [46]	Wrong RCODE	✓
NSD	DNAME not applied recursively [65]	Wrong Answer	✓
	Wrong RCODE when * is in Rdata [64]	Wrong RCODE	✓
	Used NS records below delegation <sup>◊</sup> [67]	Wrong Answer	✓
	Wrong RCODE for synthesized record <sup>◊</sup> [66]	Wrong RCODE	✓
POWERDNS	CNAME followed when not required [62]	Wrong Answer	✓
	pdnsutil check-zone DNAME-at-apex <sup>◊</sup> [63]	Preprocessor Bug	✓
KNOT	Incorrect record synthesis [58]	Wrong Answer	✓
	DNAME not applied recursively [61]	Wrong Answer	✓
	Used records below delegation [59]	Wrong Answer	✓
	Error in DNAME-DNAME loop KNOT test [60]	Faulty KNOT Test	✓
	Wrong RCODE for synthesized record <sup>◊</sup> [91]	Wrong RCODE	✓
COREDNS	NXDOMAIN for existing domain [53]	Wrong RCODE	✓
	Wrong RCODE for CNAME target [55]	Wrong RCODE	✓
	Wildcard CNAME loops & DNAME loops [52]	Server Crash	✓
	Wrong RCODE for synthesized record [57]	Wrong RCODE	✓
	CNAME followed when not required [56]	Wrong Answer	✓
	Sibling glue records not returned [54]	Wrong Additional	✓
YADIFA	CNAME chains not followed [70]	Wrong Answer	✓
	Wrong RCODE for CNAME target [69]	Wrong RCODE	✓
	Used records below delegation [68]	Wrong Answer	✓
MARADNS <sup>†</sup>	AA flag set for zone cut NS RRs	Wrong Answer	✓
	Used records below delegation	Wrong Answer	✓
TRUSTDNS <sup>†</sup>	Wildcard match only one label [49]	Wrong Answer	✓
	Used records below delegation [51]	Wrong Answer	✓
	AA flag set for zone cut NS RRs [50]	Wrong Flag	✓
	CNAME loops crash the server [48]	Server Crash	✓

Table 6: Summary of the bugs found by FERRET across the eight implementations. Status column represents whether the developers responded and acknowledged (✓) and also fixed (☒) to the filed bug report. The <sup>†</sup> symbol denotes implementations with unreported issues due to missing or unimplemented features. The <sup>◊</sup> symbol denotes the bugs found exclusively using testing with invalid zone files. We reported all the bugs FERRET identified to the respective developers before publishing this paper.

```

test.com. SOA ...
foo.test.com. DNAME bar.test.com.
cs.foo.test.com. AAAA 1:db8::2:1
Query: <www.foo.test.com., CNAME>

```

BIND and POWERDNS accepted the zone file but NSD and KNOT did not. FERRET chose the above query as the representative from the query EC  $\langle \alpha.\text{foo}.\text{test}.\text{com}. , \text{CNAME} \rangle$  generated by GROOT, where  $\alpha$  represents any sequence of labels that does not start with cs. BIND responded with:

```

"rcode NXDOMAIN",
";ANSWER",
"foo.test.com. 500 IN DNAME bar.test.com.",
"www.foo.test.com. 500 IN CNAME www.bar.test.com.",

```

The response from POWERDNS was the same but with a NOERROR RCODE. The RCODE is important as resolvers can use QNAME minimization (RFC 7816 [6]) to wrongly conclude

domain (non-)existence if an incorrect RCODE is returned. However, since the RFCs do not describe this subtle case, the intended behavior is unclear. Since the query is not relevant to the AAAA record, which violates the validity condition, to further investigate this issue we decided to remove that record and check the responses from NSD and KNOT. Both responded with the same response as BIND, leading us to (wrongly) conclude that the issue was with POWERDNS.

To our surprise, after reporting the issue to POWERDNS they responded: “The PowerDNS behavior looks correct to me. Are you sure BIND, NSD, and Knot all return NXDOMAIN on a CNAME query in this context?” BIND and KNOT noticed the issue we filed on POWERDNS’s GitHub and fixed the bug almost immediately, even before we filed reports on their repositories. After some back and forth with the NSD developers they concurred saying: “If you are right that the other implementations do this, then we can do that too; that makes less unexpected surprises in packet responses.”

Max Length ( $n$ )	2	3	4	5
No. of Tests	52	618	12673	646K (51K tested)
Test generation time	10m	40m	6h	14d
No. of Tests Failing	12	224	8240	41173
No. of Fingerprints	9	22	76	115
No. of Bugs	4	14	24	27

Table 7: Results summary for different bounds.

**Bug #5: POWERDNS pdnsutil bug.** FERRET generated the following test case and POWERDNS returned an incorrect response, exposing a bug in its zone-file preprocessor.

dept.com.	SOA	...
dept.com.	DNAME	dept.edu.
host.dept.com.	A	1.1.1.1
Query: <host.dept.com., A>		

The zone file is considered invalid as it violates condition vii in Table 2. nsd-checkzone and kzonecheck preprocessors reject the zone file but named-checkzone and pdnsutil do not raise any errors or warnings and accept the zone file. When queried for the A record, POWERDNS returned this record even though it should have used the DNAME record. POWERDNS has a long-standing open issue about handling DNAME occlusion (records below a DNAME, which should be ignored), and pdnsutil generally gives a warning but did not in this specific case. We filed a bug report for this test and the developers confirmed a bug in pdnsutil when the DNAME is at the apex of the zone. This is now fixed and pdnsutil gives a warning as in other occlusion cases.

#### 5.4 Small-scope Property Validation

Finally, we performed an experiment to validate the small-scope property that justifies our approach — many interesting behaviors can be covered with small tests. We used FERRET to generate valid tests where the length of each domain name and the number of records in the zone were limited to  $n$ , for different values of  $n$ . Table 7 shows the results. For example, when  $n = 2$  there are 52 feasible paths through the model. FERRET generated the corresponding 52 tests in 10 minutes, out of which 12 had more than one group, and these 12 fell into 9 fingerprints. By inspecting those failed tests, we identified 4 unique bugs, which are a subset of the ones identified by our evaluation described in § 5.1, where  $n = 4$ .

Our experiment identifies two distinct forms of small-scope property. First, the DNS query resolution protocol itself, as represented by our logical model, has a small-scope property. In particular, when  $n = 2$  all leaf nodes in Figure 5 are covered by at least one test, except for the R1 leaf, and all leaf nodes are covered when  $n$  is 3 or higher. Hence, although we are restricted to generating small zones, we can still cover all return points in our formal model, each of which represents a distinct RFC behavior.

Second, the DNS nameserver implementations have a small-scope property. In part the fact that we have identified dozens of subtle new errors is evidence that small tests can explore interesting behaviors. The results in Table 7 add further evidence. As we increase the size of  $n$  from 2 to 3 to 4, the number of bugs identified goes from 4 to 14 to 24. In the  $n = 5$  case, FERRET generated over 646K tests and took almost 14 days to finish. The distribution of tests across model cases is similar to the  $n = 4$  breakdown shown in Table 4, where the majority of tests fall into the E1, E4 and R1 cases. We randomly sampled 50K tests to run from these three cases, according to their proportions. The other cases totalled to around 1000 tests, so we ran all of them. Out of the resulting 115 fingerprints, 50 fingerprints were in common with the fingerprints of  $n = 4$ . We therefore decided to examine the remaining 65 fingerprints to search for new bugs. For these 65 fingerprints, the median number of tests in each fingerprint was 3, and the mode was 1. We found three bugs that we did not find with  $n = 4$ , but all three bugs were covered by the tests for invalid zones with  $n = 4$  (§ 5.2). In other words, increasing  $n$  from 4 to 5 has so far not uncovered any new errors in the DNS nameserver implementations.

## 6 Discussion

Our SCALE approach worked surprisingly well at identifying subtle errors in implementations. This was not obvious from the beginning, since each implementation can have very different control logic compared to one another and compared to our formal model. And yet seemingly the tests derived from paths through our formal RFC model frequently uncover bugs in rare control paths for these implementations.

On the other hand, this approach is not a panacea. We found situations where one path in the model corresponds to multiple paths in an implementation due to the internal data structures that it uses to represent different record types, which can lead to FERRET missing some issues. This showed up, for example, with empty non-terminals (ENTs) – domain names that own no resource records but have subdomains that do. Since there is no explicit branch that differentiates empty non-terminals in the model, FERRET did not generate test cases where the zone file had both an ENT and a query targeting that ENT. However, by manually testing a few such cases, we found two more bugs in COREDNS. Going forward it may be possible to extend FERRET to find more cases like this by adding additional non-semantic branches to the model to expose behavior thought to be error-prone.

More generally, we believe our SCALE approach to RFC compliance testing and “ferreting” out bugs through (i) symbolic execution of a small formal model to jointly generate configurations together with inputs, combined with (ii) differential testing, and (iii) fingerprinting, could be useful more broadly beyond the DNS. For instance, there are many other complex and distributed protocols used at different network layers such as routing protocols like BGP and OSPF,

flow control protocols like PFC, new transport layer protocols such as QUIC, and many more. It would be interesting future work to apply the SCALE methodology beyond DNS.

## 7 Related Work

FERRET and SCALE are related to several lines of prior work in DNS and in automated testing.

**Verified DNS implementations.** One approach is to build, from scratch, a nameserver implementation verified to be correct. This approach has found some success in other domains, for example, in operating system microkernels [73] using proof assistants such as Coq [79]. Ironsides [12] is an implementation of a DNS resolver and authoritative nameserver that uses SPARK [3] to prove the absence of dataflow errors such as buffer overflows. While this work is promising, it does not formalize the DNS RFC semantics and thus cannot provide any functional correctness guarantees. Moreover, open source implementations such as BIND [23] are already used pervasively in the Internet. Providing a new verified implementation does not help these existing deployments.

**Models for DNS.** In our prior work on the GROOT zone-file verifier [71] we provided the first formalization of DNS semantics. However, it was a paper formalism and was only used to prove the correctness of the equivalence-class generation algorithm that forms the core of GROOT’s approach to verifying zone files. Indeed, GROOT assumes that DNS implementations conform to the DNS RFCs. Our work is therefore complementary, but we used GROOT’s logical model as a basis for our executable Zen model. We also leveraged GROOT’s equivalence-class generation algorithm to create queries for invalid zone files.

**Fuzz testing.** Fuzz testing with semi-random and/or grammar-based tests has seen success in recent years [1, 5, 40, 78, 101]. However, as mentioned in § 1, fuzzing cannot easily be used in our setting due to the need to navigate complex constraints and dependencies, and hence existing fuzzers for DNS [10, 89, 99] are limited to testing DNS parsers and use a fixed zone file.

**Symbolic execution.** Symbolic execution [36, 37], which systematically solves for inputs that take different execution paths in a program, has also been successful [9, 11]. However, as described in § 1, due to the scale and complexity of DNS nameserver implementations, symbolic execution has been used only on individual functions and has avoided the need to generate zone files [93]. Our SCALE approach uses symbolic execution to drive test generation, but it does so on an executable model of the RFC behavior, which is significantly smaller and simpler than an implementation and has carefully chosen data representations that are amenable to symbolic execution. As a result, symbolic execution on our model is tractable and allows us to jointly generate (small) zone files and DNS queries that exercise interesting behaviors.

**Model- and specification-based testing.** In model-based testing (MBT) [8, 88, 90, 104] a user builds an abstract model of the system to test (e.g., a finite state machine [8, 104]). A tester implementation then generates paths through this abstract model and creates concrete tests by “filling in” missing information from the abstract example. Closest to our work are model-based testers for black-box network functions (e.g., [30, 98]), which also use symbolic execution to generate tests. However, they respectively use finite-state machine models [30] and a domain-specific language for specifying network function behavior [98], while we have implemented a full functional model of DNS in a general modeling language [4]. Further, their setting does not require generating configurations, which is the key technical challenge for testing protocols like DNS.

Specification-based testing leverages a user-provided specification of the valid inputs to a function. Most commonly, tests are generated by finding inputs that satisfy a given precondition [7]. Like SCALE these approaches typically rely on a small-scope hypothesis [43] and hence focus on generating small inputs. Recent work has developed an approach to automated testing for QUIC implementations [81, 82] that leverages a formal specification, but in a very different way than in our approach. Specifically, the specification models the party that is interacting with the implementation being tested and is used to generate valid responses.

Finally, recent works automatically learn protocol models from implementations [31] or RFCs [105]. We could potentially adopt these techniques in the future to reduce the burden of producing our formal model.

## 8 Conclusion

Despite its importance as the “phonebook” of the Internet, DNS is fraught with implementation bugs that can impact millions of users. In this paper, we introduced FERRET, the first automatic test generator for RFC compliance of DNS nameserver implementations. The SCALE approach underlying FERRET uses symbolic execution of a formal model to jointly generate configurations together with inputs. FERRET combines this technique with differential testing and fingerprinting to identify and automatically triage implementation errors. In total FERRET identified 30 new bugs, including at least two for each of the 8 implementations that we tested. We believe that this combination of techniques can generalize to “ferret” out subtle RFC-compliance bugs in large implementation code bases for other network protocols that use configurations.

## Acknowledgements

We thank our shepherd Phillipa Gill and the anonymous reviewers for their insightful comments. We also thank the DNS developers and the DNS-OARC community for their feedback on the bug reports. This work was partially supported by NSF grants CNS-1704336 and CNS-1901510.

## References

- [1] American Fuzzing Lop (AFL). AFL 2018.  
<https://lcamtuf.coredump.cx/afl/>.
- [2] Amazon. Route 53.  
<https://aws.amazon.com/route53/>.
- [3] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, London, GBR, 2012.
- [4] Ryan Beckett and Ratul Mahajan. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks, HotNets '20*, page 8–15, New York, NY, USA, 2020. Association for Computing Machinery.
- [5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.
- [6] Stéphane Bortzmeyer. DNS Query Name Minimisation to Improve Privacy. RFC 7816, March 2016.
- [7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02*, page 123–133, New York, NY, USA, 2002. Association for Computing Machinery.
- [8] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. A formal tls handshake model in Int. In John P. Gallagher, Rob van Glabbeek, and Wendelin Serwe, editors, *Proceedings Third Workshop on Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation*, Thessaloniki, Greece, 20th April 2018, volume 268 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–40, Greece, 2018. Open Publishing Association.
- [9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, page 209–224, USA, 2008. USENIX Association.
- [10] Frederic Cambus. Fuzzing dns zone parsers.  
<https://www.cambus.net/fuzzing-dns-zone-parsers/>.
- [11] Marco Canini, Vojin Jovanović, Daniele Venzano, Dejan Novaković, and Dejan Kostić. Online testing of federated and heterogeneous distributed systems. In *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11*, page 434–435, New York, NY, USA, 2011. Association for Computing Machinery.
- [12] M. Carlisle and B. Fagin. Ironsides: Dns with no single-packet denial of service or remote code execution vulnerabilities. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 839–844, Anaheim, CA, USA, 2012. IEE.
- [13] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.
- [14] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.
- [15] Bind Community. Bind gitlab issues.  
<https://gitlab.isc.org/isc-projects/bind9/-/issues>.
- [16] CoreDNS community. Coredns.  
<https://coredns.io/>.  
Code commit used: <https://github.com/coredns/coredns/tree/6edc8fe7f6c2f57844c8ee7f7f5deef71085ebe8>.
- [17] Dnspython Community. Dnspython.  
<https://dnspython.readthedocs.io/en/latest/index.html>.
- [18] Knot community. kzonecheck – knot dns zone file checking tool.  
[https://www.knot-dns.cz/docs/2.5/html/man\\_kzonecheck.html](https://www.knot-dns.cz/docs/2.5/html/man_kzonecheck.html).
- [19] NSD Community. Nsd github issues.  
<https://github.com/NLnetLabs/nsd/issues>.
- [20] PowerDNS community. Pdnsutil.  
<https://doc.powerdns.com/authoritative/manpages/pdnsutil.1.html>.
- [21] PowerDNS Community. Powerdns.  
<https://www.powerdns.com/>.  
Code commit used: <https://github.com/PowerDNS/pdns/tree/a03aaad7554483ee6efe72a81eda00a9d1a94fe5>.
- [22] PowerDNS Community. Powerdns github issues.  
<https://github.com/PowerDNS/pdns/issues?q=is%3Aissue+is%3Aopen+label%3Aauth>.

- [23] Internet Systems Consortium. Bind 9. <https://www.isc.org/bind/>.  
Code commit used: <https://gitlab.isc.org/isc-projects/bind9/-/tree/dbcf683c1a57f49876e329fca183cb39d20ca3a4>.
- [24] Internet Systems Consortium. named-checkzone(8). <https://linux.die.net/man/8/named-checkzone>.
- [25] CZ.NIC. Knot. <https://www.knot-dns.cz/>.  
Code commit used: <https://gitlab.nic.cz/knot/knot-dns/-/tree/563fcdd886b5d5c52bceeb8fda3c4bda59ece73e>.
- [26] National Vulnerability Database. CVE-2021-25215 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2021-25215>.
- [27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [28] Robert Elz and Randy Bush. Clarifications to the DNS Specification. RFC 2181, July 1997.
- [29] EURid.eu. Yadifa. <https://www.yadifa.eu/>.  
Code commit used: <https://github.com/yadifa/yadifa/tree/dc5bed2fb8ec204af9b65eeb91934c2c85098cbb>.
- [30] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing Context-Dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, March 2016. USENIX Association.
- [31] Tiago Ferreira, Harrison Brewton, Loris D’Antoni, and Alexandra Silva. Prognosis: Closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM ’21, page 762–774, New York, NY, USA, 2021. Association for Computing Machinery.
- [32] Jonathan Foote. How to fuzz a server with american fuzzy lop. <https://www.fastly.com/blog/how-fuzz-server-american-fuzzy-lop>, 2015.
- [33] Benjamin Fry and Community. Trust-dns. <http://trust-dns.org/>.  
Code commit used: <https://github.com/bluejekyll/trust-dns/tree/7d9b186121fb5cb331cf2ec6baa47846b83de8fc>.
- [34] James Fryman. Dns outage post mortem. <https://github.blog/2014-01-18-dns-outage-post-mortem/>, 2014.
- [35] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: What’s decidable? In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing*, HVC’12, page 209–226, Berlin, Heidelberg, 2012. Springer-Verlag.
- [36] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’08, page 206–215, New York, NY, USA, 2008. Association for Computing Machinery.
- [37] Patrice Godefroid, Nils Karlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.
- [38] Suzanne Goldlust, Michał Kępień, Peter Davies, and Everett Fulton. CVE-2021-25215: An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself. <https://kb.isc.org/v1/docs/cve-2021-25215>.
- [39] Google. Cloud dns. <https://cloud.google.com/dns>.
- [40] Sam Hocevar. zzuf: multi-purpose fuzzer. <http://caca.zoy.org/wiki/zzuf/>, 2007.
- [41] Paul E. Hoffman, Andrew Sullivan, and Kazunori Fujiwara. DNS Terminology. RFC 8499, January 2019.
- [42] Dyn Inc. Dynamic dns. <https://account.dyn.com/>.
- [43] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.
- [44] Siva Kakarla, Mark Andrews, Michał Kępień, Peter Davies, and Michal Nowak. [CVE-2021-25215] An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself. <https://gitlab.isc.org/isc-projects/bind9/-/issues/2540>.

- [45] Siva Kesava R Kakarla and Mark Andrews. Glue records can be returned when the name server's name is same as the zone origin.  
<https://gitlab.isc.org/isc-projects/bind9/-/issues/2385>.
- [46] Siva Kesava R Kakarla, Mark Andrews, and Michał Kępień. DNAME: synthesized CNAME might be perfect answer to CNAME query.  
<https://gitlab.isc.org/isc-projects/bind9/-/issues/2284>.
- [47] Siva Kesava R Kakarla, Mark Andrews, and Michał Kępień. Sibling (In-bailiwick rule of RFC 8499) domain IP records not returned.  
<https://gitlab.isc.org/isc-projects/bind9/-/issues/2384>.
- [48] Siva Kesava R Kakarla and Benjamin Fry. CNAME loops throws off the server.  
<https://github.com/bluejekyll/trust-dns/issues/1283>.
- [49] Siva Kesava R Kakarla and Benjamin Fry. Wildcards match only one label.  
<https://github.com/bluejekyll/trust-dns/issues/1342>.
- [50] Siva Kesava R Kakarla and Benjamin Fry. Zone cut NS RRs returned as authoritative records.  
<https://github.com/bluejekyll/trust-dns/issues/1273>.
- [51] Siva Kesava R Kakarla, Benjamin Fry, and Jonas Bushart. Glue records returned as authoritative records by the server.  
<https://github.com/bluejekyll/trust-dns/issues/1272>.
- [52] Siva Kesava R Kakarla and Miek Gieben. Handling wildcard CNAME loops.  
<https://github.com/coredns/coredns/issues/4378>.
- [53] Siva Kesava R Kakarla and Miek Gieben. NXDOMAIN returned when the domain exists.  
<https://github.com/coredns/coredns/issues/4374>.
- [54] Siva Kesava R Kakarla and Miek Gieben. Sibling (In-bailiwick rule of RFC 8499) domain IP records can also be returned along with NS records.  
<https://github.com/coredns/coredns/issues/4377>.
- [55] Siva Kesava R Kakarla and Chris O'Haver. Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR rcode.  
<https://github.com/coredns/coredns/issues/4288>.
- [56] Siva Kesava R Kakarla, Chris O'Haver, and Kohei Yoshida. CNAME need not be followed after a synthesized CNAME for a CNAME query.  
<https://github.com/coredns/coredns/issues/4398>.
- [57] Siva Kesava R Kakarla, Chris O'Haver, and Kohei Yoshida. Return code for synthesized CNAME records (from wildcards and DNAMEs).  
<https://github.com/coredns/coredns/issues/4341>.
- [58] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. Record incorrectly synthesized from wildcard record.  
<https://gitlab.nic.cz/knot/knot-dns/-/issues/715>.
- [59] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. Records below delegation are not ignored (kzonecheck also does not raise any issue).  
<https://gitlab.nic.cz/knot/knot-dns/-/issues/713>.
- [60] Siva Kesava R Kakarla, Libor Peltan, Daniel Salzman, and mscbg. DNAME-DNAME loop test case is not a loop.  
<https://gitlab.nic.cz/knot/knot-dns/-/issues/703>.
- [61] Siva Kesava R Kakarla, Libor Peltan, Daniel Salzman, and Vladimír Čunát. DNAME not applied more than once to resolve the query.  
<https://gitlab.nic.cz/knot/knot-dns/-/issues/714>.
- [62] Siva Kesava R Kakarla and Peter van Dijk. CNAME need not be followed after a synthesized CNAME for a CNAME query.  
<https://github.com/PowerDNS/pdns/issues/9886>.
- [63] Siva Kesava R Kakarla and Peter van Dijk. pdnsutil DNAME checks have issues.  
<https://github.com/PowerDNS/pdns/issues/9734>.
- [64] Siva Kesava R Kakarla and Wouter Wijngaards. '\*' in Rdata causes the return code to be NOERROR instead of NX.  
<https://github.com/NLnetLabs/nsd/issues/152>.

- [65] Siva Kesava R Kakarla and Wouter Wijngaards. DNAME not applied more than once to resolve the query.  
<https://github.com/NLnetLabs/nsd/issues/151>.
- [66] Siva Kesava R Kakarla and Wouter Wijngaards. DNAME: synthesized CNAME might be perfect answer to CNAME query.  
<https://github.com/NLnetLabs/nsd/issues/140>.
- [67] Siva Kesava R Kakarla and Wouter Wijngaards. NS Records below delegation are not ignored (nsd-checkzone also does not raise any issue).  
<https://github.com/NLnetLabs/nsd/issues/174>.
- [68] Siva Kesava R Kakarla and yadifa. Records below delegation are not ignored.  
<https://github.com/yadifa/yadifa/issues/12>.
- [69] Siva Kesava R Kakarla, yadifa, and edfeu. Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR.  
<https://github.com/yadifa/yadifa/issues/11>.
- [70] Siva Kesava R Kakarla, yadifa, and edfeu. Why are CNAME chains not followed?  
<https://github.com/yadifa/yadifa/issues/10>.
- [71] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 310–328, New York, NY, USA, 2020. Association for Computing Machinery.
- [72] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
- [73] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammadika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
- [74] Eduard Kovacs. Dns servers crash due to bind security flaw.  
<https://www.securityweek.com/dns-servers-crash-due-bind-security-flaw>, 2018.
- [75] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open dns resolvers. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 355–368, New York, NY, USA, 2015. Association for Computing Machinery.
- [76] NLnet Labs. Nsd.  
<https://nlnetlabs.nl/projects/nsd/about/>. Code commit used: <https://github.com/NLnetLabs/nsd/tree/4043a5ab7be7abaec969011e48e4d0d60a0056a6>.
- [77] NLnet Labs. nsd-checkzone - nsd zone file syntax checker.  
<https://www.nlnetlabs.nl/documentation/nsd/nsd-checkzone/>.
- [78] Hyojeong Lee, Jeff Seibert, Dylan Fistovic, Charles Killian, and Cristina Nita-Rotaru. Gatling: Automatic performance attack discovery in large-scale distributed systems. *ACM Trans. Inf. Syst. Secur.*, 17(4), April 2015.
- [79] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris Sud, 2004.
- [80] Edward P. Lewis. The Role of Wildcards in the Domain Name System. RFC 4592, July 2006.
- [81] Kenneth L. McMillan and Lenore D. Zuck. Compositional testing of internet protocols. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 161–174, 2019.
- [82] Kenneth L. McMillan and Lenore D. Zuck. Formal specification and testing of quic. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 227–240, New York, NY, USA, 2019. Association for Computing Machinery.
- [83] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239):2, March 2014.
- [84] Microsoft. Github, inc.  
<https://github.com/>.
- [85] Microsoft. Microsoft dns.  
[https://en.wikipedia.org/wiki/Microsoft\\_DNS](https://en.wikipedia.org/wiki/Microsoft_DNS).
- [86] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, November 1987.
- [87] Paul Mockapetris. Domain names - implementation and specification. RFC 1035, November 1987.

- [88] B. Neelakantan and S. V. Raghavan. *Protocol Conformance Testing — A Survey*, pages 175–191. Springer US, Boston, MA, 1995.
- [89] NMAP Organization. Dns-fuzz. <https://nmap.org/nsedoc/scripts/dns-fuzz.html>.
- [90] Javier Paris and Thomas Arts. Automatic testing of tcp/ip implementations using quickcheck. In *Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG, ERLANG ’09*, page 83–92, New York, NY, USA, 2009. Association for Computing Machinery.
- [91] Libor Peltan and Daniel Salzman. DNAME: synthesized CNAME might be perfect answer to CNAME query. [https://gitlab.nic.cz/knot/knot-dns/-/merge\\_requests/1217](https://gitlab.nic.cz/knot/knot-dns/-/merge_requests/1217).
- [92] Libor Peltans. Nsd and knot discussion. <https://github.com/NLnetLabs/nsd/issues/142#issuecomment-732753256>.
- [93] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., August 2015. USENIX Association.
- [94] Fahmida Y. Rashid. Isc updates critical dos bug in bind dns software. <https://www.infoworld.com/article/3126472/isc-updates-critical-dos-bug-in-bind-dns-software.html>, 2016.
- [95] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model finding for recursive functions in smt. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, pages 133–151, Cham, 2016. Springer International Publishing.
- [96] Scott Rose and Wouter Wijngaards. DNAME Redirection in the DNS. RFC 6672, June 2012.
- [97] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashoq Muhaimen, and Ramesh K. Sitaraman. Akamai dns: Providing authoritative answers to the world’s queries. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM ’20*, page 465–478, New York, NY, USA, 2020. Association for Computing Machinery.
- [98] Harsha Sharma, Wenfei Wu, and Bangwen Deng. Symbolic execution for network functions with time-driven logic. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.
- [99] Robert Swiecki and et al. Honggfuzz - security oriented software fuzzer. <https://github.com/google/honggfuzz/tree/master/examples/bind>.
- [100] Dmitriy Zaporozhets Sytse "Sid" Sijbrandij. Gitlab, inc. <https://gitlab.com/>.
- [101] Peach Tech. Peach fuzzer platform. [peach.tech/products/peach-fuzzer/](https://peach.tech/products/peach-fuzzer/) [peach.tech/products/peach-fuzzer/](https://peach.tech/products/peach-fuzzer/).
- [102] Sam Trenholme. Maradns. <https://maradns.samiam.org/>. Code commit used: <https://github.com/samboy/MaraDNS/tree/3ec477f227b2bf6947be8fbe8fd0ab73130227d0>.
- [103] Liam Tung. Azure global outage: Our dns update mangled domain records, says microsoft. <https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/>, 2019.
- [104] Margus Veane, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, pages 39–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [105] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM ’21*, page 272–286, New York, NY, USA, 2021. Association for Computing Machinery.
- [106] Dan York. Hbo now dnssec misconfiguration makes site unavailable from comcast networks (fixed now). <https://www.internetsociety.org/blog/2015/03/hbo-now-dnssec-misconfiguration-makes-site-unavailable-from-comcast-networks-fixed-now/>.



# Decentralized cloud wide-area network traffic engineering with BLASTSHIELD

Umesh Krishnaswamy

Rachee Singh

Nikolaj Bjørner

Himanshu Raj

Microsoft

## Abstract

Cloud networks are increasingly managed by centralized software defined controllers. Centralized traffic engineering controllers achieve higher network throughput than decentralized implementations, but are a single point of failure in the network. Large scale networks require controllers with isolated fault domains to contain the *blast radius* of faults. In this work, we present BLASTSHIELD, Microsoft’s software-defined decentralized WAN traffic engineering system. BLASTSHIELD *slices* the WAN into smaller fault domains, each managed by its own slice controller. Slice controllers independently engineer traffic in their slices to maximize global network throughput without relying on hierarchical or central coordination. BLASTSHIELD is fully deployed in Microsoft’s WAN and carries a majority of the backbone traffic. BLASTSHIELD achieves similar network throughput as the previous generation centralized controller and reduces traffic loss from controller failures by 60%.

## 1 Introduction

Cloud wide-area networks (WANs) enable low-latency and high bandwidth cloud applications like live-video, geo-replication, and other business critical workloads. Cloud WANs are billion-dollar assets, and annually cost a hundred million dollars to maintain. To efficiently utilize their infrastructure investment, cloud providers employ centralized, software-defined traffic engineering (TE) systems. Centralized TE leverages global views of the topology and demands to maximize the network throughput.

**Maximum throughput, but at what cost?** The paradigm shift in WAN TE from fully decentralized switch-native protocols (*e.g.*, RSVP-TE [4]) to centralized TE controllers was driven by the throughput gains made possible by centralization [16]. After a decade of operating the software-defined WAN (SWAN) in Microsoft’s backbone network, we claim that it is more important that the centralized TE controller does not become a single point of failure in the system. The impact of a TE controller fault needs to be lowered along with achieving high throughput.

**Controller replication does not guarantee availability.** Our operational experience with SWAN has taught us that regardless of good engineering practices (*e.g.*, code reviews, safe deployment, testing and verification), software systems will fail

in production in unforeseen ways, often due to complex interactions of multiple faults. While it is hard to eliminate faults, it is crucial to contain the damage when faults inevitably occur. Despite fault-tolerant components of the SWAN TE system and replication of the centralized TE controller, an unforeseen cascade of faults led to an outage of global scope in the SWAN TE system.

In this work, we first describe the operational experiences that led us to migrate away from SWAN, the fully centralized TE system in the Microsoft cloud network (§ 2). Second, to reason about the availability of large-scale wide-area TE systems, we define *blast radius* of a TE controller as the fraction of customer or tier-0 traffic at risk due to its failure. We developed BLASTSHIELD, a WAN TE system that reduces the blast radius by slicing the global cloud WAN into smaller fault domains or *slices* (§ 3). BLASTSHIELD dials back from fully centralized to slice-decentralized TE by striking a balance between the centralized vs. distributed design principles.

BLASTSHIELD slices are independent, and do not rely on hierarchical or central coordination. Multiple WAN slices and controllers raise unique implementation challenges for BLASTSHIELD. In SWAN, a centralized controller with global view of the network, programmed TE routes in all WAN routers. In contrast, BLASTSHIELD slice controllers work independently — each with its own version of code, configuration, and view of the global network topology. Inconsistent views of the network topology can cause routing loops for inter-slice traffic in the cloud WAN. The failure of a slice controller on the path could blackhole traffic. BLASTSHIELD solves these challenges by developing a robust inter-slice routing mechanism that falls back on switch-native protocol routes in case of slice controller failures (§ 4 and § 5).

We have been operating Microsoft’s backbone with BLASTSHIELD since 2020. We find that BLASTSHIELD allows us to deploy changes to the network safely without the risk of global impact. While any change in network configuration or software is accompanied by risk, the ability to deploy changes without global risk is a significant advantage. Quantitatively, BLASTSHIELD reduces the risk of traffic loss due to failure of a TE controller by 60%, compared to SWAN (§ 6).

## 2 Background and Motivation

In this section, we describe an outage in the SWAN network that motivated the design of BLASTSHIELD. This outage was caused by a cascade of several independent failures and its

ripple effects persisted long after the root cause was resolved. The experience of resolving this incident urged us to survey the components at risk in SWAN and mechanisms to mitigate the risks. We define metrics to quantify the availability of TE controllers and design a TE system robust to global-scale outages like the one SWAN experienced.

## 2.1 Bad luck comes in threes

Prior to the development of BLASTSHIELD, a series of three unfortunate events occurred causing a SWAN outage of global scope. Global SWAN outages lasting more than a few minutes result in loss of several terabytes of network traffic, and are instantly observed by a global audience.

**Controller removes all routes.** A partially failed web request triggered the first bug that led the SWAN controller to remove all its TE routes from WAN routers. In the absence of controller routes, the traffic gets routed over shortest paths computed by the IGP [18]. This type of fallback is acceptable at a small scale, but not as a network-wide replacement.

**Incorrect IGP shortest paths.** Second, there were two links with misconfigured IGP link weights. The misconfiguration was inconsequential while the controller routes were present. When the controller removed its routes, these links incorrectly became a part of many shortest paths, consequently attracting more traffic than their capacity.

**Delayed controller response time.** An automatic recovery process could have restored the controller routes in 3 minutes, but a second controller bug incorrectly assumed that the recovering routers were undergoing maintenance, and held back programming routes on them. The longer recovery caused some internal workloads to dynamically change their traffic class to a higher tier, worsening the load and congestion in the network. The combination of these three cascading faults amplified the amount of traffic affected by the outage.

With the luxury of hindsight, we extract three key lessons from the SWAN incident:

1. **All changes have risk.** Global changes are antithetical to the availability of large-scale systems. We need an ability to gradually deploy changes, starting with staging which are production-like but without real customers, to low impact, and finally high impact regions. Global centralized TE precludes piece-wise rollout of changes.
2. **Configuration and software bugs are inevitable.** The outage occurred due to configuration and software bugs that escaped sandbox validation. While validation can be effective, it remains inherently best effort. In a nutshell, critical infrastructure like SWAN should not presume perfect pre-deployment validation.
3. **Global optimization does not preclude multiple controllers.** In the scenario, non-leader replicas of the controller had an accurate view of the network, and could have

optimized traffic correctly. By partitioning the scope of TE controllers, a faulty leader in one region of the WAN would not impact controllers in other regions.

## 2.2 Blast Radius, Ripple and Shielding

While faults and small-scale outages occur and get rectified rapidly in our network, what stood out about the SWAN outage incident was its global scope. We define the following terms to quantify the scope of wide-area traffic engineering outages. In later sections, we use these terms to evaluate the reduction in the scope of potential outages when we deploy the new TE system, BLASTSHIELD.

**Definition 1 (Blast Radius)** *is the fraction of customer or tier-0 traffic at risk by a TE controller failure.*

The service level objective (SLO) is the daily average of the hourly percentage of successfully transmitted bytes. Customer or tier-0 traffic has the highest SLO of 99.999%. Discretionary traffic tiers, tier-1 and tier-2, have a lower SLO of 99.9%. Half the traffic in our network is tier-0. The TE controller routes traffic on engineered paths to optimize for congestion, latency, and diversity. When a TE controller fails by withdrawing its routes or programming incorrect routes or stops programming the network, the ensuing tier-0 loss is the blast radius of the controller.

**Definition 2 (Blast Ripple)** *of a controller failure is the service level degradation experienced by components that are not governed by the failing TE controller.*

The blast or failure of a TE controller can cause *ripples* and impact traffic not managed by the failing controller. The impact of the ripple is proportional to the amount of tier-0 traffic affected that is not managed by the failing controller.

**Definition 3 (Blast Shielding)** *is the engineering practice that minimizes the blast radius of failing components while meeting operational constraints like cost and complexity.*

We note that blast shielding does not ensure that the overall system is fault tolerant in achieving the service level objective. Fault tolerance allows the system to operate even if its components fail [3]. Table 1 covers mitigation in Microsoft’s TE deployment to achieve fault tolerance and blast shielding. We highlight faults that were not addressed in SWAN’s original design and are a focus of this work with  $\blacktriangleright$ .

## 3 Slicing the cloud WAN

The global scope of the SWAN outage inspired the design of BLASTSHIELD, the WAN traffic engineering system that has replaced SWAN in Microsoft’s backbone network. BLASTSHIELD views the WAN as a collection of sites. Each site

Fault	Mitigation
Controller hardware, cluster, or site failure.	Automatic migration to geo-redundant cluster.
Network fault, <i>e.g.</i> , link failure, forwarding fault, router reboot.	Per-router agents perform local repair autonomously without controller intervention. Controller does global repair in the next TE iteration.
Network device disconnects or is unreachable by controller.	Router agents retain last programming. Controller reconnects via router management plane. Router is treated as down if failure persists. Rollback routes if disconnection is during new route programming.
Invalid, inconsistent, outdated programming by controller.	Router agents perform data plane verification. Controller programs agents with latest inputs every 3 minutes.
TE optimization failure <i>e.g.</i> , a controller withdraws its routes, or programs incorrect routes. $\blacklozenge$	Divide the WAN into subgraphs with a controller per subgraph managing a small fault domain.
Malicious router agent <i>e.g.</i> , agent stalls the controller from programming other routers. $\blacklozenge$	Decrease agent-controller interaction to defined subgraphs of the network.
Byzantine controller fault, <i>e.g.</i> , a controller sabotages other controllers. $\blacklozenge$	Controllers acquire network inputs independently.
Zero-day fault in multiple controllers. $\blacklozenge$	Diverge configurations in TE controllers.

Table 1: Fault types and their mitigation. New fault types handled by this paper are marked with  $\blacklozenge$ .

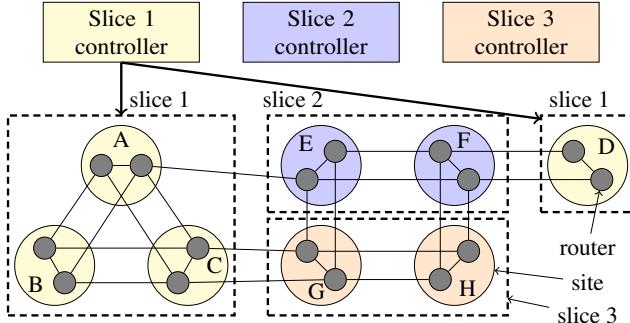


Figure 1: The WAN is divided into slices. Each slice is managed by a dedicated slice controller. Slice 1 consists of routers in sites A–D, slices 2 and 3 have routers in sites E–F and G–H.

consists of multiple WAN routers. WAN routers connect to other routers in the network like the datacenter fabric with a high bandwidth interconnect. WAN routers also transit traffic that is not from a directly connected datacenter. WAN sites at submarine landing terminals and optical transit sites do not have datacenters attached to them.

**WAN Slices.** BLASTSHIELD divides the WAN into *slices* or subgraphs of routers, each controlled by a dedicated slice controller. A slice is a logical partitioning of the WAN into disjoint sets of routers where each router belongs to exactly one slice. A slice can consist of a single router or all routers, or anything in between. Routers do not have any slice-specific configuration. In Fig. 1, slice 1 consists of routers in sites A–D. A slice can have multiple strongly connected components of routers. Slice 1 has two strongly connected components, the routers in sites A–C and D, respectively. Controllers 2 and 3 manage routers in sites E–F and G–H, respectively. The count and composition of slices is not limited by the design but dictated by operational choice.

**Enforcing slice isolation.** Only the slice’s owning controller

programs routers in the slice. All traffic from slice routers to any destination is engineered by the slice controller. This includes traffic that originates in datacenters directly connected to slice routers and the traffic originating in upstream slice routers. Each slice is a separate deployment and can be patched independently. Slices can inherit common configuration but BLASTSHIELD applies slice-specific configuration independently. Slice controllers do not communicate with another slice controller. This further isolates faults and prevents byzantine controllers bringing the entire system down. Slice controllers operate with a global view of the network by acquiring global topology and demand inputs. Each slice controller makes traffic engineering decisions based on expected conditions in local and remote slices. Controllers anticipate what other controllers do given the same inputs. While deviations between flow allocations computed by different controllers are possible, they are not disruptive to BLASTSHIELD’s operation.

**How many slices?** The number of BLASTSHIELD WAN slices decide the system’s operating point on an important tradeoff between network throughput and blast radius. A single slice enables the TE formulation to achieve maximum network throughput through centralization, but exposes the network to the risk of global blast radius. In contrast, several BLASTSHIELD slices reduce the blast radius of slice controllers but may also reduce the achievable network throughput. Additionally, several WAN slices increase the operational overhead of configuring and maintaining slice controllers. There is a sweet spot for the number of slices that limits the risk of changes and keeps operational overhead manageable. We empirically derive the number of BLASTSHIELD slices for Microsoft’s network and strike a balance between blast radius and network throughput (§ 6).

## 4 BLASTSHIELD System Design

In this section we present the design of BLASTSHIELD and describe the design choices that motivated our design.

### 4.1 System overview

Each BLASTSHIELD slice controller is a collection of four services: topology service, demand predictor, traffic engineering scheduler, and route programmer (Fig. 2). In addition to the controller services that run on off-router compute nodes, a router agent runs on all WAN routers.

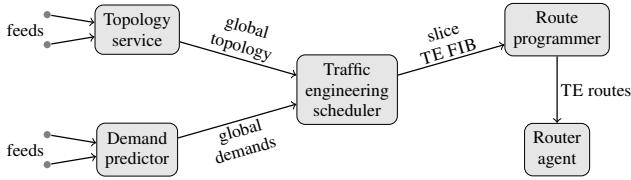


Figure 2: The slice controller consists of topology service, demand predictor, traffic engineering scheduler, and route programmer. Together, they compute traffic engineering routes and program slice routers through router agents.

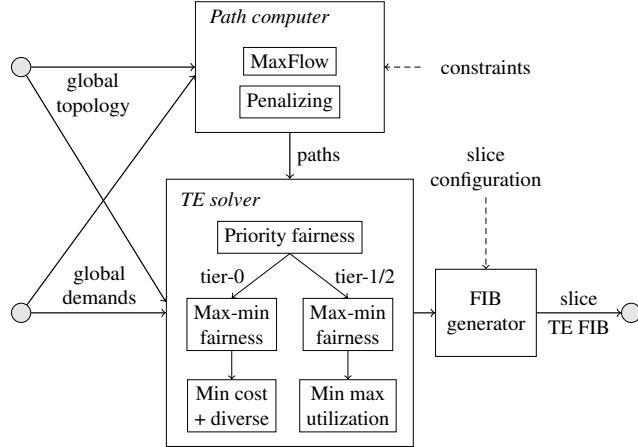


Figure 3: Traffic engineering scheduler computes routes that optimize paths for flows by traffic tier. Each controller performs global optimization based on its view of the entire network, but only programs routers belonging to its slice.

**Topology Service** synthesizes the global network topology using graph metadata, link state, and router agent input feeds. Graph metadata consists of routers, links, and sites. BGP-LS [15] is the primary source of dynamic link state information *e.g.*, link bandwidths, interface addresses, and segment identifiers [11]. The router agent feed is only used to acquire the health of the router agent; a router must have a functioning agent to be used for traffic engineering.

**Demand Predictor** predicts upcoming global network demands using a real-time traffic matrices measured by

sFlow [26] and host-level packet counters. Each network demand is identified by the tuple: source router, destination site, and traffic class. Traffic class is a differentiated service queue name *e.g.*, voice, interactive, best-effort, or scavenger [5]. Tier-0 traffic uses best-effort or higher traffic classes. Tier-1 and tier-2 use the scavenger traffic class. The data feeds of the demand predictor are independently scaled out and not part of the controller.

**Traffic Engineering Scheduler** forms the core of the BLASTSHIELD system (Fig. 3). It ingests global network topology and global demands from topology service and demand predictor respectively. The path computer calculates paths using the dynamic topology for the source-destination pairs in the global demands. MaxFlow path computer uses maximum flow algorithms [14], and penalizing path computer computes risk diverse shortest paths using Dijkstra. Path *constraints*, described later in §§ 5.1 and 5.2, limit allowed paths in order to support the routing in BLASTSHIELD.

TE solver consists of a chain of linear programming optimization steps that place demands on multiple paths with unequal weights between demand source and destination pairs. It places tier-0 demands on paths with diversity protection that minimize latency subject to approximate max-min fairness. Lower priority demands in tier-1 and tier-2 classes are placed on paths that minimize the maximum link utilization. For brevity, we exclude the optimization problem formulations, which are previously described in [6, 16, 21, 25].

The FIB generator mechanically converts the output of the TE solver, called the *solver result*, into TE routes. The *slice configuration* specifies the subset of routers for which routes are generated. The FIB generator transforms the solver result based on the slice configuration, and produces routes only for the routers in the slice. The network is re-optimized every 3 minutes, or on topology change, whichever occurs first.

**Route Programmer** programs traffic engineering routes in the router agent which in turn installs them in the router. It periodically receives the full set of routes for all slice routers from the traffic engineering scheduler. This is called the traffic engineering forwarding information base (TE FIB). The FIB is organized into per-router flow and group tables (see Fig. 4). The route programmer updates all slice router agents in parallel using an update procedure, called *make-before-break*. The principle is to make all new traffic engineered paths before placing traffic on them. Intermediate FIBs build new paths, transfer traffic to the new paths, and tear down unused paths.

**Router Agent** runs on all WAN routers. It installs TE routes, monitors the end-to-end liveness of TE paths (*tunnels*), and modifies ingress routes based on liveness information. Route installation on the router requires translating the FIB into router platform-specific API calls. Router agents have a platform-dependent module to handle this translation. The router agent verifies tunnels within the slice using probes generated natively or with BFD [22] from tunnel ingress points.

Flows are unequally hashed to live paths based on the path weight, flow 5-tuple, and traffic class. If a path goes down, the agent proportionally distributes the weight of the down path to remaining up paths. If no path is up, then the ingress route is withdrawn, and packets are forwarded using switch-native protocol routes. This is called *local repair*.

## 4.2 Design considerations

**Global solution at local instances.** Each BLASTSHIELD slice controller consumes global network topology and demands. The solver of each controller computes flow allocations for the entire network. Therefore, each slice controller produces the same solver result if its inputs and solver software versions are the same. In practice, inputs and software versions can differ, and we study the impact of these differences in § 6.2. Although a slice controller only programs the WAN routers in its slice, it optimizes flow with a global view. Slice controllers do not communicate with each other but gather inputs from the network. Performing global optimization at each slice controller is beneficial while deploying changes to the network. Some faults involve complex interactions that only occur in unique parts of the WAN. Global inputs increase the coverage of code paths while new software or configuration changes are being deployed in small blast radius slices.

**Slices as isolated routing domains.** In centralized TE systems, a single controller is responsible for programming all WAN routers with the TE routes. BLASTSHIELD replaces the centralized controller with multiple slice controllers that can only program the routers within their slice. By preventing slice controllers from programming routers outside their slice, we enforce fault isolation between slices. In addition, the routing mechanisms described in § 5 ensure that the failure of one controller does not impede other controllers *e.g.*, the failure of a downstream slice controller on an inter-slice route in the WAN does not lead to blackholing of traffic. Similarly, slice controllers with inconsistent views of the network, route packets to their destination without centralized control.

**Fault tolerant design.** All services run on multiple machines in at least two geographically separate clusters. Topology service instances are fully active, but elect a leader to avoid oscillations if two instances report different topologies due to faults or transients. The traffic engineering scheduler and route programmer elect leaders, and switchover in case of failure. The route programmer handles all the faults and inconsistencies that can happen during programming, *e.g.*, router agents are unresponsive or have faults before, during, or after route programming. Reliable controller-agent communication is achieved by using network control traffic class, and redundant data and management plane connections. The router agent can react to network faults even when it is disconnected from the router programmer.

**Decoupling TE scalability from blast shielding.** BLASTSHIELD employs slice controllers to reduce the blast radius of faults in our network. We handle scale along several dimensions, unrelated to blast shielding. But slices also provide the following scaling benefits. The total number of tunnels in the network decreases because an inter-slice path is a sequence of intra-slice tunnels in BLASTSHIELD, whereas in SWAN it required its own tunnel. Second, shorter tunnels decrease tunnel probe round-trip times and speed up local repair.

## 5 Routing and forwarding in BLASTSHIELD

The routing of *intra-slice* flows in BLASTSHIELD is the same as SWAN. In this section, we describe BLASTSHIELD’s extensions to enable routing and forwarding of *inter-slice* flows *i.e.*, flows whose traffic engineered paths span multiple slices. § 5.1 describes inter-slice routing, the approach we deployed, and § 5.2 describes a source routing approach that was evaluated but not deployed.

### 5.1 Inter-slice routing

In SWAN, packets are routed using a combination of switch-native protocols and the TE controller. WAN routers connected to the datacenter fabric advertise datacenter routes with themselves as the BGP [27] next hop. BGP receivers recursively lookup the route for this BGP next hop and find multiple available routes: the shortest path route computed by the IGP, or the route programmed by the TE controller which leverages traffic engineered paths. TE routes have higher precedence than the IGP routes. The TE route encapsulates packets using Multiprotocol Label Switching (MPLS) [28] path labels from a label range reserved for the TE controller.

BLASTSHIELD routes inter-slice flows *i.e.*, flows whose traffic engineered paths span multiple slices, using *slice-local encapsulation* till the slice boundary. Slice controllers add encapsulation headers while the packet is within the slice but ensure that the packets arrive at the next slice in their *native encapsulation* *i.e.*, the encapsulation in which the packets entered the WAN. Each slice controller is only responsible for routing traffic to the ingress router of the next slice. Packets are encapsulated with an MPLS path label at the time of BGP route lookup on the WAN ingress router or the intermediate slice ingress routers. In both scenarios, transit routers forward the packet using the MPLS path label, and the label is popped by the penultimate router — either at a slice boundary or at the destination. Intra-slice traffic is split across TE paths only once at the WAN ingress router. Inter-slice traffic can also be split at the ingress router of an intermediate slice.

**Inter-slice forwarding** In Fig. 4, all four slice controllers determine that the demand from  $a$  to  $z$  should be placed on paths  $abegjuwxz$ ,  $acdmoqstyz$ , and  $acdmnikvzy$  with weights 0.3, 0.42, and 0.28 respectively. Slice 1 programs  $abe$  with weight

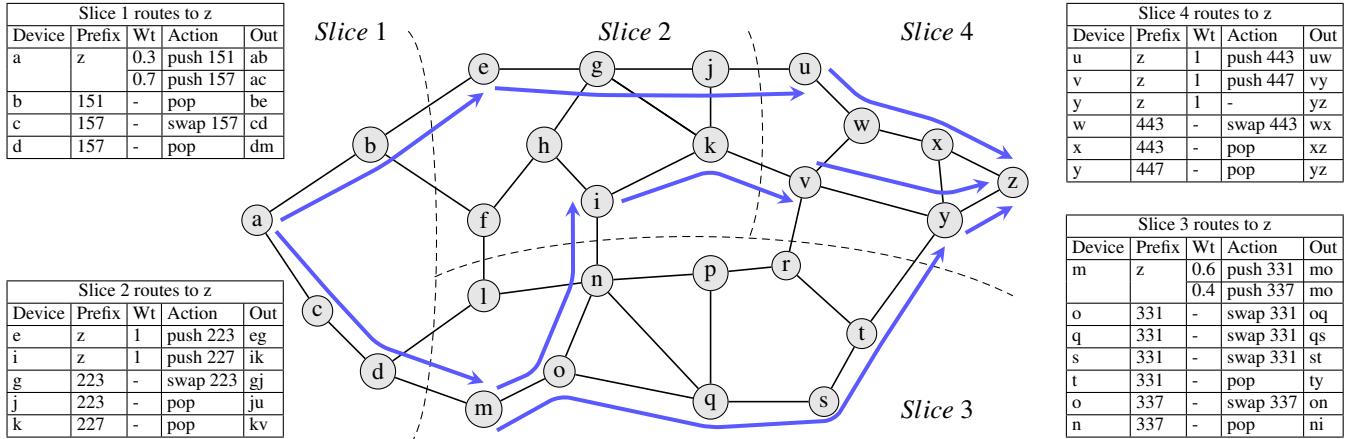


Figure 4: Inter-slice routing using an example router-level network graph divided into four slices. The tables represent TE FIBs programmed by slice controllers using inter-slice routing. Each slice controller programs the path segment within its slice. For the path *abegjuwzxz*, slice 1 programs *abe*, slice 2 programs *egju*, and slice 3 programs *uwxz*. Traffic arriving at slice ingress routers get encapsulated and split over different paths. Transit routers guide the packet along the path specified by the MPLS label. Packets return to native encapsulation at the next slice and the WAN exit.

0.3, and *acd*<sub>m</sub> with weight 0.7. Slice 2 programs *egju* and *ikv*. Slice 3 programs *moqsty* with weight 0.6, and *moni* with weight 0.4, and slice 4 programs *uwxz*, *yz*, and *yz*. Controllers only need to install routes in their slice routers.

If any downstream slice controller fails to program routes to the destination, packets are forwarded using protocol routes along the shortest paths to the destination. Since we enable segment routing [11] with the IGP, the IGP route changes the packet encapsulation and routes the packet to the destination. For example, if the slice 2 controller withdraws all routes due to a failure, the inter-slice traffic uses shortest paths to the destination, *z*. This is the blast ripple of a down controller. In § 6.1, we will discuss how to define slice boundaries to decrease the blast ripple. Downstream slice controllers may have slightly inconsistent views due to network events like link flaps. Inter-slice traffic will be forwarded on shortest paths while the controllers converge. We show results on the alignment of multiple controllers in § 6.2.

**Preventing routing loops.** Unlike the TE controller in SWAN, a BLASTSHIELD slice controller is only responsible for routing packets within the slice and not until the packets' destination. Since each slice is its own routing domain, inconsistent views of the global network graph in different slice controllers can lead to routing loops.

BLASTSHIELD avoids routing loops by enforcing *enter-leave constraints* on inter-slice next hops. These constraints define the set of inter-slice next hops for all source-destination pairs in the network. The constraints ensure loop-free paths and are calculated offline using a static network graph. The path computer calculates paths on the dynamic network graph, and only allow paths that satisfy the enter-leave constraints. However, enter-leave constraints should not be overly restrictive. For example, a potential approach to preventing routing loops can limit inter-slice next hops to be on the minimum

spanning tree from the source router to the destination. But this approach restricts inter-slice paths to go through a few links and causes bottlenecks.

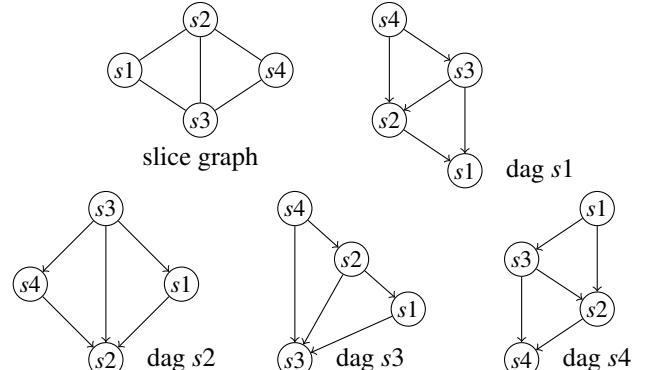


Figure 5: Enter-leave constraints restrict paths to achieve loop-free routing. Slice graph is a component level graph of Fig. 4. Slice DAGs are constructed from shortest path distances in the slice graph. Router-level paths must follow DAG edges when crossing slice boundaries. Path *acdmonikvyz* is allowed for TE because *s1* → *s3* → *s2* → *s4* is a path in DAG *s4*. Path *abfhinprvyz* is not allowed for TE because *s2* → *s3* is not present in DAG *s4*.

**Computing enter-leave constraints.** An offline generator computes enter-leave constraints from the static router-level network graph to prevent inter-slice routing loops. It first constructs a slice graph from the network graph, where each slice node represents a strongly connected component (SCC) after removing all inter-slice links. Figure 5 is the slice graph of Fig. 4, formed by removing inter-slice links *be*, *bf*, *dl*, *dm*, *fl*, *in*, *ju*, *kv*, *rv*, and *ty*, and calculating SCCs. A slice can contribute one or more SCCs as nodes to the slice graph. A link between the slice graph nodes aggregates all links between SCCs in the network graph. Link weights in the slice graph are computed from link weights in the network graph.

The enter-leave constraint generator then constructs per-destination slice DAGs based on the shortest path distances in the slice graph. The enter-leave constraints come out directly from the slice DAGs. In Fig. 5, the slice DAG for  $s4$  says that paths from any node in  $s1$  to any node in  $s4$  can only have inter-slice transitions:  $s1 \rightarrow s2 \rightarrow s4$ ,  $s1 \rightarrow s3 \rightarrow s4$ , and  $s1 \rightarrow s3 \rightarrow s2 \rightarrow s4$ . No controller, no matter its topology, can use any other inter-slice transition.

The path computer blacklists edges excluded by enter-leave constraints in the dynamic network graph before computing TE paths. Since the slice DAG is loop-free, paths computed by any slice controller are also loop-free. This ensures that even if slice controllers have inconsistent views of the dynamic network graph, they will arrive at loop free routes. Enter-leave constraints place restrictions on TE paths, and reduce the number of paths available to place demands. We evaluate the percentage of allowed paths vs. computed paths without constraints in § 6.1.

**Verifying enter-leave constraints.** Due to the negative impact of routing loops in production, and because they are global configuration, enter-leave constraints are verified offline before deployment. Enter-leave constraints are updated when there are newly provisioned routers or inter-slice links in the network. They do not need to be updated for newly provisioned intra-slice links.

We use the following formalism to define correct inter-slice routing. Let  $\mathcal{R}$  be the set of defined route keys, where route key is a tuple of (router, destination prefix), **end** be the terminating route key, **null** be the undefined route key, and *ttl* be the packet time to live. Let  $f : \mathcal{R} \rightarrow \mathcal{R}$ , where  $f(\mathbf{null}) = \mathbf{null}$ ,  $f(\mathbf{end}) = \mathbf{end}$ . Routing is a repeated application of  $f()$ , till  $f^n(x) = \mathbf{end}$  where  $n$  ranges over  $1 \leq n \leq \text{ttl}$ . The collection of TE, BGP, and the IGP routes, and their union are examples of routing functions. The routing function is complete, loops, or blackholes, if:

$$\begin{aligned} \forall x, \exists n : f^n(x) &= \mathbf{end} && (\text{complete}) \\ \exists x, n : f^n(x) &= x && (\text{routing loop}) \\ \exists x, n : f^n(x) &= \mathbf{null} && (\text{blackhole}) \end{aligned}$$

where  $x$  ranges over  $\mathcal{R} \setminus \{\mathbf{end}, \mathbf{null}\}$  and  $n$  ranges over  $[1.. \text{ttl}]$ . Enter-leave constraints are verified using this formalism to detect routing loops.

## 5.2 Why not source routing?

In this section, we describe an alternate approach that leverages the capabilities of segment routing (SR) [11], and why we did not adopt this approach.

**Loose source routing with SR.** SR is a source-based routing technique that allows senders to specify the packets' route through the network by leveraging the MPLS forwarding plane. An SR router subjects arriving packets to a policy and encapsulates the matching packets in an MPLS label stack, each label represents a *segment* in the SR-path. A *node segment* causes the packet to be routed on least-cost paths computed

by the IGP to the router identified by the node segment. An *adjacency segment* causes the packet to use a specified link for its next hop.

An IGP path computer models the modified Dijkstra shortest path first algorithm [18]. Coupled with segment identifiers from topology service (§ 4.1), it implements *loose source routing*. In place of explicitly listing adjacency segments of hop-by-hop links of a path, loose source routing uses a node segment when it exactly represents the sequence of the hop-by-hop links of the path. Figure 6 shows an example of loose source routing for the same paths shown in Fig. 4. The path *begjuwxz* is composed of two shortest path segments *begju* and *uwxz*. Hence *a* encapsulates with label stack of  $[n(u) n(z)]$  to route to *z*, where  $n()$  is the node segment identifier of a router.

**Packet encapsulations reduce hashing entropy.** To achieve balanced utilizations across links in the WAN, the cloud network employs two load balancing mechanisms. Link aggregation group hashing sprays packets on member links of a port-channel. Equal cost multi-path hashing sprays packets on the next hops of a group of traffic engineering routes. The packet processor uses fields from the packet headers to hash the packet to different output ports with the goal of maximizing entropy in the hash calculation. To achieve high entropy, the outermost IPv4/IPv6 source and destination addresses under stack of MPLS header encapsulations should be used to calculate the hash. A deep MPLS label stack can impair the ability of the packet processor to extract the relevant fields in the IP header.

The *depth limit* is the maximum number of MPLS encapsulations a packet can have while still allowing the packet processor to extract the header fields of the original (*i.e.*, prior to MPLS encapsulations) packet. The depth limit is switch platform-dependent [2, 8, 20]. We note that if the packets entering the WAN are already encapsulated in MPLS, the depth limit available to source routing is further reduced.

**Why select inter-slice routing?** Based on the current generation of platforms across different regions of our cloud WAN, the depth limit is four labels. Paths that require more labels cannot be used for TE. Figure 7 studies the label stack depth needed to encode paths computed by the path computer for current and future evolutions of the WAN. In source routing, 45% of computed paths can be used for TE. For comparison, 69% of computed paths can be used for TE in inter-slice routing (see § 6.1).

Second, in source routing, a downstream slice can only transit upstream flows. In inter-slice routing, the downstream slice is free to rebalance the traffic to correct errors made upstream or mitigate for local slice conditions. This kind of control is not available with source routing.

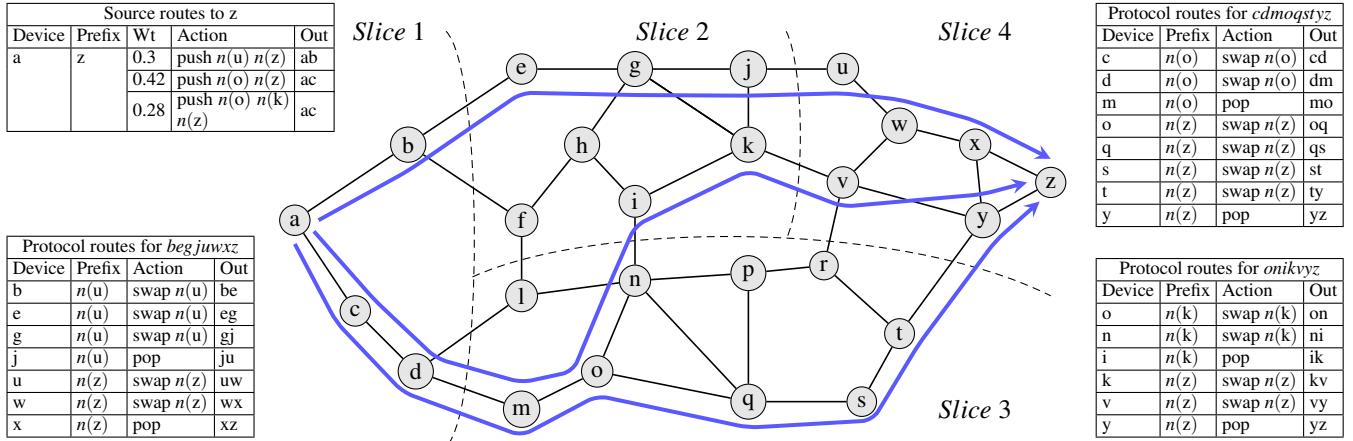


Figure 6: Source routing. Slice 1 controller programs ingress routes to z using loose source routing. The IGP with segment routing takes care of transit routes. The path *begjuwxz* is composed of two shortest path segments *begju* and *uwxz*. Hence the label stack for the path is  $n(u)$   $n(z)$ , where  $n()$  is the node segment identifier of a router. Weights of intra-slice links are 1 and inter-slice links are 5.

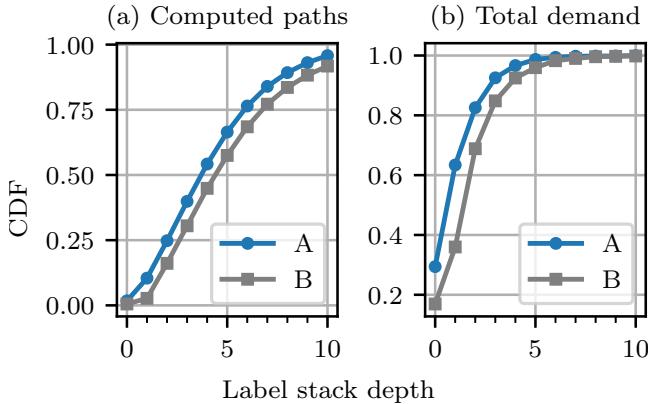


Figure 7: Cumulative distribution function of (a) computed paths and (b) total demand, by label stack depth for inputs A and B of increasing sizes. If depth limit is four, 45% of computed paths and 93% of the total demand map to allowed paths for input B.

## 6 Evaluating BLASTSHIELD in production

The incremental deployment of BLASTSHIELD began in 2020 and today BLASTSHIELD has replaced the legacy SWAN traffic engineering system in Microsoft’s cloud network. In § 6.1, we evaluate the benefits and costs of WAN slicing using demands and topology inputs from the Microsoft backbone network for the month of July 2021. The benefit of slice-decentralized traffic engineering is the reduction in traffic loss from a slice failure. Its cost is the reduction in TE throughput due to enter-leave constraints. We quantify cost and benefit as we incrementally divide the global network into ten slices. In § 6.2, we evaluate the stochastic effects caused by multiple and independent BLASTSHIELD controllers. We show that despite the controllers having different configurations, software versions and network topology snapshots, they arrive at nearly similar flow allocations.

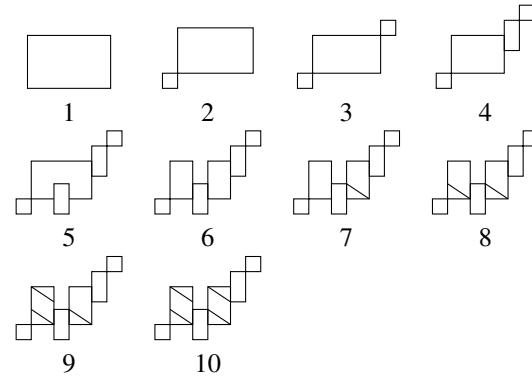


Table 2: Ten slice configurations of the global cloud network. In (1) the entire network is one slice. Slices 2–6 are formed by grouping routers in geographies. Slices 7–10 are created by further subdividing the two largest geographies, Europe and North America.

### 6.1 Availability vs. throughput trade-offs

We incrementally carve out slices from the global cloud network as shown in Table 2. We consider ten different slicing configurations with increasing number of slices from 1 to 10. Slice configuration 1 represents centralized traffic engineering as in SWAN. Slice configurations 2–6 are formed by drawing slice boundaries around large geographical regions like APAC, EMEA, India, North America, Oceania, and South America. In Table 2, slice configuration 2 represents the network divided into two slices: India and the rest of the world, configuration 3 represents India, Oceania, and the rest of the world, and so on. Slices 7–10 are formed by additionally dividing the two largest geographies, Europe and North America, into smaller slices. In our network, configurations 1–6 tend to have higher intra-slice traffic in comparison to inter-slice traffic. Slices have up to three strongly connected components, arising from disconnected sites and router planes.

**Availability gains from decentralized TE.** The key benefit

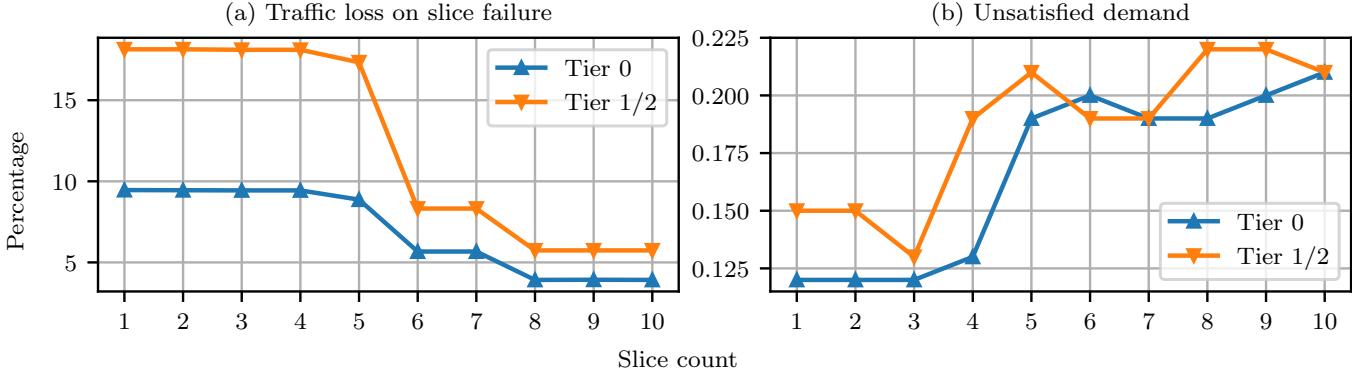


Figure 8: Benefit of BLASTSHIELD compared to its cost as a function of slice count: (a) Traffic loss from worst case single slice failure as a percentage of requested demand, (b) Unsatisfied demand due to enter-leave constraints as a percentage of requested demand.

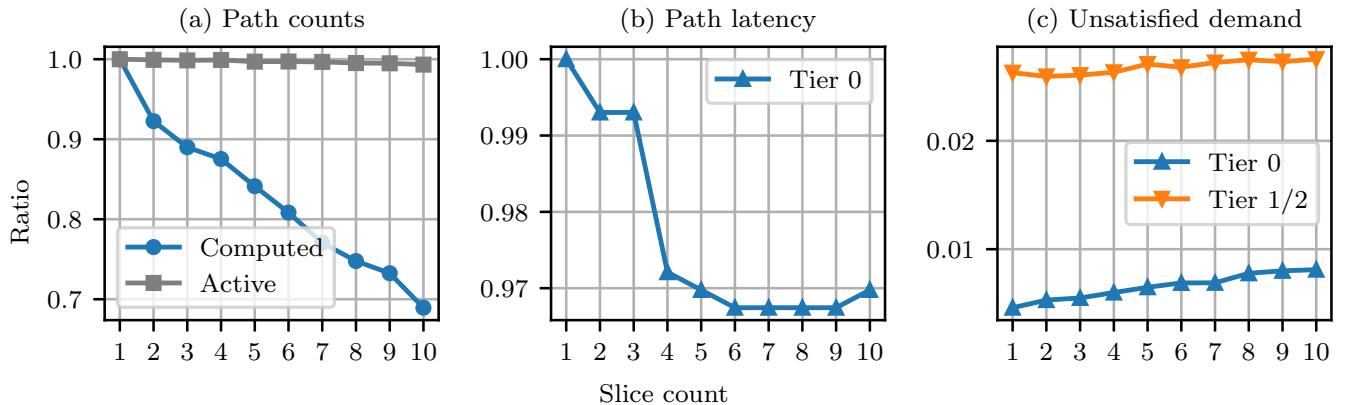


Figure 9: Stress-testing BLASTSHIELD with worst-case failures and 2 $\times$  demands. (a) Computed paths are the count of paths computed with enter-leave constraints. Active paths are the count of paths used for traffic engineering. (b) Path latency is the traffic weighted average latency of active paths for tier-0 demands. (c) Unsatisfied demand is the unallocated demand per traffic tier. All values, except unsatisfied demand, are normalized to corresponding values for one slice; the latter is a ratio of unsatisfied to requested demand.

of BLASTSHIELD’s slicing is the reduction in blast radius when a slice controller fails. We consider the failure where the slice controller removes all programmed TE routes. This causes the traffic to fall back on protocol routes and the ensuing traffic loss is the impact of the slice failure. We measure the traffic loss using a network simulator because the scenarios we are testing cannot be replicated in production. The inputs to the simulation are the production network demands, topology, TE and the IGP routes, and the network simulator models routing, forwarding, and queuing behavior. The simulator is used internally for capacity planning and operational safety checks, and hence is a well-tested proxy for the production network.

Figure 8 (a) shows the impact of the worst-case single slice failure when BLASTSHIELD is operating with 1–10 slices. We keep the demands and topology fixed in this experiment. For each slice configuration, we fail the largest slice by demand. The network uses the IGP routes of the failed slice and TE routes of the remaining slices (if any) to allocate the remaining demands. The traffic losses are caused by congestion due to shortest path routing over IGP routes. There are no losses due to traffic blackholes or routing loops. Figure 8 (a) shows that with ten slices, tier-0 traffic loss due to slice failure, which is

the metric for blast radius, decreases by 60%, from 9.5% to 3.9%. Tier-1 and tier-2 traffic loss reduction is greater at 70% (18.1% to 5.7%) because they map to scavenger traffic class and experience more congestion losses when the failed slice uses IGP routes. Slices 2–4 show little improvement because the largest slice can still cause an overly large failure. The improvements come at six and eight slices with the breakup of Europe and North America into separate and smaller slices.

**Throughput cost of decentralized TE.** The key reason why inter-slice routing in BLASTSHIELD can have lower throughput than SWAN is due to the enter-leave constraints (§ 5.1). These constraints decrease the choice of paths available for placing demands, which in turn decreases the demands that can be allocated. Figure 8 (b) shows unsatisfied demand from enter-leave constraints as a percentage of requested demand. We calculate worst case unallocated demand from 20 variations of the network topology, each variation has multiple shared risk failures that reduce the available capacity. Without constraints, the worst-case unsatisfied demand is 0.27% of the requested demand, and with ten slices it increases to 0.42%. The increase in unsatisfied demand of 0.15% is much smaller than the 18% traffic loss reduction from slice failure. Addi-

tional capacity can be provisioned to decrease the unsatisfied demand.

**Stress testing BLASTSHIELD.** We oversubscribe the network by doubling the bandwidth values in requested demands, and test with variations of the production network with multiple shared risk group failures in hot spots of the topology. The purpose of the stress test is to evaluate the performance of enter-leave constraints in the presence of significant over-subscription. Figure 9 shows the impact of slicing on paths computed by the BLASTSHIELD path computer. Since the constraints enforce a shortest path order when crossing slice boundaries, they exclude paths that would otherwise be allowed. At ten slices, computed paths decrease by 31% when compared with one slice. The number of paths actively used for carrying traffic decreases slightly — by < 1% due to some demands remaining completely unsatisfied, or diverse paths not getting found. Figure 9 shows that the traffic weighted path latency of tier-0 demands decreases by 3% because the computed paths are skewed towards shortest paths. Finally, unsatisfied demands as a percentage of requested demands increases 16% from 3.1% to 3.6%. Unsatisfied demand increases at half the rate of computed path decrease which is well controlled. In practice, the percentage of computed paths allowed by enter-leave constraints are used to determine whether a slice strategy is appropriate.

## 6.2 Stochastic effects of multiple controllers

Prior to the deployment of BLASTSHIELD, the centralized SWAN controller programmed new TE routes for the entire cloud network. BLASTSHIELD replaces the centralized controller with multiple slice controllers that snapshot the network topology and demands at different times. Moreover, the controllers may re-run the TE optimization and program their slice routers at different times. We study the impact of the temporally staggered operation of slice controllers to ask: can multiple slice controllers work harmoniously and not be discordant?

We reserve 15% scratch capacity in order to support the high SLO of tier-0 traffic. Transient traffic bursts and hashing polarization can cause link utilization to differ significantly from expected values. The scratch capacity is used to avoid congestion losses in these conditions. BLASTSHIELD uses this scratch capacity to deal with differences that arise with multiple controllers.

**Symphony or cacophony of controllers?** Path weights decide the split of traffic across paths and are the ultimate result of TE optimization. The weight of a path is the fraction of demand placed on it. BLASTSHIELD programs the newly computed path weights every 3 minutes. Since all slice controllers solve the TE problem for the entire network, we measure if the path weights that different controllers compute diverge from each other. We quantify the *path weight difference* as

the root mean squared error between path weight time series from two controllers. A path weight difference of zero implies that the controllers are perfectly aligned. Non-zero path weight difference implies that the controllers are setting aside different link bandwidths for a flow which can cause congestion.

We measure the path weight difference between six different BLASTSHIELD controller pairs in the production network over a 30-day period. There were days when the controllers were operating with different configurations, different software versions, in addition to network topology and demand changes that happen throughout the day. Figure 10 shows that only 2% of paths and 3% of total demand have path weight difference of  $\geq 0.15$ . Inter-slice demands make up 48% of paths but 10% of total demand because of the slicing strategy. Since intra-slice traffic dominates, the impact of the path weight difference is limited. The slicing strategy and scratch capacity allow multiple controllers to operate without coordination.

**Solver stability.** Different path weights for *slightly perturbed* inputs can create an operational challenge for BLASTSHIELD. We constrain the solver models to make their solutions stable — the tier-0 objective function minimizes demand weighted latency after solving for max-min fairness. In practice, this makes the solver results more stable when subjected to input perturbations. We do not allow non-determinism in the TE solver *e.g.*, no parallel primal and dual simplex invocation in the linear programming solver to pick the first result, since they will produce different solution vectors that result in different path weights.

We evaluate the stability of the solver results using the normalized autocorrelation function (ACF)  $\rho(\tau)$ . ACF is the correlation of a time series to a delayed version of itself, as a function of the delay,  $\tau$ . In Fig. 11, we calculate ACF for the hour-long path weight time series of all paths in the production network over a 24-hour period. ACF values range  $[-1, 1]$ , and 1 implies perfect correlation.

Demand and network topology changes also affect path weight ACF. So perfect correlation is not possible in an operational network. Figure 11 (a) is an example path weight time series with ACF(30 minutes) of 0.65 showing steady values of the same path weight interspersed by occasional gyrations. Figure 11 (b) shows that mean ACF is 0.75–0.63 for lags of 3–30 minutes. This reaffirms the data in Fig. 10 that path weights from independent BLASTSHIELD controllers are predominantly the same.

## 7 Discussion

In this section, we discuss our operational experience with BLASTSHIELD and describe safe deployment of software and configuration in BLASTSHIELD slices. We consider the implications of byzantine slice controllers, and the safeguards in place to prevent damage from them.

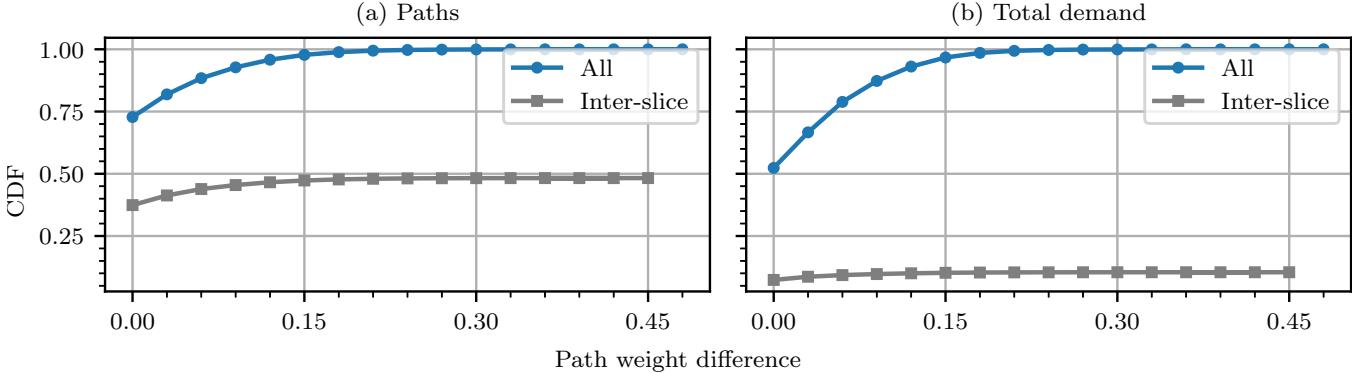


Figure 10: Cumulative distribution function of (a) paths and (b) total demand, by path weight difference, for all demands and inter-slice demands, measured for six controller pairs in the production network over a 30-day period. 98% of paths and 97% of total demand have path weight difference  $\leq 0.15$ . Inter-slice demands make up 48% of paths but 10% of total demand.

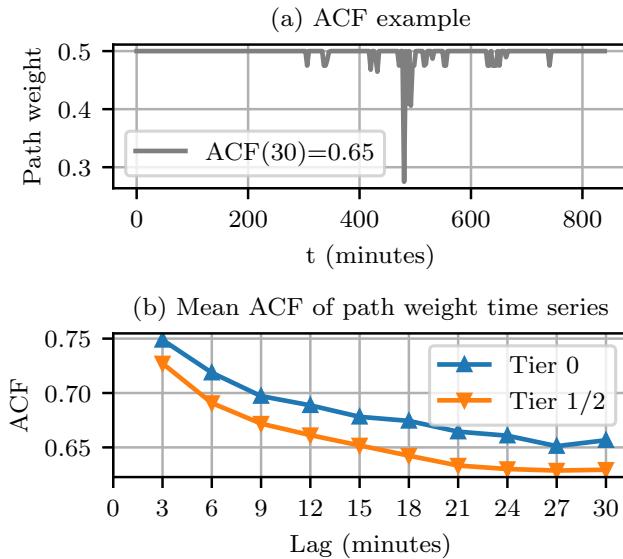


Figure 11: Autocorrelation function (ACF) measures self-similarity with a delayed version, and ranges  $[-1, 1]$  with 1 being perfect correlation. (a) Example path weight time series with ACF(30 minutes) of 0.65. (b) Mean ACF of path weight time series averaged over all paths in the production network over a 24-hour period by traffic tier.

## 7.1 Operational experience

BLASTSHIELD has been in operation for two years. Migration from SWAN to BLASTSHIELD was carried out over a number of months. The first step was to deploy inter-slice routing and forwarding functionality in the SWAN controller and router agents. This was the riskiest step and preceded by many months of testing in a virtualized emulation environment of the production network with fault injection. Each slice migration involved preparing a new BLASTSHIELD controller, excluding a set of routers from the slice configuration of the SWAN controller, and adding them to the new controller.

To support deployment of new software and configuration changes, we define slices that range from low to high impact. *Safe deployment* is a partial ordering of slices based on their blast radius. BLASTSHIELD has two staging sites with

a staging controller, and new software and configuration is first deployed here. The next slice has the smallest production scope. We assign routers in geo-redundant site pairs to separate slices for additional safety. Deployment progresses to the next slice in the sort order after a sufficient probationary period. The process continues till either all slices receive the new version of software or a failure happens in a slice, which may trigger a rollback of this version from all slices.

Enter-leave constraints have been updated multiple times to pick up newly provisioned routers and links. In one instance, the constraints affected traffic engineering for an inter-datacenter pair by excluding too many links. New constraint configuration to correct the error and reverse an inter-slice traffic flow was deployed without incident.

We have introduced new hardware platforms, router agents, and controller services that would be considered high risk in the SWAN paradigm. BLASTSHIELD allowed us to introduce new implementations in isolated slices with very small blast radius and no inter-slice traffic. Initially the slice only served intra-slice traffic. Inter-slice traffic was introduced after the slice had been in operation for many months. Outages caused by failures in the new slices never had a global impact.

Slice controller environments are used by additional services to decrease their blast radius. For example, discretionary flows can be throttled at the sending host to control congestion in the network. Bandwidth is allocated to discretionary flows by global optimization but each controller only serves bandwidth pools for a smaller fault domain.

## 7.2 Byzantine slice controllers

A byzantine controller is an unreliable controller that is disseminating false information or sabotaging the operation of other slices in the network [24]. A controller that only impacts its own traffic is not considered byzantine in our analysis.

Resistance to byzantine slice controllers is baked into the BLASTSHIELD design. BLASTSHIELD does not allow any inter-controller interaction. Each controller uses its own services to get demand and topology inputs. It calculates TE

routes by sensing the state of the network, and does not rely on communication with other controllers. Route programmers of a WAN slice do not communicate with router agents in other slices, and thus are unaffected by unreliable agents in other slices. Access control lists on slice routers prevent another slice controller from attempting to program them.

Despite these protections, a byzantine controller may route traffic in a way that causes congestion in downstream slices. A slice controller estimates the demands at the slice boundary based on the assumption that all slices are well behaved *i.e.*, they use the same algorithm and configuration as itself. Byzantine slice can violate this assumption. The impact of a byzantine controller’s actions are limited to the remote traffic from the byzantine slice. WAN traffic patterns inform the creation of slices that minimize inter-slice traffic [30].

We note that non-byzantine controller faults are also possible. Faulty controller may withdraw all its routes and congest links in its own or other slices. A faulty controller may loop or blackhole packets. While we have safety checks and routing constraints that prevent such conditions, if a controller manages to bypass the checks, human intervention is required. We mitigate these failures by pausing the faulty controllers, and restoring the network programming to last known good FIB.

## 8 Related work

B4 [17, 19] and EBB [10] are two examples of operational networks that use software-defined traffic engineering. [17] states that site-level domain controllers were large blast radius and faults caused widespread impact to traffic passing through the affected site, which led them to divide a site into two or four control domains, each managed by a separate domain controller. Similarly, in BLASTSHIELD, we assign routers in a site to separate slice controllers. [17] uses a central controller to calculate tunnel split groups and the sequencing of traffic engineering operations, and a large fleet of domain controllers to do route calculation and programming. BLASTSHIELD does not use any central controllers and each slice controller performs global traffic engineering calculation and slice-local route programming. It should be noted that the network architectures of BLASTSHIELD and B4 are quite different. [10] uses a centralized controller and segment routing, which we evaluated but did not select because of label stack depth and lack of control in intermediate slices.

Prior work on software-defined traffic engineering [1, 7, 23, 25, 29] focus on the optimization problem of maximizing utilization, guarantee fairness, preventing congestion under faults, or dynamic pricing without considering how they would be deployed. They all assume a centralized controller will perform the optimization for the entire network without considering what happens when the controller fails. BLASTSHIELD can be used in conjunction with these works to make them deployable in operational networks.

Inter-slice routing is similar to pathlet routing [13] but without any controller interaction or dissemination protocol. [9, 12] study consistent updates and loop avoidance with a centralized controller, but not multiple controllers with inconsistent views. BLASTSHIELD adopts a stricter approach of not communicating with another controller to avoid additional failure modes from faults in the communication, and because the information a controller needs can be acquired from the network.

## 9 Conclusion

In this work, we motivate the design of a decentralized traffic engineering system for large-scale cloud WANs using our operational experience with SWAN. We propose BLASTSHIELD, Microsoft’s new global TE system that decentralizes the TE controller with WAN slicing and implements loop-free inter-slice routing. BLASTSHIELD achieves similar throughput as fully centralized TE implementations while significantly reducing the blast radius of faults in TE controllers. We have been operating Microsoft’s WAN with BLASTSHIELD, and it has substantially lowered the risk of configuration changes causing large outages.

**Acknowledgements.** We thank our colleagues for their significant contributions to BLASTSHIELD: Amin Ahmadi Adl, Ashlesha Atrey, Jeff Cox, Shubhangi Gupta, Guruprasad Hiriyanaiyah, Luis Irún-Briz, Karthick Jayaraman, Srikanth Kandula, Pranav Khanna, Sonal Kothari, Nishchay Kumar, Erica Lan, Dave Maltz, Paul Mattes, Antra Mishra, Zahira Nasrin, Paul Pal, Francesco De Paolis, Rohit Pujar, Rejimon Radhakrishnan, Prabhakar Reddy, Newton Sanches, Anubha Sewlani, Sailaja Vellanki, Wei Wang, and Li-Fen Wu. We also thank our shepherd, Stefan Schmid, and the anonymous reviewers who gave us invaluable feedback.

## References

- [1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *Proceedings of USENIX NSDI*, pages 175–200, April 2021.
- [2] Port channels and LACP load balancing hashing algorithms. <https://www.arista.com/en/um-eos/eos-port-channels-and-lacp>, accessed February 2022.
- [3] Algirdas Avižienis. Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, December 1976.
- [4] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels. RFC 3209, December 2001.

- [5] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, December 1998.
- [6] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 29–43, August 2019.
- [7] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. Lancet: Better network resilience by designing for pruned failure sets. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), December 2019.
- [8] Implementing Cisco Express Forwarding. <https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5500/ip-addresses/66x/b-ip-addresses-cg-ncs5500-66x/m-implementing-cisco-express-forwarding-ncs5500.html>, accessed February 2022.
- [9] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 133–143, June 2016.
- [10] Mikel Jimenez Fernandez and Henry Kwok. Building express backbone: Facebook's new long-haul network, May 2017. <https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/>.
- [11] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Brune Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. RFC 8402, July 2018.
- [12] Klaus-Tycho Forster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking*, May 2016.
- [13] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of ACM SIGCOMM*, pages 111–122, August 2009.
- [14] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, László Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI Layout (Algorithms and Combinatorics)*, volume 9, pages 101–164. Springer-Verlag, 1990.
- [15] Hannes Gredler, Jan Medved, Stefano Previdi, Adrian Farrel, and Saikat Ray. North-bound distribution of link-state and traffic engineering (TE) information using BGP. RFC 7752, March 2016.
- [16] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of ACM SIGCOMM*, pages 15–26, August 2013.
- [17] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of ACM SIGCOMM*, pages 74–87, August 2018.
- [18] Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473). ISO/IEC 10589:2002, November 2002. <https://www.iso.org/standard/30932.html>.
- [19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hözle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of ACM SIGCOMM*, pages 3–14, August 2013.
- [20] Understanding the algorithm used to load balance traffic on MX series routers. <https://www.juniper.net/documentation/us/en/software/junos/sampling-forwarding-monitoring/topics/concept/hash-computation-mpcs-understanding.html>, accessed February 2022.
- [21] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 253–264, August 2005.
- [22] Dave Katz and Dave Ward. Bidirectional Forwarding Detection. RFC 5880, June 2010.
- [23] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *Proceedings of USENIX NSDI*, pages 157–170, April 2018.
- [24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

- [25] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of ACM SIGCOMM*, pages 527–538, August 2014.
- [26] Peter Phaal and Marc Levine. sFlow version 5, July 2004.
- [27] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.
- [28] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture. RFC 3031, January 2001.
- [29] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with Cascara. In *Proceedings of USENIX NSDI*, pages 201–216, April 2021.
- [30] Rachee Singh, Nikolaj Bjørner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with Shoofly. In *Proceedings of ACM SIGCOMM*, pages 534–546, August 2021.

# Detecting Ephemeral Optical Events with OpTel

Congcong Miao<sup>1</sup>, Minggang Chen<sup>1</sup>, Arpit Gupta<sup>2</sup>, Zili Meng<sup>3</sup>, Lianjin Ye<sup>3</sup>, Jingyu Xiao<sup>3</sup>,  
Jie Chen<sup>1</sup>, Zekun He<sup>1</sup>, Xulong Luo<sup>1</sup>, Jilong Wang<sup>3,4,5</sup>, Heng Yu<sup>3</sup>

<sup>1</sup>Tencent, <sup>2</sup>UC Santa Barbara, <sup>3</sup>Tsinghua University, <sup>4</sup>BNRist, <sup>5</sup>Peng Cheng Laboratory

## Abstract

Degradation or failure events in optical backbone networks affect the service level agreements for cloud services. It is critical to detect and troubleshoot these events promptly to minimize their impact. Existing telemetry systems rely on arcane tools (e.g., SNMP) and vendor-specific controllers to collect optical data, which affects both the flexibility and scale of these systems. As a result, they fail to collect the required data on time to detect and troubleshoot degradation or failure events in a timely fashion. This paper presents the design and implementation of OpTel, an optical telemetry system that uses a centralized vendor-agnostic controller to collect optical data in a streaming fashion. More specifically, it offers flexible vendor-agnostic interfaces between the optical devices and the controller and offloads data-management tasks (e.g., creating a queryable database) from the devices to the controller. As a result, OpTel enables the collection of fine-grained optical telemetry data at the one-second granularity. It has been running in Tencent’s optical backbone network for the past six months. The fine-grained data collection enables the detection of short-lived events (i.e., ephemeral events). Compared to existing telemetry systems, OpTel accurately detects  $2\times$  more optical events. It also enables troubleshooting of these optical events in a few seconds, which is orders of magnitude faster than the state-of-the-art.

## 1 Introduction

Cloud service providers, such as Google, Microsoft, and Tencent, have embraced the approach of setting up as many data centers as possible across metro areas [6, 20, 21, 23, 26, 38]. Such an approach enables cloud providers to physically get closer to the end-users, which in turn enables a wide range of applications with diverse bandwidth and latency requirements [29, 45]. The optical backbone network that interconnects these geographically distributed data centers is critical for ensuring reliable exchange of terabits of data every day [2, 3, 24, 25, 27]. Under the hood, the optical back-

bone network is composed of optical hardware (e.g., optical transponders, amplifiers, wavelength (de-)multiplexers), and fiber cables. Degradation or failure of any of these components (i.e., optical events) would degrade the inter-DC connectivity, which in turn affects the service level agreements (SLAs) for cloud services [5, 18, 20, 49]. Therefore, to improve the reliability and availability of the optical backbone network, it is critical to promptly detect and troubleshoot optical events.

Unfortunately, existing telemetry systems are not designed for such fast-paced detection and troubleshooting of optical events. More concretely, they collect sampled or aggregated data from optical devices. Such coarse-grained data is not suited for either detecting short-lived optical events or troubleshooting related optical events to various stakeholders (i.e., application developers, data center tenants, etc.). Figure 1(a) illustrates the limitations of existing telemetry systems. Here, when a customer reports degradation in the quality of experience for video streaming service (e.g., rebuffering), attributable to a short-lived optical event lasting few tens of seconds. The network operator that looks at the telemetry data collected by the existing telemetry systems at the 15-minute granularity cannot detect or troubleshoot such a short-lived optical event. The current telemetry systems are slow in detecting and troubleshooting the more disruptive persistent events as well. Network operators need to query data from multiple vendor-specific controllers to stitch a holistic view of the underlying network, which is tedious and prone to errors. Our analysis of the trouble tickets dataset shows that it takes hours to days to troubleshoot the optical hardware failures. Though we witnessed the development of various network telemetry systems, such as Sonata [19], Marple [33], PathDump [42], OmniMon [22], etc., that offer packet-level network streaming analytics at scale, they are not suited for diagnosing degradation or failure events in optical networks.

The limitations of the existing telemetry systems are attributable to three key factors. First, the optical backbone network uses devices from multiple vendors (i.e., vendor-free optical systems in § 2.1), and the current telemetry systems develop interfaces for vendor-specific controllers to ac-

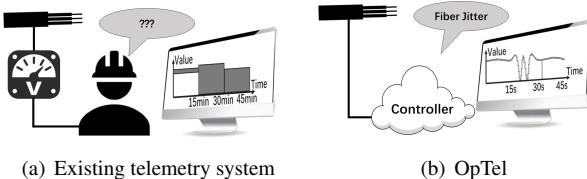


Figure 1: (a) Existing telemetry systems fail to detect ephemeral events and are slow in detecting and troubleshooting persistent events. (b) OpTel detects and troubleshoots both ephemeral and persistent events promptly with the one-second granularity data.

cess the optical data. Though vendor diversity is critical for cloud providers to deter vendor monopolies and avoid concurrent failures, *fragmented design* of existing telemetry systems is undesirable. It inhibits accessing optical data directly or extracting a consistent network view. Second, the existing telemetry systems rely on *arcane tool*, i.e., SNMP, to collect data from different devices. SNMP performs various data-management tasks, such as creating a local MIB database [35], supporting read and write operations to this database, etc., locally on the optical devices. Both faster reads (queries) and writes to this database will cause higher CPU usage. Given the limited resources at the device, it is not possible to query this data at higher frequencies with the SNMP protocol. Third, the vendor-specific controllers run on physical servers with fixed compute and memory resources. Such *inelastic resource allocation* for the existing telemetry pipelines creates multiple bottlenecks with the increasing number of optical devices or data-collection frequencies.

In this paper, we present the design and implementation of OpTel (Figure 1(b)), an optical telemetry system for optical networks. The proposed system offers direct access to optical data in a vendor-agnostic manner and offloads data-management tasks from the optical devices to cloud-based controllers that can easily scale with network size and collection frequency. We highlight the salient feature of the proposed system below.

**Vendor-agnostic centralized control.** OpTel shunts away vendor-specific controllers and replaces them with a single centralized controller that directly interfaces with optical devices in a vendor-agnostic manner. To enable such a vendor-agnostic design, we develop a standardized model for optical devices. This device-level model consists of two essential parts: logic and data model. Here, the logic model identifies key components common across devices from different vendors and standardizes their workflow. The data model specifies the configurable parameters for each component.

**Streamline telemetry pipeline at optical devices.** OpTel replaces SNMP (pull-based) protocol with a “push-based” telemetry pipeline. More concretely, it offloads the compute-intense data-management tasks from the optical devices to cloud-based controllers, with access to an elastic pool of re-

sources. Such streamlining of the telemetry pipeline offloads resource-intense operations to the cloud, enabling OpTel to collect fine-grained optical data at higher frequencies from resource-constrained optical devices. The telemetry pipeline at optical devices consists of the following key parts: telemetry manager, telemetry agent, cache, and aggregator. Here, the telemetry manager interfaces with the centralized controller and is responsible for receiving configurations from the controller and configuring other parts. The telemetry agent reads data from different modules and stores them into the local cache. The aggregator is responsible for pushing the data in the cache to the centralized controller.

The rest of the paper presents the background and motivation in Section 2, details the design and implementation in Section 3. We demonstrate how OpTel enables collecting fine-grained telemetry data at the one-second granularity and how such a dataset empowers network operators to promptly detect and troubleshoot optical events, both persistent and ephemeral, in Section 4. We have been running OpTel in Tencent’s optical backbone network for the past six months. We report our experience of collecting and analyzing the telemetry data at scale. Notably, we demonstrate that access to such fine-grained data enables us to establish temporal relationships between different optical events.

## 2 Background and Motivation

We first provide an overview of the optical backbone network (§ 2.1). We then discuss why existing telemetry systems fail to promptly detect and troubleshoot optical events (§ 2.2).

### 2.1 Optical Backbone Network

The optical backbone network interconnects different data center sites, carrying terabytes of traffic each day. Figure 2 zooms-in into a specific link (i.e., an optical transport system) interconnecting two data center sites. Each link consists of an optical line system (OLS) and multiple optical transponder units (OTUs). Each OTU receives the electrical signal from the data center router (DR), and converts it into a specific wavelength, called an optical *channel*, and vice versa. When router ports have a lower capacity than the optical channel, the OTU encapsulates and multiplexes multiple router ports onto the channel.

The OLS contains two optical *segments*, one for each direction of network traffic. Each segment carries multiple optical channels, with wavelength division multiplexing/demultiplexing (MUX/DMUX) combining/splitting these channels and booster amplifier (BA) at the transmitting end and preamplifier (PA) at the receiving end. Segments also have in-line amplifiers (LAs) that amplify the signal in the optical domain to deal with long-haul transmission loss. Each part of the segment is called a *span*. As a special case, segment yields span if the OLS does not have LA. Optical Supervisory

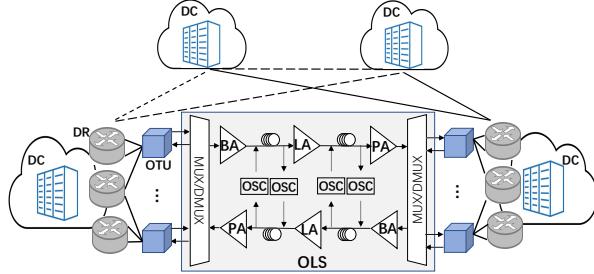


Figure 2: An overview of the optical backbone network.

Channel (OSC) is an additional channel that does not carry any payload traffic and monitors each span.

Most cloud providers use optical devices from multiple vendors. Vendor diversity is intentional to deter vendor monopolies and avoid concurrent failures. Typically, cloud providers, including Tencent, embrace a *vendor-free design* of the optical transport system (i.e., vendor-free optical system), where they purchase optical line systems and optical transponder units from different vendors [11].

## 2.2 Monitoring Optical Backbone Network

Any degradation or failure events in the optical backbone network can affect the SLAs for various cloud services. Thus, it is critical for network operators to promptly detect and diagnose such optical events, which in turn requires collecting fine-grained optical data from the underlying optical devices at high frequency. The existing telemetry systems are not designed to support such intense data-collection requirements. We identify three key factors that inhibit existing telemetry systems to scale flexible data collection.

**Highly fragmented design.** The control plane for most optical backbone networks is highly fragmented as it relies on vendor-specific controllers to manage individual devices. The existing telemetry systems inherited this fragmented design, where a centralized controller interfaces with vendor-specific controllers to collect the required telemetry data. Such a fragmented design inhibits flexible and direct access to fine-grained optical data at scale. Each vendor-specific subsystem implements its workflow to collect the data from individual devices, affecting how frequently each subsystem reports the telemetry data. Additionally, the data schemas across vendors are different, which further inhibits supporting a consistent representation of the collected data.

To illustrate the impact of each of these factors, we use the metric, *polling delay*, which measures the difference in time when the centralized controller sends the poll request to vendor-specific controllers and when it receives the requested data. We have performed the measurement studies of two subsystems provided by vendor 1 and vendor 2. For confidentiality, we omit the vendor name. We observe it takes about 3 minutes and 7 minutes to complete the collection of 5 indicators from vendor 1 and vendor 2 respectively. Here, each indicator represents the type of data, such as SNR, Q-factor,

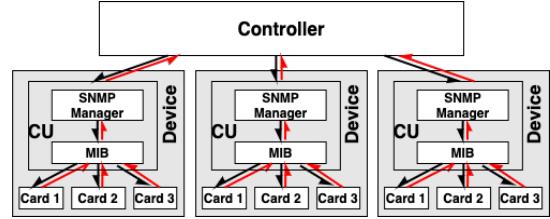


Figure 3: SNMP’s data collection workflow

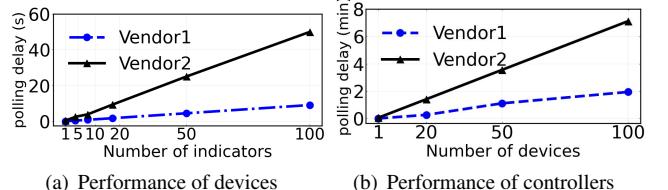


Figure 4: Performance of optical devices and vendor-specific controllers from two vendors.

etc., collected from the devices. The difference in polling delay across two vendors is attributable to an artifact of different data-collection workflow each applied within its subsystem. Such high variance in polling delays across different vendors makes it hard for network operators to extract a consistent (synchronized) view of the network, affecting their ability to troubleshoot various optical events.

**Reliance on arcane data-collection tools.** Most existing telemetry systems for optical backbone networks rely on SNMP [10], which is not suited for high-frequency data collection. SNMP performs various data-management tasks locally on the optical device. More concretely, it creates and updates a local queryable database (MIB) on the device, and handles controller’s queries. Figure 3 shows SNMP’s data collection workflow. Here, to simplify exposition, we divide the optical device into control and data plane. Here, the control plane consists of SNMP manager and MIBs and the data plane comprises of multiple line cards. The black and red arrows represent the control and data flows respectively. Once the SNMP manager receives an SNMP GET request from the controller, it traverses the table in MIB database [35] one by one to get the function to obtain the data from the line card and then reports the requested data. This process is slow and consumes a significant number of CPU cycles, making it difficult to scale data collection frequency with SNMP.

Figure 4(a) shows how polling delay changes as the number of indicators increases. We observe that the relationship between polling delay and the number of indicators is linear. Our interactions with vendors revealed that this linear relationship is attributable to their choice of serializing read request for multiple indicators, reading only one at a time. This design choice limits SNMP’s CPU usage, which competes with device’s data-plane operations. Such a long polling delay with SNMP inhibits existing telemetry systems to collect fine-grained optical data at higher frequencies.

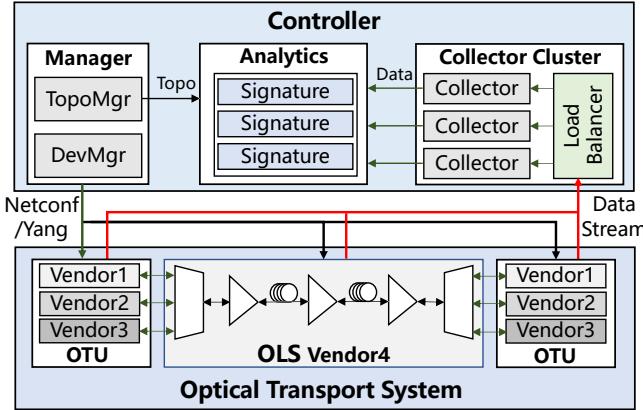


Figure 5: Architecture of OpTel.

**Inelastic resource allocation for the telemetry pipeline.** Telemetry systems need to concurrently collect data from all the underlying optical devices to ensure a consistent and fine-grained view of the network. However, we observe that the vendor-specific controllers run on physical servers with fixed compute and memory resources. Such an inelastic design makes these controllers a bottleneck in existing telemetry pipelines as the number of optical devices or the collection frequency increases.

Unsurprisingly, we observe a linear relationship between polling delay and the number of devices in Figure 4(b). This behavior is also attributable to vendors’ choice to serialize requests at the controller. Such serialization ensures that the controller can handle all the incoming requests with a fixed set of resources at the cost of longer polling delays, which affects the ability to construct a consistent view of the network at fine time scales. More concretely, it is impossible to correlate the optical data across two different optical devices on a short time scale, affecting the troubleshooting capabilities of the existing telemetry systems.

### 3 OpTel’s Design and Implementation

We now describe how the proposed system, OpTel, addresses the limitations of existing telemetry systems described above. We first state its design goals in Section § 3.1, and then describe how it achieves these goals in Section § 3.2 and § 3.3.

#### 3.1 Design Goals

OpTel’s goal is to extract multiple indicators from all the devices in the optical backbone at finer time granularities, i.e., order of seconds. Such a dataset is critical for timely detection and diagnosis of various disruptive events in the backbone network. OpTel addresses the limitations of existing telemetry systems to achieve this goal. More concretely, to address the fragmentation issue, it bypasses vendor-specific controllers to collect the telemetry data directly from the optical devices in a vendor-agnostic manner. To address the scalability issues, it

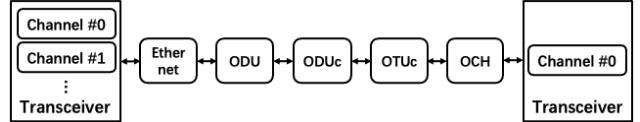


Figure 6: The logic model of OTU.

streamlines the telemetry pipeline such that it performs all the complex data-management tasks to a centralized controller running in the cloud. Such a design ensures that the data collection pipeline is not bottlenecked by limited compute resources at the individual devices. The centralized controller has access to an elastic pool of resources in the cloud.

### 3.2 Vendor-agnostic Centralized Control

Figure 5 presents OpTel’s architecture. Here, the centralized controller directly interfaces with the optical devices in a vendor-agnostic manner. We developed a standardized model that abstracts away the vendor-specific details for the controller. We now describe how we develop the vendor-agnostic device model and how it enables collecting data directly from the optical devices.

#### 3.2.1 Standardized Model for Optical Devices

In vendor-free optical systems, the operation performed by different optical devices is similar at a high level, but the specific logic and workflow vary across vendors. Such heterogeneity across devices from different vendors complicates the design of vendor-agnostic interfaces. We develop a standardized model for optical devices that abstracts away the vendor-specific details to address this challenge. It consists of two parts: *logic model* and *data model*. Here, the logic model identifies key components common across devices from different vendors and standardizes their workflow. The data model specifies the configurable parameters for each component.

**Logic model.** The first challenge in developing vendor-agnostic interfaces is that the physical components and their workflow are proprietary to each vendor. To address this challenge, the logic model first identifies a group of logical components that are common across devices from different vendors. It then standardizes the workflow between these components. To illustrate, consider the case of optical transponder units, i.e., OTUs. Figure 6 shows OTU’s logic model. Here, the logic model first identifies four logical components across all vendors: Ethernet, optical data unit (ODU, ODUc), optical transport unit (OTUc), and optical channel (OCH). Recall that an OTU encapsulates and multiplexes multiple router ports onto an optical channel. The ODUc is a high-order data unit after combining the payload data from multiple router ports. The logic model then specifies the workflow between these components. For example, the mapping between Ethernet and ODU represents an encapsulation of an Ethernet frame into an ODU frame. Such an abstraction enables the standardized representation of different optical devices.

**Data model.** The second challenge in enabling vendor-agnostic interfaces is that the capability of physical components inside the device is different across vendors, although their functions are the same. For example, the range of gain of an optical amplifier provided by vendor 1 is 15-25 dB, while it might be 20-30 dB from vendor 2. This heterogeneity complicates managing these devices in a vendor-agnostic manner. We design a component data model with specific descriptions of configurable parameters of each component. When each device connects to the controller, the controller obtains the specification datasheets from the device and initializes the corresponding value of configurable parameters. Such an approach simplifies the management complexity of heterogeneous devices, regardless of the capability of physical components inside the device.

We have developed a model for each device type for our optical backbone network. Our experience using these models in production settings was smooth, demonstrating their generalizability.

### 3.2.2 Centralized Data Collection

The standardized model allows the centralized controller to access the telemetry data directly, enabling OpTel to shunt away vendor-specific controllers. The centralized controller consists of three key modules: *global manager*, *scalable collector* and *real-time analytics*, to perform detecting and troubleshooting optical events at scale in a timely manner.

**Global manager.** It consists of two parts: device manager (DevMgr) and topology manager (TopoMgr). The DevMgr is responsible for configuring the underlying optical devices. For each device, it leverages the relevant standardized model to configure devices in a vendor-agnostic manner. It completes this process by issuing a Yang file [7] to the device through the vendor-neutral Netconf protocol [13]. The TopoMgr maintains a physical topology of optical devices to provide a network-wide view of the optical networks and thus helps the real-time analytics to troubleshoot the optical events at scale. To illustrate how TopoMgr aids troubleshooting, consider the case of degradation in a fiber cable. Here, as it is not possible to directly collect the data from the cable, the analytics can instead use the TopoMgr to identify the two terminal devices at each end of the cable. It can then query the transmit (Tx) and receive (Rx) power data from these devices for troubleshooting.

**Scalable collector.** This module is a cluster of multiple collector nodes designed to handle changes in the number of indicators, collection frequency, or the number of optical devices over time. With the aid of the cloud’s elastic pool of resources, it can scale horizontally by adding (or removing) collector nodes over time. It relies on a load balancer to distribute the load among individual collectors within the cluster. It is robust against the failure of a particular collector node.

**Real-time analytics.** It performs the task of promptly de-

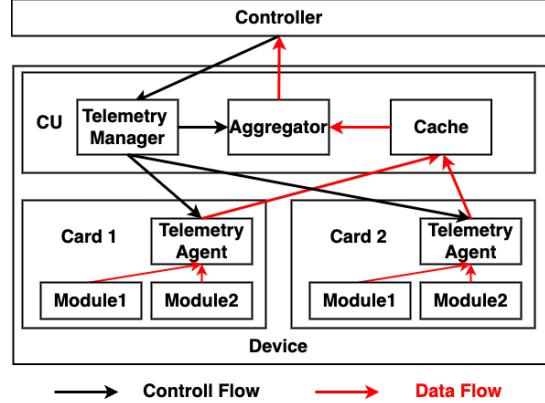


Figure 7: Push-based optical telemetry

tecting and troubleshooting optical events by combining the optical data from the collector and the topology information from the global manager. The workflow of real-time analytics consists of two parts: detection and troubleshooting. To detect degradation or failure events, it monitors the values of optical data in real-time and raises the alarm if the value exceeds a pre-specified threshold. In parallel, it starts the troubleshooting process. Rather than manually troubleshooting the optical event, it leverages the signatures of previous optical events for diagnosis. To illustrate, consider the case when the received optical power becomes zero for a device. The analytics module first raises the alarm with a message, *the receiver can not receive the light* and then begins troubleshooting. It matches the collected data with previous signatures. If it finds the match, it simply sends troubleshooting report to the operator. If the collected data does not match a pre-existing signature, it lets operators manually express their queries for troubleshooting. It automatically updates the relevant signatures for future events. Our deployment experience shows that is possible to troubleshoot most of the optical events using existing signatures.

### 3.3 Streamlined Telemetry Pipeline

Promptly detecting and troubleshooting optical events requires collecting fine-grained data from the underlying optical devices. The widely used SNMP is flawed in performance because it performs various data-management tasks locally on the resource-constrained optical devices (Figure 3). In contrast, OpTel offloads compute-intense operations from the optical devices to the centralized controllers by push-based telemetry pipeline, enabling to collect the fine-grained optical data at higher frequencies. Figure 7 depicts the architecture of push-based optical telemetry. The telemetry pipeline at optical devices consists of the following key parts: *telemetry manager*, *cache* and *aggregator* in the control unit (CU), and *telemetry agents* in the line cards. The telemetry manager is responsible for the configurations of other parts, i.e., telemetry agent and aggregator. The telemetry agent reads data from different modules and stores them into the local cache. The

aggregator is responsible for pushing the data in the cache to the centralized controller. In the following, we will describe them in detail.

**Telemetry manager.** The offloading of compute-intense data-management tasks from the optical devices to the controller requires preliminary configurations at the device. The telemetry manager firstly interfaces with the centralized controller to obtain the YANG file [7] and then parses the YANG file to configure the telemetry agent and aggregator. The aggregator is configured to periodically initiate a connection to push the optical data from the local cache to the controller. As for the telemetry agent in the line card, it is configured in three parts: the destination of data (i.e., cache), the source of data (i.e., modules in the line card), and the periodicity that the telemetry agent should push the data.

However, based on the real-world deployment experiences, we observed that configuring the same periodicity for pushing data at the telemetry agent and aggregator may result in the frequent data loss in the controller. This phenomenon is attributable to the different timing mechanisms. Generally, the CU always runs a Linux operating system and enables network time protocol (NTP) [30] to keep timing. However, some line cards are the embedded equipment without running a Linux operating system, resulting in it being unable to keep timing through NTP. Thus, these line cards keep timing through the crystal oscillator. The frequency deviation inside the crystal oscillator will lead to the timing inaccuracies [44]. Therefore, the performance data pushed by the telemetry agent is not strictly periodic. Slower timing will result in the data not being stored in the local cache, which in turn causes data loss in the controller. For example, assume that the controller needs to collect the data from the device at the one-second granularity. The telemetry manager configures the telemetry agent and aggregator to push the data every one second. However, the frequency deviation inside the crystal oscillator may result in the timing in the line card slower than that in the CU. It will take more than one second for the telemetry agent to push the data to the local cache. Therefore, the aggregator will push the empty data to the controller. Motivated by this, we always configure the data pushed in the telemetry agent at a higher frequency.

**Telemetry agent.** Once configured, the telemetry agent periodically performs the card-level data collection through the vendor-specific protocols and pushes the data to the local cache. Specifically, the values of data are generated in two ways: instant value and accumulated value.

*Instant Value.* It is a sampled data in a given time interval. The receiver captures the physical analog signal and then translates it into the digital value, which is further stored in the RAM. Figure 8(a) describes the process of generating instant value of the received signal in the physical layer. The PIN photodiode firstly captures the light signal and transforms it into the analog current. An analog-to-digital converter (ADC) is applied to convert the analog current into a digital value

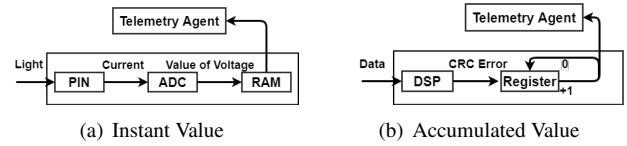


Figure 8: **The process of generating specific values.** (a) Instant value records the performance in the physical layer; (b) Accumulated value records the performance in the data link and network layer.

of voltage which is further stored in the RAM. The telemetry agent periodically reads RAM to collect the data through the vendor-specific protocol. Note that the value in RAM will be replaced frequently, thus enables the data to be collected at higher frequencies. In our work, the instant value records the performance in the physical layer. We use transmit/receive (Tx/Rx) power to detect optical events, and signal-to-noise ratio (SNR) and quality factor (Q-factor) to check the ability of the optical system to transmit data (Figure 9(a) and 9(b)).

*Accumulated Value.* It is a counting value accumulated across the whole timeline. The digital signal processor (DSP) processes the received digital signal and counts in a certain way. Figure 8(b) describes the process of generating the accumulated value of the CRC error. The register counts the volume of CRC errors in the whole time interval. The telemetry agent periodically reads the register to collect the value. After that, the register will be reset to its initial value. The accumulated value records the performance in the data link and network layer, such as CRC error and post forward error correction (FEC). We use them to differentiate optical events according to the influence of optical events in the data link and network layer (Figure 9(c), 9(d)).

**Cache.** The local cache serves as data storage that stores the performance data received from the telemetry agent and then bundles data at the device level. It is compatible with the performance data pushed by the different agents at different frequencies. Generally, the data for a single indicator stored in the telemetry cache is more fine-grained since the frequency of the telemetry agent pushing the data is higher than that of the aggregator reading the data. The data in the local cache will be cleaned after being read.

**Aggregator.** The aggregator periodically initiates a connection to get the bulked data from the local cache. Since the data provided by the cache is more fine-grained, the aggregator should merge the data to get representative statistics and push them to the controller through the gRPC protocol [1].

## 4 Evaluation

OpTel has been running in Tencent’s backbone network for the past six months, demonstrating its deployability in production settings. In this section, we show how the proposed streamlined telemetry pipeline enables collecting all possible indicators from all the optical devices in the network at the

one-second frequency (Section 4.2). We then show how such fine-grained data enables the detection of ephemeral optical events (Section 4.3). We investigate how ephemeral events help predict more disruptive future events, illustrating the utility of such a fine-grained telemetry system (Section 4.4). We also demonstrate how such fine-grained data enables troubleshooting optical events in the order of few seconds, which is orders of magnitude faster than possible with the existing telemetry systems (Section 4.5).

## 4.1 Setup

### 4.1.1 Dataset

We use OpTel to curate three datasets. Here, we collect the data for six months (July–December, 2020) from Tencent’s optical backbone network. This backbone has O(50) links, O(100) spans, O(100) segments, O(1000) optical channels, and O(1000) optical devices from O(10) vendors. For confidentiality reasons, we do not report the exact numbers.

**Optical telemetry dataset.** We curate this dataset by collecting all indicators from all the optical devices at one-second granularity using OpTel. We collect the Tx/Rx power levels, SNR, and Q-factor from the physical layer. From the data link layer, we collect the Post Forward Error Correction Bit Error Rate (FEC BER) [31], loss and error frame rate (i.e., the ratio of the number of Rx vs. Tx frames and Error vs. Rx frames). We also collect cyclic redundancy check (CRC) error rate [40] from the network layer. Here, the physical layer indicators are “instant” values, and the rest are the “accumulated” values (§ 3.3). Also, note that since the OTU encapsulates and multiplexes payload from router ports, we collect the data link and network layer indicators directly from the OTU (§ 3.2.1). However, it is not efficient to only focus on the Tx/Rx power in OTU or BA/PA to troubleshoot optical events. For example, if there are several spans and LAs in a segment, and the Rx power of PA becomes 0 while the Tx power of BA does not change, we can not distinguish which span is responsible for the event. Thus, we combine the Tx/Rx power of OSC for span-level monitoring. The detailed origins of telemetry data are shown in Figure 17 in appendix A.

**Location dataset.** We use OpTel’s TopoMgr to curate this dataset. It maintains a topology of the devices to provide a network-wide view to establish a relationship between different devices. Such relationships are critical for troubleshooting as indicators from a single device are often not enough to diagnose various optical events. For example, diagnosing degradation events in fiber cables requires data from both ends of the fiber cable.

**Trouble tickets dataset.** We collect this data from the network management platform at Tencent. We first filter out the events related to the optical networks (see appendix B for details) and then categorize these optical events into a small number of classes, i.e., fiber cable, hardware, and power

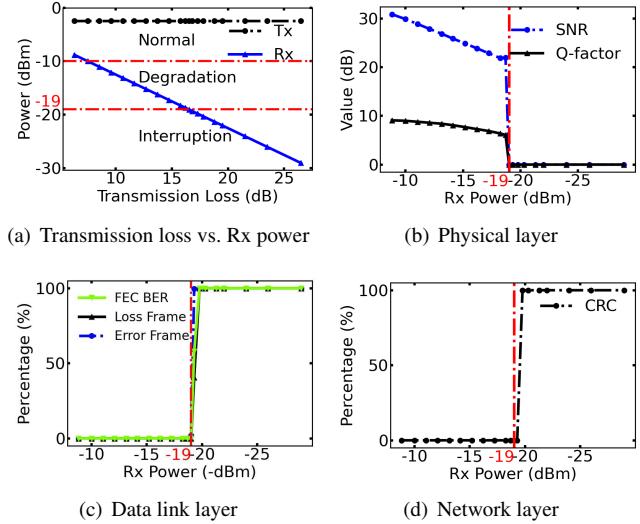


Figure 9: An example of the physical/data link/network layer behaviors with the increase of transmission loss.

events. Each ticket contains a timestamp recording the event’s start time with detailed messages and a corresponding timestamp recording the localization of the event, i.e., event name (e.g., optical fiber jitter, amplifier instability, etc.). Note, troubleshooting optical events requires much manual effort in existing telemetry systems. We use this data to learn signatures of different optical events and show the time efficiency of OpTel on troubleshooting optical events by comparing it with the existing telemetry system.

### 4.1.2 Optical Events

We now present how we categorize optical events on the basis of their impact (degradation vs. interruption events) and duration (ephemeral vs. persistent events).

**Interruption vs. Degradation events.** To categorize optical events on the basis of their impact, we investigate the relationship between indicators at the physical, data link, and network layer (see Figure 9). Specifically, we take an optical transport system as an example, and fix the Tx power and iteratively adjust the transmission loss of optical fiber to simulate the degradation/failure event. Figure 9(a) shows a linear relationship between the transmission loss and values of Rx power in OTU based on the formula  $Rx\ power\ (dBm) = Tx\ power\ (dBm) - transmission\ loss\ (dB)$ . We observe similar trends in PA (not shown for brevity). After establishing the relationship between transmission loss and Rx power, we study how degradation in the power level at the receiver affects SNR and Q-factor at the physical layer (Figure 9(b)); FEC BER, loss frame and error frame rate at the data link layer (Figure 9(c)); and the CRC at the network layer (Figure 9(d)).

For higher Rx power levels (i.e., around -9 dBm), the values of SNR and Q-factor are high with SNR=31 dB and Q-factor=9 dB. The SNR and Q-factor indicate the ability of the

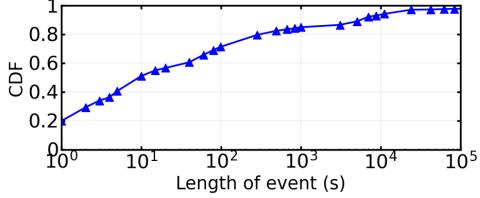


Figure 10: The CDF of optical events’ duration.

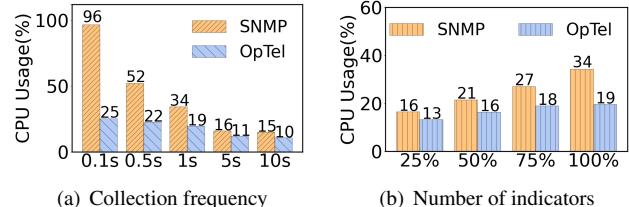
system to transmit data. Higher values for these physical layer indicators imply higher possibility of correctly decoding the transmitted ‘1’s and ‘0’s signal, and vice versa [18]. As Rx power decreases, SNR and Q-factor decrease linearly. But the values of data link and network layer indicators do not change as Rx power level above a specific threshold guarantees correct decoding of the transmitted signals. When the Rx power is below the threshold (i.e., -19 dBm), the Q-factor and SNR are below the sensitivity of transceiver and thus, it reports 0 to represent the abnormal state of optical system. For links with low SNR and Q-factor, the post-FEC BER increases because the number of error bits exceeds its error correction capability. Consequently, the receiver can not restore the transmitted data, resulting in nearly 100% of frame loss and error in the data link layer (Figure 9(c)) and packet loss due to CRC error in the network layer (Figure 9(d)).

Given these observations, we conclude that the Tx/Rx power level is the key indicator of optical-layer performance. This reinforces the prior study [53]. Thus, We use the changes of Tx/Rx power level to detect optical events and divide optical events into two broad categories on the basis of their impact: *degradation* and *interruption*. Generally, there is a conservative deployment of the optical transport system, with redundancy baked in at the Rx power. The degradation event occurs when the optical system transitions to an abnormal state, evident from smaller values for physical layer indicators Tx/Rx power level, Q-factor, SNR, etc. However, here such anomalies do not affect the data transmission at the data link or network layer. In contrast, the interruption events are where further degradation in the physical layer starts affecting data transmission. Note that fluctuations in Rx/Tx power levels are common in production networks. Based on the network operator’s experiences, we treat any fluctuation within 1 dB range as normal.

**Ephemeral vs. Persistent events.** Optical events not only vary in terms of impact but also in duration. Figure 10 shows the duration of optical events (both interruption and degradation). We observe that the event duration exhibits long-tail behavior. Interestingly, we observe that 20% of events only last for one second and more than 50% of them last for less than ten seconds, indicating the prevalence of such transient optical events in the optical backbone. These observations demonstrate the utility of OpTel’s ability to detect such short-lived events that go unnoticed with the existing telemetry systems. Given these observations, we divide optical events into two categories based on their duration. We call all the

Table 1: The proportion of four types of optical events.

Type	P-I	P-D	E-I	E-D	Total
Percentage	44.63%	4.28%	16.85%	34.24%	100%



(a) Collection frequency

(b) Number of indicators

Figure 11: CPU usage of the device with different collection frequencies and numbers of indicators (normalized).

optical events that last less than ten seconds as *ephemeral events* and the rest as *persistent events*.

Overall, we consider four different types of optical events based on the combination of their impact and duration: *ephemeral degradation (E-D)*, *persistent degradation (P-D)*, *ephemeral interruption (E-I)* and *persistent interruption (P-I)*. Table 1 shows the prevalence of each of these event types in our dataset. For confidentiality, we do not report the exact numbers. We observe that optical events of the type P-I contribute 44.63% to the total, followed by the E-D events, which contribute about one-third to the total. The P-D events are the least prevalent, only contributing 4.28% to the total events. Note that more than 50% of optical events are ephemeral.

## 4.2 Data Collection Overheads

Intuitively, we expect collecting optical data at higher frequencies (i.e., order of seconds) to be prohibitively expensive. We now demonstrate how OpTel’s streamlined telemetry pipeline makes such high-frequency data collection feasible. We compare OpTel’s overhead, quantified in terms of CPU usage at the optical devices, with existing SNMP-based telemetry systems for different collection frequencies (i.e., 0.1s, 0.5s, 1s, 5s, and 10s) and the number of indicators (i.e., 25%, 50%, 75%, and 100% of the total). We can configure the same device to either use OpTel’s or conventional SNMP-based telemetry pipelines in our current deployment. Such flexibility enables us to report the CPU usage for these two different pipelines for the same set of optical devices.

Figure 11(a) shows that CPU usage increases with collection frequency for both pipeline, but the rate of change for OpTel is marginal. More specifically, the increase in collection frequency from 1 second to 0.1 seconds raises SNMP-based pipeline’s CPU usage from 34% to 96%. Such high CPU usage highlights SNMP’s struggles to handle polling requests at such high frequencies. In contrast, OpTel’s CPU usage only increases by 6% from 19% to 25%, demonstrating its efficacy. As shown in Figure 3, the polling-based SNMP consumes a significant number of CPU cycles to collect data, including receiving the request from the controller, traversing the MIB

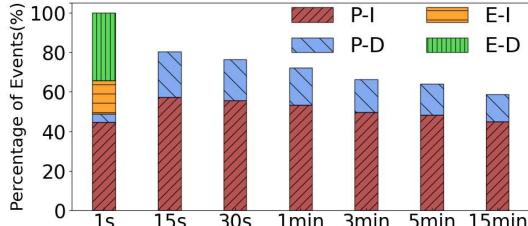


Figure 12: The percentage of events detected with the decrease of collection frequencies (y axes is normalized by total events detected with one-second granularity data).

database [35], and then requesting data from the line cards. The reduction in the CPU usage of OpTel is attributable to the offloading of compute-intense data-management tasks from the optical devices to the controller. As shown in Figure 7, once configured, the device only needs to periodically initiate a connection to push the data to the controller. This process does not introduce much CPU overhead on the device.

Figure 11(b) shows that CPU usage increases with the number of indicators, but the rate of change for SNMP-based pipeline is greater than OpTel’s. Specifically, the CPU usage is 16% if we collected 25% of total indicators with an SNMP-based pipeline. However, the CPU usage increases to 34% if all indicators are collected. In contrast, the CPU usage is only 19% for OpTel. Recall that vendors limit SNMP’s CPU usage at the cost of longer polling delays. Figure 4(a) depicts that it takes tens of seconds to complete a polling period. The high polling frequency results in a new polling request starting before the previous polling request has ended. There will be a lot of concurrent polling requests, resulting in high CPU usage. The current design choice of SNMP is not scalable to collect a large number of indicators at higher frequencies. In contrast, OpTel streamlines the telemetry pipeline by offloading resource-intense operations to the cloud. Therefore, OpTel maintains low CPU usage at the device with the increasing collection frequencies and indicators.

### 4.3 Detecting Optical Events with OpTel

Detecting optical events is essential for troubleshooting the related network disruptions to various stakeholders. We evaluate the efficiency and accuracy of OpTel on detecting optical events by comparing with existing telemetry systems.

**High Efficiency.** We firstly study the efficiency of OpTel on detecting optical events by comparing with different collection frequencies, i.e., time intervals varying orders of minutes (1min, 3min, 5min, and 15min) and seconds (1s, 15s, and 30s). Previous works only took advantage of minute-level data to study operational optical networks [8, 18, 39, 52]. To the best of our knowledge, no prior work uses second-level data to detect optical events in operational optical networks. We introduce them in experiments to further demonstrate the relationship between the number of detected events and collection frequency. Specifically, we take the data collected at

Table 2: The comparison of detected optical events with OpTel and the existing telemetry system. *UND* means undetected optical events.

OpTel (1 second)	Existing system (15 minutes)			
	P-I	P-D	UND	Total
P-I	<b>33.80%</b>	0	10.83%	44.63%
P-D	0	<b>1.92%</b>	2.36%	4.28%
E-I	<u>11.00%</u>	0	5.85%	16.85%
E-D	0	<u>11.88%</u>	22.36%	34.24%
Total	44.80%	13.80%	41.40%	100%

the one-second granularity as the ground truth and simulate the detection of events with different collection frequencies.

Figure 12 demonstrates that OpTel achieves high efficiency on detecting optical events. For confidentiality, we do not report the detailed number of events. As the figure shows, the total number of detected events decreases when the collection frequency decreases. Specifically, OpTel outperforms the collection frequencies with 15 seconds, 1 minute, and 15 minutes by 25%, 39%, and 71%, respectively. This phenomenon proves the efficiency of OpTel to detect them. Another observation is that the collection frequencies lower than 15 seconds can not detect ephemeral optical events, and the number of persistent events (i.e., P-I and P-D) decreases when the collection frequency decreases. OpTel takes advantage of the one-second granularity data to exactly detect these ephemeral events. Surprisingly, we observe that the number of persistent events detected with the 15-second granularity data is more than that with the 1-second granularity data. This phenomenon implies that the majority of ephemeral events are wrongly identified as persistent events, i.e., E-I and P-I are wrongly identified as E-D and P-D, respectively. In other words, a portion of persistent events detected with the coarse-grained data are not actually persistent. This motivates us to learn the accuracy of detecting optical events by OpTel.

**Full Accuracy.** We then study the accuracy of OpTel on detecting optical events by comparing with existing telemetry systems. We take the existing system with 15-minute granularity data as an example since it is widely studied for optical layer in previous works [8, 18, 39, 52]. Similarly, we regard the one-second granularity data as the ground truth and simulate the detection of optical events. The results are shown in Table 2. For confidentiality, the number of events is normalized by the total number of optical events detected with the 1-second granularity data. Each row represents the events detected by OpTel, while each column represents the events detected by the existing telemetry system with 15-minute granularity data. We observe that the existing system can only correctly detect 35.72% of optical events (shown in **Bold**), while 41.40% of optical events are not detected (*UND* column) and 22.88% of optical events are wrongly detected (shown in underlined). Specifically, for P-D events, only less than 50% of P-D events can be accurately identified by the existing telemetry system while the rest can not be detected. As for ephemeral events, they are either identified as persistent events or not detected

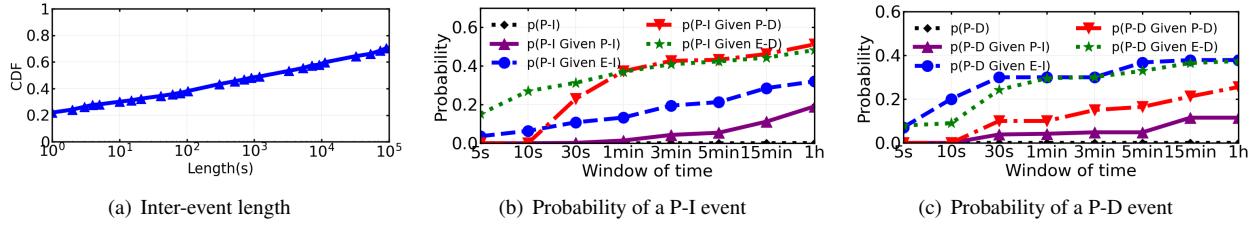


Figure 13: (a) The CDF of length of inter-events (Log-scale x-axes); (b) Probability of a P-I event in a given time window after different types of events; (c) Probability of a P-D event in a given time window after different types of events.

by the existing telemetry system. As for E-D, 22.36% of E-D events can not be detected, occupying about two thirds of total E-D events. It can be caused by several reasons. For example, if there are several E-D events in one 15-minute time interval, only one E-D event will be identified as a P-D event, and the rest can not be identified. In general, our OpTel with 1-second granularity data accurately detects all optical events, especially for ephemeral events.

#### 4.4 Predicting Future Optical Events

We evaluate the possibility of predicting future events based on the current event within a short time by OpTel. Figure 13(a) depicts the CDF of length of inter-events. A surprising observation is that 20% of inter-event lengths are only one second. This phenomenon suggests that these events occur in bursts and demonstrates the utility of the optical telemetry system on collecting data at the one-second granularity. Another observation is that 50% of inter-event lengths are less than 1000 seconds, suggesting a high probability of an optical event within about 15 minutes after the current event.

We focus on predicting persistent events as they represent a more prolonged impairment or loss in network capacity and are more predictable. Taking the P-I event as an example, we first compute the probability of a persistent interruption event within a window of time and call it  $p(\text{P-I})$ . For  $x \in \{\text{P-I}, \text{E-I}, \text{P-D}, \text{E-D}\}$ ,  $p(\text{P-I} \text{ given } x)$  indicates the probability of observing a P-I event given a prior  $x$  event within the same window. Fig 13(b) and 13(c) depict the average probabilities across all spans as a function of window size, from 5 seconds to 1 hour. In contrast to the previous work [17], our works focus on taking advantage of one-second granularity data and the ephemeral events to achieve the short-term predictions.

As expected,  $p(\text{P-D})$  and  $p(\text{P-I})$  increase as window size increases; the larger the window of time, the higher possibility of a persistent event occurs within that window. For a window of 1 hour, the probability of a persistent event occurrence is less than 1%. This suggests a low probability of having a persistent event in the 1-hour window. However, there is a significant jump in the probability if there has been another event in the past, e.g., E-D, E-I, and P-D. For example, for a window of 1 hour, the probability of persistent event occurrence increases to about 50% if there has been an E-D event within that window. Meanwhile, we observe that the events

have a strong relation in a short time window. For example, for a window of 5 seconds in Figure 13(b), the probability of P-I occurrence increases to about 20% if there has been an E-D event within that window, while for a window of 1 minute, the probability increases to 40%. This indicates that the E-D event is strongly related to the future P-I event. As for the prediction of P-D events in Figure 13(c), the probability of P-D occurrence increases to 30% if there has been an ephemeral event (i.e., E-D and E-I) within 30 seconds, and the possibility does not increase much with a larger window of time. This suggests that the P-D event always happens after the ephemeral event within 30 seconds. Another observation is that the past P-I event is less predictive of the future persistent events, indicating that the P-I events are memoryless.

OpTel demonstrates a high possibility for predicting future events at the second-level granularity. Thus, network operators could take the fine-grained IP layer network management. First, network operators should monitor the ephemeral and degradation events and raise alarms when they occur. Then, appropriate actions should be taken since the failure probability of IP layer link will increase. For example, they could improve traffic engineering so that important traffic should be dispatched away from the corresponding link.

#### 4.5 Troubleshooting Events with OpTel

**Characterizing failure signals.** We demonstrate the effectiveness of OpTel on unveiling the signatures of optical events, and thus we could quickly troubleshoot the optical events based on the observed signatures. We use the Tx/Rx power as the primary method to unveil the signatures of optical events since it is the key indicator of optical-layer performance [53]. For some optical events such as fiber events, we learn the signatures based on the network operator's experience (Figure 14). However, for some optical events such as hardware failures, we should traverse the trouble tickets dataset to learn the signatures. To locate optical events accurately, we take advantage of the centralized controller to conduct inter-device analysis by combining the Tx/Rx power from three sources, i.e., OSC, OTU, and BA/PA. For confidentiality, we do not show all signatures of optical events and take an example for each category of the events.

**(a) Fiber cable: optical fiber jitter before interruption (Figure 13(b)).** We firstly unveil the most frequent optical

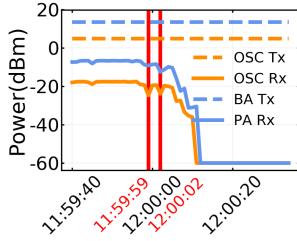


Figure 14: Fiber

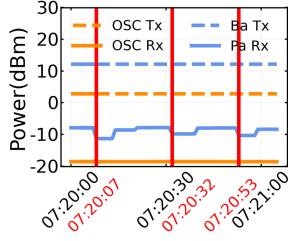


Figure 15: Amplifier

events, i.e., fiber cable events. As shown in Figure 14, on the one hand, we observe that the Tx powers of BA and OSC remain unchanged during the timeline. However, the Rx powers at both PA and OSC is down to -60 dBm after 12:00:10 (The receiver records a predefined minimum value when the received power is below its sensitivity.). On the other hand, the timestamp of Rx power changes in OSC and PA is consistent with several ephemeral degradation events at 11:59:59 and 12:00:02 before the interruption. Thus, we can localize the optical event as an optical fiber event since the sources of the light work well and the probability of two receivers at PA and OSC having problems at the same time is relatively low.

**(b) Hardware: amplifier instability.** We then unveil one example of hardware failures, i.e., amplifier failure. Since the curve is quite similar, we only select a 1-minute time interval, as shown in Figure 15. We observe that the Rx power of PA changes periodically with about a 3 dB drop at 7:20:07, 7:20:32, and 7:20:53, while the indicators of the rest remain unchanged. The stable Tx/Rx values of OSC indicate a normal state of fiber cable, while the stable Tx values of BA indicate that BA works well. Thus, we can localize the optical event as the instability of PA, i.e., amplifier failure.

**(c) Power: power outage at site of LA.** We unveil one example of power events, i.e., the power outage at the site of the in line amplifier (LA). In a long-haul transmission system, there is a relay site containing LA that amplifies the signal to deal with long-haul transmission loss. Figure 18 in Appendix C depicts the detailed origins of Tx/Rx power in Figure 16. In Figure 16(a), We observe that the Tx power of OSC can not be collected after 03:32:31 while the Tx power of BA remains almost unchanged. Surprisingly, the Rx power values of both OSC and PA become -60 dBm after 03:32:32. Thus, we locate the power outage event at LA in the site, and the delay of Rx power is mainly due to energy storage of components in the device, such as capacitance and inductance [12]. The similar results in Figure 16(b) further prove this phenomenon. Thus, we localize the optical event as a power outage at site of LA.

These signatures of optical events present the necessity of indicators to be collected at the second-level granularity which can not be demonstrated in the existing telemetry system. OpTel unveils the signatures of optical events, which presents the superiority of optical telemetry to collect data at the one-second granularity and a centralized controller to conduct inter-device analysis in real time.

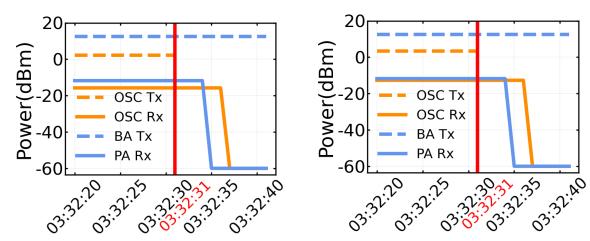


Figure 16: The power outage at site of LA.

We finally evaluate the time efficiency on troubleshooting optical events by comparing OpTel with the existing telemetry system. For the existing system, it takes much manual effort to troubleshoot the optical events, and we calculate the total time of troubleshooting the optical events based on timestamps recorded in the trouble tickets dataset. As for OpTel, based on signatures learned before, OpTel conducts inter-device analysis in a centralized controller to detect and troubleshoot optical events in a timely manner. Table 3 presents the comparison of the total time of troubleshooting optical events between OpTel and the existing telemetry system across all event categories. We do not report events that have not happened in the studied dataset. There are several observations. Firstly, 89.7% of optical events (i.e., P-D, P-I, E-D, and E-I) are caused by fiber cable, including fiber cut, fiber jitter, and fiber bent/degradation. The existing telemetry system takes about 5min ~10min to troubleshoot a fiber cut event. However, it can not troubleshoot the optical events caused by fiber jitter or degradation. In contrast, OpTel only takes several seconds to troubleshoot all the events related to the fiber cable. Secondly, 7.8% of optical events are caused by hardware, and it takes quite a long time, i.e., hours ~days and much manual effort to troubleshoot them. Some hardware events, such as amplifier instability (Figure 15) can not be troubleshooted in the existing telemetry system. In contrast, OpTel only takes about 2s ~60s to troubleshoot all the events related to hardware, reducing the time by as much as two to four orders of magnitude. The total time of troubleshooting events by OpTel is related to the time length of the signature. For example, OpTel takes 2s to troubleshoot amplifier malfunction and 60s to troubleshoot amplifier instability since we need to take about 60s to get the value change patterns of Rx power in PA to troubleshoot the optical event (Figure 15). As for the power events, OpTel is efficient to troubleshoot these events within a minute. In summary, OpTel takes advantage of the one-second granularity data to learn the signatures of the optical events and thus troubleshoots optical events at scale in a timely manner.

## 5 Related Work

**Network streaming telemetry.** Previous works have extensively studied the design of network streaming telemetry systems. End-host-based network streaming telemetry sys-

Table 3: The comparison of the total time of troubleshooting events between OpTel and the existing telemetry system.

Event category	Percentage	Event type (Detect)	Event name (Troubleshoot)	Existing telemetry system	OpTel
Fiber cable	89.7%	PI	Fiber cut	5min~10min	10s
		EI / ED	Fiber jitter	UNK	3s
		PD	Fiber bent / degradation	UNK	10s
Hardware	7.8%	PI / ED	Amplifier malfunction / instability	hours~days / UNK	2s~60s
		PI / ED	OSC malfunction / instability	hours~days / UNK	2s~60s
		PI	OTU malfunction	hours ~days	2s~60s
Power	2.5%	PI	Power outage	hours	10s~30s
		PI	Power down	hours	10s~30s

tems [4, 16, 32, 41] performed flow-level tracking but had to deal with a limited view of the network. Switch-based network telemetry systems usually offered a coarse-grained view of the network, collecting aggregated or sampled data from the network [36, 43, 48]. Systems supporting packet-level analytics offered limited flexibility as they only supported a limited set of analytics queries [28, 33, 50, 51]. More recently, hybrid telemetry systems [19, 22, 42] struck a balance between flexibility and scale, supporting dataflow operators over packet fields at scale. Though these works enabled packet-level or flow-level network streaming analytics, they were not suited for ingesting physical, data link, and network link layer data to diagnose optical events. Previous works [34, 37] did propose a telemetry system explicitly designed for optical networks. However, they evaluated the proposed artifacts in lab environments, making it difficult to assess their performance in production settings. In contrast, OpTel demonstrates the feasibility to collect fine-grained optical telemetry data at higher frequencies (i.e., one-second granularity) by running in production at Tencent’s optical backbone network for six months.

**Optical layer control.** Several works have studied the control interface of optical networks. Cox [11] proposed an ultimate goal of controlling the open optical line system (i.e., vendor-free optical system) in Microsoft’s optical backbone by a unified SDN controller and discussed some issues surrounding the effort. Filer et al. [15] expressed a long-term goal of unifying the optical control plane and pointed out the challenges in properly controlling the plurality of optical source and line system options. They recognized Yang model [7] and SNMP [10] as potential starting points for a standard data model and control interface. In contrast to previous works which only provided the preliminary idea, we demonstrate the feasibility of a centralized control of vendor-free optical networks by designing a standardized model for devices that abstracts away the vendor-specific details.

**Optical layer characterization.** Previous work [9, 14, 46, 47] characterized the dispersion (e.g., polarization mode dispersion, chromatic dispersion) of the deployed fiber cable. Our work complements these efforts by investigating similar phenomena (and more) for a much larger deployment. Ghobadi et al. [17] reported a three-month study of Q-factor data from Microsoft’s optical backbone and evaluated whether fiber segments can support higher-order modulations to increase network bandwidth. The following work RADWAN [39] pro-

vided a traffic engineering system that dynamically adapted link rates according to the SNR to enhance network throughput and availability. These works took advantage of one coarsely sampled indicator. In contrast, our work benefits from the fine-grained data and a centralized controller to support inter-device analysis to detect and troubleshoot optical events. We leave correlations of IP layer performance and optical events to future work.

**Diagnosis optical events.** Ghobadi et al. [18] studied Q-factor data from Microsoft’s optical backbone network and observed that network outages could be predicted based on the values drops in optical signal quality. RAIL [53] regarded RxPower as a key indicator of optical layer performance and found that instances of low Rx power could cause packet corruption. CorrOpt [52] used an optical layer monitor with Tx and Rx power to help determine the root cause of packet corruption in DCNs. These previous works adopted SNMP optical MIB [35] to poll optical performance indicators spanning from 5 minutes to 15 minutes. As a result, their works were slow in detecting persistent events and not capable of detecting ephemeral events. In contrast, OpTel is an optical telemetry system that supports one-second granularity optical data collection. Meanwhile, based on the signature learned from such fine-grained data, OpTel is able to detect and troubleshoot optical events in a timely manner.

## 6 Conclusion

This paper presents OpTel, an optical telemetry system that uses a centralized vendor-agnostic controller to collect optical data in a streaming fashion. More specifically, it offers flexible vendor-agnostic interfaces between the devices and the controller and offloads data-management tasks from the devices to the controller. As a result, OpTel enables the collection of fine-grained optical telemetry data at the one-second granularity. It has been running in Tencent’s optical backbone network for the past six months. Compared to existing telemetry systems, OpTel accurately detects 2× more optical events, half of which are ephemeral events. OpTel also enables troubleshooting of these optical events in a few seconds, which is orders of magnitude faster than the state-of-the-art.

*This work does not raise any ethical issues.*

## Acknowledgments

We sincerely thank our shepherd Manya Ghobadi, Walter Willinger, Gilberto Mayor, Kevin Schmidt, and the anonymous reviewers for their valuable feedback on earlier versions of this paper. We also thank teams at Tencent for their contributions to the work. Zekun He and Jilong Wang are corresponding authors. This work was supported in part by the National Key Research and Development Program of China under Grant No. 2020YFE0200500. Arpit Gupta was supported by NSF/Intel Partnership on Machine Learning for Wireless Networking Program under Award 2003257, “ML-WiNS: RL-based Self-driving Wireless Network Management System for QoE Optimization”.

## References

- [1] grpc: a high performance, open-source universal rpc framework. <https://grpc.io/>.
- [2] The need for otn in data center interconnect (dci) transport, 2016.
- [3] Inter-datacenter bulk transfers with codedbulk. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association.
- [4] ALIPOURFARD, O., MOSHREF, M., ZHOU, Y., YANG, T., AND YU, M. A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research* (2018), pp. 1–14.
- [5] 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 2010 Conference* (2010), pp. 63–74.
- [6] ARNOLD, T., HE, J., JIANG, W., CALDER, M., CUNHA, I., GIOSTAS, V., AND KATZ-BASSETT, E. Cloud provider connectivity in the flat internet. In *Proceedings of the ACM Internet Measurement Conference* (2020), pp. 230–246.
- [7] BJORKLUND, M., ET AL. Yang-a data modeling language for the network configuration protocol (netconf).
- [8] BOGLE, J., BHATIA, N., GHOBADI, M., MENACHE, I., BJØRNER, N., VALADARSKY, A., AND SCHAPIRA, M. Teavar: striking the right utilization-availability balance in wan traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 29–43.
- [9] BOHATA, J., JAROS, J., PISARIK, S., ZVANOVEC, S., AND KOMANEC, M. Long-term polarization mode dispersion evolution and accelerated aging in old optical cables. *IEEE Photonics Technology Letters* 29, 6 (2017), 519–522.
- [10] CASE, J., FEDOR, M. S., SCHOFFSTALL, M. L., AND DAVIN, J. Simple network management protocol (snmp). *RFC 1098* (1989), 1–34.
- [11] COX, J. Sdn control of a coherent open line system. In *Optical Fiber Communication Conference* (2015), Optical Society of America, pp. M3H–4.
- [12] DI VENTRA, M., PERSHIN, Y. V., AND CHUA, L. O. Circuit elements with memory: memristors, memcapacitors, and meminductors. *Proceedings of the IEEE* 97, 10 (2009), 1717–1724.
- [13] ENNS, R., BJORKLUND, M., SCHOENWAELDER, J., AND BIERMAN, A. Network configuration protocol (netconf).
- [14] FEUERSTEIN, R. J. Field measurements of deployed fiber. In *National Fiber Optic Engineers Conference* (2005), Optical Society of America, p. NThC4.
- [15] FILER, M., GAUDETTE, J., GHOBADI, M., MAHAJAN, R., ISSENHUTH, T., KLICKERS, B., AND COX, J. Elastic optical networking in the microsoft cloud. *Journal of Optical Communications and Networking* 8, 7 (2016), A45–A54.
- [16] GENG, Y., LIU, S., YIN, Z., NAIK, A., PRABHAKAR, B., ROSENBLUM, M., AND VAHDAT, A. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 549–564.
- [17] GHOBADI, M., GAUDETTE, J., MAHAJAN, R., PHANISHAYEE, A., KLICKERS, B., AND KILPER, D. Evaluation of elastic modulation gains in microsoft’s optical backbone in north america. In *2016 Optical Fiber Communications Conference and Exhibition (OFC)* (2016), IEEE, pp. 1–3.
- [18] GHOBADI, M., AND MAHAJAN, R. Optical layer failures in a large backbone. In *Proceedings of the 2016 Internet Measurement Conference* (2016), pp. 461–467.
- [19] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication* (2018), pp. 357–371.
- [20] 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 2013 Conference on SIGCOMM* (2013), pp. 15–26.
- [21] HONG, C.-Y., MANDAL, S., AL-FARES, M., ZHU, M., ALIMI, R., BHAGAT, C., JAIN, S., KAIMAL, J., LIANG, S., MENDELEV, K., ET AL. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 74–87.
- [22] HUANG, Q., SUN, H., LEE, P. P., BAI, W., ZHU, F., AND BAO, Y. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 404–421.
- [23] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ET AL. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.
- [24] JIN, X., LI, Y., WEI, D., LI, S., GAO, J., XU, L., LI, G., XU, W., AND REXFORD, J. Optimizing bulk transfers with software-defined optical wan. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication* (2016), pp. 87–100.
- [25] KACHRIS, C., AND TOMKOS, I. A survey on optical interconnects for data centers. *IEEE Communications Surveys & Tutorials* 14, 4 (2012), 1021–1036.
- [26] LABOVITZ, C., IEKEL-JOHNSON, S., MCPHERSON, D., OBERHEIDE, J., AND JAHANIAN, F. Internet inter-domain traffic. *ACM SIGCOMM Computer Communication Review* 40, 4 (2010), 75–86.
- [27] LAOUTARIS, N., SIRIVIANOS, M., YANG, X., AND RODRIGUEZ, P. Inter-datacenter bulk transfers with netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference* (2011), pp. 74–85.
- [28] 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), pp. 101–114.
- [29] MCQUISTIN, S., UPPU, S. P., AND FLORES, M. Taming anycast in the wild internet. In *Proceedings of the Internet Measurement Conference* (2019), pp. 165–178.
- [30] MILLS, D. *RFC1305: Network Time Protocol (Version 3) Specification, Implementation*. RFC Editor, 1992.

- [31] MIZUOCHI, T. Recent progress in forward error correction and its interplay with transmission impairments. *IEEE Journal of Selected Topics in Quantum Electronics* 12, 4 (2006), 544–554.
- [32] 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), pp. 129–143.
- [33] 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), pp. 85–98.
- [34] PAOLUCCI, F., SGAMBELLURI, A., CUGINI, F., AND CASTOLDI, P. Network telemetry streaming services in sdn-based disaggregated optical networks. *Journal of Lightwave Technology* 36, 15 (2018), 3142–3149.
- [35] PRESUHN, R., CASE, J., MCCLOGHRIE, K., ROSE, M., AND WALDBUSSEN, S. Management information base (mib) for the simple network management protocol (snmp). Tech. rep., STD 62, RFC 3418, December, 2002.
- [36] 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. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 407–418.
- [37] SADASIVARAO, A., JAIN, S., SYED, S., PITHEWAN, K., KANTAK, P., LU, B., AND PARASCHIS, L. High performance streaming telemetry in optical transport networks. In *Optical Fiber Communication Conference* (2018), Optical Society of America, pp. Tu3D–3.
- [38] SAEED, A., GUPTA, V., GOYAL, P., SHARIF, M., PAN, R., AMMAR, M., ZEGURA, E., JANG, K., ALIZADEH, M., KABBANI, A., ET AL. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 735–749.
- [39] SINGH, R., GHOBADI, M., FOERSTER, K.-T., FILER, M., AND GILL, P. Radwan: rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 547–560.
- [40] STONE, J., AND PARTRIDGE, C. When the crc and tcp checksum disagree. *ACM SIGCOMM computer communication review* 30, 4 (2000), 309–319.
- [41] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 233–248.
- [42] 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), pp. 453–456.
- [43] TILMANS, O., BÜHLER, T., POESE, I., VISSICCHIO, S., AND VAN-BEVER, L. Stroboscope: Declarative traffic mirroring on a budget. In *Proc. of NSDI* (2018).
- [44] WALLS, F. L., AND VIG, J. R. Fundamental limits on the frequency stabilities of crystal oscillators. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 42, 4 (1995), 576–589.
- [45] WOHLFART, F., CHATZIS, N., DABANOGLU, C., CARLE, G., AND WILLINGER, W. Leveraging interconnections for performance: the serving infrastructure of a large cdn. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 206–220.
- [46] WOODWARD, S., NELSON, L., FEUER, M., ZHOU, X., MAGILL, P., FOO, S., HANSON, D., SUN, H., MOYER, M., AND O’SULLIVAN, M. Characterization of real-time pmd and chromatic dispersion monitoring in a high-pmd 46-gb/s transmission system. *IEEE Photonics Technology Letters* 20, 24 (2008), 2048–2050.
- [47] WOODWARD, S. L., NELSON, L. E., SCHNEIDER, C. R., KNOX, L. A., O’SULLIVAN, M., LAPERLE, C., MOYER, M., AND FOO, S. Long-term observation of pmd and sop on installed fiber routes. *IEEE Photonics Technology Letters* 26, 3 (2013), 213–216.
- [48] YU, D., ZHU, Y., ARZANI, B., FONSECA, R., ZHANG, T., DENG, K., AND YUAN, L. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 207–220.
- [49] ZHONG, Z., GHOBADI, M., KHADDAJ, A., LEACH, J., XIA, Y., AND ZHANG, Y. Arrow: Restoration-aware traffic engineering.
- [50] ZHOU, Y., SUN, C., LIU, H. H., MIAO, R., BAI, S., LI, B., ZHENG, Z., ZHU, L., SHEN, Z., XI, Y., ET AL. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 76–89.
- [51] 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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 479–491.
- [52] ZHUO, D., GHOBADI, M., MAHAJAN, R., FÖRSTER, K.-T., KRISHNAMURTHY, A., AND ANDERSON, T. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 362–375.
- [53] ZHUO, D., GHOBADI, M., MAHAJAN, R., PHANISHAYEE, A., ZOU, X. K., GUAN, H., KRISHNAMURTHY, A., AND ANDERSON, T. {RAIL}: A case for redundant arrays of inexpensive links in data center networks. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 561–576.

## A The origins of telemetry data collected from optical device

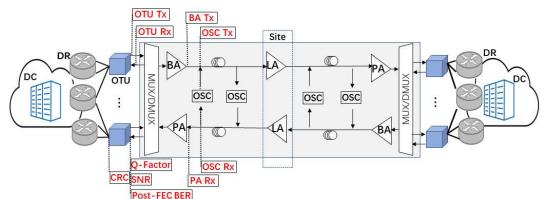


Figure 17: The origins of telemetry data collected from optical device

## B Filtering out the network events from trouble tickets dataset related to optical events

Since the tickets in trouble ticket dataset describe whole network events, each ticket contains a timestamp that records the start time of the network event and the detailed alarm message and corresponding a timestamp recording the end time of the event with the causes of the failures. After manually reviewing a number of tickets, we observed that most optical events

had been saliently described in the trouble tickets. Filtering out and grouping these tickets requires a lot of effort. We design a two-layer filtration. Specifically, in the first layer, we filter out the trouble tickets related to the optical backbone network by matching keywords, phrases, and regular expressions to get a set of optical trouble tickets. In the second layer, by manually reviewing the optical trouble tickets, we observe that the optical events can be categorized into a small number of classes, i.e., fiber cable, hardware and power event. We classify these tickets based on matching keywords or phrases. In some instances, there may be multiple tickets pertaining to the same failure event. Grouping these multiple tickets into a single event requires some piece of information to be repeated in each ticket.

## C Data collection point of power event.

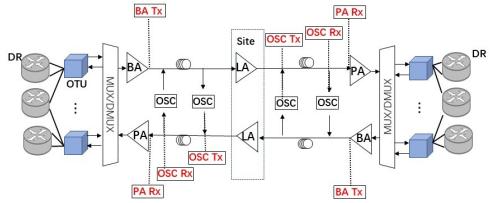


Figure 18: Schematic diagram of data collection

The performance data shown in 16(a) is collected from the top part of Figure 18, and the performance data shown in 16(b) is collected from the bottom part of Figure 18. Note, the LAs in the site share the electrical power sources.



# Bluebird: High-performance SDN for Bare-metal Cloud Services

Manikandan Arumugam<sup>1</sup>, Deepak Bansal<sup>3</sup>, Navdeep Bhatia<sup>1</sup>, James Boerner<sup>3</sup>, Simon Capper<sup>1</sup>, Changhoon Kim<sup>2</sup>, Sarah McClure<sup>3</sup>, Neeraj Motwani<sup>3</sup>, Ranga Narasimhan<sup>3</sup>, Urvish Panchal<sup>1</sup>, Tommaso Pimpo<sup>3</sup>, Ariff Premji<sup>1</sup>, Pranjal Shrivastava<sup>3</sup>, and Rishabh Tewari<sup>3</sup>

*Arista<sup>1</sup>, Intel<sup>2</sup>, Microsoft<sup>3</sup>*

## Abstract

The bare-metal cloud service is a type of IaaS (Infrastructure as a Service) that offers dedicated server hardware to customers along with access to other shared infrastructure in the cloud, including network and storage.

This paper presents our experiences in designing, implementing, and deploying Bluebird, the high-performance network virtualization system for the bare-metal cloud service on Azure. Bluebird's data plane is built using high-performance programmable switch ASICs. This design allows us to ensure the high performance, scale, and custom forwarding capabilities necessary for network virtualization on Azure. Bluebird employs a few well-established technical principles in the control plane that ensure scalability and high availability, including route caching, device abstraction, and architectural decoupling of switch-local agents from a remote controller.

The Bluebird system has been running on Azure for more than two years. During this time, it has served thousands of bare-metal tenant nodes and delivered full line-rate NIC speed of bare-metal servers of up to 100Gb/s while ensuring less than 1μs of maximum latency at each Bluebird-enabled SDN switch. We share our experiences of running bare-metal services on Azure, along with the P4 data plane program used in the Bluebird-enabled switches.

## 1 Introduction

For some time now, Software Defined Networks (SDNs) have been foundational in enabling virtualized networks for customer workloads in multi-tenant clouds. Traditionally, the data plane of SDNs has been implemented in software as part of the end-host networking stack, typically leveraging virtual switches in hypervisors such as Open V-Switch (OVS) [16] or user-level networking libraries such as DPDK [17]. Given that scale and performance needs have grown over the years, the mechanisms available to offload such software-based packet processing have also evolved. These include solutions such as smartNICs [15], which leverage Switch-on-a-Chip (SoCs), ASICs, and FPGAs to perform packet processing at line rate without incurring significant overhead [58–60].

Today, cloud customers have even more demanding workloads in the cloud as they look to migrate their line-of-business applications and begin to phase out their own data centers.

These workloads require complete control of the hardware, and in many cases, custom hardware to be hosted in the cloud. For example, workloads such as those for NetApp, Cray, SAP, and HPC [13, 53] require custom hardware. We refer to the cloud offering for supporting such workloads as bare-metal cloud services or hardware as a service (HWaaS).

Bare-metal workloads are not well-supported by traditional SDN stack implementations. In general, bare-metal servers do not offer the necessary opportunities for integration with the networking stack on the host or NIC, calling instead for a "bump-in-the-wire" approach that has no dependency on the host hardware. Since the Top-of-Rack (ToR) switches are the first network hop connected to these hosts, the ToR offers an excellent opportunity to implement this "bump in the wire."

In this paper, we introduce Bluebird, a ToR-based SDN solution that is broadly deployed in one of the largest public cloud infrastructures to enable bare-metal workloads. We also discuss the challenges, design, and operational experiences in building and designing such a solution.

Bluebird is based on programmable ASICs such as the now largely available Barefoot Tofino chipset [27] as well as upcoming merchant silicon offerings like Broadcom's SmartTOR ASIC [6]. The programmability of high-speed networking ASICs, along with the increase in scale, have made the ToR-based "bump-in-the-wire" approach feasible. Cloud providers must be able to evolve the SDN capabilities of the platform as customer requirements change. Without programmable chips, a cloud provider may have to wait for an 18-24 month technology cycle before changing their service offerings. Unless, of course, the cloud provider undertakes an off-cycle, expensive hardware replacement. Additionally, several SDN functions, such as load balancing, NAT, etc., require flow state tracking. ToR ASICs such as Barefoot Network's Tofino and Broadcom's SmartToR are now able to track millions of flows, which is critical to enable network virtualization in a ToR.

Bluebird is able to achieve line-rate throughput and deliver latencies of less than 1μs that are on par with non-virtualized environments. By leveraging route caching mechanisms, Bluebird can scale to the largest virtualized networks that exist in a public cloud. The rest of this paper is organized as follows: §2 reviews different SDN implementations; §3 describes goals and rationales behind Bluebird; §4 presents the design at the

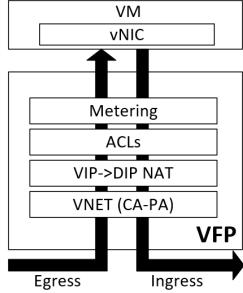


Figure 1: Virtual Filtering Platform (VFP) design.

base of our solution; §5 investigates performance; §6 explores operationalization and experience; §7 discusses related work; and §8 concludes this paper.

## 2 Background

In this section, we present an overview of the SDN stack implementations coexisting within Azure and how they compare with Bluebird to enable bare-metal services. Table 1 summarizes the comparison.

### 2.1 Host SDN

First, we consider end-host based software solutions. Figure 1 shows the Host SDN model, where the SDN software stack runs on the host machine hypervisor. Packets to and from the VMs are processed by a programmable virtual switch (vSwitch) operating within the hypervisor environment. The vSwitch’s design is critical to the effectiveness of this solution since it is responsible for implementing the forwarding policy while still minimizing overhead. Since the host has already processed every packet reaching the physical forwarding layer, the physical networking equipment can be relatively simple. However, processing packets in software is expensive and competes with client software for host resources. This results in reduced revenue and has a negative impact on network efficiency as congestion increases CPU use [15].

Using Hyper-V as the hypervisor and the Virtual Filtering Platform (VFP) [14] as the vSwitch, Azure primarily employs the Host SDN model across its fleet (Figure 1). The VFP implements SDN policies by acting as a programmable platform accessible to the controllers running Azure’s SDN. The VFP is organized in layers, which are stateful flow tables that enforce the controller’s policy. Each layer implements a specific set of inbound and outbound rules that can filter and transform packets. A packet traverses the layers in order, matching one rule per layer by searching by rule priority.

Figure 1 shows a common layer configuration. Following the inbound order, the Virtual Network (VNET) layer provides tunneling for packets coming from Customer Address (CA) space to the Physical Address space (PA). Inbound packets are decapsulated while outbound packets are encapsulated.

The next layer is the Ananta [61] NAT load balancer layer, which NATs inbound packets from a Virtual IP (VIP) to a Direct IP (DIP). The Access Control Lists (ACLs) layer is a stateful firewall, while the metering layer is for billing and is placed as the last layer between the VFP and the VM.

While the Host SDN model is used widely in Azure, it is not well-suited for bare-metal workloads. Such workloads are not Hyper-V based, leaving no environment to implement the VFP. Furthermore, the network performance is expected to be on par with deployments in non-virtualized environments, precluding a software-based solution.

### 2.2 SmartNIC-based SDN

A smartNIC is a programmable network interface card (NIC) that supports the network processing operations usually performed by the host CPU. SmartNICs can be configured for both control plane and data plane operations. They can make use of various technologies depending on the requirements: Application-specific Integrated Circuit (ASIC), System-on-chip (SoC), and Field-programmable Gate Arrays (FPGA).

Compared to VFPs, smartNICs offer lower latency and higher throughput while maintaining the same scalability and programmability level. This allows the host CPU to offload some costly network operations, resulting in reduced processing time and improved power efficiency. However, in some cases, only a subset of tasks is offloaded to the smartNIC, which in itself requires careful orchestration between the hypervisor and the smartNIC operating system.

While smartNICs have proven to be effective overall, they are not a good fit for the bare-metal model given the integration requirements. This is mostly due to the complexities that would be introduced at the hypervisor and network stack of the host. Additionally, the SDN stack implementation should be decoupled from any particular or specialized bare-metal appliance to ensure that a single, general approach will be compatible with all bare-metal services.

### 2.3 SDN on ToR

Traditionally, data centers are built using fixed-function switches. In such environments, the cloud intelligence resides in the host or hypervisor and not in the network. The host uses smartNICs and VFPs to define a clear separation between the control and data plane.

SDN on ToR refers to the use of programmable switches to execute policy on the ToR switch instead of the host. In order to use a programmable ToR, the target function(s) must be well-defined and limited in scope. A well-targeted SDN ToR application can reduce development time and complexity.

Most, if not all, SDN policies could be supported by programmable switches. However, we have taken an incremental approach where only a small subset of features is initially introduced. This set will be expanded once it has met certain

	<b>End-host software stack per core (§2.1)</b>	<b>SoC-based smart-NIC (§2.2)</b>	<b>ASIC-based smart-NIC (§2.2)</b>	<b>FPGA-based smart-NIC (§2.2)</b>	<b>Programmable ASIC-based ToR (Bluebird) (§2.3)</b>
Max Throughput	< 40Gbps & 10's of Mpps	Up to ~100Gbps & ~100Mpps	Up to 200Gbps & 100-200Mpps	Up to 200Gbps & 100-200Mpps	6.4-12.8Tbps & 5-7Bpps
Latency	< 100 $\mu$ sec	> 1 $\mu$ sec	> 1 $\mu$ sec	> 1 $\mu$ sec	< 1 $\mu$ sec
Scale	GBs of DRAM + traditional cache hierarchy	8 GBs of DRAM + traditional cache hierarchy	Tens of MBs on chip cache + GBs of DRAM	Tens of MBs on chip BRAM + GBs of DRAM	12 stages of high-throughput pipeline and each pipeline has 24 TCAMs and 80 SRAM blocks
Cost per 100Gbps	Medium (including capex and opportunity cost)	Medium	Medium	Medium	Low
Power consumption per 100Gbps	~500-700W per server including the NIC	~500-700W per server including the NIC	~500-700W per server including the NIC	~500-700W per server including the NIC	~300W for the system that includes 64 ports of 100GbE (~5W/100GbE)

Table 1: Comparison of SDN stack implementations.

standards of quality and reliability. In Bluebird, the primary objective is for the ToR to maintain a high number of CA-to-PA mappings while ensuring hardware-like performance.

An SDN ToR gives us the ability to collapse multiple functions into one network element. In the Bluebird model, we collapse two key roles into a single device: 1) logical network isolation between customers via Virtual Routing and Forwarding (VRF) instances and 2) the association of one or more CA-to-PA mappings per VRF per customer. Implementing these two functions separately on different devices (such as routers implementing VRFs with tunnels to servers running software gateways) incurs the cost of routers and additional servers. By collapsing the routing and CA-to-PA mapping tasks onto a single device, we reduce the number of hops, encapsulations, and consequently latency for bare-metal workloads. Implementing these functionalities directly on the SDN switch results in increased performance and scalability.

To implement an SDN ToR for use with Bluebird, a single VRF is allocated per customer to guarantee logical isolation between customers. Since the goal of the SDN ToR is to connect a customer’s bare-metal instance to their VNET, each VRF is programmed with CA-to-PA mappings in the form of VXLAN [48] static routes that associate the bare-metal host to its VNET. Customized P4 programming is used to perform the necessary encapsulation for packets destined to the VNET. This allows the communication between bare-metal (BM) and VMs and between BM and BM to happen at hardware speeds. Additionally, the number of programmable routes is extended through an onboard cache that increases the otherwise limited number of routes the switch ASIC on-chip memory can hold. The route-cache solution is discussed in more detail in §4.

## 2.4 SDN Servers

The bump-in-the-wire method could have also been approached by assigning dedicated ports on a custom DPDK-enabled SDN server attached to the bare-metal appliance, making use of DPDK-enabled smartNICs on this server to

perform the bump-in-the-wire function. However, based on the power consumption data in the Table 1, one can deduce that dedicated servers that carry out bump-in-the-wire operations make the power overhead a non-starter. The performance and scale that an SDN ToR offers in a <500W power envelope makes a strong case for using a dedicated SDN ToR.

## 3 Design Goals and Rationale

In developing an SDN solution for bare-metal workloads in Azure, we had the following objectives:

### 1. Programmability

SDN for bare-metal workloads needs to be able to evolve along with the rest of the SDN stack. The VFP [14] model enables many configurable virtual network features. As requirements and policies change over time, SDN for bare-metal should maintain interoperability with the existing stack. This is achieved through the ToRs’ programmability which provides control at every stage of packet processing.

### 2. Scalability

The most significant disadvantage of SDN on ToR compared to host implementations is the limited scale. Memory linearly scales with the number of hosts. Consequently, route capacity can quickly become a bottleneck in resource-limited ToRs. Accordingly, we developed a cache system that extends the hardware capacity of our ToRs and allows us to meet our scalability and performance requirements.

### 3. Latency and throughput

Bare-metal workloads typically demand high bandwidth, low latency, and deterministic behavior. To meet these requirements, we have used programmable high-speed network ASICs since they offer consistent latency, high throughput, and sustained performance.

### 4. High availability

To avoid customer impact due to hardware failure or maintenance, the SDN for bare-metal solutions must have high

availability. To support this requirement, we designed redundancy into Bluebird as described in §6.

#### 5. Multitenancy support

Azure supports a large number of customers and tenants who have the ability to create, modify, and delete virtual networks rapidly. When supporting multitenancy, isolation is critical for providing an experience indistinguishable from dedicated networks and servers.

#### 6. Minimal overhead on host resources

With the VFP model, VMs running on the host compete with the SDN stack for hardware resources. The introduction of AccelNet [15] and FPGA-based smartNICs has significantly reduced the overhead, but the VFP still stays on the host to process the first packet in the flow. With bare-metal workloads, customers expect the performance to be similar to that of direct access to the underlying hardware.

#### 7. Seamless integration

Bare-metal workloads run on a wide array of architectures and operating systems. Integrating a new workload in the Azure network should be possible without change to the bare-metal server. Bluebird decouples the workload architecture from the SDN stack and enables consistent virtualization of a diverse set of bare-metal workloads.

#### 8. External network access

Given that bare-metal hosts may require Internet or external network access, a form of Network Address Translation, directly available on the SDN ToR, should be supported.

#### 9. Interoperability

As we introduce programmable ToRs to support bare-metal workloads, these ToRs need to transparently operate with the existing SDN stack to ensure communication between the physical and virtual address space. Interaction with the VFP §4 is of primary importance to realize a heterogeneous system like the one proposed in this paper.

## 4 System Design

In network system design, there is often a trade-off between the cost of a device, its memory (or route capacity), and features intrinsic to the Network Processing Unit (NPU) or ASIC itself. Internet core routers are generally feature-rich, designed to support large route tables, able to move large amounts of network traffic ( $>30$  Tbps), and usually quite expensive. Alternatively, the ToRs in data centers are cost-effective but have fewer features and support smaller routing tables. In our design, we needed an SDN ToR with a reasonably large VNET routing table but with the cost efficiencies of a typical data-center-class ToR. Bare-metal hosts would connect directly to such SDN ToRs and use a specialized route table to communicate with Azure VMs in a virtualized address space.

Before Bluebird was introduced, Azure supported on-prem bare-metal to VNET connectivity through a software-gateway model. In this model, traffic is encapsulated on a router and

forwarded to one or more software-gateways. The software gateways, implemented on standard servers, hold large numbers of CA-to-PA mappings associating an on-prem customer to their VNET. However, with the introduction of the workloads for NetApp, it became clear that the gateway model would not meet the throughput and performance requirements. The NetApp bare-metal service required at least 240Gbps of throughput with a latency ceiling of no more than 4ms. With this in mind, we decided to adopt the SDN ToR model and program the CA-to-PA VNET routes directly onto the ToR, avoiding the software gateway altogether. This improves the throughput as it is now limited only by the throughput of the SDN ToR, which in Bluebird’s case is 6.4Tbps.

A considerable effort was put into the planning of how on-chip resources had to be arranged in the pipeline to meet our needs. In particular, we had to decide how to allocate the on-chip switch memory to maximize the number of CA-to-PA mappings which are represented in the VXLAN Tunnel Endpoints (VTEP) resource table. Using Tofino’s P4 programmable pipeline, we reduced the IPv4 and IPv6 unicast route table size and significantly increased the VXLAN VTEP table scale from 16K to 192K entries. The ability to support 192K CA-to-PA mappings offered greater flexibility in customer address choices since many specific routes (/32 IPv4 or /128 IPv6 routes) could be used to point to customer VNETs rather than limiting to a smaller number of aggregate routes. In order to make the design future proof, we gave ourselves the maximum allowable table space on the chip in the event that more specific routes were dominant.

The reduction in IPv4 and IPv6 table space also allowed us to extend the VXLAN Network Identifier field space as well as add support for an IPv6 underlay. The custom P4 programmability proved to be extremely valuable in helping us achieve our scale objectives.

While the P4 profile we implemented to give the SDN ToR a large VXLAN VTEP table was adequate when we launched the service, we had to start planning for growth. The VXLAN VTEP route capacity of the ToR was further enhanced with the introduction of a route cache system. The route cache mechanism allowed the VXLAN VTEP table capacity to grow beyond the hardware limit of 192K entries. The caching solution is described later in this section (§4.4). P4 programming flexibility also allowed for other quick packet header manipulations that helped in the rapid development of this service, namely overwriting the inner Ethernet header with the destination VM’s MAC address while modifying the VXLAN UDP source port to a custom value.

In the remainder of this section, we discuss the packet flow, related packet transformations, and the control plane design.

### 4.1 Packet Flow

In this section we provide an in-depth discussion on the packet flow. In order to achieve customer isolation at a logical

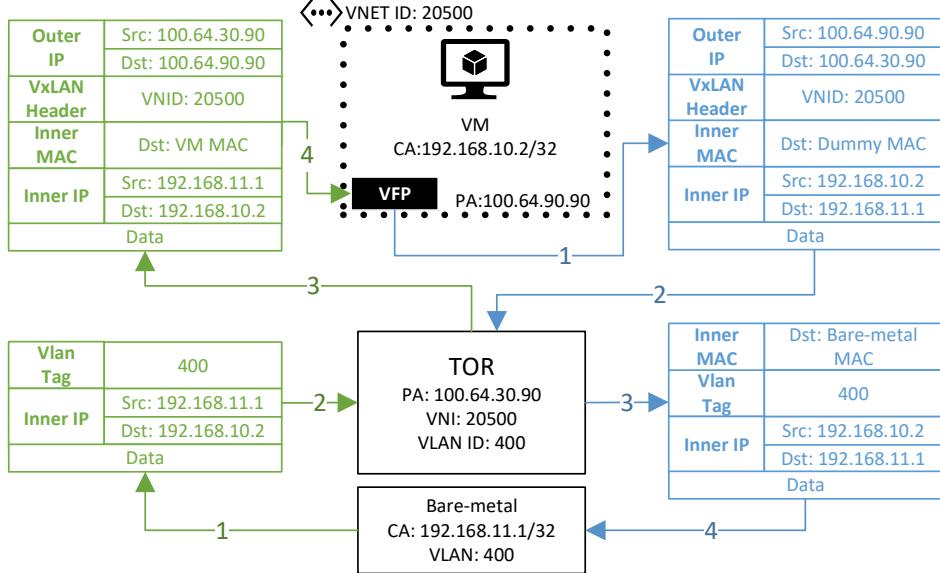


Figure 2: Packet flow between a VM and a bare-metal server.

layer within a common fabric, we identify each customer’s VNET by a Virtual Network Identifier (VNI) associated with a unique Virtual Routing and Forwarding (VRF) instance.

**Bare-metal to VM.** When a bare-metal server sends a packet to an Azure VM, a VLAN tag is added to the outbound packet. The ToR then receives the packet on the virtual interface specified by the VLAN tag. Each interface on the ToR has an associated VRF configured to route the packet to the customer VM. The routes in the VRF associate a VM’s IP with the routable IP of the host containing the VM, the VNI of the VNET, and the MAC address of the VM. When the packet reaches the destination host, the VFP decapsulates the packet and uses the MAC to switch the packet to the correct VM. This flow is explained with an example below.

In the green flow shown in Figure 2 at point (1), the bare-metal server sends a packet to the VM through the VLAN 400 interface. The packet is then received on the associated interface on the server’s ToR at (2). On the ToR, VLAN 400 is configured to be associated with the VRF 20500, which contains the routes programmed by Bluebird to route the packet to the VM. At (3), the ToR rewrites the inner destination MAC with the VM’s MAC contained in the VRF. At (4), the ToR, configured to use the loopback interface as the VXLAN source interface, encapsulates the original frame in a VXLAN frame containing its own PA 100.64.30.90 (loopback IP address) as outer source IP and the VM’s host PA 100.64.90.90 as the outer destination IP.

**VM to Bare-metal.** When an Azure VM sends a packet to a bare-metal server, a Bluebird-provisioned rule instructs the VFP on the host to encapsulate the Ethernet frame in a UDP datagram containing the customer VNI and the IP of the bare-metal’s ToR as the destination VTEP IP. The SDN ToR decapsulates the packet and uses the VNI to identify the VRF.

Outer IP	Src: 100.64.30.90
	Dst: 100.64.90.90
VxLAN Header	VNI: 20500
Inner MAC	Dst: VM MAC
Inner IP	Src: 192.168.11.1
	Dst: 192.168.10.2
Data	

Inner MAC	Dst: Bare-metal MAC
Vlan Tag	400
Inner IP	Src: 192.168.10.2
	Dst: 192.168.11.1
Data	

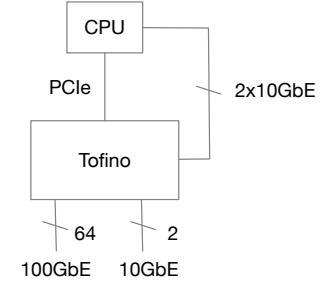


Figure 3: Front panel and CPU interfaces in an Arista 7170.

Within the VRF, a route lookup is performed to identify the next-hop to which the packet is subsequently be forwarded.

In the blue flow in Figure 2, a VM sends a packet to the bare-metal server where the inner source and destination IPs are respectively set to the VM’s CA 192.168.10.2 and the server’s CA 192.168.11.1. The destination MAC is set to a dummy value. In the VFP (1) the packet is encapsulated in a VXLAN frame containing the VM’s VNI 20500, the outer source IP pointing to the host PA 100.64.90.90, and the PA 100.64.30.90 of bare-metal’s ToR as destination IP. At (2), the bare-metal’s ToR receives the packet and decapsulates it. The virtual network identifier is used to find the VRF associated with the customer virtual network. At (3), the switch learns the destination MAC through ARP and adds a VLAN tag pointing to the configured VLAN interface 400, and at (4), the packet is routed to the bare-metal server.

## 4.2 Platform Selection

Using a switch ASIC with a programmable P4 pipeline, we were able to quickly prototype packet formats that would interoperate with Azure’s VFP. Additionally, based on the initial requirements, we needed support for at least 192K CA-to-PA mappings. A variety of silicon offerings could meet most of our requirements at the time, but the scale of CA-to-PA mappings pointed our investigation towards (Intel) Barefoot Network’s Tofino-1 chipset.

The Tofino-1 is a 6.4Tbps single-chip solution with 12 programmable stages, 256x25/10G SerDes, and a software-defined P4 packet processing pipeline. On the Arista 7170 switch, the Tofino-1 is coupled with a Quad-core 2.2GHz Intel Pentium CPU with additional 2x10GbE ports from the switch ASIC wired directly into the CPU (as shown in Figure 3).

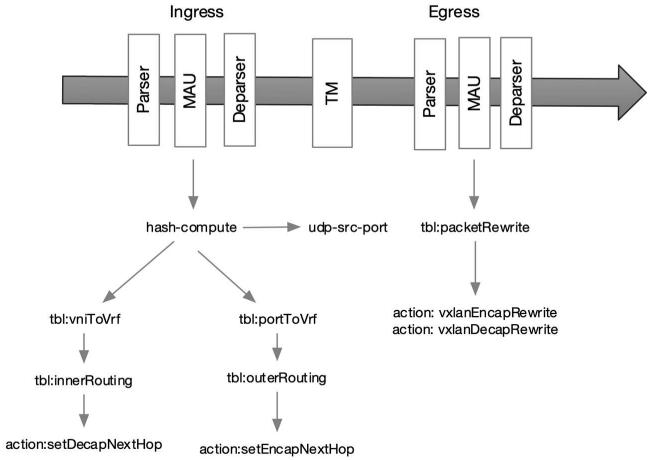


Figure 4: P4 programming pipeline.

These additional ASIC-to-CPU 10GbE ports provide a fast path for route-cached packets to be processed on the CPU.

### 4.3 P4 Pipeline Design

P4 facilitated rapid prototyping and quick iteration in finding a balance between desirable features and the available chip resources. For instance, while the P4 programming needed to create CA-PA mappings was easy to define, special considerations had to be given to whether the underlay would be IPv4 or IPv6. In a simplistic model, having an IPv6 underlay would significantly reduce the number of CA-to-PA mappings since the entries in the forwarding table would need additional resources to support IPv6 PA destinations. However, with our custom P4 pipeline, we were able to completely decouple the underlay route scale from the overlay route scale thus giving us the maximum number of CA-to-PA mappings independent of whether the underlay used is IPv4 or IPv6.

To describe the packet transformations used in our flows, we used the P4 programmable pipeline on Tofino, which consists of an ingress and egress block (see Figure 4), with each block comprising a sequence of sub-blocks: Parser, Match-Action-Unit (MAU), and Deparser. The Parser block parses the packet and extracts the relevant headers. After that, the MAU performs table lookups and manipulates the packet. The Deparser then reassembles and sends the packet to the Traffic Manager (TM). Finally, packet queuing, replication, and scheduling are done by the TM. A part of the P4 program [1], which illustrates one of the inner-MAC rewrite transformations implemented in our pipeline, is open-sourced.

In the sample P4 program shared [1], the parser excludes regular IP packets and keeps the VXLAN encapsulated packets. After each packet is decapsulated, a route lookup is done on the inner destination IP which determines the action to be taken and the data to be rewritten. In the egress logic, several fields, such as the inner-MAC address, are rewritten based on the bridged metadata received from the ingress pipeline.

```
ip route vrf VNET-A
192.168.10.2/32 vtep 10.100.2.4 vni 20500
router-mac-address 00:12:23:45:A2:9F
```

Figure 5: Static VXLAN route configuration on ToR

While the P4 example provided gives the reader a high-level view of the flexibility available in packet manipulation, the actual P4 code used for this service is more intricate and optimized, allowing for 192K mappings and additional features.

The flexibility that P4 provides allowed us to give the ToR a ‘personality’ based on the application. A ToR can be preloaded with a custom P4 profile, making the ToR suitable for a given application. For example, we use a ‘bare-metal’ profile when the ToR is used for Bluebird workloads and a ‘NAT-profile’ when source NAT is required. Each profile results in a different P4 program getting activated in hardware.

The rest of the packet transformations are presented in detail in the remainder of this section.

**Inner Destination MAC Rewrite.** When a BM sends traffic to a VM, the receiving hypervisor’s VFP uses the inner destination MAC (DMAC) to forward the packet to the destination VM. If the DMAC is unknown, the VFP drops the packet. For this reason, the CA-to-PA routes have the form of static routes extended with additional fields. An extended route, defined on the SDN ToR, contains the VTEP as the PA address, the VNI for the VNET, and the destination MAC of the VM. When a packet matches a route, the ToR overwrites the DMAC with the MAC of the VM. For example, the route in Figure 5 points to a VNET VM with MAC 00:12:23:45:A2:9F at address 192.168.10.2 residing on a VXLAN with VNI 20500 and reachable host (PA) at 10.100.2.4.

**Limiting the Range of the VXLAN Source Port.** In a traditional VXLAN, the ToR imposes a VXLAN UDP source port value derived from the incoming packet’s entropy. The source port is calculated per packet. This is done to help with hash-based ECMP load-balancing schemes employed by network chipsets, ensuring that VXLAN packets are properly load-balanced across the network. To aid the VFP on the VM host in identifying BM-sourced VXLAN packets, we limit the range of source port values usable by the SDN ToR. Limits in imposable ports can be set with a simple CLI command.

**Inner Destination MAC Masking.** We also needed customization on the SDN ToR to ignore the inner destination MAC address arriving in the VXLAN packet. This is in the direction of the VNET to the SDN ToR, where the inner destination MAC address is usually resolved by an ARP exchange between two VXLAN hosts. Specifically, an ARP request/reply exchange would have to take place, ensuring that the end-hosts are aware of each other’s MAC addresses. The entire ARP resolution step can be skipped if the SDN ToR absorbs all packets regardless of the DMAC value, as described in packet flow in §4.1. The VFP transmitter on the VNET simply writes a bogus inner DMAC value in the

packet destined to the SDN ToR. The SDN ToR is made to ignore this bogus MAC and proceeds to route all the received frames regardless. The SDN ToR accomplishes this task using a wildcard bitmask applied using a CLI command.

#### 4.4 Route Cache

Although we were able to make room for additional mappings using a custom P4 pipeline, we were faced with the challenge of increasing the scale beyond what the chip hardware could support. At this point, we had a working P4 pipeline that was programmed to fit 192K CA-PA mappings, support an IPv6 underlay, and offer 1:1 static NAT to BM hosts.

As the number of bare-metal customers grew, we realized that the 192K CA-PA upper-limit of the Tofino would soon become a bottleneck. We considered a few alternatives, one of which was to quickly onboard the next-generation Tofino (Tofino-2), which was known to support up to 1.5M CA-PA mappings in hardware. However, we ended up pursuing the route cache feature since it would be valuable regardless of the scale of the underlying Tofino ASIC. Route caching gives us a five-fold increase over the original 192K mappings. The route cache feature is an implementation of the familiar statistical multiplexing model whereby a significantly higher number of customers can share a finite resource, as long as not all customers are active at the same time. In other words, if we know that our customers are not always using all available hardware entries, we can reassign those unused entries to other active users. A primary enabler for the route cache concept is the way Bluebird provisions SDN entries. Bluebird does not preconfigure the SDN ToR but instead dynamically provisions mappings as needed allowing the route cache logic to continuously determine which entries are to remain in hardware or moved to software.

Before describing the route cache feature, it is important first to understand the Software Forwarding Engine (SFE). The SFE is a DPDK-enabled packet processing function provided by the ToR's CPU which has a packet forwarding rate of 200K pps. CPU bound packets use the 2x10GbE interfaces connecting the Tofino to the CPU to reach the SFE. A software agent on the ToR monitors hardware entries on the switch ASIC and moves inactive entries to the SFE. SFE resources are CPU and memory bound. With 16GB of memory, up to 600K mappings can be stored in the SFE, whereas 1.5M mappings can be stored with at least 32GB of memory. The number of mappings compared to the total memory is low because the memory also supports the switch operating system, i.e., EOS (Extensible Operating System). The portion of memory used by the SFE is relatively small: about 3GB out of the available 32GB total memory is used to store mappings. The rest of the memory is used for running the operating system and other agents such as the routing agent, platform agent, etc.

Bluebird configures static VXLAN routes/mappings and

Threshold level	Utilization	Idle time
low	85%	1100s
medium	90%	300s
high	95%	100s

Table 2: Default caching thresholds.

specifies whether the route/mapping is cacheable or not. The mapping itself is programmed by Bluebird using a JSON RPC call as described later in this section. A mapping is identified as active if there was a packet that recently used the prefix programmed in the VNET route. We will use the concept of a ‘hitbit’ to note when a route entry is touched by a packet, either in software or hardware. EOS then maintains this hitbit for all hardware and software (SFE) entries. The hardware hitbit is triggered when mappings are used for hardware-based forwarding. A SFE hitbit is triggered when a packet takes the software DPDK path, indicating that this software mapping now needs to be upgraded from the SFE to hardware.

To select the mappings that are evicted from the hardware, we use a Least Recently Used (LRU) eviction algorithm. LRU entries are found by polling the hitbit property maintained in the hardware for each prefix. The flows going through the SDN ToR are monitored using an age-based idle timer. No packet state is maintained, nor are the packets examined. We considered other hardware eviction algorithms but ultimately rejected them. These included 1) tracking TCP flows and 2) tracking flows that carried the most traffic volume.

These options were rejected because tracking TCP flows is expensive from two perspectives: 1) the need to send packets containing TCP flags S, F, R to an agent on the switch for tracking purposes, and 2) flows are expensive to store as the access key would comprise of the source IP/port and destination IP/port. For routing purposes, the switch only needs the destination IP and the added flow state becomes unnecessary overhead.

We decided to implement the idle-timer based approach since it was simpler and it met our requirements. The idle timer based approach is the simplest and most efficient because it does not require tracking individual flows or state. In the future, other algorithms may be considered as we learn more about customer traffic patterns.

In the CLI, we can specify CA-PA mapping entries as cache candidates. All candidate mappings become a part of the route cache, which means that these prefixes can be downgraded to software if they become inactive and can be upgraded to hardware if they become active. The aging time of hardware routes and how many of these entries remain in hardware can be configured as a percentage of total hardware capacity. Finally, the default threshold values are listed in Table 2.

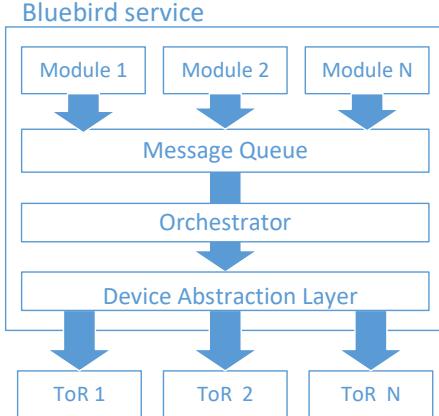


Figure 6: Bluebird service structure.

## 4.5 Control Plane and Policy Provisioning

In the Azure host SDN, a controller running on each host programs the VFP. However, since this is no longer an option for the bare-metal workloads, two alternatives for hosting SDN logic were considered; an agent on the ToR or programming the ToR via an external service. We rejected the idea of having an agent on the ToR since it did not meet our performance goals. The agent would compete for resources with latency-sensitive operations running on the ToR, and resource consumption could increase as requirements changed over time. On the other hand, an external service does not need immediate proximity to the ToR because configuration on a ToR has less stringent latency requirements than typical data plane operations. Moreover, a separate service also has advantages in terms of fault tolerance and deployment time. As a result, we introduced the Bluebird Service (BBS) (Figure 6).

### Provisioning SDN Policy on ToR

BBS is a lightweight, multi-tenant, stateless microservice that configures the ToRs with virtual network policies for each customer (Figure 6). Each policy is represented as a JSON object that specifies a ToR’s desired state, called the goal-state. Azure SDN services send their goal-states to BBS’ message queue leading to the orchestrator module which parses and consolidates the goal-states into ToR configurations. Configurations are then transferred to the device abstraction layer (DAL), which keeps the SDN business logic independent from ToR implementations. In the DAL, they undergo a conversion process that results in a sequence of commands for the targeted ToR. The list of allowed commands is strictly limited to the operations needed for bare-metal provisioning, reducing the possibility of interference with other automation systems responsible for software upgrading or traffic shifting. In the case of the Arista 7170, BBS uses the JSON-RPC 2.0 protocol over HTTPS. Each JSON payload contains an ordered list of commands using Arista EOS CLI syntax.

BBS performs a sync-check on all ToR configurations at defined intervals. At every sync, it calculates the delta between a

ToR’s configuration and its target configuration and performs a reconciliation in case of differences. Each configuration request is atomic, and configurations are versioned to avoid inconsistent states due to out-of-order execution. BBS also ensures state consistency between multiple ToRs if they are part of a high availability network configuration (see §6.1) in which they are seen as one logical entity.

To prevent resource exhaustion due to extremely large VNets generating a high number of routes, BBS limits the number of programmable routes per VNET. The default limit is maintained as a function of the ToR’s capacity. When more routes are needed, the limit can be raised to match the requirements and, in some cases, customers are migrated to dedicated ToRs.

Azure regions protect from failures through increased redundancy. Each region is subdivided into several distinct physical locations called availability zones (AZs). Each zone is made up of one or more data centers (DCs) equipped with independent power, cooling, and networking. BBS is deployed per AZ within an Azure Service Fabric [33] ring organized as a series of active and inactive instances. Additionally, BBS’s scope is not limited to a single AZ and can target any AZ within the same region. However, this is limited only to scenarios in which another AZ is severely impacted and the local BBS is unable to function.

## 5 Performance

Over the past two years, Azure has been deploying bare-metal services on SDN-ToRs in over 42 data centers. In addition, Azure has successfully onboarded several HWaaS native applications such as Cray ClusterStor, and NetApp Files [13, 53]. As of today, we have powered several thousands of bare-metal servers and serve thousands of terabytes of traffic per day. Although a significant number of customers are adopting these bare-metal services, the number of routes tied to these workloads has yet to grow to the point of exceeding the route cache threshold of 85% capacity which would trigger the use of route cache. We estimate that the threshold will be exceeded within a year and believe that the route cache feature will play an important role in the future of the bare-metal service offering as the number of provisioned VNET routes outpaces the growth in hardware capacity of SDN ToRs.

We have compared the performance of bare-metal servers with VMs using Azure accelerated networking, both running in an Azure data center on Intel Xeon E5-2673 v4 (Broadwell at 2.3 GHz) CPUs with 40Gbps NICs and Windows Server 2019. We measured throughput, CPU overhead, and latency, with both solutions performing similarly and with no appreciable difference.

This section presents a performance analysis specifically focusing on route cache that is carried out through the use of synthetic testing tools and production data where available.

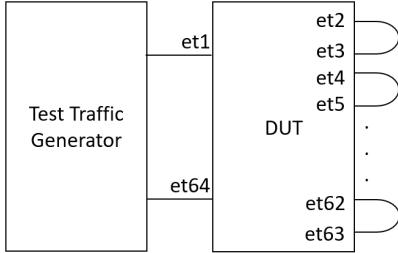


Figure 7: Test topology.

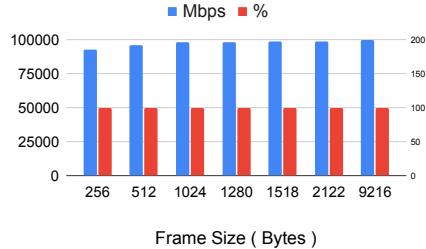


Figure 8: Throughput in Mbps on the left y-axis and in % on the right y-axis per frame size.

## 5.1 Hardware Performance

To measure the port-to-port latency within a SDN ToR, we connected a traffic generator directly to the Device Under Test (DUT). Figure 8 shows that the Intel Tofino ASIC performs at wire-speed with a consistent port-to-port latency of  $<1\mu s$ . The Tofino chip demonstrated deterministic performance across all cases of packet manipulation required for Bluebird. Additionally, even during heavy traffic load, minimal differences were observed between the min and max throughput values. Goodput was tested by passing L2 frames of various size through a DUT in a snake topology (Figure 7).

In Figure 8, 100% goodput is sustained across all the tested packet sizes up to 9216 bytes, with the only expected exception for packets of 256 bytes. This is due to the smaller relative size difference between header and payload. This performance is in line with our requirements to support bare-metal workloads that are bandwidth and latency-sensitive.

From a power utilization perspective, although the ToR behaves like a bump in the wire, it is not adding any power draw over a regular data center design. The typical/max power draw of BlueBird ToRs is 271W/571W. This is in the same range as other ToRs used in Azure with the same bandwidth (64x100G) which have a typical/max power draw of 314W/616W.

## 5.2 Performance Impact of Route Caching

The route cache feature is responsible for moving route entries from the SFE to the hardware and vice versa while ensuring the process is transparent to the customer. The CPU on the ToR provides a DPDK-enabled packet processing function, helping to meet our stringent latency requirements. As men-

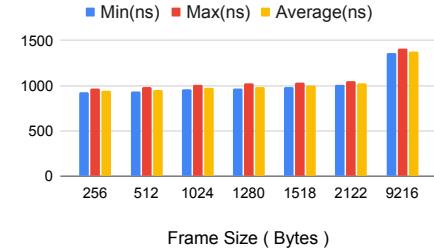


Figure 9: Latency measurements in nanoseconds (y-axis) per frame size (x-axis).

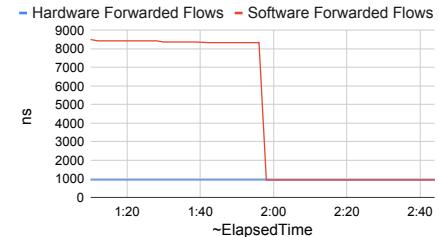


Figure 10: Latency comparison between software forwarded and hardware forwarded flows in nanoseconds per frame.

tioned earlier, we use 2x10G high-bandwidth links between the CPU and the ASIC. Since the traffic consists exclusively of TCP flows, our system has enough time to move the entries from the SFE to hardware by the time the TCP 3-way handshake is complete.

There are two primary factors that contribute to the observed latency when route-caching is performed; 1) the latency experienced while packets are being forwarded in software by the SFE and 2) the time taken to move a route entry from the SFE to the hardware.

To accurately measure the latency experienced by packets forwarded in software, we used instrumented packets that were generated by a traffic analyzer and forcefully forwarded them via the SFE. This was done by defining 5000 routes on the SDN ToR and sending traffic to each one of these entries while deliberately preventing the entries from being programmed into the hardware ASIC. To ensure that these software entries would not get programmed in the hardware, we temporarily disabled the route cache feature after the route entries were activated in the SFE. This approach ensured that the route entries in the SFE remained in software while we recorded latency measurements. In this state, where the routes were present only in the SFE, we found that packets experienced an increased latency of about  $8\mu s$  when compared to the latency experienced by packets using hardware entries.

Under normal circumstances, and assuming the route-entry used is in the SFE, we expect the packets to be software-routed for only a short time. The first packet that triggers a hitbit recording immediately kicks off the hardware programming for the SFE route. While this transition is hard to measure using generic traffic analyzers due to their lack of precision, we were able to inspect system logs on the SDN ToR to

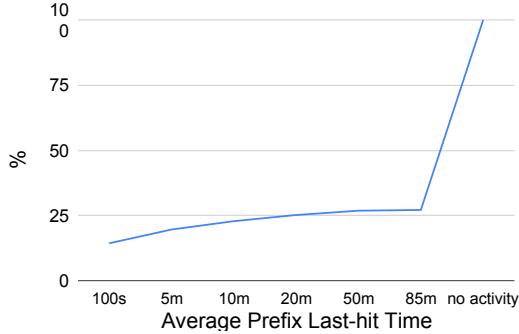


Figure 11: Prefix utilization based on last-hit time.

learn that this transition time is <2ms. To measure this, we compared the logged timestamps between when the packet hit the SFE and when the route appeared in hardware.

### 5.3 Validating Route Cache

The premise of designing and implementing the route cache model was based on a theoretical assumption that not all bare-metal customers would require hardware table entries at the same time. In order to prove that route caching would work, we had to find a way to record the age, or the last-time a given hardware entry was used. In other words, if an entry had aged and it's recorded "last time used" was old, then that entry could be demoted and moved to the SFE. Furthermore, we created the following time-buckets for age categories; 0s-100s, 101s-5m, 5m-10m, 10m-20m, 20m-50m, 50m-85m and a 'no activity' bucket for all prefixes that have not been touched, in hardware, for over 85 minutes. Grouping the prefix counts by last-time-used gives us further flexibility in tuning how aggressively we want to move entries to the SFE.

After weeks of data gathering from production SDN ToRs upgraded with the route cache feature, the results were inline with our expectations. Figure 11 shows the hardware table utilization across the route-entry last-hit time bins for the entire SDN ToR fleet. We see that 10% of all the prefixes in the hardware were utilized or 'hit' in the last 100 seconds.

Bluebird does implement an inherent artificial hardware provisioning constraint, to mostly protect against complete hardware table exhaustion on the SDN ToR. This protection was put in place to ensure that the 192K entries are never consumed. However, what we learned was that the hardware table utilization is in fact only 50% utilized and no more than 20-25% of the prefixes are active at any given time (Figure 11). This means that that a majority of the prefixes (75-80%) that are in hardware can in fact be moved to the SFE.

Armed with this data, we can now remove the Bluebird provisioning constraints and allow for provisioning of more than 192K entries knowing that 75% of the prefixes will most likely reside in the SFE.

Based on Figure 11, we can conclude that entries categorized under 'no activity' are good candidates to migrate to

the SFE leaving newly-vacated hardware entries available to other customers. Since only 20-25% of the entries are used at any given time, route caching allows us to increase our scale by 4-5x.

## 6 Operationalization and Experiences

Bluebird has now been deployed at scale in multiple data centers for various bare-metal workloads. The service has brought together high-throughput and low-latency bare-metal offerings to existing cloud customers without compromising scale or reliability. Bluebird has accomplished all the goals that we set out in §3. In order to make Bluebird successful, we adopted well-known operational models including continuous integration for both service delivery and feature development, ensuring redundancy in all aspects, planned failure for maintenance purposes, and incorporating monitoring and alerting.

During the feature development phase we used a P4 emulator [2] to simulate the entire pipeline in software which gave us a glimpse into the complete lifecycle of a packet. Having this flexibility removed the need for hardware at every stage of the development cycle. The software tools helped implement all the SDN ToR features, simulated the hardware, and allowed for rapid prototyping without any of the usual and costly hardware resource dependencies. Furthermore, P4 provided the flexibility of software with hardware-level performance. For example, routing look-up decisions would generally occur at a certain, fixed point in an ASIC's pipeline. In the case of P4, we had the flexibility of doing a routing look-up after the parser stage or in the MAU, giving us the luxury of normalizing the contents of an incoming packet and acting on any portion of the inner or outer IP header (see Figure 4). It was this flexibility in P4 that also allowed us to limit the UDP source-port values described earlier.

For workloads where data-plane redundancy was required, we paired two SDN ToRs using Multi-chassis Link Aggregation Group (MLAG) (Figure 12). While MLAG seemed like a natural choice for first-hop redundancy at layer-2, we did not want BBS to be concerned with the details of MLAG itself or make MLAG a design requirement moving forward. Hence, we created a common anycast loopback IP to represent both members within a redundant pair of SDN ToRs. BBS configures each member like any other SDN-ToR using a shared anycast loopback IP address representing the SDN ToR's physical address. If traffic were to arrive on a member of the SDN ToR pair with failed links to the bare-metal server, MLAG would locally switch the packets to the neighboring SDN ToR with active links to the bare-metal host.

While the P4 emulator tools helped with the development of the software features running on the SDN ToR, a different software emulator was used to help operationalize the SDN ToR in the network. A Docker container emulating a complete SDN ToR was used to speed up the management plane bring-up between Bluebird and the SDN ToR. Since the

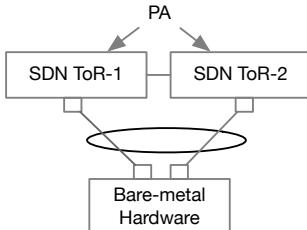


Figure 12: Bare-metal redundancy.

container version of the SDN ToR had all the properties of the hardware based SDN ToR (minus the hardware), the Bluebird management plane was developed and tested against this container. The Docker container was also useful when testing new customer scenarios before taking them to production.

For monitoring and alerting, we built our system to gather numerous metrics per ToR, BM server, and BBS. These metrics are collected in a centralized monitoring system, combined, and further transformed to create alerts. The actions based on the collected metrics differ based on the conditions or events configured. The metrics and alerts are also frequently added and updated as more experience is gained from production. Additionally, they are aggregated in customizable dashboards used to drive decision-making.

## 6.1 Lessons Learned

These are some of the lessons learned since the release of Bluebird:

- **Data-plane packet mirroring for debugging:** Having the ability to inspect data plane traffic during troubleshooting proved helpful. All data plane packets can be mirrored to the ToR CPU to inspect traffic entering or leaving the SDN ToR. This provides an additional layer of visibility to all the interface and protocol-level counters that are available on the switch. We did not expect this feature would be used as often as it has for debugging issues in production.
- **ASIC with re-configurable programming pipeline:** The Tofino ASIC, with its re-configurable pipeline, allowed us to develop features like route cache even after Bluebird was deployed. The VXLAN source port offset feature, described in §4.3, was possible because of the malleability of the pipeline. Making the decision to use a P4 ASIC proved to be useful as it allowed us to develop all the features that would otherwise have not been possible.
- **ASIC emulators can be used to speed up software development on ToR:** We used a P4 emulator during the ToR packet pipeline development process. Because of this, the development team did not have to wait for the hardware to be available. Moreover, the end-to-end packet flow could be tested by generating actual data-plane packets. This enabled us to test smaller parts of the P4 code without having an end-to-end pipeline ready.

- **Virtualized ToR image for control-plane testing:** Since the availability of the hardware ToR switch for lab testing and development is limited, having a VM image of the ToR was extremely useful to test the route programming service along with the control-plane interaction.

- **Need for 64-bit OS:** With a 32-bit OS, only 4GB of virtual memory could be addressed giving us 600k route cache entries. Hence, moving to a 64-bit OS and increasing the amount of RAM on the SDN-ToR was required to support 1.5M route-cache entries.

- **Limited control-plane vocabulary:** We limited the scope of commands that BBS was allowed to execute on the SDN ToR. The commands issued by BBS are strictly related to adding/deleting customer VRFs and mappings. This limits the damage that can be caused by bad actors and avoids interference with the rest of the automation framework. All other provisioning, maintenance, and monitoring functions are performed by the larger automation framework.

- **Software coordination at scale:** BBS runs on a server-class machine that is far more capable than the comparatively underpowered ToR's hardware. This difference is also reflected in the number of concurrent connections possible, which, if left unchecked, can impact programming time due to BBS exceeding the ToR connection limit. Consequently, requests from BBS are sent to a queue and batched to keep the number of connections within the limits of the switch.

- **Customer traffic should be agnostic of ToR availability:** MLAG helped us abstract out whether a ToR was in maintenance or not. This made the BBS less disruptive to deploy and maintain for redundant workloads.

- **Reconciliation is necessary:** Restoring an outdated ToR configuration can lead to failures and programming conflicts. Reconciliation is necessary to ensure that errors introduced by outdated configurations are repaired. A reconciliation process running after a configuration is restored guarantees eventual consistency and transient conflicts. For instance, CA-to-PA mappings received from upstream are compared with the ToR's current configuration, and stale mappings are removed in the process. Reconciliation is also performed against BBS' internal cache and state. Restoring a state after a service fail-over is also a source of incoherent configurations. Missed notifications during downtime are, in fact, a common occurrence in a live service.

- **Artificial limits can cause overheads:** During analysis of scale requirements from production data on the size of VNETs, we concluded that we had to limit the per-customer mappings until the route cache feature was enabled. As a result, an upper limit was put to stop one customer from monopolizing an entire ToR. This artificial limit soon became an operational overhead since we had to increase it per customer on an on-demand basis and in many cases the new limit was barely above the imposed value.

- **State to reduce reconciliation time:** BBS initially was

developed as a fully stateless service. After each restart, the switch configuration would simply be reconstructed through data received from upstream and downstream components. However, this significantly increased the rehydration and reconciliation time. Eventually, we moved to a stateful model to reduce the time consuming interactions with the other components. In the new model, BBS maintains a versioned representation of the switch configuration and communication is established only when strictly necessary.

- **For bug fixes, prefer a new image over a patch:** During the initial deployment stages, for quick bug fixes, we deployed bug-fix patches on top of the ToR OS image. But as the number of ToRs grew, it became cumbersome to deploy patches throughout the fleet of devices and keep a track of them. So, we decided to have bug fixes in new images only. Now we upgrade the devices more often but in a manner that is easier to track. This decision has helped us improve the quality of the code overall.
- **Unmodified Linux kernel:** The ToR OS uses an unmodified Linux kernel. Because of this, we were able to use open source tools like tcpdump, iperf, etc. for debugging without any issues. Also, we are able to run Docker containers on top of the ToR OS for SSH user certificate rotation.

## 7 Related Work

A practical implementation of Bluebird relies on the ability to enable custom and dynamic SDN policies in a ToR, enabled by recent work in programmable switches [4, 5]. As discussed in §2, many other forms of hardware can be used to implement the SDN stack including smartNICs [58–60] and servers running any one of a variety of software network processing systems such as [17, 25, 41, 62]. While there is a vast array of prior work in this space, the state-of-the-art software solutions are not able to meet the throughput of a programmable switch and require far more power. Work in network function virtualization [20, 43, 49, 54, 66, 68, 70] shows that these software-based approaches can be feasible at scale, though they do not meet our stringent requirements. Similarly, smartNICs have been used to offload various custom network operations [15, 44]. However, the bare-metal model precludes these options as they reside at the host. One may also adopt a hybrid approach, leveraging commodity switches and software [3, 18], but again the power consumption of a server is higher than a ToR switch.

Programmable switches have been used for a wide variety of other applications such as caching [32, 45, 47], telemetry [24, 57], consensus [11, 31], machine learning [63], and various network functions [37, 39, 52]. Despite the well-known and strict resource constraints in programmable switches [65], these systems demonstrate that non-trivial computations can be done in the network at line rate on these devices.

Bluebird leverages this speed while overcoming the state

limitations of Tofino switches by using the switch CPU and memory for cached flows. As the scale of the necessary state continues to grow, upgrading to the Tofino-2 [28] or adopting a switch memory extension may be helpful [40]. With increased traffic engineering and the rise of SDN, the limits of in-switch memory have become a noticeable issue prompting investigations into the practicality of caching for traditional routes [38, 46] and flow policies [9, 36]. We do not have to implement the complex dependency logic of [36] since the CAPA mappings are non-overlapping. Similarly, [38, 46] and [9] capitalize on relationships between entries in a FIB or open-flow table [51] which are not present in Bluebird’s in-switch data. Other SDN rule distribution techniques [34, 35, 55] do not apply to the Bluebird design as there is only one switch on route to implement the necessarily policies.

Since the early and largely academic SDN designs [7, 19, 22, 23, 42, 51], hyperscale cloud networks have adopted SDN to virtualize and isolate tenant networks [10, 12, 14, 21, 30, 56, 61], implementing at various layers of the stack. Regardless of multi-tenancy, SDN is used to operate large single-tenant networks with high-level intent and to implement arbitrary traffic engineering [8, 26, 29, 64, 67, 69, 71]. Bluebird is a new addition to our SDN deployment accommodating the unique requirements of baremetal customers: the servers must be connected to virtual networks, but we cannot enforce any SDN policies at the host in an approach such as [50]. This encourages new in-ToR support so that the additional costs of running the necessary network functions on a server similarly to [50] can be avoided.

## 8 Conclusions and Future Work

We have presented our experiences designing, implementing, and deploying Bluebird, a high-performance network virtualization system for bare-metal cloud services on Azure. Bluebird has been running in Azure data centers for more than two years and has served demanding workloads like those for Netapp, Cray, and SAP.

The abstraction layer in Bluebird’s control plane allows us to handle different switches with minimal change. By using high-performance programmable ASICs, we rearranged ToR’s resources to increase route capacity. On top of that, we have implemented a cache system that extends the capacity even further while incurring a negligible performance penalty. The support for additional routes has allowed us to improve performance by removing software gateways. Lastly, our design decouples the SDN stack from the bare-metal services and facilitates the introduction of new and diverse workloads.

In the future, as we learn more about customer traffic, we will explore ways to improve the cache system, such as by considering a different eviction algorithm.

## References

- [1] <https://github.com/aristanetworks/p4-vxlanencapdecap/blob/main/switch-vxlan.p4>.
- [2] P4 behavioral model, 2021. <https://github.com/p4lang/behavioral-model>.
- [3] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guohan Lu, Lihua Yuan, and Jennifer Rexford. A scalable VPN gateway for multi-tenant cloud services. *SIGCOMM Comput. Commun. Rev.*, 48(1):49–55, April 2018.
- [4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
- [5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, page 99–110. Association for Computing Machinery, 2013.
- [6] Broadcom. Trident SmartToR, 2021. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/smarttor>.
- [7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’07, page 1–12. Association for Computing Machinery, 2007.
- [8] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. FBOSS: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’18, page 342–356. Association for Computing Machinery, 2018.
- [9] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM ’11, page 254–265. Association for Computing Machinery, 2011.
- [10] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauchi Zermenio, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooster, 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, April 2018. USENIX Association.
- [11] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, August 2020.
- [12] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hieltscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jannah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.
- [13] Hewlett Packard Enterprise. Cray clusterstor e1000 storage systems, 2021. <https://buy.hpe.com/us/en/enterprise-solutions/storage-solutions/cray-clusterstor-storage-systems/cray-clusterstor-e1000-storage-systems/cray-clusterstor-e1000-storage-systems/p/1012842049>.
- [14] Daniel 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, March 2017. USENIX Association.
- [15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert 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, April 2018. USENIX Association.
- [16] Linux Foundation. Open vSwitch, 2016. <https://www.nvidia.com/en-us/networking/ethernet/connectx-5/>.
- [17] Linux Foundation. Data plane development kit (DPDK), 2021. <http://www.dpdk.org>.
- [18] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, page 27–38. Association for Computing Machinery, 2014.
- [19] Yashar Ganjali and Amin Tootoonchian. HyperFlow: A distributed control plane for OpenFlow. In *2010 Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN 10)*, San Jose, CA, April 2010. USENIX Association.
- [20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, page 163–174. Association for Computing Machinery, 2014.

- [21] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, March 2011.
- [22] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. 35(5):41–54, October 2005.
- [23] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.
- [24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, page 357–371. Association for Computing Machinery, 2018.
- [25] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI’12*, page 135–148, USA, 2012. USENIX Association.
- [26] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google’s software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, page 74–87. Association for Computing Machinery, 2018.
- [27] Intel. Intel Tofino, 2021. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html>.
- [28] Intel. Intel Tofino 2, 2021. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html>.
- [29] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM ’13*, page 3–14. Association for Computing Machinery, 2013.
- [30] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, April 2013. USENIX Association.
- [31] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.
- [32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP ’17*, page 121–136. Association for Computing Machinery, 2017.
- [33] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ah-san, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys ’18*. Association for Computing Machinery, 2018.
- [34] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT ’13*, page 13–24. Association for Computing Machinery, 2013.
- [35] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *2013 Proceedings IEEE INFOCOM*, pages 545–549, 2013.
- [36] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. CacheFlow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research, SOSR ’16*. Association for Computing Machinery, 2016.
- [37] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research, SOSR ’16*. Association for Computing Machinery, 2016.
- [38] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *Proceedings of the 10th International Conference on Passive and Active Network Measurement, PAM ’09*, page 3–12, Berlin, Heidelberg, 2009. Springer-Verlag.
- [39] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyasa Sekar, and Srinivasan Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM ’20*, page 90–106. Association for Computing Machinery, 2020.
- [40] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic external memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot*

- Topics in Networks*, HotNets '18, page 1–7. Association for Computing Machinery, 2018.
- [41] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.
- [42] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 351–364, USA, 2010. USENIX Association.
- [43] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 255–273, Santa Clara, CA, March 2016. USENIX Association.
- [44] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14. Association for Computing Machinery, 2016.
- [45] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGARCH Comput. Archit. News*, 45(1):795–809, April 2017.
- [46] Yaoqing Liu, Syed Obaid Amin, and Lan Wang. Efficient FIB caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, 43(1):14–21, January 2013.
- [47] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.
- [48] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, August 2014.
- [49] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling fast, dynamic network processing with ClickOS. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 67–72. Association for Computing Machinery, 2013.
- [50] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413. Association for Computing Machinery, 2019.
- [51] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.
- [52] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan 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, page 15–28. Association for Computing Machinery, 2017.
- [53] Microsoft. Azure NetApp files, 2021. <https://azure.microsoft.com/en-us/services/netapp/#overview>.
- [54] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.
- [55] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 157–170, Lombard, IL, April 2013. USENIX Association.
- [56] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: A scalable multi-tenant network architecture for virtualized datacenters. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 62–73. Association for Computing Machinery, 2011.
- [57] 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*, SIGCOMM '17, page 85–98. Association for Computing Machinery, 2017.
- [58] NVIDIA. Connectx-5, 2021. <https://www.nvidia.com/en-us/networking/ethernet/connectx-5/>.
- [59] NVIDIA. Data processing units, 2021. <https://www.nvidia.com/en-us/networking/products/data-processing-unit/>.
- [60] NVIDIA. Innova-2 flex, 2021. <https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/>.
- [61] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218. Association for Computing Machinery, 2013.
- [62] Luigi Rizzo and Matteo Landi. Netmap: Memory mapped access to network devices. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 422–423. Association for Computing Machinery, 2011.
- [63] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling

- distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.
- [64] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 418–431. Association for Computing Machinery, 2017.
  - [65] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon 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, March 2017. USENIX Association.
  - [66] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else’s problem: Network processing as a cloud service. *SIGCOMM Comput. Commun. Rev.*, 42(4):13–24, August 2012.
  - [67] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hözle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google’s datacenter network. *Commun. ACM*, 59(9):88–97, August 2016.
  - [68] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. SNF: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, page 296–310. Association for Computing Machinery, 2020.
  - [69] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at Facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, page 426–439. Association for Computing Machinery, 2016.
  - [70] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE'11*, page 12, USA, 2011. USENIX Association.
  - [71] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 432–445. Association for Computing Machinery, 2017.

# Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling

Yifan Li<sup>‡,†</sup>, Jiaqi Gao<sup>†</sup>, Ennan Zhai<sup>†</sup>, Mengqi Liu<sup>†</sup>, Kun Liu<sup>†</sup>, Hongqiang Harry Liu<sup>†</sup>

<sup>‡</sup>Tsinghua University <sup>†</sup>Alibaba Group

## Abstract

Programmable switches are widely deployed in Alibaba’s edge networks. To enable the processing of packets at line rate, our programmers use P4 language to offload network functions onto these switches. As we were developing increasingly more complex offloaded network functions, we realized that our development needs to follow a certain set of constraints in order to fit the P4 programs into available hardware resources. Not adhering to these constraints results in *fitting issues*, making the program uncomplilable. Therefore, we decide to build a system (called Cetus) that automatically converts an uncomplilable P4 program into a functionally identical but compilable P4 program. In this paper, we share our experience in the building and using of Cetus at Alibaba. Our design insights for this system come from our investigation of the past fitting issues of our production P4 programs. We found that the long dependency chains between actions in our production P4 programs are creating difficulties for the programs to comply with the hardware resources of programmable switching ASICs, resulting in the majority of our fitting issues. Guided by this finding, we designed the core approach of Cetus to efficiently synthesize a compilable program by shortening the lengthy dependency chains. We have been using Cetus in our production P4 program development for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from  $O(\text{day})$  to  $O(\text{min})$ ). In this paper we share several real cases addressed by Cetus, along with its performance evaluation.

## 1 Introduction

Programmable switches allow network programmers to use P4 language to offload network functions to data planes, enabling these functions to process packets at line rate. As one of the largest global service providers, Alibaba has widely deployed programmable switches in its edge networks [20, 27]. By Jan 2021, we have built  $O(100)$  PoP (point of presence) nodes and  $O(1000)$  edge sites in total, and the majority of them have employed programmable switches to implement a group of network functions, including firewall, DDoS defense, and load balancer. Figure 1 shows an example of the architecture of network functions within a single programmable switch in our edge networks. In this architecture, our programmers offload multiple network functions to a single programmable switch, enabling these network functions to process packets at Tbps speeds and saving CPU resources on the end-servers in edge networks.

While our business significantly benefits from the deploy-

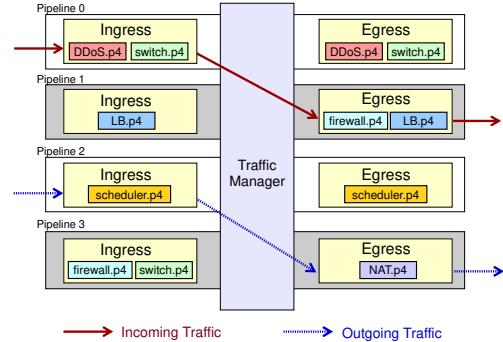


Figure 1: A gateway P4 program example deployed in Alibaba’s edge network. In our edge network scenario, our programmers put various network functions in a single switch.

ment of programmable switches, nevertheless, we still encounter a tough problem. Our P4 program development—e.g., implementation of new network functions and update of the existing network functions via P4—needs to take into account the various constraints of programmable switching ASICs; neglecting these constraints often results in programs that cannot fit on the hardware and hence cannot compile. We call this problem as *fitting issue*.

Fitting a P4 program is hard to our programmers, because (1) programmable switching ASICs have various hardware resources, each with unique size and constraints, and (2) resources are sometimes correlated, reducing the resource *A* usage of a program coming at the cost of increasing the usage of resource *B*. Our programmers, therefore, usually fall into time consuming trial and error program “reshaping” cycles, significantly delaying their development time. On the other hand, it is impractical to require our programmers to learn all hardware constraints.

Alibaba therefore decided to build a system (called Cetus) that automatically converts an uncomplilable P4 program  $P$  into a functionally identical but compilable P4 program  $P'$ .

**State of the art.** Existing work falls into two categories. On the one hand are systems that compile a high-level abstraction to generate optimized P4 programs [10, 13, 14, 25, 30]. Although they offer good resource optimizations, we found these solutions may not be effective in our specific scenario. For example, P4All [13, 14] optimizes the resource usage among network functions by explicitly leveraging reusable data structures (e.g., bloom filters and key-value stores); however, the network functions within our production P4 programs do not share these data structures, invalidating this optimization in our case. In addition, our programmers are reluctant to use

an extension of P4 such as explicitly specifying some data structure to optimize via objective in P4All. Another state-of-the-art system, Lyra [10], merges the tables that have no dependencies with each other in order to optimize the resource usage; however, we found that merging tables while keeping the original dependencies is not enough to enable our production P4 programs to fit into the programmable ASICs. On the other hand, existing efforts like Chipmunk [11, 12] and Domino [24] improve P4 compiler to synthesize optimized switch binary code, which is different from our goal of generating optimized P4 programs.

**Our approach: Cetus.** This paper shares our experience in the building and using of Cetus at Alibaba. We first investigated our production P4 programs and their past fitting issues, in order to derive insight for our solution design. We found that the **long** dependency chains between actions in our production P4 programs were creating difficulties for the programs to comply with the hardware resources of programmable switching ASICs, resulting in the majority of fitting issues.

Guided by the above finding, we designed Cetus. For a given P4 program  $P$ , Cetus automatically merges tables to fit into fewer stages by removing dependencies between tables, thus shortening the long dependency chains (§5). Because such a method may generate many table merging options (called candidates), we propose an approach, called constraint-based filter & optimizer (§6), to drop the candidates that do not satisfy hardware resources (including memory size, PHV, and crossbar) or constraints, and then select the best one as  $P'$ . Designing such a filter & optimizer approach is non-trivial due to two challenges: (1) the large formula encoding each candidate may result in state explosion, and (2) large solution searching space in each candidate will cause long solving time. We propose PHV sharing encoding (§6.1) and two-step solving (§6.2) to address the above two challenges, respectively. With  $P'$  in hand, Cetus automatically generates a set of control plane APIs for  $P'$  to enable  $P'$  to be deployed seamlessly (§7).

Finally, we share several representative real cases addressed by Cetus (§8), along with its performance evaluation (§9). We have been using Cetus in production for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from  $O(\text{day})$  to  $O(\text{min})$ ).

## 2 Preliminary: Programmable Data Plane

We use  $\Upsilon$  to denote the name of programmable switching ASICs of Vendor A.<sup>1</sup> Our programmers compile P4 programs via  $\Upsilon$  compiler.  $\Upsilon$  chip is a physical implementation of *Protocol Independent Switch Architecture* (or PISA).  $\Upsilon$  chip’s ingress and egress consist of 12 stages, respectively. All of these stages are identical, in terms of compute units, memory types, and memory capacities.

---

<sup>1</sup>We omit the vendor name and ASIC name for the confidentiality.

## 2.1 Hardware & Constraints of $\Upsilon$ Chip

**Hardware resource.**  $\Upsilon$  chip contains various hardware resources, and each of them has unique size and characteristic. We are mainly focused on the following hardware resources:

- **Pipeline stages.** The packet processing pipeline consists of a fixed number of individual stages. A P4 program does not compile if it takes more than 12 stages in an ingress or egress pipeline in  $\Upsilon$  chip.
- **Packet header vector (PHV).** The PHV is a “bus” that carries information (from packet fields and per-packet metadata) between stages. PHV cannot carry more data than its total width. See §6.1 for more PHV details.
- **Memory.** Memory resources mainly contain SRAM and TCAM. SRAM and TCAM are around tens of Megabytes in capacity. The memory resources are equally split and attached to each stage so that each stage can only access its local memory resources.
- **Crossbar.** In each stage, the crossbar extracts fields from the PHV and sends them to the match and action units for computation. Crossbar has a size limit, so the total number of bytes assigned to a stage’s crossbar should not exceed this limit.

**Hardware constraints.** The hardware constraints, in this paper, refer to both the hardware resource characteristics (*e.g.*, in  $\Upsilon$  chip, memories are stage local, and memory can be accessed no more than once per packet), and the mappings between the P4 program elements and hardware resources (*e.g.*, a P4 table’s keys should be stored in SRAM or TCAM memory, and a packet header field should be mapped into one or multiple cells in the PHV). Understanding these hardware constraints is crucial to programming on the  $\Upsilon$  chip.

To successfully compile a P4 program via  $\Upsilon$  compiler, this program must not exceed the size of each hardware resource and comply with all constraints of  $\Upsilon$  chip.

**Fitting a P4 program in our practice.** Our production P4 programs typically pack as many functions and modules as possible, which may overuse hardware resources or violate the hardware constraints, resulting in the fitting issues. When this happens, our programmers have to ‘reshape’ the programs to fit into the programmable ASIC. Such a reshaping process is program specific. Our programmers often spend a significant amount of time reshaping our P4 programs in order to comply with the hardware resources and constraints.

## 2.2 Dependencies between Tables

A P4 program is a collection of match-action tables chained together by branching conditions. In each table, at most one action can be applied according to the match result. For a given group of actions, if there is no read-write or write-write dependency among these actions, they could be placed within the same stage. On the contrary, for example, if action  $i_1$  uses (reads or writes) a value generated by action  $i_2$ , then  $i_1$  must

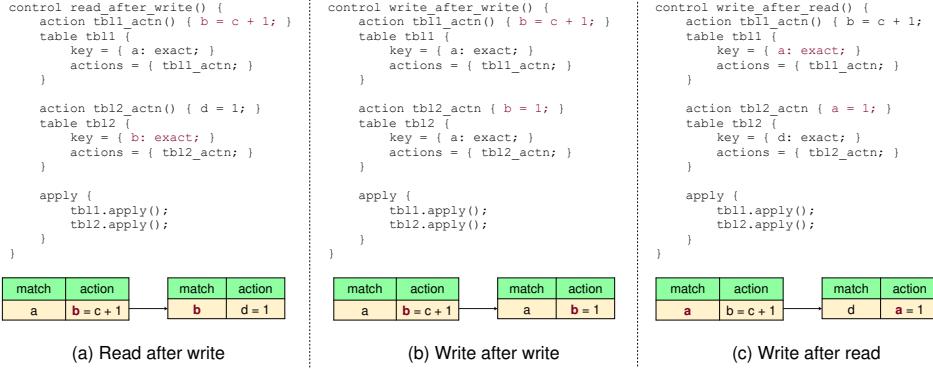


Figure 2: Three types of dependencies between actions in our production P4 programs.

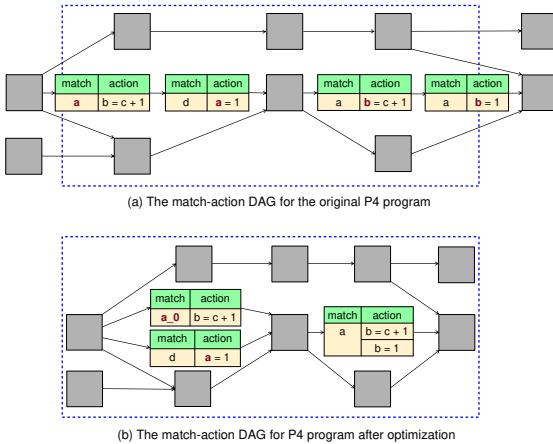


Figure 3: Examples for match-action DAGs. Rectangles represent tables. The blue dashed frame represents the architecture of Υ chip. The blue dashed frame's length and width represent the usages of stage and memory, respectively, in Υ chip. (a) shows a match-action DAG representing a given P4 program  $P$ .  $P$  does not fit in Υ chip. (b) is a match-action DAG representing  $P'$  that tweaked from  $P$ , which is compilable.

be placed in a stage after the stage of  $i_2$  in the PISA architecture. In our production P4 programs, we are mainly focused on three types of dependencies: read after write, write after write and write after read<sup>2</sup>. Figure 2 shows their examples. The tables, in Figure 2(a), (b), and (c), are not allowed to be directly placed within the same stage; otherwise, the programs' function logic is changed.

**Match-action DAG.** By tracking dependencies between actions, we can represent a P4 program in the form of a match-action directed acyclic graph or *match-action DAG*. Figure 3(a) presents such a match-action DAG.

**Diameter of a match-action DAG.** The total number of stages occupied by a P4 program  $P$  cannot be less than the *diameter* of the match-action DAG representing  $P$ . The diameter of a match-action DAG  $G$  is: the number of tables in the longest dependency chain (*i.e.*, the dependency chain containing the highest number of tables) in  $G$ . For example

P4 Programs	Network Functions	Diameter		Head, Tail Memory Percentage
		Ingress Pipeline	Egress Pipeline	
Edge vSwitch	VXLAN encapsulation	9	3	14.73%, 3.32%
	VXLAN decapsulation			
	Controlling the flow between CPU and data plane			
	Traffic statistic			
	IP packet forwarding			
CDN	ACL			
	Load balancing	10	5	0.87%, 5.04%
	Controlling the flow between CPU and data plane			
	Scheduling			
	IP packet forwarding			
Edge Gateway	DDoS defense			
	ACL			
	VXLAN packet forwarding	8	7	0.01%, 0.86%
	Traffic limit			
	Load balancing			
	ACL			

Figure 4: Our production P4 programs and their involved network functions as well as their diameters. These three programs have been deployed on almost all the programmable switches in our edge networks.

In Figure 3(a), the diameter is 7, because there are 7 tables in the longest dependency chain of the DAG. The diameter in Figure 3(b) is 5. Thus, we can say that the diameter of a match-action DAG (representing  $P$ ) must be  $\leq$  the number of stages, if  $P$  compiles.

### 3 Key Findings & Solution Intuition

In order to release our programmers from trial and error program-reshaping cycles, we need to understand the root causes resulting in fitting issues during the development of our production programs, thus exploring insights for our solution design. Specifically, we selected three mainstream P4 programs (listed in Figure 4) in our production, which were deployed in almost all the edge switches in Alibaba edge networks. We then selected all fitting issues (of these three programs) that took our programmers more than one hour to resolve, and manually analyzed how they were fixed.

We classified our analysis results into two groups. (1) **Group A:** About 80% of fitting issues were resolved by eliminating or reducing dependencies between tables (*e.g.* by reordering or merging them) that allowed us to take advantage of the parallel nature of the switch architecture. (2) **Group B:** 20% issues were resolved by fixing hardware resources and constraints that programmers were not aware of such as PHV

<sup>2</sup>We explain why write after write dependency is necessary in §5.1

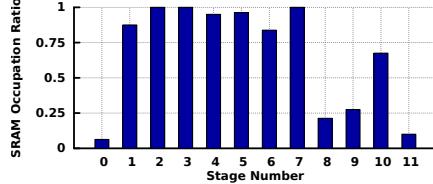


Figure 5: SRAM usage of Edge Gateway program.

allocation and stateful ALUs. We now analyze the principles behind Group A (§3.1) and Group B (§3.2).

### 3.1 Key Findings from Group A

We investigated why rearranging tables can resolve the fitting issues in this group. We found that all of these efforts (*e.g.*, reordering and merging tables) implicitly shortened the P4 programs’ diameters. For example, in one of the cases, our programmer unwittingly merged two tables by changing dependencies between their actions (as shown in Figure 7(a) example), and then found that the program compiled. While this programmer did not know the fundamental reason (*i.e.*, shortening the diameter), he succeeded after multiple reshaping cycles.

**Observation 1: Diameter is long in our production.** Why shortening the diameter can resolve the fitting issue? We found that the match-action DAG representing each of these three P4 programs had long diameters. Given that Υ chip provides 12 stages of match-action units, a long diameter should be reduced in order to make programs fit on Υ chip. As shown in Figure 3(a), blue dashed frame’s length and width represent the usages of stage and memory, respectively, in Υ chip. The program’s diameter in Figure 3(a) is too long to comply with the stage resource size.

The long diameter results from the large number of packet processing operations required by our diverse edge services. In particular, each of our P4 programs not only needs to insert various metadata into the different types of packet headers, but also filters or forwards them according to a number of service needs. For example, an input packet is first encapsulated with VXLAN, then forwarded based on some condition, next mirrored for traffic statistics and finally checked by ACL as well as distributed by the ECMP. Figure 4 details these three P4 programs’ diameters and their involved network functions. All programs shown in Figure 4 have at least a diameter of 8 in ingress, which means they occupy at least 8 stages in the ingress pipeline. It is therefore highly possible to result in fitting issues in Υ chip when new tables are added.

**Observation 2: Many available memory resources.** We also found that shortening the diameter by tweaking tables, in principle, increases the usage of memory within individual stages, as shown in Figure 3(b). Why did this memory-for-stage method work in our production? We found that both ends of the match-action DAG (tables with 0 in-degree or out-degree) use much less memory, offering flexibility for table tweaking.

At the beginning of the pipeline, our programs need to perform checking and pre-computations such as packet validation, link aggregation group checking, pre-computing hashes, and setting flag based on header’s validity; at the end of the pipeline, our programs finalize the packet processing based on the previous matching results, including marking header fields, dropping packets, and encapsulations. All these operations can be easily done in parallel, while at the same time they do not require a lot of table entries; thus, much available memory remains. Figure 4 shows the percentage of memory that both ends of DAG occupy compared with the entire program. If the memory is distributed evenly across the DAG, both ends of the DAG should occupy around 10% of memory each. Figure 5 shows the SRAM occupation ratio per stage of Edge Gateway program (*i.e.*, the third program in Figure 4). We observed that stage 0 and 11 only used less than 10% of memory. The other two programs also follow the same phenomena.

We also observed much available memory in the middle of the pipeline. Figure 5 shows tables at stage 8 and 9 take only 25% of memory. Similar phenomena also occurred in the rest of the two P4 programs listed in Figure 4. This is because, in a network function chain, we typically have a few tables that are small but critical such as a table inserting a mainstream service-shared DSCP value into the packet header as metadata. Such a table (called  $T$ ) must have (read-write or write-write) dependency relationships with the tables before and after  $T$ .

**Summary.** We now understand that our programmers unwittingly shortened the diameter of their programs by trial and error table (dependency) tweaking, luckily making their programs compile. Examples in Figure 3(a) and (b) illustrate such an intuition. We therefore derive the following key finding.

**Finding 1:** Long dependency chains between actions in our production P4 programs make the developed programs hard to fit into the programmable ASIC. We thus need to remove dependencies on the “longest path” of DAG to change the original “long, narrow” DAG to a “short, fat” DAG, as shown in Figure 3, in order to enable our developed programs to compile.

### 3.2 Key Findings from Group B

Fitting issues in Group B were caused by the violation of chip-specific resource size and hardware constraints. For example, because our programmers ignored the size of an individual stage, the program they wrote required the compiler to assign more DRAM within one stage than allowed (otherwise the dependency constraint is violated), resulting in a fitting issue; the same issue also happened for other resources such as hash units. There is no pattern to follow among these root causes. But we noticed that some of the issues in Group B were caused by same constraint violation. This means that our

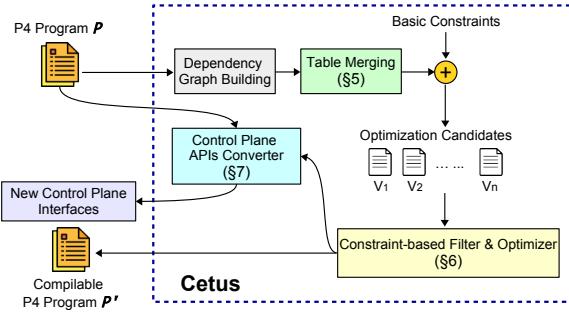


Figure 6: Cetus’s workflow overview.

programmers failed to learn or remember the fitting issues that they have ever fixed. We thus derive the following finding.

**Finding 2:** Although it might be hard for our programmers to learn all chip-specific resource size and constraints, we should avoid the fitting issues—resulting from the unfamiliarity with the resource size and constraints—that we have encountered before.

### 3.3 Our Solution Idea

Based upon our above two findings, we design the core approach of our solution, which includes the following three steps. First, for a given P4 program  $P$ , we automatically merge tables to fit into fewer stages by removing dependencies between actions, in order to shorten the long diameter of DAG representing  $P$  (driven by Finding 1). Such an approach would generate many candidate results. Second, we encode hardware resource size and hardware constraints as many as we know in our system’s backend DB to ensure that the synthesized program complies with all already-known resource size and constraints (driven by Finding 2). Finally, we check each candidate with the encoded constraints, selecting the most optimal one.

**Why the state of the art does not help?** Existing systems (*e.g.*, Lyra [10] and P4All [13, 14]) are unable to offer such a level of program optimization. Specifically, Lyra can only merge tables without dependencies. In other words, Lyra cannot merge two tables by removing dependencies between the tables; thus, Lyra is unable to shorten the diameter of the given DAG. P4All optimizes programs by reusing common data structures. In our programs (shown in Figure 4), however, the tables on the diameter do not share any data structure, invalidating P4All’s assumption.

## 4 Cetus’s Workflow Overview

We build Cetus, a synthesis system that automatically converts an uncompliable P4 program  $P$  into a functionally identical but compilable P4 program  $P'$ .

Figure 6 presents Cetus’s workflow that consists of the following main phases.

- First, given a P4 program  $P$ , Cetus generates a match-action DAG by analyzing read-write and write-write dependencies in  $P$ . Then, Cetus introduces a table merging approach (§5) to shorten the diameter of the generated DAG by removing dependencies between tables. There could be many potential table merging cases. We drop the cases that violate basic hardware constraints (*e.g.*, memory size), obtaining a group of candidate programs.
- Second, we propose a constraint-based filter and optimizer (§6) to check each candidate individually with already-known constraints, selecting the most optimal one as  $P'$ .
- Finally, Cetus automatically generates a set of control plane APIs for  $P'$  to enable  $P'$  to be deployed seamlessly (§7).

## 5 Table Merging by Dependency Removal

Cetus proposes a table merging approach to shorten the diameter by removing dependencies. Intuitively, the purpose of the table merging module is to tweak  $P$  to fit into the architecture of Y chip. This approach includes several primitives to merge tables for different types of dependencies. This section first introduces these primitives (§5.1), and then describes the entire solution (§5.2).

### 5.1 Dependency Removal Primitives

We design several dependency removal primitives in terms of dependency types, including write-after-write, write-after-read and read-after-write dependencies (shown in Figure 2). Each of the primitives takes two tables as input and returns one or two tables that can be put within one single stage. The purpose of these primitives is to reduce the number of used stages by increasing other resources’ overhead such as PHV and memory.

**Symbols.** We define the following notations: table  $t$  has  $n_m$  match fields  $\{m_{t1}, \dots, m_{tn_m}\}$ , each field  $m_{ti}$  has  $w_{ti}$  bits in width and its match type is  $p_{ti}$ , which can be *exact*, *ternary*, *etc*. It also has  $n_a$  actions  $\{a_{t1}, \dots, a_{tn_a}\}$ . If one table has no default action, we add an empty action as the default. Table  $t$  has  $l_t$  entries. Let  $P_t$  be the action parameters’ total bit width, then table  $t$ ’s total memory usage is  $l_t(\sum_{i=0}^{n_m} w_{ti} + P_t)$ .

**Write-after-write (WAW) dependency.** WAW dependency happens when one table  $t_1$  contains an action that writes the value written by another table  $t_2$ . For example, in Figure 2(b), table  $tbl2$ ’s action  $tbl2\_actn$  writes variable  $b$ , which is previously modified by table  $tbl1$ <sup>3</sup>. Since two actions are not allowed to write to the same data in a PHV word concurrently, one cannot place them in the same stage. It is also impossible to reorder them since the program’s correctness is violated.

This primitive removes WAW dependency by merging the two tables into a new table  $t'$ . The merged table  $t'$  enumerates both tables’ all action combinations. The primitive works as follows, and Figure 7(a) shows an example.

<sup>3</sup>We cannot remove  $tbl1$  because a packet can hit  $tbl1$  but miss  $tbl2$ .

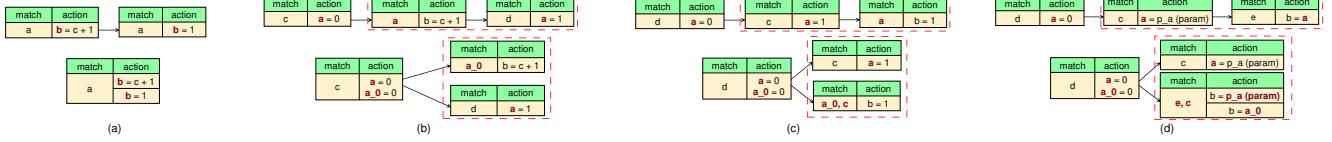


Figure 7: Examples for different dependency removal primitives. (a) WAW (b) WAR (c) RAW-match (d) RAW-action

- Merge the match fields of the two tables and generate new match fields  $\{m_{t_11}, \dots, m_{t_1n_1}, m_{t_21}, \dots, m_{t_2n_2}\}$ .
- Generate all possible combinations of the two tables' actions  $\{(a_{t_11}, a_{t_21}), (a_{t_12}, a_{t_21}), \dots, (a_{t_1n}, a_{t_2n})\}$
- Merge each pair of actions into one by appending the statements in the second action after the first one.
- When a merged action has two statements that write the same value, one from  $t_1$ , one from  $t_2$ , we keep the latter one.

*Memory usage.* Since the two tables hit and miss independently, the merged table should include all four possibilities. Thus, unless two tables have identical match fields, table  $t'$  uses ternary match field types and is deployed in the TCAM memory. In total, there are  $(l_{t_1} + 1)(l_{t_2} + 1) - 1$  entries. The total memory usage of table  $t'$  is  $l_{t_1}l_{t_2}(\sum_{i=0}^{n_{t_1}} w_{t_1i} + P_{t_1} + \sum_{i=0}^{n_{t_2}} w_{t_2i} + P_{t_2})$ .

**Write-after-read (WAR) dependency.** When one table  $t_2$  writes the variable read by  $t_1$ , WAR dependency happens. For example, in Figure 2(c), the table  $tbl2$ 's action  $tbl2\_actn$  writes variable  $a$ , which is table  $tbl1$ 's match fields. Again, we cannot reorder these two tables; however, PISA architecture allows  $t_2$  to be deployed alongside  $t_1$ . When  $t_1$  occupies multiple stages,  $t_2$  can only share  $t_1$ 's last stage and not earlier. WAR dependency does not necessarily increase the total number of stages of a program directly, but it sets a “barrier” and pushes other tables to later stages. For example, in Figure 2(c), if we have a third table  $tbl3$  that reads variable  $a$  after table  $tbl2$ , then it has to be deployed after table  $tbl1$ , even though there is no dependency between  $tbl1$  and  $tbl3$ .

For WAR dependency, let  $x$  be the shared variable. We have table  $t_1$  reads  $x$  and table  $t_2$  writes it. To remove WAR dependency, we create a new copy of the shared variable  $x'$  and modify  $t_1$  so that it reads  $x'$  instead of  $x$ . The primitive works as follows, and Figure 7(b) shows the example.

- Find the table where  $x$  is last written. If such a table exists, copy the action that writes  $x$  and modifies it to write  $x'$ . If no such table exists, such as  $x$  is a header, then we assign the value of  $x'$  in the parser.
- Modify table  $t_1$ 's match and action list so that it reads  $x'$ .

*Memory usage.* This primitive does not create a new table and the memory usage is kept the same. It may introduce PHV overhead since it creates a new variable.

**Read-after-write (RAW) dependency.** Read-after-write dependency happens when one table ( $t_2$ ) reads the value created by another one ( $t_1$ ). For example, in Figure 2(a), table  $tbl2$ 's

match fields read the value written by table  $tbl1$ 's action. The dependency can also happen when the value is read in the action field. Same as the WAW dependency, two tables with RAW dependency between them have to be placed in different stages and cannot be reordered.

This primitive removes the RAW dependency by summarizing the primitives used in WAW and WAR dependency: we first create a new table  $t'$  that summarizes the match fields of both tables and replace  $t_2$ , and then we adopt WAR dependency removal primitive to remove the dependency between  $t_1$  and  $t'$ .

Let  $x = f(\mathbf{v}_1)$  be action in  $t_1$  that modifies shared variable  $x$ . In Figure 2(a),  $\mathbf{v}_1$  is  $\{c, 1\}$ . Assume the action is executed when table  $t_1$  matches value  $\mathbf{v}_2$ , then after applying table  $t_1$ ,  $x$ 's value is:

$$x = \begin{cases} f(\mathbf{v}_1) & \text{if } (m_{t_11}, m_{t_12}, \dots, m_{t_1n}) = \mathbf{v}_2 \\ x_0 & \text{otherwise} \end{cases} \quad (1)$$

where  $x_0$  is the value of  $x$  before applying table  $t_1$ . The key of the dependency removal primitive is to encode enough information in a new table  $t'$  to compute variable  $x$  without using the result in  $t_1$ . Equation 1 shows that  $x$  depends on three sets of variables  $\mathbf{v}_1, \mathbf{v}_2, x_0$ . We can learn  $\mathbf{v}_2$  from entries in table  $t_1$ .  $x_0$  is created before  $t_1$ , so we borrow the primitive used in WAR dependency removal and create a new copy of variable  $x$ . So our challenge is reduced to understanding  $\mathbf{v}_1$ .

Theoretically, since variables in  $\mathbf{v}_1$  have fixed lengths, we can enumerate all possibilities. However, this would lead to too much memory overhead. As a result, we only remove RAW dependency when we can infer values in  $\mathbf{v}_1$  easily, such as when all of them are numbers or assigned to numbers directly. In Figure 2(a),  $\mathbf{v}_1 = \{c, 1\}$ . If we can infer the value of  $c$ , then we can merge  $tbl1$  and  $tbl2$ , otherwise, we cannot. Cetus removes dependency differently depending on whether table  $t_2$  reads  $x$  in the match or action part. If table  $t_2$  reads  $x$  in the match fields, the primitive works as follows, and Figure 7(c) shows the example tables and merged result.

- Create a copy of variable  $x$  through the method introduced in the WAR dependency removal primitive, let the copy be  $x_0$ .
- Merge the match fields of the two tables, remove  $x$ , and generate new match fields  $\{m_{t_11}, \dots, m_{t_1n_1}, m_{t_21}, \dots, m_{t_2n_2}\} - \{x\} + \mathbf{v}_1 + \{x_0\}$ .
- Remove constants from the match field. For example when  $\mathbf{v}_1$  or  $x_0$  is fixed.

	RAW	WAW	WAR
Direct stateful objects	N	N	Y
Normal & not directly involved	Y	Y	Y
Normal & directly involved	N	Y	Y

Table 1: Cetus applies primitives to different cases.

- Generate a new table  $t'$  with the new match fields. Copy table  $t_2$ 's action field to the table  $t'$ .  
If the table  $t_2$  reads  $x$  in the action field, we need to encode both branches in Equation 1 and duplicate actions that read  $x$ . The primitive works as follows, Figure 7(d) shows the example tables and merged results.
- Create a copy of variable  $x$  through the method introduced in the WAR dependency removal primitive, let the copy be  $x_0$ .
- Merge the match fields of the two tables and generate new match field  $\{m_{t_11}, \dots, m_{t_1n_1}, m_{t_21}, \dots, m_{t_2n_2}\} + \mathbf{v}_1$ .
- Remove constants from the match fields.
- For each action  $a_{t_2i}$  that reads  $x$ , replace  $x$  with new copy  $x_0$ . Create a new copy  $a'_{t_2i}$  and add  $x_0$  into its parameter. Action  $a'_{t_2i}$  is triggered when Equation 1's first condition is triggered, Action  $a_{t_2i}$  is triggered when the second condition is triggered.
- New table  $t'$  has the new match fields, all actions from table  $t_2$ , and newly generated actions  $a'_{t_2i}$ .

*Memory usage.* Memory usage varies depending on how many constants we can infer. Assume we can infer the value of  $x_0$  and  $\mathbf{v}_1$ , then the newly generated table  $t'$  takes up  $l_2(\sum_{i=0}^{n_1} w_{t_1i} + \sum_{i=0}^{n_2} w_{t_2i} + P_{t_2})$  memory. The newly generated table's match fields stays the same.

**Multiple dependencies between two tables.** Two tables can have more than one dependency and may not be limited to the same type. For example, they can have WAW and WAR dependency at the same time, or have two RAW dependencies. When dependencies have the same type, we can apply the pre-mentioned primitives directly (WAW) or recursively (RAW, WAR) to remove dependencies. For different dependency types, we choose not to remove them since the result table usually incurs too much memory overhead.

**Counters, meters, and registers.** In ASIC, stateful objects such as counters have two modes: direct and indirect. Direct counters have one-to-one mapping with table entries, while indirect ones have user-defined sizes. Depending on their mode and whether they are involved in the dependency directly (*i.e.* they write to variables read or written by another table), Cetus chose whether apply different primitives differently, and it is summarized in Table 1.

## 5.2 Table Merging Approach

Given a P4 program with  $n$  dependencies, there could be  $2^n$  different table merging strategies at most. Different strategies produce different resource-usage trade-offs among stage, PHV, and memory. Rather than sending all of them to the

constraint-based filter & optimizer module, we propose a heuristic algorithm that filters out strategies that violate basic constraints such as memory and stage.

In this approach, we only focus on comparing two metrics: stage saving and memory overhead. §6 would take more resources into account. If a strategy's memory overhead takes more stages than it can save by removing dependencies, it must end up occupying more stages than the original program, which conflicts with our goal. To sum up, given a P4 program, our heuristic algorithm runs as follows:

- Given a P4 program  $P$ , we generate its match-action DAG,  $D_P$ , and find all pairs of tables that potentially could be merged according to any of our primitives (mentioned in §5.1). Suppose we find  $n$  pairs.
- We build a binary decision tree  $T$  with  $n$  layers. Each layer represents one pair of tables, and each branch presents whether we remove the dependency of this pair of tables or not. Thus, a path from the root node of  $T$  to some leaf node of  $T$  represents a combination of table merging strategies.
- We thus run a deep-first search on  $T$ . During the searching process, we cut off the branches that violate basic memory and stage constraints. For each leaf node, we compute  $S_{\text{save}} * m > M$ , where  $S_{\text{save}}$  is the number of stages this strategy saves,  $m$  is the memory space of a single stage, and  $M$  is the memory overhead this strategy actually introduces. Note that  $S_{\text{save}}$  and  $M$  are computed by our primitives. If  $S_{\text{save}} * m > M$ , we keep this leaf node as one of our candidates used as the input of constraint-based filter & optimizer module (§6); otherwise, we drop this strategy.

## 6 Constraint-Based Filter & Optimizer

This module takes as input all candidates generated by §5, and then encodes each candidate program with all hardware resource size and constraints (stored in Cetus's backend DB) into an SMT formula. Then, we call an SMT solver (*e.g.*, Z3 [7]) to synthesize a table location plan that uses the least memory and stage resources. Finally, we realize this plan in a P4 program that specifies the locations of tables via pragma instructions.

The key challenge is how to efficiently solve these SMT formulas (each representing a candidate with all constraints). We found that the existing encoding approaches (*e.g.*, Lyra [10]) may result in state explosion, because a great number of diverse hardware resources create a huge search space that exceeds the SMT solver's searching capability.

To address the above challenge, we introduce a new approach that contributes two novel designs: (1) a new PHV encoding approach that significantly reduces the size of SMT formulas to avoid state explosion problem (§6.1); and (2) a two-step solving algorithm that decouples the solving process into table-related resource and variable-related resource solving to speed up the solving process (§6.2).

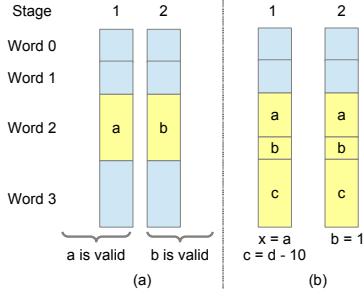


Figure 8: PHV sharing (a) across stages, (b) in one stage.

## 6.1 PHV Sharing Encoding

Packet Header Vector (PHV) serves as the bus between stages. The basic component of PHV is called word. There are tens of words with 8, 16, and 32-bit width respectively. One field can occupy one or multiple words. For example, a 48-bit source MAC field can take one 32b and one 16b word or three 16b words.

PHV is a scarce resource and needs careful planning, especially when the program is large and involves lots of headers and metadata. Simply adopting encoding approaches (*e.g.*, Lyra [10]) would waste the precious PHV spaces and fail to find a feasible solution. This is because Lyra’s encoding assumes each word is dedicated to one variable; however, PHV words can be shared across variables in the PISA architecture, both across stages and within the same stage.

**PHV sharing across stages.** Different variables can occupy the same word at different stages. As shown in Figure 8(a), after stage 2, variable  $a$  is no longer used and another variable  $b$  can take over the same word. This allows us to use only one PHV container to store two independent variables that would otherwise require two containers. This sharing requires the variables have non-overlapping lifetimes, *i.e.* from the stage they are created till the last stage they are used. Note that all packet header fields’ lifetime is the entire pipeline since they are created by the parser and consumed by the deparser. So the cross-stage sharing only applies to the metadata.

**PHV sharing within one stage.** Variables can also share the same word in the same stage as long as this sharing does not affect the correctness. Shown in Figure 8(b), variable  $a$  is read in stage 1 and variable  $b$  is assigned to a new value in stage 2. These two variables can share the same word. But variable  $c$  can not share with  $a$  at stage 1 because it was written by a subtract instruction. This is constrained by the fact that the Arithmetic Logic Unit (ALU) can perform at most one instruction to one word in one stage. The same-stage sharing applies to both header fields and metadata.

Cross-stage and same-stage sharing pack more variables into PHV, and it poses great pressure on PHV encoding. Because of the cross-stage sharing, we have to encode each stage’s PHV allocation separately. The same-stage sharing further complicates the problem since we need to consider whether each pair of variables could share the same word.

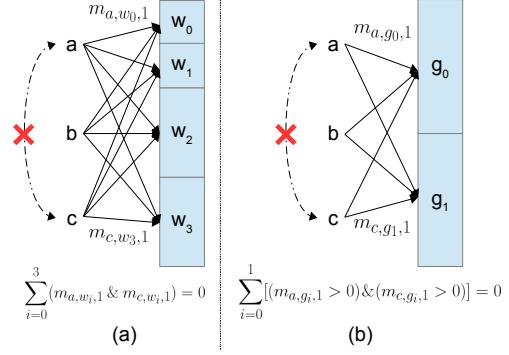


Figure 9: PHV encoding for 3 variables and 4 PHV words. (a) Strawman solution introduces 12 mapping variables and 4 rules. (b) Our solution reduces it to 6 mapping variables and 2 rules.

**A strawman solution.** A strawman solution is to encode the mapping  $m_{v,w,s}$  between the variable  $v$  and each PHV word  $w$  at stage  $s$ . It encodes the cross-stage sharing by treating each stage separately. As for same-stage sharing, when two variables  $v_1$  and  $v_2$  cannot share the same word, we can add the constraint  $m_{v_1,w,s} \& m_{v_2,w,s} = 0$ . Next, we encode constraints such as each word has its own size limit, each variable should reserve enough bits in the PHV, *etc*. However, shown in Figure 9(a), because there are tens of stages and hundreds of PHV words, this solution introduces too many such mapping variables and the search space is huge.

**Our encoding.** Our PHV sharing encoding method addresses the scalability challenge. We observe that the total number of independent mappings in the encoded formula is the key complexity contributor. Thus, our focus is to reduce the independent mappings.

For same-stage sharing, we remove the boundary between PHV words and focus on whether variables can share with each other. We noticed that at each stage, there are only a few “shareable groups”, the set of variables that can share with each other. Note that one variable can belong to multiple groups since the shareability is not transitive, *i.e.* variable  $v_1$  can share with  $v_2$  and  $v_3$  cannot conclude  $v_2$  can share with  $v_3$ . Then we can maintain the mapping between variables and these groups instead of the PHV words and restore PHV mapping afterward.

We also observe that in the encoded formula, all groups are symmetric: it does not affect the correctness when we reorder the groups. This is also another slow-down factor since it gives the SMT solver more freedom. To break the symmetry, we give preference to the groups with lower ID, the SMT solver can only use a new group until all the groups with lower ID are already assigned.

To summarize, the PHV encoding works as follows:

- (1) Given the input program  $\mathcal{P}$ , we count total number of non-assignment instructions  $I$  in each pipeline. This is the upper bound of the number of groups.

- (2) (Cross-stage sharing) For each variable  $v$ , we maintain: i) the mapping  $m_{v,g,s}$ , which denotes the number of bits  $v$  assign to group  $g$  at stage  $s$ , ii) the lifecycle  $l_v$  and  $r_v$ , which denotes the start and end stage of  $v$ .
- (4) (Same-stage sharing) If  $v_1$  cannot share with  $v_2$ , then  $(m_{v_1,g,s} > 0) \& (m_{v_2,g,s} > 0)$  is always false.
- (5) (Variable width) For each variable, if stage  $s$  is within the its lifecycle, the total number of bits in each group equals variable width  $b_v$ :  $l_v \leq s \leq r_v \rightarrow \sum_i m_{v,g_i,s} = b_v$ . Otherwise the summation is 0.
- (6) (PHV size) The summation of total number of bytes in each group should be less than PHV size.  $\sum_i \lceil \sum_j m_{v,j,g_i,s} / 8 \rceil \leq N_{PHV}$ .
- (7) (Break symmetry) We prioritize groups with lower ID:  $(\sum_j m_{v,j,g_{i+1},s}) > 0 \rightarrow (\sum_j m_{v,j,g_i,s}) > 0$ .

In Figure 9(b), because only  $a$  cannot share with  $c$ , there are at most 2 shareable groups. We introduce 2 groups  $g_0$  and  $g_1$ . Through this encoding, we can reduce the total mapping from 12 to 6. In reality, there is at least one order of magnitude fewer groups than the PHV words. This can greatly reduce the encoded formula's complexity.

## 6.2 Two-Step Solving

The PHV sharing encoding optimization can greatly reduce the encoded formula's complexity, but the SMT solver still struggles when dealing with large-scale production programs. Due to their scale, the encoded formula is still too complex. Additionally, PISA architecture's table-related resources (*i.e.* memory, table stage) and variable-related resources (*i.e.* PHV, crossbar) are orthogonal to each other: how much memory the table allocates per stage does not affect where the variable is located in the PHV. This loose coupling relationship forms a huge search space and exceeds SMT solver's searching capability under large scale programs.

While this loose coupling is the culprit, it offers us an optimization opportunity. We can safely ignore their correlation and split the SMT solving problem into two smaller problems. The two-step solving works as follows:

- Given a P4 program (*i.e.*, one of the candidates), we encode all table-related resources and constraints and find a feasible plan  $P_t$  meeting dependency and constraints.
- Upon  $P_t$ , we encode variable-related resources and constraints, and call the SMT solver to find a solution  $P_v$  capable of meeting resources (*e.g.*, PHV and crossbar) and constraints.
- If yes, with  $P_t$  and  $P_v$ , we have  $P = P_t + P_v$  as a resource allocation plan for the input P4 program, returning plan  $P$ .
- If not, we return to step 1, find another feasible plan  $P'_t, P'_v$ .
- We repeat the above process until we find a valid plan  $P$ ; otherwise, there is no valid plan for the input program.

This two-step approach can greatly improve the efficiency

of our SMT solving. This aligns with our previous findings in §3.1 that the allocation of stages and table is our major concern. Other resources still remain and are more flexible.

## 6.3 The Best Result Selection

At the end of our workflow, the constraint-based filter & optimizer module may output one or more results that meet all already-known resource size and constraints. We select the most optimal one based on our internally-defined metric calculator. However, our experience shows that the constraint-based filter & optimizer module returns only one result in most cases.

## 7 Control Plane APIs Converter

After  $P'$  is obtained, our last task is to synthesize a control plane converter, making sure that the control plane APIs generated from the original program  $P$  are compatible with  $P'$  without any modification. Although different dependency removal primitives require different converting strategies, they follow the same underlying principle: generate new table entries that replace the previous tables' dependencies.

Due to limited space, we briefly describe the API converter for a concrete case shown in Figure 7(d) when installing new table entries. The rest of cases are detailed in Appendix A.

Let  $t_1, t_2$  be the tables match  $c$  and  $e$  in program  $P$ , and  $t'_1, t'_2$  be the tables after processing. In this example,  $t'_1$  is the same as  $t_1$ . In the runtime, the converter keeps a record of existing entries in table  $t_1$  and  $t_2$  installed from the control plane.

When inserting an entry  $e_1$  to table  $t_1$ , we first insert  $e_1$  into table  $t'_1$  unmodified. Next, for each existing entry  $e_{2i}$  in table  $t_2$ , create two new entries, one hits both  $e_{2i}$  and  $e_1$ , action is  $b = p\_a$ ; one matches  $e_{2i}$  but misses  $e_1$ , action is  $b = a\_0$ . Insert all of them into table  $t'_2$ .

When inserting an entry  $e_2$  to table  $t_2$ , for each existing entry  $e_{1i}$  in table  $t_1$ , we create two new entries as well. If table  $t_1$  is empty, only create one rule that matches  $e_2$  and other fields left wildcard. Other operations such as modifying or deleting an entry follow the same principle.

## 8 Deployment Experience

Cetus has been used to facilitate the development of P4 programs at Alibaba for one year. It has effectively decreased our P4 development workload by two orders of magnitude (from  $O(\text{day})$  to  $O(\text{min})$ ). This section presents several real cases addressed by Cetus.

**Case 1: Parallelizing network functions.** A common problem our programmers frequently encountered is that implicit dependencies between actions or hardware constraints may prevent two or multiple network functions from occupying the same stages. If one of the functions contains a large table and another function consists of multiple small tables forming a long dependency chain, the total number of occupied stages could exceed the number of stages available, and our programmers had no clue on how to fix such a problem.

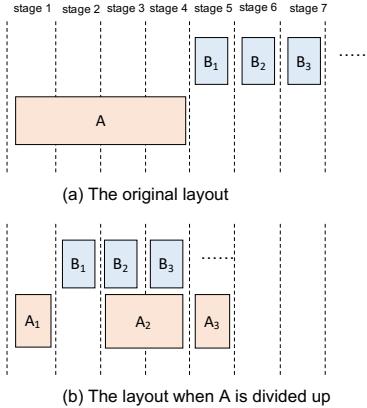


Figure 10: Parallelizing network functions via Cetus.

Figure 10 shows a real case in our edge gateway program. In the original P4 program, network function *A* only has one table *A*, which is a large table for load balancing. Function *B* consists of multiple tables like *B*<sub>1</sub>, *B*<sub>2</sub>, *B*<sub>3</sub>, etc. formulating a chain of small tables, each of which being responsible for inserting customized metadata for diverse services. However, if the program places network functions *A* and *B* as shown in Figure 10(a), a fitting issue occurs because their resource usage exceeds total stages available. From the view of our programmers, they can only do trial and error.

Through the dependency removal algorithm introduced in Section 5, Cetus can automatically address this problem by parallelizing network functions within few minutes. As shown in Figure 10(b), Cetus detected there is a deep dependency between actions of *A*<sub>1</sub> and *B*<sub>1</sub>, thus dividing function *A* into a few tables and maximizing the parallelization of table placement. We used the solution in [23] to guarantee the split tables act the same as the original one.

**Case 2: Optimizing write-after-write dependency.** Using global data is common in many programming languages and software systems. However, such practice comes with pitfalls in P4 programs. For instance, because the physical pipeline offers control registers, our programmers are allowed to explicitly drop a packet in packet validation, access control, and error handling. However, write operations to a common field issued by different modules may constitute write-after-write dependencies, which cause the number of required stages to exceed the actual stage number.

Figure 11 shows a real case. Figure 11(a) is the original P4 program. Two tables are invoked consecutively, which may call the same action to explicitly drop the packet. Because of write-after-write dependency, they must occupy two stages. Due to the “lengthy diameter” feature in our production programs, a fitting issue happened because stage resources are overly used. We therefore called Cetus to solve our fitting issue. Cetus automatically generates a program shown in Figure 11(b). We can observe that the two tables in the original program are merged into one, saving one stage to enable the program to compile. More interestingly, Cetus can also care-

```

action drop_packet() {
    eg_dprsr_md.drop_ctl = 1;
}

table color_drop() {
    key = { meta.pkt_color: exact; }
    actions = { drop_packet; NoAction; }
}

table mirror_drop() {
    key = { meta.pkt_color: exact;
            meta.mirror: exact }
    actions = { drop_packet; NoAction; }
}

control() {
    color_drop();
    mirror_drop();
}

```

(a) write-after-write dependency that requires two stages (b) merged tables that require only one stage

Figure 11: Write-after-write optimization

```

action set_flow_tag(bit<16> tag){
    meta.tag = tag;
}

table color_flow() {
    key = { meta.ingress_port: exact; }
    actions = { set_flow_tag; NoAction; }
}

action set_sample_rate(bit<16> rate){
    meta.rate = rate;
}

table sample_rate() {
    key = { meta.tag: exact; }
    actions = { set_sample_rate;
                NoAction; }
}

control() {
    color_flow();
    sample_rate();
}

```

(a) read-after-write dependency that requires 2 stages (b) merged tables that require only one stage

Figure 12: Read-after-write optimization

fully merge the match keys from the two tables. Because the *color\_drop* table does not match *meta.mirror* so the merged table used ternary to match *meta.mirror*.

**Case 3: Optimizing read-after-write dependency.** Modularization is another common paradigm in program development. By clearly defining interfaces and decoupling modules, it allows the independent design and development of individual pieces of code. However, the modularization of P4 programs often comes at the expense of RAW dependencies.

In our production P4 programs, it is common for one module to set a particular field, which is later read by another module. Figure 12(a) shows a real program example where the table *color\_flow* tags each packet depending on which port it comes from. Then, another *sample\_rate* table sets the sampling rate based on a packet’s tag. This constitutes read-after-write dependency; thus, *sample\_rate* has to be placed at least one stage later than *color\_flow*, resulting in at least two stages occupied. We found such read-after-write dependencies are quite annoying in our programs because many fitting issues were caused by this type of dependency.

With Cetus in hand, we directly applied Cetus in this scenario. Cetus automatically analyzes whether it is better to trade-off modularization for more efficient and compact code, given the limited number of physical stages in each pipeline. In particular, Cetus checks whether *meta.tag* is solely determined by *color\_flow*, and whether they are applied consecutively. If so, it merges the two tables so that the first and second lookup are performed simultaneously within one stage, as shown in Figure 12(b). As a side effect, merging these two

Program	LoC	Table Num (Ig/Eg)	Before		After		Dependency Removed			Time
			Diameter (Ig/Eg)	Stage Num	Diameter (Ig/Eg)	Stage Num	WAW	RAW	WAR	
PINT [3]	380	13 / 0	6 / 0	7	6 / 0	6	0	2	0	19s
RTT [16]	408	12 / 0	9 / 0	9	8 / 0	8	0	3	0	25s
Bier [18]	703	26 / 4	7 / 2	11	5 / 2	7	2	8	2	41s
P4_protect [17]	576	12 / 1	5 / 1	6	4 / 1	4	0	6	0	25s
Conquest [5]	847	1 / 19	1 / 7	9	1 / 6	6	0	8	0	2m51s
Beaucoup [6]	1677	25 / 0	10 / 0	12	10 / 0	11	0	1	0	6m58s
P4_switch	4701	34 / 25	8 / 5	12	8 / 5	11	0	2	0	11m30s
CDN	6342	19 / 2	10 / 2	11	10 / 1	10	0	3	0	1m27s
Edge vSwitch	2733	32 / 6	9 / 3	11	8 / 2	8	2	3	0	1m21s
Edge Gateway	4417	32 / 37	8 / 7	12	8 / 7	11	2	1	1	7m21s

Table 2: Experimental results conducted on a workstation with Intel Xeon 2.5GHZ CPU and 128GiB RAM

tables may cause the new table to occupy more memory; however, as designed in §6, Cetus is able to take both factors (*i.e.*, memory and stage) into account and produces a feasible solution if such optimization is indeed worthwhile.

**Case 4: SDE upgrade.** As the programs keep evolving, we also upgrade the runtime and development-time infrastructure, including the versions of switch OS and the P4 compiler, to enjoy the latest performance optimizations and fixes provided by the vendors. In such an upgrading case, the program must be re-fit. We can consult Cetus to pinpoint the problem and search for a feasible table layout. After being automatically annotated with pragmas, the existing P4 program was successfully compiled while keeping its code structure intact. In this way, Cetus cleared the most challenging obstacle and enabled the upgrade of the whole system.

## 9 Evaluation

Our evaluation aims to answer whether Cetus can reduce different program’s stage usage (§9.1) and how effective the optimization algorithms are (§9.2). All experiments were performed on a server with 2.5GHz CPU and 768GiB RAM.

### 9.1 Optimization

We chose 10 P4 programs, 6 open-sourced and 4 private ones, to evaluate whether Cetus can optimize and reduce their stage usage. In this evaluation, we mainly show Cetus’s stage occupation reduction capability. For each program, we record its DAG’s diameter and the number of stages it occupies in Y chip before and after optimization. We further listed which types of dependencies Cetus removed and the time it took for each program. Table 2 shows the result.

First, Cetus removed 1 to 12 table dependencies, reduced the program’s diameter by 1 to 2 and 1 to 4 stages. This shows the effectiveness of the primitives used by Cetus and our findings in §3.1 also apply to open source programs.

Second, Cetus can successfully find the best candidate at a decent speed. For simple programs, Cetus can find a plan in under a minute. For complicated ones, Cetus still managed to finish the search in minutes. Compared with the days of efforts developers spent optimizing the program manually, this is way faster and saves a lot of deployment efforts.

Third, we can see that most of the dependencies removed

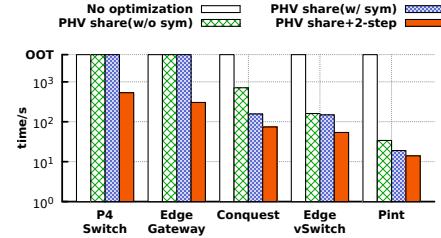


Figure 13: Time for a solution under different optimizations. were RAW dependencies. This is because of two reasons: (1) RAW dependency is common in programs. (2) RAW dependency is hard to find and also hard to remove. For example, below is a code snippet from Beaucoup [6]:

```
if (ig_md.cf_key_matched==1) {
    exec_regcoupon_merge(); // writes coupon_merge_check
}
if (ig_md.cf_decay_has_expired==1) {
    exec_counter_set_to_one();
} else {
    if (ig_md.cf_key_matched==1 && ig_md.coupon_merge_check==0) {
        exec_counter_incr();
}}
```

In the above code, the action `exec_regcoupon_merge()` writes variable `coupon_merge_check`, which is later read by the condition of action `exec_counter_incr()`. Cetus removes their dependency through the RAW dependency removal primitive, and it reduces one stage occupation. But for developers, it is hard to notice because it is spread across two different condition branches far away.

### 9.2 Performance

To further evaluate the effectiveness of the optimization techniques introduced in §6, we chose several typical programs with different scales and run experiments with different optimization techniques enabled. Starting from the naive solution with no optimization, we add vanilla PHV sharing encoding, symmetry breaking encoding, and finally two-step solving to Cetus sequentially. We set 1 hour as the timeout threshold. The result is shown in Figure 13.

Without any optimization, all programs timed out, which means it is necessary to introduce optimizations. For small-scaled programs, such as Conquest, Edge vSwitch, and Pint, adopting PHV sharing encoding can greatly improve the performance, indicates that the bottleneck lines in the complexity of the encoded SMT formula. However, for large-scale pro-

grams, such as P4 Switch and Edge Gateway, we only met the deadline after adding all three optimizations. This shows that for large scale programs, encoding optimization is not enough, the search space is still too large for the SMT solver to handle. It is necessary to leverage the key findings in §3.1 and bring in two-step solving to give a hint to the SMT solver.

## 10 Discussion and Lessons

This section discusses our lessons and limitations.

**Is  $P'$  functionally identical to  $P$ ?** In principle, Cetus’s approaches, including table merging and constraint-based filter & optimizer, can only change and optimize the location of tables, rather than the function logic of programs; thus,  $P'$  should be functionally the same as  $P$ . While we have not manually proved our approach on this property, in Alibaba, we employ a P4 verification tool, Aquila [27], to check the consistency between  $P$  and  $P'$  when Cetus generates  $P'$ . If Aquila returns “yes”, that means we can use  $P'$  to replace  $P$ . So far, we have not seen any inconsistency case.

**Can Cetus capture all hardware constraints within Y chip?** We encode constraints as many as we can; thus, we can only make sure that  $P'$  will not violate any constraints we have encountered before. With the accumulation of more and more hardware constraints, we believe the capability of Cetus will become stronger. However, we cannot guarantee every  $P'$  can compile to Y chip. We did experience few cases that  $P'$  does not compile due to unknown constraints.

**Can lengthy diameter always hold?** We cannot guarantee the lengthy diameter can always exist in our production programs in the future; however, based on our experience with Cetus so far, the stage shortage issue resulting from the lengthy diameter is still the highest priority barrier in our scenario. We thus suggest the ASIC vendor consider releasing a chip with double the number of stages and less memory.

**Cetus’s limitations.** We have the following main limitations. First, Cetus can only remove dependencies like WAW, RAW, and WAR. Cetus cannot handle more tricky cases such as removing dependency via modifying program semantic. Both RAW dependency removal algorithms require a third table in front to parallelize the latter two tables. For programs such as Syncookies [22], Cheetah [29], because they have long, chained sequential computations, the requirement of RAW dependency removal is not met, Cetus cannot perform optimizations. Second, Cetus cannot optimize a program when it occupies too many resources, since the dependency removal algorithms come at the cost of additional resources in the switch, such as PHV and memory. Third, we cannot guarantee Cetus’s implementation is bug-free although we spent a lot of time checking our implementation bugs; thus, sometimes the output  $P'$  may not be the best one. Finally, if a new programmable ASIC architecture is introduced, Cetus cannot be directly used to generate compilable programs for this new ASIC. Cetus has to encode all constraints of this new ASIC.

## 11 Related Work

**P4 program optimizers and compilers.** This type of systems optimize resource usage in programmable ASICs or simplify programmers’ tasks on expressing their coding intent. P4All [13, 14] aims to optimize resource usage by leveraging reusable data structures, such as bloom filters and key-value stores; however, our production P4 programs do not share these data structures. P4visor [31, 32] optimizes resources by merging redundant code fragments (*e.g.*, header parser and tables). P4visor is a good complementary to Cetus. Before Cetus was developed, we already built an internal system (similar to P4visor) to merge redundant code fragment.  $\mu$ P4 [26] proposes a modular way to write P4 code. Jose *et al.* [15] compiles P4 programs to architectures such as the RMT and Flex-Pipe. Domino [24] and Lyra [10] simplify data plane programming by specifying C-like new languages. Chipmunk [11, 12] leverages slicing, a domain-specific synthesis technique, to remove unnecessary resources cost by Domino. P<sup>2</sup>GO [30] proposes an idea that reduces the allocated resources of a P4 program based on traffic trace profiling. However, it might be hard for us to deploy it in our environment, because if unexpected traffic turns up after the profiling, some function might be already pruned. Different from the state of the art (that keeps the original dependencies), Cetus optimizes resource usage by removing dependencies in P4 programs.

**Network-wide configuration synthesis.** Configuration synthesis work [4, 8, 9, 19, 21, 28] offers the operator network-wide abstractions for configuration synthesis. SyNET [8] and ConfigAssure [19] offer general abstractions to synthesize the protocol configuration. Recent work [9] indicates that none of the above systems is scalable to cloud-scale networks. Propane [1, 2], Snowcap [21], and Jingjing [28] synthesize BGP, updating, and ACL configurations, respectively.

## 12 Conclusion

We have presented Cetus, the first system that releases the P4 programmers from frustrating trial and error compiling. Cetus can automatically convert an uncomplilable P4 program into a functionally identical but compilable P4 program. We have been using Cetus in our production P4 program development for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from  $O(\text{day})$  to  $O(\text{min})$ ).

*This work does not raise any ethical issues.*

## Acknowledgments

We thank our shepherd, Dejan Kostic, and NSDI’22 reviewers for their insightful comments. We also thank Vladimir Gurevich for his valuable feedback on both the technical part and the presentation of this paper. This work is supported by Alibaba Group through Alibaba Research Intern Program. Yifan Li is supported in part by the National Natural Science Foundation of China under Grant Number 61872212.

## References

- [1] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don’t mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.
- [2] Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.
- [3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 662–680, 2020.
- [4] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.
- [5] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 15–29, 2019.
- [6] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 226–239, 2020.
- [7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
- [8] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification (CAV)*, 2017.
- [9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. NetComplete: Practical network-wide configuration synthesis with autocompleteion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
- [10] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 435–450, 2020.
- [11] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)*, 2019.
- [12] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 44–61, 2020.
- [13] Mary Hogan, Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.
- [14] Mary Hogan, Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.
- [15] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.
- [16] Elie Kfouri, Jorge Crichigno, Elias Bou-Harb, and Gautam Srivastava. Dynamic router’s buffer sizing using passive measurements and p4 programmable switches.
- [17] Steffen Lindner, Daniel Merling, Marco Häberle, and Michael Menth. P4-protect: 1+ 1 path protection for p4. In *Proceedings of the 3rd P4 Workshop in Europe*, pages 21–27, 2020.
- [18] Daniel Merling, Steffen Lindner, and Michael Menth. Hardware-based evaluation of scalable and resilient multicast with bier in p4. *IEEE Access*, 9:34500–34514, 2021.
- [19] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage.*, 16(3):235–258, 2008.

- [20] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 194–206, 2021.
- [21] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 33–49, 2021.
- [22] Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle. Me love (syn-) cookies: Syn flood mitigation in programmable data planes. *arXiv preprint arXiv:2003.03221*, 2020.
- [23] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for tcam. *IEEE Micro*, 21(1):36–47, 2001.
- [24] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.
- [25] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 731–747, 2021.
- [26] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doganes, and Nate Foster. Composing dataplane programs with  $\mu$ p4. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 329–343, 2020.
- [27] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 17–32, 2021.
- [28] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the 2019 ACM SIGCOMM Conference*, pages 214–226. 2019.
- [29] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, page 2407–2422, New York, NY, USA, 2020. Association for Computing Machinery.
- [30] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *The 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.
- [31] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.
- [32] Peng Zheng, Theophilus A. Benson, and Chengchen Hu. Building and testing modular programs for programmable data planes. *IEEE J. Sel. Areas Commun.*, 38(7):1432–1447, 2020.

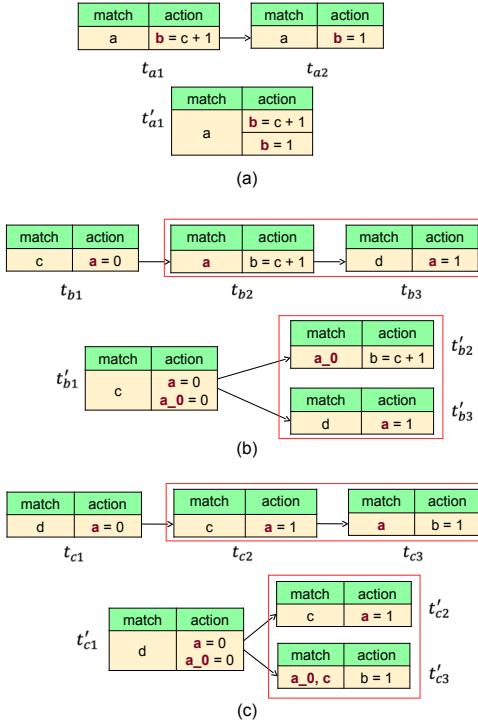


Figure 14: Examples for control plane APIs converter: (a) WAW (b) WAR (c) RAW-match.

## APPENDIX

Appendices are supporting material that has not been peer-reviewed.

### A Control Plane APIs Converter

This section details how Cetus’s control plane API converter bridges the inconsistency between the original program  $P$  and the optimized one  $P'$ . We labeled the tables in Figure 7 and show the example tables in Figure 14.

**Write-after-write dependency.** Since two tables  $t_{a1}$  and  $t_{a2}$  in Figure 14(a) share the same match field, the entries for both tables are inserted to the merged table  $t'_a$  directly. However, when two entries  $e_1, e_2$  for  $t_{a1}$  and  $t_{a2}$  respectively overlaps their match field (e.g.  $e_1$  matches 10.0.0.0/8 while  $e_2$  matches 10.0.0.0/16), entry  $e_2$  has higher priority than  $e_1$  because table  $t_{a2}$  applies later than  $t_{a1}$ .

**Write-after-read dependency.** The match field of table  $t_{b2}$  is renamed. For an entry  $e_2$  inserted to table  $t_{b2}$ , Cetus renames the match fields’ name and inserts it to table  $t'_{b2}$ . For example, in Figure 14(b), the match field  $a$  in  $e_2$  is renamed to  $a_0$ . Entries for table  $t_{b3}$  are inserted to table  $t'_{b3}$  directly.

**Read-after-write-match dependency.** In this case, Cetus records all the entries inserted to table  $t_{c2}$  and  $t_{c3}$  in a ‘logical table’ stored in memory. When a control plane application

inserts an entry  $e_2$  to table  $t_{c2}$  with match value  $c_{e2}$ , Cetus first inserts  $e_2$  to table  $t'_{c2}$  unmodified. Next, if there exists an entry recorded in logical table  $t_{c3}$  that matches the result of action in table  $t_{c2}$ , which is  $a = 1$  in Figure 14(c), then Cetus creates a new entry  $e'_2$  that matches  $c$  with value  $c_{e2}$  and ignores value of  $a_0$  and inserts it to table  $t'_{c3}$ . When an entry  $e_3$  is inserted to table  $t_{c3}$ , there are two cases. If  $e_3$  matches the result of the action in table  $t_{c2}$ , record it in the ‘logical table’ and do not insert it anywhere. Otherwise, rename the match field name of  $e_3$  from  $a$  to  $a_0$ , add another match field  $c$  in  $e_3$  but ignores the value. The ‘ignore’ can be expressed by using the wildcard if  $t'_{c3}$  uses TCAM memory, or by enumerating all possible values if it uses SRAM memory.

**Read-after-write-action dependency.** This part has been detailed in §7.

The entry removal operation is the reverse of the above actions.



# Exploiting Digital Micro-Mirror Devices for Ambient Light Communication

Talia Xu, Miguel Chávez Tapia, and Marco Zúñiga

Technical University Delft

{m.xu-2, m.a.chaveztapia, m.a.zunigazamalloa}@tudelft.nl

## Abstract

There is a growing interest in exploiting *ambient light* for wireless communication. This new research area has two key advantages: it utilizes a free portion of the spectrum and does not require modifications of the lighting infrastructure. Most existing designs, however, rely on a single type of optical surface at the transmitter: liquid crystal shutters (LCs). LCs have two inherent limitations, they cut the optical power in half, which affects the range; and they have slow time responses, which affects the data rate. We take a step back to provide a new perspective for ambient light communication with two novel contributions. First, we propose an optical model to understand the fundamental limits and opportunities of ambient light communication. Second, based on the insights of our analytical model, we build a novel platform, dubbed PhotoLink, that exploits a different type of optical surface: digital micro-mirror devices (DMDs). Considering the same scenario in terms of surface area and ambient light conditions, we benchmark the performance of PhotoLink using two types of receivers, one optimized for LCs and the other for DMDs. In both cases, PhotoLink outperforms the data rate of equivalent LC-transmitters by factors of 30 and 80: 30 kbps & 80 kbps vs. 1 kbps, while consuming less than 50 mW. Even when compared to a more sophisticated multi-cell LC platform, which has a surface area that is 500 times bigger than ours, PhotoLink's data rate is 10-fold: 80 kbps vs. 8 kbps. To the best of our knowledge this is the first work providing an optical model for ambient light communication and breaking the 10 kbps barrier for these types of links.

## 1 Introduction

In the last two decades, the adoption of wireless communication has gone through an unprecedented expansion. This ever-increasing demand has raised warnings of a looming ‘radio frequency (RF) crisis’ [5], and various alternative technologies are being explored to mitigate this risk. Among them, visible light communication (VLC) has gained significant attention due to its wide, free and unregulated spectrum. VLC is a sub-area of optical wireless communication

(OWC) that focuses on light sources that are incoherent, divergent and multichromatic (such as sunlight and artificial white light). VLC allows standard LEDs to provide illumination and communication and it is enabling several novel applications, from interactive toys [23], indoor positioning systems [27], to LiFi [20]. VLC, however, has an important limitation: it requires direct (*active*) control over the circuitry of the light source to modulate its intensity. Most of the light in our environments comes from sources we cannot control directly, not only the sun but also plenty of artificial lighting.

To exploit the vast presence of *ambient light*, researchers are investigating backscattering (*passive*) communication. Passive-VLC modulates ambient light using liquid crystal shutters (LCs). LCs can be seen as light shutters that allow (or block) the passage of light to communicate logical ones (or zeros). Recent studies report ambient light links reaching more than 50 m with data rates around 1 kbps, while consuming only a few mWs [7, 25]. Ambient light communication is a transformative eco-friendly concept because it piggybacks on top of energy that already exists, but current passive-VLC studies face two main challenges.

**Challenge 1:** *There has been no optical analysis of various passive VLC systems.* In a way, our community has rushed into the design of systems without carrying out first a proper optical analysis of the various types of ambient light and their impact on communication. Hence, several designs have been implemented reporting a wide range of (i) coverages (from a few meters to several tens of meters), (ii) data rates (from hundreds of bps to several kbps), and (iii) lighting conditions (from cloudy and sunny days to various types of artificial lighting). However, without an analytical framework, it is difficult to define a common baseline to directly compare and understand which elements contribute to such disparate performance. More importantly, we cannot provide insights about the fundamental opportunities and limits of ambient light communication.

**Challenge 2:** *Transmitters focus on a single optical device.* State-of-the-art (SoA) designs in passive VLC studies have been mainly constrained to a single type of optical surface,

the LCs, but LCs have some inherent limitations. First, even before any type of modulation begins, LCs cut the optical power in half due to the use of polarizers. This undesirable, but necessary, property of LCs reduces the communication range. Second, LCs have inherently slow rise and fall times, which has limited the data rate of all *single-cell* designs to values around 1 kbps [7, 13, 29]. Our design space could broaden greatly if we include other types of optical surfaces.

In this work, we take a step back to rethink passive-VLC. First, we propose a simple optical model to gain fundamental insights. Then, based on the outcomes of our model, we explore the use of digital micromirror devices (DMDs), which have different operating principles compared to LCs. In particular, our work makes the following contributions:

**Contribution 1 [section 2]: An optical model for ambient light communication.** Our model includes a key optical principle that has not been considered in ambient light communication: the fact that the performance depends not only on the luminous flux of the light source (output power) but also on its radiation pattern (diffused or directional). For example, this insight explains why a system tested under artificial light can perform better than under diffuse sunlight, even though diffuse sunlight can provide illumination that is an order of magnitude higher than artificial lighting.

**Contribution 2 [section 3]: A new type of transmitter device.** Our model shows that maintaining directional light patterns is central for passive links, but maintaining such directionality requires the right type of (i) *ambient light* and (ii) *transmitter* (optical surfaces with specular reflection). To attain that goal, we propose a novel transmitter based on DMDs. Inexpensive DMDs, however, are designed for video projection and provide slow update rates, around a few hundred Hz. We design a custom controller to generate carriers up to 220 kHz. Our novel transmitter provides higher contrast and faster switching speed, allowing us to increase the data rate of passive links by a factor of 80 compared to LC transmitters.

**Contribution 3 [section 4 and section 5]: An implementation and thorough evaluation of our platform.** We build two transmitters, one with a DMD and the other with an LC; and two receivers, one optimized for LCs and the other for DMDs. Using the same setup for all evaluations, in terms of surface area and illumination, our results show that (i) if we use the receiver optimized for LCs, PhotoLink attains 30 kbps for a distance of six meters and a BER below 1%, compared to the 1 kbps provided by the LC for the same range and BER [3, 7, 29], (ii) if we use the receiver optimized for DMDs, the data rate increases to 80 kbps. This performance is obtained with a power consumption around 45 mW. Furthermore, even if we compare PhotoLink with a *multi-cell* LC system having a surface area that is 500+ times bigger than ours ( $66 \text{ cm}^2$  vs.  $0.13 \text{ cm}^2$ ) [28], PhotoLink can achieve an order of magnitude higher data rate (80 kbps vs. 8 kbps). To the best of our knowledge, our work is the first to break the 10 kbps barrier with ambient light communication.

## 2 System Analysis

A passive VLC system has three basic components, the emitter (light source), the transmitter (modulating surface) and the receiver. Every SoA study adopts a different set of components. Some studies use a light bulb as the emitter, others use a flashlight or the sun. Some studies use a diffuser at the modulating surface, others use retro-reflectors or aluminium plates. Some studies use lenses at the receivers, others do not. This wide range of set ups is, in part, responsible for the equally wide range of performances reported in the literature, with data rates ranging from 0.5 kbps to 8.0 kbps to link distances ranging from 2 m to 80 m [3, 13, 25, 26, 28, 29].

Leaving aside the specific modulation methods of all these studies, we want to gain a fundamental understanding of passive systems and their components. Building upon the models developed for free-space optics [21], we propose a framework to analyze passive communication with ambient light.

### 2.1 Maintaining the luminous flux

First, let us start with a guideline that, to the best of our knowledge, has not been stated in any prior passive-VLC study: *The most important aim in passive communication is to convey as much LUMINOUS FLUX as possible from the emitter to the receiver.* The *luminous flux*, which is measured in *lumen*, is different from *illuminance*, which is measured in *lux* (lux = lumen per unit area). To compare two different systems fairly, one should know at least the area and the illuminance at the transmitter (modulating surface). This represents the amount of energy that is captured by the transmitter ( $E_C$ ). Unfortunately, few studies report these two pieces of information.

The luminous flux, however, is not the only important parameter. Equally important is the radiation pattern, which determines how much luminous flux is maintained throughout the optical link (i.e., how much of  $E_C$  is able to arrive at the receiver). To highlight the importance of the radiation pattern, Fig. 2 depicts a *specular (mirror-like)* surface under four different types of light sources. The effect on the luminous flux is shown from more to less directive:

**a) Ideal.** First, to exemplify an ideal setup, let us use a laser, which is a highly directional source where the luminous flux is hardly lost. Due to this property, lasers are used extensively for long-distance free-space communication. Lasers, however, are a fundamentally different type of light source that is not as pervasive (or safe) as natural or artificial white light, and therefore, it is considered only as a reference in this paper.

**b) Directional** (sunlight in a clear day). On a clear day, sunlight rays travel in parallel and a specular surface maintains that directionality (luminous flux) towards the receiver. *We found only one study exploiting this setup, but with LCs [7]. Our platform shows the significant gains that can be obtained in this setup using DMDs.*

**c) Lambertian** (light bulbs and flashlights). With light bulbs, only a fraction of the luminous flux radiated by the source reaches the surface (green arrows in Fig. 1c). Further-

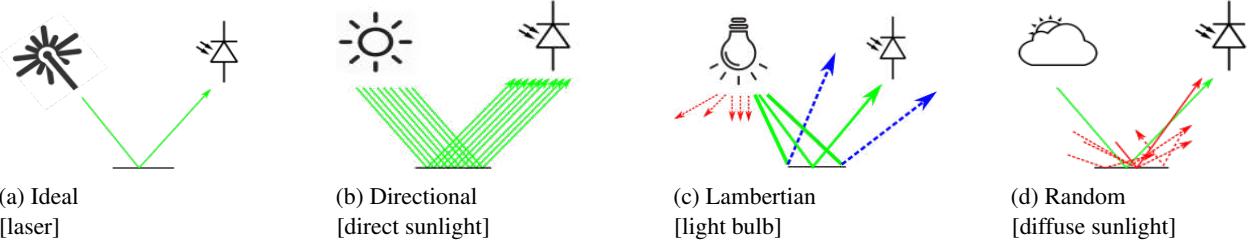


Figure 1: The effect of different radiation patterns on the luminous flux. The reflective surface is specular.

more, since rays are radiated in different angles, when the luminous flux hits the surface, some rays are lost because the impinging angle is either too broad or too narrow to hit the receiver (blue arrows). *This scenario is used by all the backscattering studies reported in the literature [13, 25, 28, 29].*

**d) Random** (sunlight in a cloudy day). Clouds scatter sunlight, emitting rays uniformly in random directions. Due to this phenomenon, only an infinitesimally small fraction of the rays will impinge the surface at the right angle to reach the receiver (green arrow in Fig. 1d). *Our model shows that this is the worst case scenario with specular surfaces. No practical links can be obtained in this setup.*

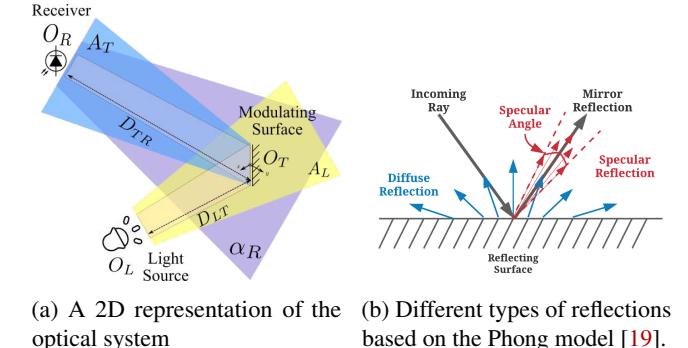
The key point of this preliminary analysis is to highlight the importance of maintaining the luminous flux throughout the optical link. In the next subsection, we present a model to capture more detailed insights with a ray-tracing simulator.

## 2.2 Ray-tracing model

A 2D representation of a typical passive system is shown in Fig. 2a. The optical link has two main parts. *First*, the link between emitter and transmitter. Light is emitted from the light source  $O_L$ , with a (yellow) wavefront represented by  $A_L$ . The modulating surface  $O_T$ , acting as a transmitter, is at a distance  $D_{LT}$  from the light source, and receives a fraction of the luminous flux emitted by  $O_L$ . *Second*, the link between transmitter and receiver. The flux reflected by the surface  $O_T$  is represented with a (blue) wavefront  $A_T$ <sup>1</sup>. The photoreceiver  $O_R$  is at a distance  $D_{TR}$  from the transmitter, and collects only a fraction of the flux reflected by  $O_T$ . Another relevant parameter is the Field-of-View (FoV) of the receiver, which is represented by  $\alpha_R$  (purple coverage). A wide FoV can cope with movements at the transmitter, but captures more noise.

Our toolbox, based on the above described model, is built upon Optometrika, a ray-tracing tool [18]. In essence, the toolbox divides the surface of the emitter, transmitter and receiver into small elements and calculates the fraction of rays that are able to reach the receiver. To assign the correct weight to each ray, Optometrika considers important optical parameters such as the angles of radiation, incidence and reflection. To analyze ambient light communication, the key inputs we need to provide to the toolbox are the radiation patterns of the emitter and the modulating surface.

<sup>1</sup>It is important to note that our model also captures the performance of retroreflectors because, from an optical perspective, the reflected radiation patterns are similar to those caused by mirrors



(a) A 2D representation of the optical system  
(b) Different types of reflections based on the Phong model [19].

Figure 2: Optical system and different reflection types.

## 2.3 Insights & Guidelines

A passive link is, in essence, a triplet  $\langle$ emitter, transmitter, receiver $\rangle$  that finetunes the parameters of each element to optimize the performance. To analyze the complete design space, including the systems proposed in prior studies, we utilize a few abstractions for the emitters and transmitters, as presented in Tables 1 and 2.

Unless indicated otherwise, our analysis assumes that (i) there is no noise, which is similar to conducting experiments in the dark, (ii) the illuminance on the transmitter is fixed at 1800 lx, to provide a common baseline for all cases and remove the trivial case where the performance is increased by increasing the illuminance, and (iii) the area of the receiver is  $1 \times 1 \text{ cm}^2$ . The selected area has no real impact on the analysis. The only assumption we make is that the transmitter's area is bigger than the receiver's, which is the case for most systems. Also, for our initial analysis, the receiver's FoV does not play a role because we assume a dark environment. In practice, the FoV plays a critical role and we will discuss it later on.

Regarding the modulating surface, we consider two main reflective patterns, as shown in Fig. 2b: *diffuse reflection*, caused by rough surfaces that reflect light in all directions, and *specular reflection*, caused by smooth surfaces. We further classify specular surfaces based on their specular angle. If the angle is zero, we call it mirror reflection.

### 2.3.1 Choosing the right emitter and transmitter

The design space of passive links can be divided into six main blocks based on the  $\langle$ emitter, transmitter $\rangle$  pair. Table 3 shows previous works categorized in this manner. Considering that direct sunlight provides tens of thousands of lx, overcast sunlight thousands of lx and light bulbs only hundreds of lx, a

Table 1: Emitters

Source	Type	Size of $O_L$	$D_{LT}$
L1	LED	5 cm × 5 cm	1 m
L2	LED	5 cm × 5 cm	4 m
L3	Diffuse Sunlight	N/A	N/A
L4	Direct Sunlight	N/A	N/A

Table 2: Transmitters

Modulating Surface	Type	Specular Angle	Size of $O_T$	Illuminance
T1	Diffuse	N/A	3 cm × 3 cm	1800 lx
T2	Specular	0.3°	3 cm × 3 cm	1800 lx
T3	Specular	1°	3 cm × 3 cm	1800 lx
T4	Specular	5°	3 cm × 3 cm	1800 lx

designer may assume that for any given surface, sunlight will always perform better than light bulbs. Similarly, considering that specular (mirror) surfaces provide stronger reflections than diffuse surfaces, a designer may assume that for any type of ambient light, a specular reflector will always perform better. Neither assumption is correct. In fact, we show that a particular combination of sunlight and specular reflectors gives the worst performance.

Fig. 3 depicts the signal strength of various scenarios as a function of the transmitter-receiver distance ( $D_{TR}$ ). We consider all six possible combinations of *emitters*: LED (L1 & L2), overcast day (L3), clear day (L4); and *transmitters*: diffuse (T1), specular (T2). Our results show four design regions, which are described next from worst to best. Our evaluation section validates many of these results empirically.

*Region 1: cloudy day & specular surface* (L3-T2 in Fig. 3a, gray area in Table 3). This region captures the scenario in Fig. 1d, where light arrives in a scattered manner and only an infinitesimal amount of the flux reaches the receiver. The signal strength of this setup is so weak and decays so fast, compared to the other scenarios, that it is not shown within the range of Fig. 3a to have a clearer view of the other regions.

*Region 2: any light & diffuse surface* (LX-T1, blue area). When a diffuse surface is used, it does not matter the radiation pattern of the light source, so long as the luminous flux at the transmitter's surface is the same. Note that all T1 curves overlap with each other in Fig. 3a. This occurs because ideal diffusers, such as paper or plaster, distribute the reflections of the impinging flux in all directions.

*Region 3: LED & specular surface* (L1/L2-T2, red area). This is the second best region, and coincidentally, the main focus of prior work using retro-reflectors. Artificial lights, however, offer a wide range of radiation patterns, resulting in widely different performance. To illustrate this point we use Fig. 3b, where two emitters are placed at 1 m and 4 m (L1 & L2). *Both emitters attain the same illuminance at the receiver* (1800 lx, a white light illuminance of 1800 lx over an  $1m^2$  surface is approximately equivalent to the power of a 25 W LED), but L2, which is *further away*, provides a stronger signal strength, which is counter-intuitive. This

Table 3: A taxonomy of passive VLC systems

Light Source \ Surface Type	RetroVLC [13] PassiveVLC [29] RetroTurbo [28] Retrol2V [25]	Diffuse
LED		
Sunlight (Cloudy Day)		Tweeting with Sunlight (TwSL) [4]
Sunlight (Clear Day)	ChromaLux [7]	Luxlink [3]

occurs because the further away the light source is, the more it behaves as a distant point source, leading to more directional beams impinging on the transmitter, and hence, less flux lost towards the receiver, c.f. Fig. 1c. In practice, L1 could be seen as a light bulb and L2 as a flashlight, which explains why studies using a flashlight attain better results [25, 28].

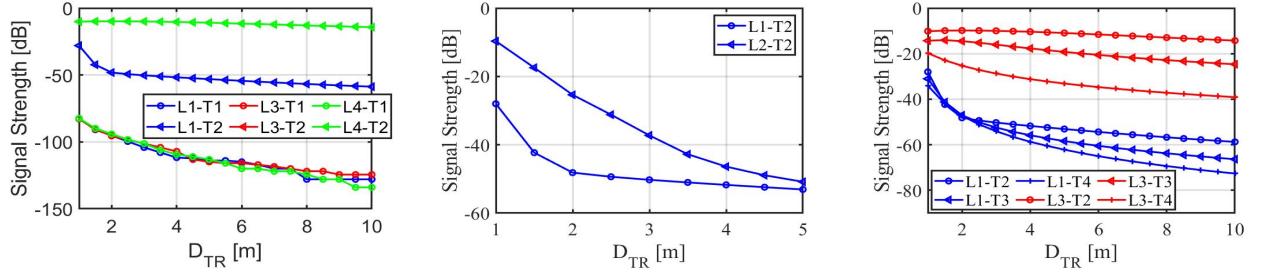
*Region 4: clear day & specular surface* (L4-T2, green area). This is the best operation region. Note that the signal strength hardly decays in Fig. 3a. This occurs because the high directionality of clear sunlight maintains the luminous flux over long distances, which is why heliographs (mirrors) used in the 1800's reached ranges beyond 100 km. This same property can increase the data rate of ambient light links. In practice, air attenuates the signal strength (similar to what happens with lasers), but the benefits of directionality remain strong.

### 2.3.2 Choosing the right specular surface

The above analysis highlights the importance of maintaining directionality throughout the optical link. However, given that there are no perfect mirror-reflectors, how critical is the specular angle? A wide specular angle can be the result of imperfections on the surface. For example, many studies use retro-reflectors, but the quality of retro-reflectors can vary. Fig. 3c shows the signal strength of surfaces with different specular angles, from narrow (T2, 0.3°) to wide (T4, 5.0°), considering an LED (L1) and direct sunlight (L3). When an LED is used (blue lines), the misaligned radiation pattern of the LED is more relevant than the specular angle, therefore, there is not much difference among the various surfaces. However, for a directional source (red lines), a large specular angle (e.g. 5° for T4) can lead to a significant decrease in the signal strength. Thus, *the more directional the rays, the more critical is the use of high-quality specular surfaces*.

### 2.3.3 Choosing the right receiver

Passive-VLC systems use cameras and photodiodes as receivers. Cameras are widely available in smartphones, but they are power hungry and slow, allowing only a few hundred frames per second. Photodiodes (PDs), on the other hand, are inexpensive, low-power and have a high bandwidth. Thus,



(a) Signal strength for different light source and surface combinations

(b) Signal strength for LEDs at different distances  $D_{LT}$ .

(c) Signal strength for different specular angles.

Figure 3: Different simulation setups.

PDs are the preferred choice for high data rate links. A key element in the PD’s design is its FoV. The FoV will not only capture the intended signal but the surrounding noise as well (purple coverage in Fig. 2a). In practice, to maximize the SNR, the receiver’s FoV should cover only the modulating surface, but that is difficult to attain. PDs with varying FoV have been used in the literature, ranging from  $1^\circ$  to close to  $90^\circ$  [3, 26]. Many studies using the wide FoV, however, were conducted at night with no interfering ambient light, which is similar to having a nearly perfect FoV of  $0^\circ$ . Given that our system is aimed at working with surrounding ambient light (noise), we borrow the design from [3], which uses a lens at the receiver to reduce the FoV, and thus, limit the noise level.

Overall, our analysis uncovers two key design guidelines. First, for the emitter-transmitter link. Direct sunlight, flashlights and light bulbs –in that order– are preferred due to their directionality. Diffuse (cloudy) daylight is the least ideal condition in spite of being the second most powerful source (after direct sunlight). Second, for the transmitter-receiver link. The more directional the light source is, the more critical is to use mirror-like reflectors. The only case where diffuse surfaces are preferred is when the impinging light is diffuse as well.

### 3 Transmitter Design

#### 3.1 LC limitations

Most passive-VLC systems using either transmissive [3, 30] or reflective (backscattering) principles [13, 29] rely on liquid crystal shutters (LCs) as the modulating surface. Unlike liquid crystal displays (LCDs), LCs do not have embedded light sources. LCs are readily available, economical, and power efficient, but they suffer from two intrinsic limitations.

##### 3.1.1 Limitation 1: High signal attenuation

LCs only allow a single polarization direction to pass through. All other directions are either fully or partially attenuated. Ambient light, however, is not polarized. This implies that only half of the power can pass through a linear polarizer. On the other hand, DMDs have microscopic mirrors with a high reflection coefficient and are polarization insensitive. For example, the DLP2000 module from Texas Instruments has an efficiency of 97% [9]. Thus, considering the same modulating area and incoming illuminance, DMDs radiate almost 100%

more light than LCs, which can be exploited to increase the range or the data rate of passive links.

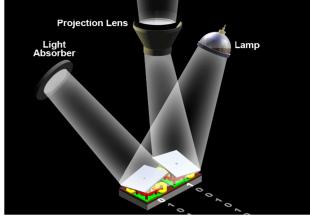
#### 3.1.2 Limitation 2: Limited bandwidth

The rise and fall times of commercial LCs take a few ms, as shown in Fig. 4d. These times limit the bandwidth to be under 1 kHz. Furthermore, LCs combine two different operation principles, an electrical signal for the rise time and mechanical inertia for the fall time. This asymmetric operation makes the fall time much slower and it is usually the main bottleneck to increase the bandwidth. Active research has been carried out to squeeze as much data rate as possible from that limited bandwidth, but community efforts are still restricted to around 1 kbps for single-cell designs [3, 7, 13, 29] and 8 kbps for more sophisticated multi-cell designs [25, 28]. DMDs, on the other hand, use the same (fast) operating principle for the rise and fall times. We exploit this fast switching speed to increase the data rate of passive links by an order of magnitude or more.

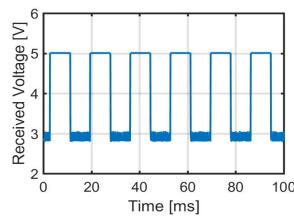
#### 3.2 DMD basics

A DMD is an optical micro-electro-mechanical system (MEMS) that contains between a few hundred thousand and several millions of highly reflective microscopic mirrors of less than 10 microns each. A DMD can be controlled by electrical pulses, which flip each mirror to one of two fixed directions, for example,  $+12^\circ$  and  $-12^\circ$ . DMDs usually come integrated within a sophisticated projector system called Digital Light Processing (DLP). Besides the DMD, the DLP has a lamp, a light absorber and a projection lens, as shown in Fig. 4a. A micro-mirror is *on* if its angle is tilted towards the projection lens, and *off* if the angle is tilted towards the light absorber. All these optical and electrical components are tightly synchronized by the DLP controller.

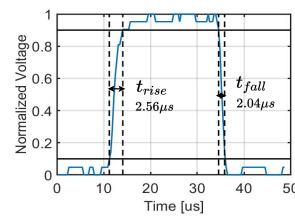
There are multiple types of DLPs, as shown in Table 4. All these DLPs tackle *Limitation-1* because DMDs have a high reflective coefficient by design, but exploiting the DMDs’ potential for higher bandwidth is harder to attain (*Limitation-2*). On one hand, there are inexpensive units, such as the DLP2000 ( $\sim €100$ ), but their screen refresh rate is too slow. The refresh rate can be seen as the equivalent of the rise (or fall) time in an LC. At 120 Hz, the DLP2000 is even



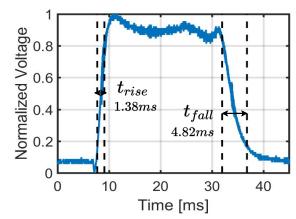
(a) States: ON/OFF (right/left pixel). Reprinted from [12].



(b) Maximum data rate of out-of-the-box DMD



(c) DMD rise/fall time with custom controller



(d) LC rise/fall time

Figure 4: DMD Pixel states and DMD and LC timing characteristics.

slower than the LC shown in Fig. 4d, which provides 320 Hz ( $\frac{1000 \times 2}{1.38 + 4.82}$ ). On the other hand, there are units providing refresh rates above 20 kHz, but with prices beyond €4K, they are prohibitively expensive compared to LC-based systems, which cost a few tens of Euros. A single DMD device (instead of an integrated unit) has comparable cost (€26) to a LC.

The inability to exploit DMDs is an important barrier in passive-VLC. While there are multiple studies utilizing LCs, there are only a few utilizing DMDs. One of those studies uses the same DMD we use, the DLP2000, but attains only a few bits per second because they only use the default refresh rate (120 Hz) and utilize a smartphone camera as a receiver, which is inherently slow [2]. The other studies utilize the more sophisticated DLP4500 (€1100) [10, 11], which provides a maximum refresh rate of 4.2 kHz. Those studies, however, do not exploit that refresh rate for digital communication, but to generate analog signals of just a few tens of Hz (sine, square, triangle, saw-tooth) for localization and audio transmissions. We design a controller for the inexpensive DMD inside the DLP2000 (€26) and increase its refresh rate to 220 kHz, almost a factor of ten faster than the most expensive DLP (DLP9500, €4400). Next, we describe the main limitation of the DLP2000 for ambient light communication, and subsequently, the design of the PhotoLink controller.

### 3.3 Limitations of inexpensive DMDs.

The DMD from the DLP2000 is the most readily available and economical product, but it is designed for display applications. Hence, for ambient light communication, logical 1s and 0s can only be conveyed as a series of white and black images in a video, which leads to the slow update rate shown in Fig. 4b. In a video application, the pixel's color is obtained by (i) multiplexing RGB beams and (ii) changing the duty cycle of the mirror for each color beam. DMDs provide incredible images, with up to 16.7 million colors, thanks to the fine-grained duty cycle provided by the micro-mirrors. *The micro-mirrors can be flipped at very high speeds between their on/off status, enabling short operational periods  $\tau$ , with  $\tau \ll T$ .* These short periods allow a large number of (primary color) combinations. The operation of DMDs, however, is designed for the human eye, which has a slow response. As long as the  $3T$  period takes less than 8.3 ms (120 Hz), people will only see high quality videos. Photodiodes, on the other hand, have MHz bandwidth and do not need to capture colors. For

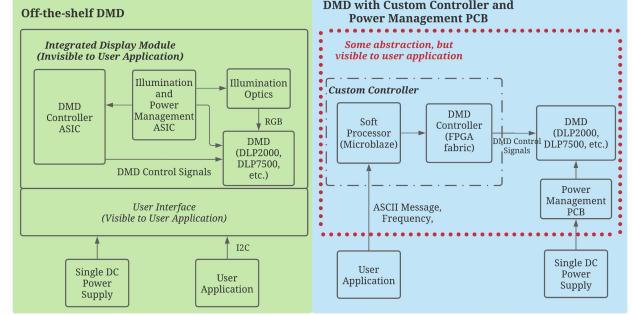


Figure 5: The block diagrams of an off-the-shelf DMD (green, left), and our custom PhotoLink controller (blue, right)

PhotoLink, we need control of  $\tau$ , not  $T$ . Thus, our goal is to remove the controller in the original system and design a new one that gets us as close as possible to the bare fast switching speed of the micro-mirror.

### 3.4 PhotoLink controller

There are two main obstacles preventing the use of inexpensive DMDs for ambient light communication: no suitable *hardware abstractions* or *operational modes*. Next, we describe each obstacle and the solutions we provide.

#### 3.4.1 Hardware abstraction

Most commercial DLPs do not expose control and power signals to user applications, as illustrated in Fig. 5. There are two ASIC components preventing direct access to these signals: the controller and power management. The *controller* implements the logic to set each micro-mirror and an I2C interface. The interface is the only means to communicate with user applications and hides all control signals. It is therefore impossible to extend beyond the supported frame rate by the controller (120 Hz for the DLP2000). The *power management* controls the DMD power and the integrated RGB light source, which is not needed for ambient light communication.

To increase the refresh rate of the DMD, we remove all hardware components from the original DLP design and use only the DMD. As shown in Fig. 5 (blue side), our main components are: (i) the power management unit, which provides the necessary voltage supplies for different DMD operations without requiring a light source; (ii) an FPGA, which supplies

Table 4: Commercial DMD Module

Name	Clock Rate	Data Bus	Screen Refresh Rate	# of Pixels	Module (DLP) Price	DMD Price	# of pins
DLP2000	60-80 MHz	12(bits)x1	120 Hz	640x360	€109.01	€26.14	42
DLP4500	80-120 MHz	24(bits)x1	4.2 kHz	912x1140	€1106.49	€144.69	80
DLP7000	200-400 MHz	16(bits)x2	32.5 kHz	1024x768	€4144.09	€866.96	203
DLP9500	200-400 MHz	16(bits)x4	23.1 kHz	1920x1080	€4403.30	€2693.38	355

the data and logic for updating the DMD; and (iii) the Microblaze module (soft-processor), which runs on the FPGA and provides a user interface but without hiding the control logic. This interface is used to configure the packet format and the transmitting frequency (explained in section 4).

### 3.4.2 Operational modes.

Creating a new hardware abstraction is necessary but insufficient to use the DLP2000 for ambient light communication. The next step is to apply the appropriate operational mode to switch the mirrors as fast as possible. The manufacturer does not disclose all the required information to tackle this step, so we base our design on two references: the data sheet of the DMD [9] and a basic description of micro-mirrors [12].

The switching of the mirrors involves two steps: the *memory state* and *micro-mirror state*. In the memory state, the value of each mirror is set (on/off), but the mirror does not tilt. In the micro-mirror state, an actuation pulse tilts the mirrors to their new value. These states define two operational modes.

**Individual pixel mode.** In this mode, every pixel acts as an individual binary reflector. This allows the DMD to be configured as a fine-grained video projector. The DLP2000 has more than 230 thousand pixels, whose memory has to be written sequentially. As a result, the memory state takes a few hundred  $\mu\text{s}$  before any actuation (transmission) can be performed.

**Global mode.** Considering that the bulk of the delay is in the *memory state*, it would be ideal to by-pass it. In ambient light communication, a fine-grained control of the DMD is not necessary, as photodiodes are used as receivers<sup>2</sup>. It is sufficient to update all pixels at once and use the DMD as a single-pixel device, which we dub the global mode. In this mode, we do not write the memory of each pixel, but instead write a global '0' or '1' to all pixels. Attaining this operation requires a careful coordination of various signals<sup>3</sup>, but the bandwidth increases dramatically compared to the original DLP design, as shown in Fig. 4c: 60 Hz vs 217.4 kHz, a factor of 3600+<sup>4</sup>. Compared to the LC, the global mode reduces the rise time by a factor of 540 (2.56  $\mu\text{s}$  vs. 1.38 ms) and the fall time by a factor of 2360 (2.04  $\mu\text{s}$  vs. 4.82 ms), which translates to almost a 1350 increase in bandwidth.

<sup>2</sup>To take advantage of the individual pixel model, a camera has to be used as a receiver, which is slow (hundreds of frames per second) and requires the use of large screens as transmitters to be efficient.

<sup>3</sup>The hardware and firmware of our controller will be made open source.

<sup>4</sup>The refresh rate of the DLP2000 is 120 Hz, which considers only the time taken by the rise or fall time, the bandwidth considers both times.

### 3.4.3 Summary of contributions.

Our novel controller allows inexpensive DMDs to be decoupled from their integrated video-projection system. We design a global mode to take full advantage of the fast switching times of micro-mirrors. Compared to LCs, our approach increases the transmitter bandwidth by more than three orders of magnitude. Our controller also achieves a higher refresh rate, even when compared to the high-end DLPs shown in Table 4. Since all DMDs manufactured by TI follow the same operating principles [12], our controller's design would also apply to those DLPs, which could allow them to increase their refresh rates to attain even a better performance than the one obtained with the low-end DLP2000.

## 4 Optical Link

### 4.1 Modulation

The majority of modulation schemes fall within two categories: amplitude-based [13, 29] and frequency-based [3]. Amplitude-based methods work well in dark scenarios but are prone to errors when external light sources are present. Frequency-based modulation, on the other hand, has the inherent property of being more resilient to external noise. However, prior LC studies using frequency-based methods had difficulties creating stable periodic signals because the rise and fall times of LCs are asymmetric [3]. DMDs have symmetric times, which allows the generation of stable periodic signals.

To increase the data rate, we use M-ary FSK (MFSK) with two bits per symbol. This high frequency band of PhotoLink (217.4 kHz) allows us to define different modulation parameters and data rates, as shown in Table 6. For example, for a data rate of 30 kbps, we set the four modulating frequencies to 15 kHz, 30 kHz, 45 kHz and 60 kHz. The different modulation parameters permit a thorough evaluation of PhotoLink under different ranges and with different receivers, as discussed in the next section. To avoid abrupt transitions between two frequency signals, the transition between the MFSK frequencies only occurs after a full oscillation period, as depicted in Fig. 6. Considering that the only prior work using MFSK for passive-VLC is [3], we use it as a baseline for comparison. We implement a similar data link layer (shown in Table 5) and receiver design (shown in Fig. 7b and described in Sec. 5). Our packet starts with a SYN symbol (00010101) that uses only the lower transmitting frequencies (00 & 01). These low frequencies have the highest amplitude, and hence, it is easier for the receiver to discover the signal and synchronize to the phase of the transmitter. The ASCII payload is preceded by a STX (Start of Text, 00000010) and followed by ETX (End

Table 5: The structure of the data link layer.

00010101	00000010	ASCII Byte Array	00000011	00010111	00010101
SYN	STX	ASCII Text Message	ETX	ETB	SYN

Table 6: Parameters for different bit rates.

Bit rate	Symbol	Frequency	# of cycle
24 kbps/30 kbps/	00	12/15/20/30/40/50 kHz	1
40 kbps/60 kbps/	01	24/30/40/60/80/100 kHz	2
80 kbps/100 kbps	10	36/45/60/90/120/150 kHz	3
	11	48/60/80/120/160/200 kHz	4

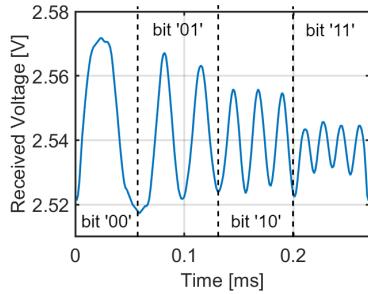


Figure 6: The received signal for different symbols for 100 kbps. Each symbol carries two bits.

of Text, 00000011) and ETB (End of Transmission Block, 00010111).

## 4.2 Demodulation

The receiver knows the transmitting frequencies and takes the following steps to demodulate the signal.

Preamble detection: A sliding window, equivalent to one symbol, applies a Fourier transform (FFT) to the received signal and decodes the symbol. Every time a byte (four symbols) is decoded, the byte is compared to SYN.

Data demodulation: After a SYN byte is identified, the receiver decodes the incoming message using the same FFT process. If an ETX is received, the packet transmission ends.

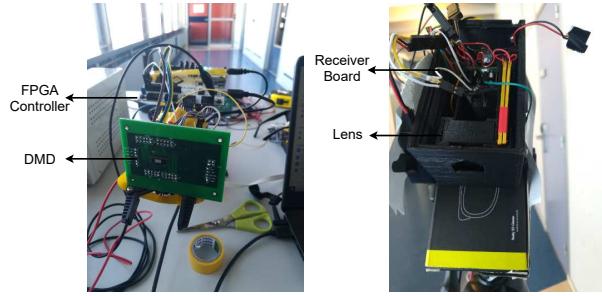
Phase correction: If a received two-bit symbol is '00', during the preamble detection or data demodulation, the receiver leverages the presence of this high-amplitude symbol to synchronize to the phase of the transmitter and adjust to any frequency shift that could have been induced by the channel.

## 5 Evaluation

Our transmitter runs the methods described in Sec. 3 using a custom FPGA controller board and a custom PCB with power management circuits for the DMD. Next, we evaluate our simulation toolbox and controller under various aspects.

### 5.1 Receiver Design & Data Rate

The design of a low-power optical receiver needs to balance a trade-off between gain and bandwidth. If we optimize for sensitivity (high-gain), small changes in light intensity can be detected, which is advantageous for long-distance communication; but the response is slow (low-bandwidth), which limits the ability to decode high frequency carriers. The opposite trade-off holds for a high-bandwidth receiver. A low-



(a) Transmitter setup

(b) Receiver setup

Figure 7: Transmitter and receiver setup

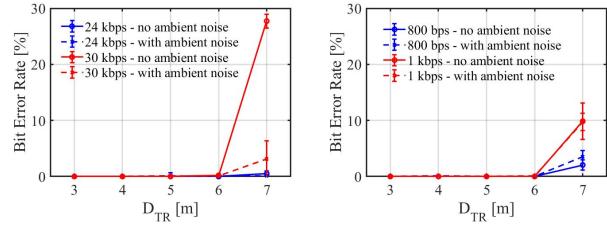


Figure 8: Bit error rate of DMD and LC with artificial light

bandwidth receiver is not a concern when LCs are used as transmitters, as the bandwidth of the LC is low, but it can severely restrict the performance of DMDs. In this subsection, we compare the performance of PhotoLink with LCs used in state-of-the-art studies. we quantify the performance of PhotoLink with two receivers, one optimized for LC operation and the other for DMD. LCs can be used in different ways depending on the type of application: as part of a reflective tag, where either a diffusive or retro-reflective material is placed behind the LC to reflect light, or as part of a transmissive tag, where the LC is used solely as an optical shutter without additional reflective surfaces. To ensure a fair comparison between DMDs and LCs, in the following evaluation, we carry out experiments with both optical devices, but without adding any additional surfaces.

Experiment 1: Receiver optimized for LCs. PhotoLink increases the data rate by a factor of 30.

In this experiment, we use a receiver similar to the one used in [3], consisting of a convex lens with a diameter of 2.5 cm and a TEPT4400 photosensor placed at the focal distance of the lens. The TEPT4400 is a high-gain low-bandwidth photoresistor well-suited for long-range communication with LCs. Using the same illumination environment, we test this receiver using a DMD and an LC as transmitters. The LC and the DMD have the same physical setup, surface area, modulation and demodulation schemes.

Table 7: Rise and fall time for different sensors and resistors

Photoreceiver	Feedback resistor	rise time	fall time
TEPT4400	69/50/20 k $\Omega$	100/64/24 $\mu$ s	140/105/48 $\mu$ s
PD204-6C	1000/400/100 k $\Omega$	6.4/3.5/2.5 $\mu$ s	6.2/3.2/2.1 $\mu$ s

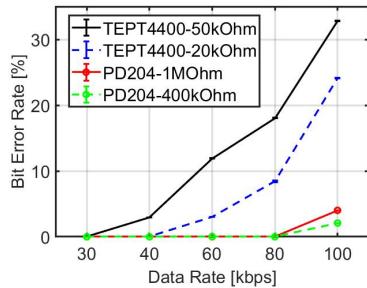


Figure 9: Bit error rate with different sensors and resistors.

We evaluate the DMD and LC in two scenarios. First, we use a bike flashlight (Simson USB Headlight "Future") to illuminate the transmitting surface in a dark room, such that repeatable experimental results can be obtained. The flashlight is placed 1 m away from the transmitter, and the illuminance at the transmitter is 1800 lx. Then, we use the same setup but turn on the indoor lights located on the ceiling and allow natural ambient light to enter the room (in addition to the flashlight). These additional light rays are not aligned with the receiver and act as ambient noise. The illuminance of the ambient noise (excluding the flashlight) is around 700 lx at the transmitter.

In each experiment, a "Hello world!" packet is sent 100 times. Each experiment is repeated 30 times, and the mean and standard variation of the bit error rates are shown in Fig. 8. With the DMD, we obtain an average BER of less than 1% at 7 m for a data rate of 24 kbps and 6 m for a data rate of 30 kbps. With the LC, we obtain an average BER of less than 1% at 6 m for a data rate of 800 bps and 1 kbps. The data rates achieved with the LC are in line with what has been reported for single-pixel systems [3, 7, 13, 29]. Overall, under the same illumination and modulation conditions, DMDs achieve a data rate of more than 30 times that of LCs.

*Experiment 2: Receiver optimized for DMDs. PhotoLink increases the data rate by a factor of 80.*

Considering that the maximum data rate achieved by the SoA is 8 kbps [28], a 30 kbps link is a significant improvement. However, using a receiver optimized for LCs does not exploit fully the capabilities of DMDs. Note from Table 6 that the maximum frequency used for a 30 kbps data rate is 60 kHz, but as stated in section 3, the global mode can reach frequencies above 200 kHz. The limitation of low-bandwidth sensors is that they cannot capture fast transitions: even though the transmitted signal has rise and fall times below 3  $\mu$ s (Fig. 1c), the received signal delivers rise and fall times above 100  $\mu$ s.

At the core of this phenomenon are two parameters, the parasitic capacitance  $C_P$ , which is inherent to the sensor and cannot be modified; and the feedback resistor  $R_F$ , which can be modified. A large  $R_F$  and  $C_P$  improve the receiver's SNR

(high-gain), but reduces the bandwidth. We analyze the effect of the feedback resistor on two photosensors: the TEPT4400 (high  $C_P$ , low-bandwidth) and the PD204-6C (low  $C_P$ , high-bandwidth). Table 7 shows the ability of each receiver-resistor pair to measure the fast DMD transitions for the rise and fall times. The first pair is the configuration used for LCs in [3], and thus, we use it as our baseline. We can see that the PD204 has a bandwidth that is big enough to capture transitions in the few microseconds range.

To showcase the importance of designing an optimal receiver for DMDs, we select four pairs from Table 7, and repeat the same experiment and setup described in Sec. 5.1 but for a fixed distance  $d_{tr}=2$  m. The results are presented in Fig. 9. We know from *Experiment 1* that the TEPT4400 with a 69 k $\Omega$  resistor can attain 30 kbps (baseline). A 50 k $\Omega$  resistor is not low enough to increase the bandwidth significantly, but a resistor of 20 k $\Omega$  can increase the data rate to 40 kbps. The lower capacitance of the PD204, however, increases the bandwidth to a value that is high enough to double the data rate, reaching 80 kbps. Note that the improvement in data rate comes at the cost of reducing the range (lower gain). For the TEPT4400, the range is reduced from 6 m (30 kbps) to 2 m (40 kbps). For the PD204 with 100 k $\Omega$  (last pair in Table 7), the data rate reaches 100 kbps but at ranges shorter than 2 m, and thus, is not presented in Fig. 9. The limited range, however, is not a fundamental problem because it can be increased by adding more amplifier stages at the receiver (our receiver has a single stage) or by using focusing lenses at the transmitter. In the case of the LCs, the bandwidth of any photodetector is much higher than the the bandwidth of the LCs, however, that is not the case for DMDs. We expect that an even higher data rate can be achieved if a photodetector with a high gain bandwidth is used together with multiple stages of signal amplification. The data rate, on the other hand, has been a fundamental limitation for passive-VLC and PhotoLink provides a ten-fold improvement over the most sophisticated LC-system in the SoA. Regarding the cost, DMD-based and LC-based systems can make use of the same photo-receivers, a DMD (€ 28.62) costs € 22 more than an LC (€ 6.56). We use an Artix-7 FPGA, which cost € 50, but a less expensive controller can be used as well. A microcontroller that costs € 13.6 was used in Luxlink [22].

## 5.2 Analyzing the Luminous Flux

Our work has two main contributions, the controller evaluated in the prior subsection and the toolbox presented in section 2. The main insight of our toolbox is the importance of maintaining the luminous flux throughout the optical link. To capture this phenomena, we consider two scenarios.

*Scenario 1: Normalized flux (indoor setup).* In this scenario, we use the baseline receiver (TEPT4400 with a 69 k $\Omega$  resistor) and transmit a fixed carrier frequency. The frequency is empirically chosen to be 30 kHz because this signal can be clearly detected at 4 m without saturating the receiver at 1 m. To calculate the amount of luminous flux maintained

in the optical link, the signal intensity measured at 4 m is normalized with respect to the intensity measured at 1 m for *the same* light source. This normalization process and careful setup quantify the impact of the radiation pattern of each light source independent of its emitted power.

Under this setup, we evaluate four different light sources *indoors*, as shown in Table 8, and simulate the same illumination conditions with our toolbox. In the test setups, the direct and diffuse sunlight arrive at the DMD through a large glass window. To obtain realistic simulation results, we apply the parameters in Table 9, which correspond to the actual physical properties of PhotoLink. The normalization method is also applied to the simulations<sup>5</sup>, and the results are shown in Fig. 10. The plots show a strong agreement between the experimental and simulated flux under all illumination conditions. With diffuse sunlight, we are not able to detect a signal even at 1 m, despite measuring a 2000 lx illuminance on the surface of the DMD. This aligns with our analysis in Sec. 2, which states that diffuse light has the lowest performance with reflective surfaces. The results also show that direct sunlight performs best at retaining the luminous flux (losing 30% at 4 m), followed by artificial lights (losing more than 80%). And with artificial lights, more luminous flux is retained at the receiver when the light is placed further away from the transmitter (setup 2). All these results are in agreement with the insights provided by our model in Sec. 2.

*Scenario 2: Absolute flux (outdoor setup).* In this scenario, we do not perform a normalization process, instead, we transmit 100 packets of "Hello world!" at 30 kbps and present the received voltage and BER. We evaluate the two best light sources identified by our toolbox, flashlight and direct sunlight. The evaluation with direct sunlight is done *outdoors* during a clear day with good sunlight (several thousand lux), and then, moving the setup indoors and placing a flashlight in a dark room.

With sunlight, a BER of  $0.9 \times 10^{-3}$  is achieved at 1 m and a BER of  $0.8 \times 10^{-3}$  is achieved at 2 m. The errors can be attributed to the fact that a link in the outdoor environment is subjected to occasional disturbance. With flashlight, the BER is 0 at 1 m, however, at 4 m, the BER increases to  $19.4 \times 10^{-3}$ . In Fig. 11, we present a direct comparison of the flux reaching the receiver with the flashlight and sunlight using the SYN symbols in the packet. At 1 m, the flux reaching the sensor with the flashlight is slightly lower than that with sunlight (around 0.18 V vs. 0.2 V), which shows that the flashlight and sunlight result in similar voltage range. However, at 4 m, the luminous flux reduces by 60% with flashlight, due to a less directional pattern, while direct sunlight loses only 10%. This result highlights the importance of expanding passive-

<sup>5</sup>Since photodiodes have a quasi-linear response to light intensity, we assume a linear correlation between the obtained signal strength in the toolbox and the voltage obtained in our experiment.

<sup>6</sup>Note that the flashlight loss is higher than the loss predicted in Fig. 10 for 1 m because we place the light closer to the DMD, 30 cm instead of 1 m

Table 8: Measuring the power drop-off with respect to distance

Setup	Light Source	$D_{LM}$	$D_{TR}$	Measured Normalized Signal	Simulated Normalized Signal
1	Direct Sunlight	N/A		0.70	0.73
2	Flashlight	4 m	1 m	0.17	0.20
3	Flashlight	1 m	and 4 m	0.04	0.06
4	Diffuse Sunlight	N/A		N/A	N/A

Table 9: Key parameters used in simulator

Light Source	Dimension	2.7 m x 2.7 m
	Half Angle	30°
Modulating Surface	Dimension	4.8 mm x 2.7 mm
	Light-absorbing area	8 mm by 8 mm
Receiver	Spreading Angle	0.3°
	Lens Dimension	2.5 cm
	Tangent Sphere Radius	(4 cm, -5 cm)
	Lens Material	bk7
	FoV	1°
	Photodiode Diameter	3 mm

VLC studies towards the exploitation of natural light.

### 5.3 Issues with DMDs

DMDs have not been designed for ambient light communication, and hence, present some limitations. We now discuss what we consider the main shortcomings of this MEMS technology for passive-vlc.

*Issue 1: Directionality.* DMDs operate with two fixed angles, which raises up the issue of directionality. If the light source changes its location, the impinging light rays will no longer be aligned with the predefined angles at the DMD, breaking the link. This issue can be overcome with light concentrators. As a proof of concept, we build a simple concentrator with two optical components, as show in Fig. 12a. The first component is a Compound Parabolic Concentrator (CPC). The CPC is a special parabolic lens that collects light from different angles and concentrates it on a small output area. We use a CPC with an input and output circular area of 14 mm and 4 mm diameter respectively, and a concentration factor of 10. The second component is a ball lens of 8 mm diameter, which is used as a coupler and collimator, to further focus the collected light. We manufacture a 3D-case to align the CPC and the ball lens, and aim its output to the DMD.

Fig. 12b presents the results obtained with and without the light concentrator. A flashlight is positioned at different incidence angles and the signal strength is measured at the receiver. Without the concentrator, the signal strength decays below 0.9 with deviations around +/- 1 degree. With the concentrator, the signal remains above 0.9 for angles around +/- 10 degrees. This is a simple implementation, more sophisticated designs can increase the FoV to any desired degree. Thus, while an ideal DMD design for passive-VLC could consider flexible angles, it is not a strict requirement.

*Issue 2: Power consumption.* Perhaps the most limiting factor of current DMD designs is the relationship between its small area and relatively high power consumption. The

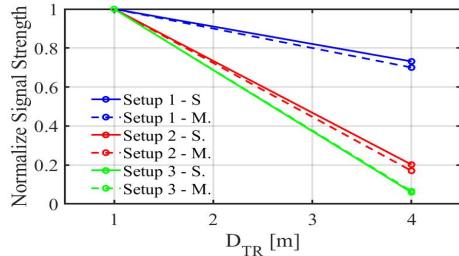


Figure 10: Simulated and measured voltage dropoff.

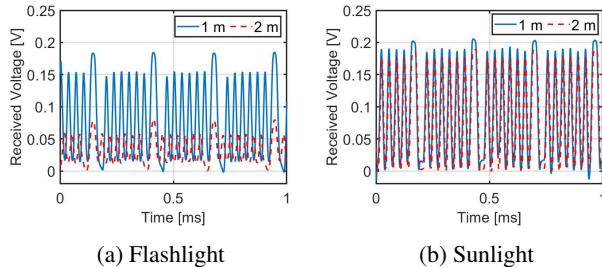
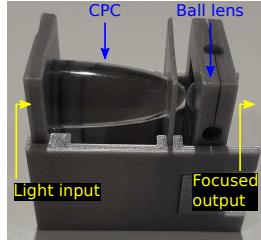


Figure 11: Signal strength

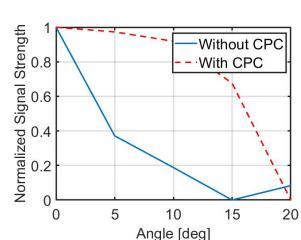
performance of passive-VLC depends on the area of the transmitting surface. For example, a standard light bulb consumes 1 watt to emit 90 lumen, But with an area of  $13 \text{ mm}^2$ , the DLP2000 would emit only between 0.1 and 0.5 lumen.

Regarding the power consumption, current DMD designs have two levels of overhead. The first level is related to the *memory state*, which is not required for PhotoLink and consumes 57 mW in the DLP2000. The second level is the actuation of the micromirrors and consumes 45.5 mW. We are, thus, left with a surface that emits between 0.1 and 0.5 lumen (depending on daylight conditions), while consuming 45 mW. Considering that LCs consume less than 1 mW, and that low-power LEDs consume less than 100 mW, it is central to consider power consumption in the comparison with LCs and LEDs. To perform that analysis, let us consider a low-power LED that has been used in prior VLC studies [8], the VLMB1500, which consumes 75 mW and emits 0.2 lumen. We perform a theoretical comparison between LCs, DMDs and LEDs based on the Shannon-Hartley theorem  $C = B \log_2(1 + SNR)$ . The analysis assumes a luminous flux of 0.2 lumen for the DMD.

First, let us consider LCs, which have areas that are two orders of magnitude bigger than DMDs, and hence, receive two orders of magnitude more lumen. Assuming that the LC receives 20.2 lumen on its incoming area (20 from the light source and 0.2 from the ‘extra’ LED), the outgoing flux would be 10.1 lumen because LCs cut the power in half (Limitation-1 in section 3). Hence, the SNR of an LC-system would increase by a factor of 50 ( $10.1/0.2$ ), but the SNR only contributes logarithmically to the capacity. Overall, the extra SNR would contribute with a factor of 6, compared to the factor of 1350 contributed by the BW of the DMD, making the DMD still two orders of magnitude more competitive.



(a) Design



(b) Evaluation

Figure 12: CPC

To consider the option of using the LED with active-VLC, we measure its rise and fall times, which are  $3.5 \mu\text{s}$  and  $1.6 \mu\text{s}$ : a period of  $5.1 \mu\text{s}$  compared to the period of  $4.6 \mu\text{s}$  for DMDs. Recalling that the LED and DMD emit a similar lumen, the DMD is only slightly more competitive. This result, however, should consider that current DMDs are not designed for ambient light communication. The mirrors of the DLP have a size of microns and use *electrostatic actuation*, larger area mirrors (bigger than 2 mm) “benefit from *electromagnetic actuation proportional to the mirror area*” [17], leading to bigger surfaces with lower power consumption.

Thus far, the passive-VLC community has faced a major obstacle, even with big LC surfaces, no system can provide data rates above 10 kbps. PhotoLink shows that current (sub-optimal) DMDs can provide 100 kbps. Synergies with MEMS researchers could enable the design of bigger modulating surfaces to create wireless networks operating solely with natural light and with low power budgets.

## 6 Related Work

*Passive VLC systems using LCs.* There have been several studies on passive VLC systems, which are summarized in Table 10. To date, LCs have been widely used as an optical transmitter in SoA passive VLC systems. There are two categories: one uses LCs in combination with retro-reflectors, where the light source and the receiver are co-located; the other adopts only the LC as a transmitter, where the light source and the receiver can be placed at different locations, opening up the possibility to take advantage of natural light.

The studies in the former category typically have a constrained data rate and range. The earlier studies [13, 29], achieve a data rate of 0.5 kbps and 1 kbps with ranges up to a few meters. More recently, RetroI2V [25] achieves a range of 80 m. However, it uses a powerful 30 W light to achieve a data rate around 1 kbps. RetroTurbo [28] has a surface area of  $66 \text{ cm}^2$  and uses an advanced modulation scheme to overcome the slow time response of LCs. RetroTurbo achieves a data rate of 8 kbps with a moderate light source (4W flashlight) up to 7.5 m [25]. However, retro-reflectors cannot be used with ambient light, as the light source and the receiver have to be co-located. On the other hand, the studies in the latter category take advantage of the strong illumination from sunlight and are able to achieve a long range without a dedicated illuminator, such as in the case of Luxlink [3] and Chromalux [7]. Luxlink is able to reach a range of 65 m with sunlight, but the

Table 10: Comparison of PhotoLink with the most relevant systems in the state of the art.

Name	$O_L$	$O_L$ Power or Illuminance	$D_{LT}$	$O_T$	Surface Type	$O_R$	FoV	Data Rate	Range
RetroVLC [13]	LED	12 W	Variable <sup>1</sup>	LC+RR <sup>2</sup>	Specular	Photodiode	50°	0.5 kbps <sup>2</sup>	2.4 m
PassiveVLC [29]	Flashlight	3 W	Variable	LC+RR	Specular	PD	4°	1 kbps	1 m
RetroTurbo [28]	Flashlight	4 W	Variable	LC+RR	Specular	PD	20°	8(4) kbps	7.5(10.5) m
RetroI2V [25]	Flashlight	30 W	Variable	LC+RR	Specular	PD	30°	125(1000) bps	101(80) m
Chromalux [7]	Sunlight(Direct) Flashlight	3-6 klx 400-700 lx	N/A N/S	LC and Metal Sheet	Specular	Color Sensor	Variable	1 kbps	50 m 10 m
Luxlink [3]	Sunlight(Direct) LED	10-26 klx 2 klx	N/A N/S <sup>3</sup>	LC and Diffuser	Diffuse	PD	1°	80 bps 1 kbps	65 m 3 m
TwSL [4]	Sunlight(Diffuse)	3 klx	N/A	Paper	Diffuse	PD	N/S	127 bps	4 m
[10]	LED	15 W	cms	DMD	Specular	PD	N/S	4.2 kbps	170 cm
[11]	LED	15 W	cms	DMD	Specular	PD	N/S	9 bps	2.5 m
[2]	Sunlight(Direct)	330 lux	N/A	DMD	Specular	Camera	N/S	1 bps	60 cm
PhotoLink	Flashlight	1800 lx	1 m	DMD	Specular	PD	1°	30(80) kbps	6(2) m

<sup>1</sup> For work involving retro-reflectors as a transmitter,  $D_{LT} = D_{TR}$ .

<sup>2</sup> RR stands for Retro-reflectors.

<sup>3</sup> For work involving retro-reflectors, uplink data rate is quoted.

<sup>4</sup> N/S stands for ‘not specified’.

data rate is limited to 80 bps. It also demonstrates that with an LED, a data rate of 1 kbps can be achieved up to 3 m. Chromalux [7] takes advantage of the transient state in LCs, and is able to achieve a range of 50 m with a data rate of 1 kbps with sunlight, and up to 10 m with a flashlight. While LCs are energy efficient, they suffer from a high attenuation loss due to the use of polarizers, and a limited bandwidth because of its slow rise and fall times. On the other hand, a higher data rate can be achieved with DMDs, but using more power. And in addition to demonstrating a novel system, we provide an analytical framework to understand the performance of different studies.

**Applications of DMDs.** Like LCs, the main application of DMDs is video projection, and thanks to their competitive advantages (high reflective efficiency and switching times) they dominate the market. But DMDs are also used in other applications: microscopy, holography, data storage, and also as spatial modulators with lasers [6]. The use of DMDs for ambient light modulation, however, is restricted to a handful of studies involving localization [10] and communication [2, 11]. And all these studies suffer from a limited data rate (1 bps, 9 bps and 4 kbps), as well as a limited communication range (60 cm, 2.5 m and 170 cm). These systems use the off-the-shelf DMD controllers with their default refresh rates, which fail to take advantage of the fast switching times of the DMDs.

**Channel modelling for VLC systems.** There have been an array of studies on channel modelling techniques for indoor active VLC systems [21], several of which are ray-tracing based [15, 16]. The focus of those studies is to achieve an accurate impulse response considering the dynamics of the VLC system and its indoor environment. They remain a theoretical exercise in most cases, as an accurate description of the indoor space is difficult to obtain. This differs from our work,

as our study focuses on the interactions between different types of surfaces and ambient light.

**Ambient RF backscatter systems.** In RF backscatter, passive devices can communicate with each other utilizing surrounding RF sources. The first study exploited TV tower signals and showed a data rate of 1 kbps at distances of 2.5 feet outdoors and 1.5 feet indoors [14]. Subsequent studies have shown that WiFi, BLE and LoRa signals can also be backscattered, attaining even higher data rates and/or ranges [1, 24]. RF backscattering is an exciting area but requires *man-made* signal (radio towers and antennas), and antennas typically have a limited bandwidth. Ambient light backscattering not only allows exploiting a different part of the electromagnetic spectrum but it can also exploit *natural* sunlight.

## 7 Conclusion

In this work, we propose an optical model to analyze ambient light communication, and based on the insights it provides, we explore the use of a DMD as an optical transmitter. We propose a novel platform that optimizes the retention of the luminous flux to attain the best optical performance. This approach allows us to achieve a data rate that is 30 times higher compared to LCs under the same working conditions. Furthermore, with optimally designed receivers, the data rate reaches 80 kbps. While current DMD designs still face limitations to operate with ambient light, it is a component that expands the possibilities of the nascent area of Passive-VLC.

## Acknowledgments

The authors would like to thank the reviewers and shepherd, Kurtis Heimerl, for their feedback. This work has been funded by the European Union’s H2020 programme under the Marie Skłodowska Curie grant agreement ENLIGHTEM No. 814215, and by the Dutch Research Council (NWO) with a TOP-Grant with project number 612.001.854.

## References

- [1] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. volume 45, page 283–296, New York, NY, USA, aug 2015. Association for Computing Machinery.
- [2] Roy Blokker. Communication with ambient light using digital micromirror devices. Master’s thesis, Delft University of Technology, 2021.
- [3] Rens Bloom, Marco Zúñiga Zamalloa, and Chaitra Pai. Luxlink: Creating a wireless link from ambient light. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys ’19, page 166–178, New York, NY, USA, 2019. Association for Computing Machinery.
- [4] Rens Bloom, Marco Zuniga, Qing Wang, and Domenico Giustiniano. Tweeting with sunlight: Encoding data on mobile objects. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1324–1332, 2019.
- [5] Cisco. Cisco annual internet report - cisco annual internet report (2018–2023) white paper.
- [6] Dana Dudley, Walter Duncan, and John Slaughter. Emerging digital micromirror device (dmd) applications.
- [7] Seyed Keyarash Ghiasi, Marco A. Zúñiga Zamalloa, and Koen Langendoen. A principled design for passive light communication. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom ’21, page 121–133, New York, NY, USA, 2021. Association for Computing Machinery.
- [8] Tilahun Zerihun Gutema, Harald Haas, and Wasiu O. Popoola. Bias point optimisation in lifi for capacity enhancement. *Journal of Lightwave Technology*, 39(15):5021–5027, 2021.
- [9] Texas Instrument. Dlp2000 (.2 nhd) dmd datasheet, 2019. <https://www.ti.com/lit/ds/symlink/dlp2000.pdf>.
- [10] Motoi Kodama and Shinichiro Haruyama. Visible light communication using two different polarized dmd projectors for seamless location services. In *Proceedings of the Fifth International Conference on Network, Communication and Computing*, ICNCC ’16, page 272–276, New York, NY, USA, 2016. Association for Computing Machinery.
- [11] Motoi Kodama and Shinichiro Haruyama. Pulse width modulated visible light communication using digital micro-mirror device projector for voice information guidance system. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0793–0799, 2019.
- [12] Benjamin Lee. Introduction to ±12 degree orthogonal digital micromirror devices (dmds), 2018. <https://www.ti.com/lit/an/dlpa008b/dlpa008b.pdf>.
- [13] Jiangtao Li, Angli Liu, Guobin Shen, Liqun Li, Chao Sun, and Feng Zhao. *Retro-VLC*: Enabling battery-free duplex visible light communication for mobile and iot applications. In Justin Manweiler and Romit Roy Choudhury, editors, *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile 2015, Santa Fe, NM, USA, February 12-13, 2015*, pages 21–26. ACM, 2015.
- [14] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Golakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. *SIGCOMM Comput. Commun. Rev.*, 43(4):39–50, aug 2013.
- [15] Francisco J. Lopez-Hernandez, Rafael Perez-Jiminez, and Asuncion Santamaría. Ray-tracing algorithms for fast calculation of the channel impulse response on diffuse IR wireless indoor channels. *Optical Engineering*, 39:2775–2780, October 2000.
- [16] Farshad Miramirkhani and Murat Uysal. Channel modeling and characterization for visible light communications. *IEEE Photonics Journal*, 7(6):1–16, 2015.
- [17] Pamela Rae Patterson, Dooyoung Hah, Makoto Fujino, Wibool Piyawattanametha, and Ming C. Wu. Scanning micromirrors: an overview. In Yoshitada Katagiri, editor, *Optomechatronic Micro/Nano Components, Devices, and Systems*, volume 5604, pages 195 – 207. International Society for Optics and Photonics, SPIE, 2004.
- [18] Yury Petrov. Optometrika, howpublished = <https://github.com/caiuspetronius/optometrika>, 2014.
- [19] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.
- [20] PureLiFi. <http://purelifi.com/>, 2021.
- [21] A.M. Ramirez-Aguilera, J.M. Luna-Rivera, V. Guerra, J. Rabadan, R. Perez-Jimenez, and F.J. Lopez-Hernandez. A review of indoor channel modeling techniques for visible light communications. In *2018 IEEE 10th Latin-American Conference on Communications (LATINCOM)*, pages 1–6, 2018.
- [22] Yu-Xuan Ren, Rong-De Lu, and Lei Gong. Tailoring light with a digital micromirror device. *Annalen der Physik*, 527(7-8):447–470, 2015.

- [23] Nils Ole Tippenhauer, Domenico Giustiniano, and Stefan Mangold. Toys communicating with leds: Enabling toy cars interaction. In *2012 IEEE Consumer Communications and Networking Conference (CCNC)*, pages 48–49, 2012.
- [24] Nguyen Van Huynh, Dinh Thai Hoang, Xiao Lu, Dusit Niyato, Ping Wang, and Dong In Kim. Ambient backscatter communications: A contemporary survey. *IEEE Communications Surveys Tutorials*, 20(4):2889–2922, 2018.
- [25] Purui Wang, Lilei Feng, Guojun Chen, Chenren Xu, Yue Wu, Kenuo Xu, Guobin Shen, Kuntai Du, Gang Huang, and Xuanzhe Liu. Renovating road signs for infrastructure-to-vehicle networking: A visible light backscatter communication and networking approach. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom ’20, New York, NY, USA, 2020. Association for Computing Machinery.
- [26] Zixiong Wang, Dobroslav Tsonev, Stefan Videv, and Harald Haas. On the design of a solar-panel receiver for optical wireless communications with simultaneous energy harvesting. *IEEE Journal on Selected Areas in Communications*, 33(8):1612–1623, 2015.
- [27] Maury Wright. Philips lighting deploys led-based indoor positioning in carrefou, 2015. <https://goo.gl/a0tGIj>.
- [28] Yue Wu, Purui Wang, Kenuo Xu, Lilei Feng, and Chenren Xu. Turboboosting visible light backscatter communication. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’20, page 186–197, New York, NY, USA, 2020. Association for Computing Machinery.
- [29] Xieyang Xu, Yang Shen, Junrui Yang, Chenren Xu, Guobin Shen, Guojun Chen, and Yunzhe Ni. Passivevlc: Enabling practical visible light backscatter communication for battery-free iot applications. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom ’17, page 180–192, New York, NY, USA, 2017. Association for Computing Machinery.
- [30] Zhice Yang, Zeyu Wang, Jiansong Zhang, Chenyu Huang, and Qian Zhang. Wearables can afford: Lightweight indoor positioning with visible light. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’15, page 317–330, New York, NY, USA, 2015. Association for Computing Machinery.

# Whisper: IoT in the TV White Space Spectrum

Tusher Chakraborty, Heping Shi, Zerina Kapetanovic<sup>†</sup>, Bodhi Priyantha, Deepak Vasisht<sup>‡</sup>, Binh Vu, Parag Pandit, Prasad Pillai, Yaswant Chabria, Andrew Nelson, Michael Daum, and Ranveer Chandra

*Microsoft, <sup>†</sup>University of Washington, <sup>‡</sup>UIUC*

## Abstract

The deployment of Internet of Things (IoT) networks has rapidly increased over recent years – to connect homes, cities, farms, and many other industries. Today, these networks rely on connectivity solutions, such as LoRaWAN, operating in the ISM bands. Our experience from deployments in multiple countries has shown that such networks are bottlenecked by range and bandwidth. Therefore, we propose a new connectivity solution operating in TV White Space (TVWS) spectrum, where narrowband devices configured for IoT can opportunistically transmit data, while protecting incumbents from receiving harmful interference. The lower frequency of operation extends the range by a factor of five over ISM bands. In less-densely populated area where larger swaths of such bandwidth are available, TVWS-based IoT networks can support many more devices simultaneously and larger transmission size per device. Our early experimental field work was incorporated into a petition to the US FCC, and further work influenced the subsequent regulations permitting the use of IoT devices in TVWS. We highlight the technical challenges and our solutions involved in deploying IoT devices in the shared spectrum and complying with the FCC rules.

## 1 Introduction

The growth in IoT is accelerating and expanding across a wide variety of industries. Networking has emerged as a fundamental challenge for IoT. Current solutions like LoRaWAN rely on narrowband (NB) connectivity in the ISM bands, such as US915, EU868, CN779, and so on [23]. However, as IoT networks continue to expand, they run into bottlenecks of these networking solutions. Consider an agriculture scenario, where IoT devices are used to enable precision agriculture techniques on farms in remote areas. These farms can span tens of thousands of acres. LoRaWAN has a communication range of up to 2.5 miles [4]. To connect such vast coverage areas, multiple LoRa gateways need to be deployed and maintained, where deploying a single gateway may cost thousands of US dollars. Besides, setting up proper backhaul connectivity for a gateway is cumbersome in remote areas. It holds for several other scenarios, such as oil and gas fields, power grids, wind farms, and so on. Furthermore, ISM bands have a limited bandwidth, e.g., only 8 MHz in EU868 and 26 MHz is US915 where bandwidth allocated for downlink communication is even smaller (see Section 2). Therefore, it becomes challenging to support IoT applications such as

heatmap-based monitoring and plant stress monitoring using cameras, where comparatively larger volumes of data traffic is required to be transmitted over a low data rate long-range network [7, 17, 18]. In such cases, the combination of longer-range and larger available bandwidth is required.

To bridge the gap, we envision enabling IoT networks over the TV white spaces (TVWS). TV white spaces are the allocated, but unused channels in the VHF and UHF broadcast TV bands that can be leveraged for both high and low bandwidth data transmission. There are several advantages of utilizing TVWS spectrum for a NB IoT deployment over ISM bands. As the TV band spectrum consists of lower frequencies than the 800/900 MHz ISM bands, it facilitates longer-range connectivity which extends to 10s of miles, with non-line of sight (NLOS) operations, and even through some obstructions. Consequently, it opens the door of covering a large-area IoT deployment with one or a minimal number of gateways. It even facilitates higher data rates for distant IoT clients which in turn provides power savings on the IoT device. In addition, in less densely populated areas, there are typically several unused TV channels available for use by TVWS devices. The actual number of available channels vary by location. In the aggregate, these available channels can offer a large bandwidth, and hence support for many more simultaneous communication channels and increased traffic.

However, in the way of realizing this vision, the challenges are twofold – regulatory and technical. While operating in a dynamic spectrum as the unlicensed user, the precondition is to ensure the protection of incumbents from receiving harmful interference and the inability to claim protection from interference. Although this challenge has been addressed in the case of unlicensed broadband devices operating in the TVWS [5], it is non-trivial to extend the same to NB devices due to their limited power budget and distinct regulations for NB operation, such as channel occupancy limit (Section 3). We identify three corresponding challenges below:

- First, in a large-area deployment, the data rate of sparsely deployed clients (Figure 7c) served by a single gateway becomes highly variable as the data rate is inversely proportional to the distance. Single configuration setting of slow data rate (longer range) for all the clients, will result in throughput loss and power overhead. On the other hand, mainstream IoT MAC protocols, mainly designed for ISM bands, cannot make the best utilization of wide TVWS spectrum in serving large traffic even using a gateway with multi-data rate support on a single channel (Section 7.2).

- The second challenge is handling the spatio-temporal dynamism in TVWS channel availability and quality. The dynamism is mainly due to the channel occupancy by nearby licensed users (e.g., TV station, wireless microphone, etc.) and unpredictable unlicensed users (e.g., TVWS broadband network). Moreover, the long spatial separation between the gateway and client devices implies that uplink and downlink may operate on different channels with dissimilar quality. Finally, the power constraint of IoT client devices adds a curb on dynamic spectrum access and management.
- The final challenge is to develop an efficient carrier sensing solution to detect the presence of an interfering RF transmission from incumbents – both licensed and unlicensed – in a dynamic spectrum. With LoRa modulation, the devices can communicate even when the signal level is below the ambient RF noise floor. Hence, the conventional approach of simply measuring RF energy level in a NB channel to detect RF interference does not work.

Over the years, we have worked on devising Whisper, an end-to-end IoT network system over the TVWS spectrum which addresses both regulatory and technical challenges. Our early field work, authorized under an experimental license, led to a proposal on NB TVWS device operations which was the part of a broader 2018 petition for rulemaking to the US Federal Communications Commission (FCC) for expanding its rules for TVWS devices. Later, our work supported the FCC’s December 2020 decision to adopt regulations on NB white space devices to operate in the VHF and UHF bands below 602 MHz [10]. In addition, we make the following contributions through this work.

**Whisper Protocol:** We design a new MAC protocol for a star-topology IoT network operating in the TVWS spectrum. Our Frequency Time Division Multiple Access (FTDMA) based design supports larger traffic along with diversity in the data rate of sparsely deployed clients. It leverages a dynamic binary counting table with the linear Diophantine equation for formalizing and optimally limiting the channel occupancy to protect the incumbents. The protocol further incorporates a smart approach for handling the dynamism in the TVWS spectrum given the power limitation of IoT devices.

**Whisper Hardware:** We design and develop a NB Whisper radio that operates in the continuous spectrum ranging from 150MHz to 960MHz. The radio uses LoRa modulation at the physical layer. Given that and the above-mentioned challenge in corresponding carrier sensing, we further develop a spectrum sensing module that uses a locally generated signal by a Whisper radio to measure the RF interference from the incumbents in individual NB TVWS channels.

**Real-world Deployment:** Finally, we make a real-world deployment of Whisper as an end-to-end IoT network system for more than 2.5 months. The deployment covers 17 fields in a 8500 acre farm with single gateway and 20 IoT devices. The sensor data collected via Whisper is used by third-party users in multiple agriculture applications including food tracing,

Data type	Area (acre)	#gateways	#clients	Traffic (bytes/hr)	Prominent issue
Sensor	1700	3	11	3.2k	Range
Sensor	350	2	20	0.9k	Range
Sensor	700	3	9	0.5k	Range
Image	8500	2(Abortive)	20	550k	Bandwidth

Table 1: Setup of multiple real-world IoT deployments where Whisper would benefit. These are representative data from our deployments across the globe using ISM band LoRa.

data-driven farming [32], and carbon monitoring.

From our real-world deployment, we find that Whisper facilitates at least 5x range improvement over LoRa operating in the 800/900 MHz ISM band and at least 3x over state-of-the-art modulation techniques proposed for NB operation in TVWS [27, 28]. Furthermore, our simulation shows that using only 3 white TV channels (6 MHz each), Whisper can handle at least 5x traffic compared to ISM band.

## 2 Motivation from Real-world Experience

The need for deploying IoT devices in TVWS spectrum is motivated by the bottlenecks experienced in our real-world deployments using LoRa operating in the ISM bands. We highlight two application scenarios here.

The first application scenario emanates from one of our earlier projects, FarmBeats, that aims to enable data-driven agriculture [32]. To do so, we deploy sensors across a farm and aggregate data in a star-topology LoRa network operating in 800/900 MHz ISM bands. We have made more than 30 research deployments in farms across the globe (including US, South Asia, Europe, Africa, Asia Pacific, etc.) over a period of 4 years. The top three entries in Table 1 are representative of the setup of these deployments. The major challenge we have experienced in these settings is the relatively short range of the communication link compared to the size of a farm and sparsely deployed IoT clients. The average maximum achieved range is 1.12 miles combining both NLOS and LOS settings. Consequently, we need to deploy multiple gateways to cover a farm, even when we need to support just a small number of sensors spread across the farm. Whereas, TVWS spectrum offers a range of tens of miles, and thus, reduces the number of gateways as we show in Section 7.

In the second application scenario (bottom entry in Table 1), we study the feasibility of monitoring plant stress using a camera in US915 ISM band [24]. Each client sends an image of  $\sim 25$  kB where the gateway can expect at least 22 images per hour. Here, the foremost problem is sending a large number of confirmed-up frames as the ISM band suffers from the paucity of bandwidth in two levels. First, the dwell time restriction of 0.4 sec enforces sending an image in a large number of small uplink frames. It, in turn, increases the load on the downlink (allocated bandwidth is 4 MHz) with a large number of ACKs. For example, if we consider the median uplink data rate (DR2) supported in US915 band, it takes around 230 uplink frames for an image [3]. On the downlink side, with a

comparable data rate (DR12), a gateway can serve ACKs for maximum 7 images per hour without even considering any frame loss and the inefficiency of existing MAC protocols in handling confirmed-up frames [3, 16]. Multiple existing research work report similar problem [7, 17, 18]. Even one of the largest commercial LoRaWAN service providers, The Things Network, recommends finding an alternative platform in such scenarios [22]. Furthermore, with the aforementioned data rates, the maximum range can be up to a mile. In this scenario, TVWS spectrum offers a larger bandwidth that can easily handle the aforementioned traffic (Section 7.2).

### 3 Regulating NB Operation in TVWS

Although FCC has adopted regulations on NB operation in TVWS spectrum in 2020, we have been working with FCC on it for more than four years. FCC’s Office of Engineering Technology granted us several experimental licenses for operating NB IoT TVWS devices in an agricultural setting. From the beginning, the experimental NB TVWS transmitter and network architecture have been designed with the understanding that the primary users of these frequency bands must be protected from receiving harmful interference. Based on the experience gained over the course of the field tests, we filed a petition for rulemaking at the FCC for expanding TVWS operations that included NB. Next, we describe the key regulations mandated by FCC for NB operation in TVWS spectrum [10].

- Incumbents are protected through a geolocation and database method. The location of the NB is provided through an incorporated geolocation decision, typically GPS [9]. The geolocation information is provided to a white spaces database (WSDB). The WSDB combines information on incumbent users from the FCC Licensing and Management System that is updated daily; information from other users input directly, such as wireless microphones that is updated hourly; and a calculation engine that determines the list of available channels for the TVWS device operating at that location.
- A TV channel in the US is 6 MHz wide. The conducted power and power spectral density limits for broadband TVWS devices are based on 100 kHz. Thus, the channel size limit for a NB device is proposed to be 100 kHz. The proposed channel plan requires NB TVWS devices to operate at least 250 kHz from the edge of a 6 MHz TV channel. It implies that NB devices are permitted to operate within 55 possible 100 kHz NB channels in the center 5.5 MHz of each TV channel.
- FCC limits transmissions by a NB TVWS device on each NB channel to a total of 36 sec per hour. It means that different NB channels may be required for the uplink/downlink data communication and interaction with the WSDB.
- The conducted power and power spectral density limits for NB devices per 100 kHz are the same as for broadband

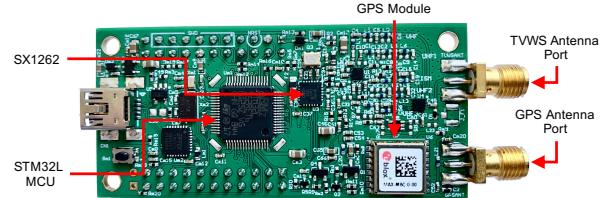


Figure 1: Whisper radio fabricated as an industrial-grade module that can operate from 150MHz to 960MHz.

TVWS devices. Here, the EIRP of a NB TVWS device can be up to 18.6 dBm/100 kHz. The rules for protecting incumbents are based on a scenario where each of the 55 channels of 100 kHz in a TV channel is concurrently being used at its conducted power limit.

We highlight that the NB devices can operate in 174-216 MHz in the VHF band and 470-602 MHz in the UHF band.

### 4 System Design

Whisper is a new IoT system that can support long-range communication at large-scale in the TVWS spectrum. We design Whisper to have two key components similar to a classic star-topology IoT network: an IoT client radio and gateway. In the following subsections, we describe each component.

#### 4.1 Whisper radio

Whisper requires a radio that can operate over the whole TVWS spectrum in both VHF and UHF bands as mentioned in Section 3. We further intend to use LoRa modulation in the physical layer, which is very popular for long-range and low-power wide area network (LPWAN). Since commercial off-the-shelf LoRa radios are designed to operate over narrow ISM bands, we develop a custom NB IoT radio that can operate over a wider TVWS spectrum. Figure 1 shows the fabricated radio which can operate from 150 MHz to 960 MHz including the upper-VHF, UHF, and ISM bands. For modulation and demodulation of LoRa signal, our radio incorporates an SX1262 radio transceiver manufactured by Semtech [12].

Unlike an off-the-shelf radio designed using SX1262 for narrow ISM bands, our design for a wider TVWS spectrum requires careful consideration of avoiding leakage and harmonics in an adjacent TV channel or other licensed bands within the spectrum. The power amplifier stage of the off-the-shelf radio chips typically has low linearity to reduce power consumption. This non-linearity results in RF signals at harmonics of the carrier frequency. In a design for a wide spectrum, harmonics of the lower carrier frequencies can lie within the higher frequencies of the spectrum, resulting in spurious RF emissions. As a remedy, our radio design incorporates a collection of electronically switchable RF filters between the SX1262 radio transceiver and the antenna. The filter cut-off frequencies are selected in a way such that, with the appropriate filter selected, the harmonics of any carrier

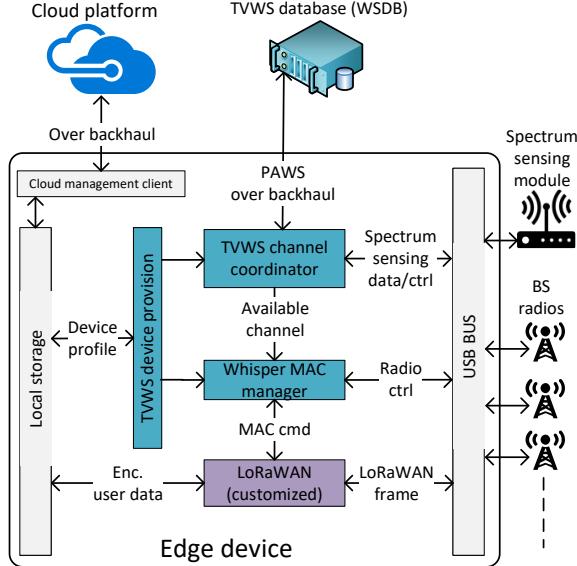


Figure 2: Gateway architecture. Whisper implements it to enable the gateway operation in the TVWS spectrum.

signal within 150 MHz to 960 MHz are filtered out preventing spurious RF emissions. We further carry out laboratory experimentation to ensure that the adjacent channel emissions limits of Whisper radio comply with FCC regulations. Find the setup and results of the experimentation in Appendix A.

Whisper radio has a low-power GPS module to provide its geo-coordinates and height to the WSDB, as required under the FCC rules for unlicensed white space devices (Section 3). To control all of these components and execute our communication protocol, we use an ultra-low-power microcontroller, STM32L151RE based on ARM cortex-M3 architecture [30].

The maximum current consumption by Whisper radio is 119 mA in transmission (at 20 dBm transmit power) and 12 mA in reception with 3.3V power supply. Although the TX energy consumption seems to be higher compared to off-the-shelf ISM band LoRa radio, Whisper radio consumes less energy per given throughput while communicating at a distance of more than a mile. Because the lower frequency of TVWS spectrum enables higher data rate, i.e., less time on air, at a longer distance compared to the ISM band (Section 7.1.4).

## 4.2 Whisper Gateway

Using the Whisper radio as an IoT client device, we also need a gateway to enable end-to-end communication. Figure 2 shows the architecture of Whisper gateway. It consists of multiple Whisper radios, an edge device, and a spectrum sensing module. First, the radios integrated with the gateway are referred to as base station (BS) radios and follow a similar design to the Whisper radio mentioned above. The number of BS radios can be adjusted depending on the scale of application. Next, the edge device is off-the-shelf can be a single board computer (e.g. Raspberry Pi, Up Board, etc.) or even

a laptop PC. Finally, the spectrum sensing module and BS radios are connected to the edge device through a USB hub.

The edge device has the Whisper MAC manager at its core, which facilitates IoT communication over the TVWS spectrum. It administrates the medium access by the clients (Section 5), coordinates network bootstrap followed by data communication (Section 5.4), and handles the dynamism in TVWS spectrum (Section 6). To do so, it requires exchanging MAC commands with the clients. Here, MAC commands and user data are wrapped in standard LoRaWAN frame format [11]. To be specific, we use the security provided by LoRaWAN along with its frame format, however, not any associated MAC protocol. Consequently, we customize the MAC commands according to our protocol.

TVWS channel coordinator prepares the list of available channels to be used in the network. To do so, it directly communicates with the TVWS database (WSDB) for TV channel availability in the region using the standard protocol to access white space (PAWS) [8]. Furthermore, it conducts real-time screening of the uplink NB channels using the spectrum sensing module (Section 6).

## 5 Whisper MAC Protocol

With TVWS, we can reduce the number of gateways in a large-area deployment and still support endpoint devices dispersed at varying distances. A question that comes up is, why can we not take a similar approach that is used in mainstream LPWAN communication such as pure or slotted ALOHA? Unfortunately, these protocols do not perform well in our targeted scenarios [1, 14, 16, 26]. In particular, these protocols are not suitable for applications requiring confirmed-up frames (e.g., cameras for plant stress monitoring) and higher traffic load in a single-gateway network (Section 7.2). Besides, a gateway following these protocols appears to be even more inefficient in handling the diversity in data rate demands by the clients [13, 19]. Most importantly, the dynamic nature of the TVWS spectrum is not considered in existing MACs, since these are primarily designed for ISM band operation.

We design and implement FTDMA based MAC protocol to address these challenges. However, note that there are two apparent overheads of FTDMA based protocol for LPWAN: synchronization frame and scheduling control frame. These are neutralized by leveraging the complementary advantages from FCC-mandated compulsory parts of the system. Here, we utilize the onboard GPS module, mounted to comply with FCC regulations detailed in Section 3, for synchronization purposes. Furthermore, we piggyback the scheduling info in the FCC mandated regulatory control frames.

### 5.1 FTDMA structure

Whisper’s MAC protocol has a custom FTDMA structure at its core. We first introduce the structure and corresponding

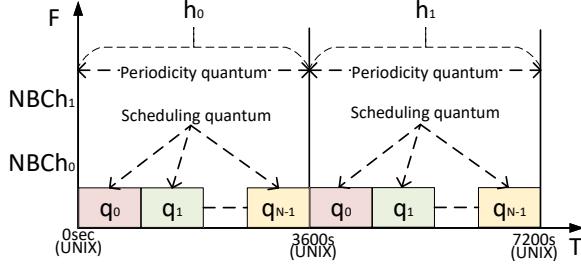


Figure 3: FTDMA structure of Whisper MAC. For communication, each client uses a slot that is a group of consecutive scheduling quanta inside a periodicity quantum.

components. Figure 3 shows a high-level depiction of the proposed FTDMA structure. The T-axis represents time tracked in seconds in the form of a UNIX timestamp. The origin point of the T-axis is starting of the UNIX timestamp which is universal. We divide the T-axis into two dividing time units: periodicity quantum and scheduling quantum. The required periodicity of periodic traffic is basically a multiple of the periodicity quantum. Each periodicity quantum is further divided into an equal number of scheduling quantum. Scheduling quantum ( $q$ ) is the minimum time precision required in the scheduling. Both periodicity and scheduling quantum can be adjusted based on the computation capability of the edge device and the required precision of scheduling parameters for the application scenario. For the application in our real-world evaluation, we use the hour as the periodicity quantum. To keep the coherence in the rest of the paper, we use hour ( $h$ ) in place of periodicity quantum. Here, each hour in T-axis is denoted with  $h_n$  (Figure 3), where  $n$  is the number of hours from the origin point of the T-axis. As mentioned above, we further break each hour down into  $N$  scheduling quanta where  $n^{th}$  one is denoted with  $q_n$ . Now, the F-axis represents the NB channels to be used. Note that if the gateway has multiple BS radios, the T-axis, as well as the set of scheduling quanta, is separate for each radio, however, the F-axis is shared. This allows multiple BS radios to operate simultaneously across different NB channels. The final component of the FTDMA structure is the slot, a group of *consecutive scheduling quanta*. For communication, each client is allocated at least one slot. Note that, in a slot, not more than one uplink (downlink) channel is used. However, multiple uplink (downlink) frames can be transmitted in a slot.

The IoT clients can generate two different patterns of traffic: periodic and event-driven. Here, the timing required for communication depends on the traffic pattern. Depending on the data rate and size of the payload, the time length required for communication varies. We formulate these requirements as the slot requirement in our FTDMA structure. To fulfill the requirement, the slot allocation algorithm of Whisper MAC optimally allocates slots along with communication channels in compliance with the occupancy limit. In the following subsections, we describe the slot allocation algorithm in detail for the two aforementioned traffic patterns.

## 5.2 Slot allocation for periodic traffic

A client generating periodic traffic requires slots at a regular periodicity. We define the requirement as  $\sigma$  scheduling quanta are required with the periodicity of  $p$ , where  $\sigma$  depends on the number of frames, both uplink and downlink, to be communicated and system processing time. One or multiple slots can be allocated having at least  $\sigma$  scheduling quanta in total. Now, the frames to be communicated can have a variable size depending on the data rate and payload size. Therefore, each frame requires at least a certain number of scheduling quanta in a slot for communication. Here, we define continuity ( $\alpha$ ) as the number of scheduling quanta in a slot. Now,  $\alpha_{min}$  is the minimum continuity required for the communication of a frame. Next, we delineate how the slot allocation algorithm of Whisper MAC allocates slots in three phases: scheduling quantum selection, channel selection, and slot allocation.

### 5.2.1 Scheduling quantum selection

The goal of the scheduling quantum selection process is to find a set of  $\sigma$  scheduling quanta at every  $p$  hours which are not a part of the existing allocated slots. As the T-axis is separate for each BS radio, we here describe the quantum selection process for single BS radio. We first define assignment,  $A < h_s, p >$ , for a scheduling quantum which implies it is occupied at every  $p$  hour starting from  $h_s$  hour to fulfill a slot requirement. When two assignments of a quantum take place in the same hour, we call it a collision. Consequently, two communication slots corresponding to these assignments collide which is not desirable. Hence, given the new slot requirement, the quantum selection process makes sure that the new assignment for a quantum does not collide with the existing ones. To do so, it leverages the linear Diophantine equation. According to the theorem for solution to linear Diophantine equation, two assignments  $A_1 < h_{s1}, p_1 >$  and  $A_2 < h_{s2}, p_2 >$  collides iff  $|h_{s1} - h_{s2}|$  is a multiple of  $gcd(p_1, p_2)$ . For now, we assume that  $p \in \mathbb{Z}^+, 1hr \leq p \leq 24hr$  (cases outside this boundary are discussed in Appendix B.4). Given the boundary of  $p$ ,  $h_s \in \mathbb{Z}^+, 0hr \leq h_s \leq 23hr$ .

Now, what if any scheduling quantum is not found collision-free for the new assignment? In this case, we modify the new assignment by making it silent in the hours of collision with existing assignments. As a result, the new client will halt its transmission in the slots of those hours. It is apparent that silencing has an effect on the throughput of the new client, and thus, it should be minimum. Here, we introduce a metric  $\omega_q\%$  that measures the frequency of silencing. It can be calculated from the generic solution of the Diophantine equation. See Appendix B.1 for more details.

The scheduling quantum selection process tunes the value of  $h_s$  from  $h_0$  to  $h_{(p-1)}$  for the required periodicity  $p$  and makes the list of collision-free scheduling quanta with and without modifying the new assignment to be silent. For the slot selection algorithm, it sets a priority value ( $\rho_q \in \mathbb{R}, 0 \leq$

$A_{c0}$	$\tau$	$A_{c1}$	$A_{c0}$	$\tau$	$A_{c1}$	$A_{c0}$	$\tau$	$A_{c1}$	$A_{c1}$	$A_{c0}$	$\tau$
0	6	0	0	1	6	0	0	0	0	0	0
1	6	0	1	6	0	1	6	0	0	1	6
		1	0	9	1	0	9	1	0	0	12
								1	0	1	18
								1	1	0	21

Figure 4: Construction of channel occupancy table. Whisper uses it to formulate and optimally limit the channel occupancy.

$\rho_q \leq 1$ ) to every scheduling quantum in the list as following.

$$\rho_q = \begin{cases} 1, & \text{has old } A[], \text{ no collision with new } A \\ 2/3, & \text{does not have assignment} \\ \frac{1-\text{Max}(\omega_q\%)}{3}, & \text{has old } A[] \text{ and collides with new } A \end{cases}$$

A scheduling quantum having existing assignments and no collision with the new assignment gets the higher priority compared to the one with no existing assignment. It ensures optimal usage of a quantum by grouping non-colliding assignments together. Finally, a quantum having existing assignments colliding with the new one gets the lowest priority depending on the frequency of silence.

### 5.2.2 Channel selection

The channel selection process, completely independent of quantum selection, optimally finds the channels for new slot requirement complying with the occupancy limit. First off, if two assignments use the same channel in the same hour, it is a collision in channel usage. However, the collision is safe, i.e., assignments are valid, as long as the total occupancy of the channel in that hour is not more than 36s (Section 3). Based on it, validating a new assignment for a downlink channel is challenging. A downlink channel might be assigned to multiple clients, and these assignments can collide in different combinations and hours. Furthermore, in a gateway having multiple BS radios, although the T-axis is separate for each BS radio, the F-axis is shared, and the channel occupancy is calculated in aggregate for all BS radios. Now, simply avoiding the collision in channel usage by making the assignment silent in the colliding hour or using spare channels result in significant throughput loss and wastage of bandwidth respectively. Therefore, we need to formulate the channel occupancy and optimally limit the same despite collision among assignments in channel usage. Here, we propose binary counting based dynamic table on top of the linear Diophantine equation to validate an assignment for a downlink channel.

Now, we illustrate the dynamic construction of channel occupancy table with Figure 4. We here define the assignment for channel as  $A_c < A, \tau, q[] >$ , where  $\tau$  denotes the channel occupancy (in seconds) per hour for the assignment, and  $q[]$  denotes the list of scheduling quanta associated with it.  $q[]$  is used in tracking the channel overlap in FDMA. As shown in Figure 4, each entry in the table contains the colliding assignments and corresponding total occupancy ( $\tau$ ) in the colliding hour of the assignments. For example, an entry  $< A_{c3} = 1, A_{c2} = 0, A_{c1} = 1, A_{c0} = 1, \tau = 17 >$  means  $A_{c3}, A_{c1}$ , and  $A_{c0}$  collide in channel usage, and the channel occupancy

is 17s in the colliding hour. Now, if there are  $n$  assignments for a channel, there are  $2^n$  possible combinations of assignments. However, the assignments in a possible combination may not collide, and thus it can be ruled out for future computation. For example,  $A_{c0}$  and  $A_{c1}$  do not collide, and thus the last entry in the second table has  $\tau = 0$  that is removed for any further development. Find step by step description of table construction with example in Appendix B.2

The channel selection process leverages the channel occupancy table in finding a valid channel despite the collision between new and existing assignments in an hour. However, the question arises which channel is optimal to select? To do so, we here utilize a priority variable ( $\rho_c$ ) for a channel similar to ( $\rho_q$ ). However,  $\omega_c\%$  is treated differently compared to  $\omega_q\%$ . In case of a collision between the new and existing assignments in channel usage, although the occupancy becomes higher only in the hour of colliding occurrence, the channel remains under-utilized in the rest of the hours of non-colliding occurrences. Consequently, a higher value of  $\omega_c\%$  results in lesser under-utilized hours. Hence, instead of  $\frac{1-\text{Max}(\omega_q\%)}{3}$ , we use  $\frac{\text{Avg}(\omega_c\%)}{3}$  in calculating  $\rho_c$  for a channel. Although it cannot ensure complete eradication of under-utilized hours for a channel, we later show how the channel can be utilized further in these hours for event-driven traffic. In aggregate, for a given assignment  $A_c$ , the channel selection process finds a valid downlink channel using the occupancy table with the maximum possible  $\rho_c$ .

### 5.2.3 Slot allocation

The slot requirement from a client can be satisfied with a single or a combination of slots. From a high level, the slot allocation algorithm works in two steps. First, it prepares different possible combinations of slots using the scheduling quantum selection and downlink channel selection (find the pseudo code in Appendix B.3). Then, it selects the best combination of slots based on the following weight function.

$$f = w_1 \rho_q + w_2 \rho_c + w_3 \alpha - w_4 \psi$$

We have discussed  $\rho_q$  and  $\rho_c$  earlier. Positive weight on higher continuity ( $\alpha$ ) value takes lesser slots to fulfill the requirement of  $\sigma$  quanta, which, in turn, saves the power of the client. On the other hand,  $\psi$  quantifies the wastage of quanta, i.e., how many extra quanta are used in total for a combination of slots. Consequently, a negative weight on  $\psi$  reduces the wastage in bandwidth-sensitive applications. All weights can be further tuned based on the nature of the application.

## 5.3 Slot allocation for event-driven traffic

In event-driven traffic, a frame is generated based on an event-trigger, e.g., rain monitoring. The frame needs to reach the gateway before a certain time. Therefore, we define the requirement as  $\sigma$  scheduling quanta are required with  $\alpha_{min}$  con-

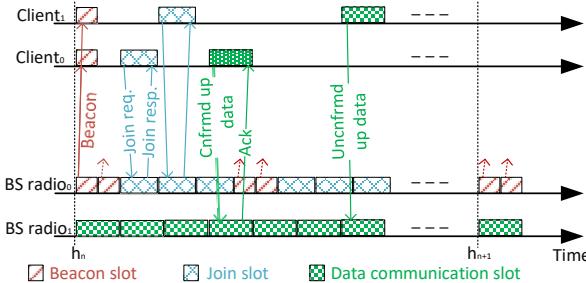


Figure 5: Time sequence diagram of Whisper protocol in a simple example scenario. This example depicts bootstrapping of two clients followed by data communication with the gateway having two BS radios.

tinuity before the expiry,  $\epsilon$ . The underneath concept of slot allocation algorithm for event-driven traffic is mostly similar to the periodic traffic. We point out the *distinct factors* below.

During scheduling quantum selection, we check the collision of an assignment for event-driven traffic with the existing assignments for both types of traffic. However, there is no notion of silencing the new assignment for event-driven traffic as its time-sensitive. Accordingly,  $p_q$  has three different priority levels - highest priority to no collision with existing assignments, then to no existing assignments, and lowest to collision only with existing assignments for periodic traffic. Here, in the last case, even if the existing periodic assignment ends up transmitting frames in the same channel together with the new event-driven assignment, according to the nature of LoRa physical layer, there is a probability of one getting successfully demodulated. In this way, the scheduling quantum selection process enlists all quanta before the expiry,  $\epsilon$ .

Although the underneath concept remains similar, the channel selection for event-driven traffic is more simplistic than periodic traffic. Here, rather than using the channel occupancy table, the validity of a new assignment for a channel is probed by simply checking collision with individual existing assignment for both types of traffic in the hour of the new assignment. If the new assignment collides with existing assignments in channel usage and the total  $\tau$  of all colliding assignments including the new one remains within the limit, we consider it valid. The priority of a channel ( $p_c$ ) for a valid new assignment is decided differently – highest priority to no collision with existing assignments, then to collision with existing assignments, and lowest to no existing assignments. It facilitates the utilization of a channel in no or low utilization hours of the periodic assignments.

Finally, we make a modification in the weight function for slot allocation as following.

$$g = w_1 p_q + w_2 p_c + w_3 \alpha - w_4 \psi - w_5 \delta$$

where  $\delta$  is the time difference between selected scheduling quantum and the expiry ( $\epsilon$ ). A negative weight on  $\delta$  makes the choice for immediate slot less greedy. It, in turn, facilitates serving multiple concurrent requests of event-driven traffic without colliding in the immediate slot.

## 5.4 Bootstrap and Data Communication

Hitherto we discuss the process of slot allocation for controlling the medium access by the clients. We now describe how Whisper MAC manages client bootstrapping and data communication using the slot allocation. Client bootstrapping is a critical part of a TVWS based IoT network since it is required to comply with FCC regulations given the power constraint of the IoT devices. Recall that, each NB TVWS device must provide the WSDB its geo-coordinates and height to obtain the list of available channels for transmission at that location. Here, the clients can register via the gateway as they do not have Internet connectivity. Nevertheless, a key question remains. How does a client share its location with the gateway without knowing the available channel and time for transmitting registration request?

According to the FCC regulations, a client that is not registered in the WSDB can transmit only its location information or network join request on the channels registered against the gateway. To do so, a client requires to know the channels registered against the gateway where the list of these channels may even vary over time. As a remedy, Whisper MAC utilizes broadcast beacons that embed the info of join slots for clients to transmit WSDB registration, i.e., network join request. In the join slot, a client transmits the join request along with its location for WSDB registration and slot requirement for the data communication. Note that the beacon slots have predefined and fixed timing and channels. Therefore, the time synced (using GPS) clients can listen for the beacon frames without draining its power in random or continuous scanning. Figure 5 depicts an example scenario of client bootstrapping followed by data communication. Due to space limitations, we put the corresponding implementation details in Appendix B.5.

After receiving the WSDB registration response, i.e., join response, a client is ready for the data transmission in the allocated slots (piggybacked as MAC command in the response frame) until the expiry of channel. A TVWS device is required to contact (poll) the WSDB once every 24 hours and check the list of available channels at the location. The poll request is piggybacked as a MAC command in a confirmed-up data frame. The expiry generally gets extended after the polling, and updated expiry is sent in the downlink ACK. However, polling is not required for a client generating event-driven traffic as it freshly joins the network whenever it has a data frame to transmit.

## 6 Dynamic Spectrum Access

The dynamic nature of TVWS spectrum can lead to bottlenecks when it is utilized for NB IoT networks. First, the long distance between the gateway and a client brings about separate uplink and downlink having dissimilar quality at their corresponding locations. Second, the IoT clients are power

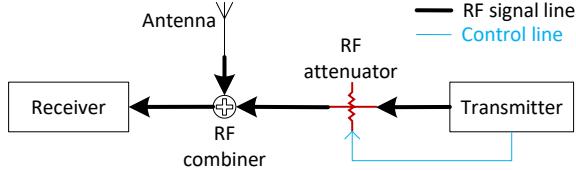


Figure 6: Block diagram of our spectrum sensing module that measures interference from the incumbent users.

constrained. Hence, any extra transmission and carrier sensing activity by the clients for spectrum management adds power overhead. Whisper addresses these challenges by implementing spectrum awareness capability and then incorporating smart spectrum exploitation approach in the protocol.

## 6.1 Spectrum awareness

Uplink communication is inevitably significant in most of the IoT applications as IoT traffic is push-based. The quality of this communication link directly relies on the interference nearby the gateway. Hence, we first try to enable spectrum awareness at the gateway. A part of spectrum awareness, both uplink and downlink, is achieved by using the WSDB. However, it is not enough to accumulate info about the real-time activity of incumbent users – both licensed and unlicensed – in the carrier. To do so, Whisper performs spectrum sensing.

First off, how can the interference level in a NB channel be measured given the Whisper radio uses LoRa modulation and its minimum detectable signal (MDS) is very low ( $-149$  dBm)? The LoRa modulation enables frame reception even when the carrier signal level is below the ambient RF noise floor. Consequently, the conventional approach of measuring RF energy in a NB channel as an indication of interference level is not viable here. Instead, we design a custom off-the-shelf spectrum sensing module to evaluate the level of interference in NB TVWS channels. It can evaluate the impact of interference from licensed and unlicensed cognitive radio users at a very low signal level.

Spectrum sensing module locally generates RF signal with a controlled amplitude to determine MDS in a NB channel. The MDS gives the perception of interference from incumbent users in the NB channel – a lower MDS implies a lower RF interference. Figure 6 depicts the block diagram of the spectrum sensing module. It consists of a transmitter, a controlled attenuator, an RF combiner, and a receiver. We use two Whisper radios as the transmitter and receiver. The attenuator is controlled by the transmitter. We modify transmitter radio firmware to set the value of attenuator. To estimate the MDS in a channel, we tune the transmitter and receiver to the specific channel frequency, and the transmitter starts transmitting test frames. The RF signal of a test frame is attenuated by the attenuator to yield a weak signal similar to a reception from a remote transmitter, which is a commonly practised technique in laboratory emulation [15]. The combiner combines this weak signal with the ambient RF signals (includes inter-

ference from incumbent users) picked up by the antenna. The attenuator value is varied to identify the maximum attenuation that results in 90% successful frame reception. The lowest RSSI of the received frames is the MDS under the interference. Note that the spectrum sensing module only senses the RF channel. The generated RF signal is only used internally, and thus has no effect on data communication and channel occupancy limit. The spectrum sensing module is a part of the gateway as mentioned in Section 4. The channel control unit utilizes it every hour to update the interference level from the incumbent users.

## 6.2 Spectrum exploitation

The interference in the downlink at the location of a client cannot be measured with the spectrum sensing module, given the long distance. Furthermore, carrier sensing at the client’s end would add huge power overhead. Here, Whisper enables the spectrum exploitation approach to mitigate the effect of interference through dynamic channel assignment that does not require any additional frame transmission.

Before jumping into the details, we first introduce the related terminologies and notations. For each slot, we define a channel-tube ( $ChT$ ) that has two channels for a confirmed-up and one channel for an unconfirmed-up data frame. For a slot used in periodic traffic,  $N_{ChT}$  channel-tubes are assigned by adjusting the  $A_c$  for the associated channels during the slot allocation described in Section 5. In every occurrence of the slot, a  $ChT$  is picked in a sequential round-robin manner for communication. In case of event-driven traffic, a  $ChT$  is assigned which is different from last ( $N_{ChT} - 1$ )  $ChTs$ . In this way, when a slot completes hopping across all  $N_{ChT}$   $ChTs$ , we call it a hopping cycle. Note that, a client may have multiple slots (as mentioned in Section 5) and each slot has an individual set of  $N_{ChT}$   $ChTs$ . Furthermore,  $ChTs$  are preferably selected from different TV channels to facilitate robustness against broadband interference. Next, we delineate how the set of  $ChT$  is dynamically updated in three phases.

**1. Monitoring phase:** During this phase, the performance of assigned  $ChTs$  for all the slots of a client is monitored at the gateway, and a flag is raised in case of substandard performance. It is raised based on the overflow of a bucket that is filled up with the tickets for an event of missing frame in a  $ChT$ . The number of tickets for a missing event is equal to the number of previous consecutive occurrences starting from the current event in the  $ChT$ . In the event of successful communication in a  $ChT$ , we remove its tickets in a similar manner. Now, how do we detect an incident of the missing frame? We here leverage the frame count block of the standard LoRaWAN frame header. Finally, we need to determine the optimal bucket size ( $S_{bukt}$ ). It is directly related to  $\sum_{i=1}^{ND_{ChT}} k_i$  where  $ND_{ChT}$  is number of distinct  $ChTs$  and  $k$  is number of times it is used in a cycle.  $S_{bukt}$  depends on the length of hopping cycle too. For example, a client with a smaller cycle

overflows the bucket faster in presence of a short-term interference, whereas it gets slower in case of a longer cycle with long-term interference. Both these scenarios are undesired. Here, we express  $S_{buk_i}$  as  $m \sum_{i=1}^{ND_{ChT}} k_i$  for different bands of cycle. From our emulation and real-world deployments, we found the following optimal values of  $m$  with  $N_{ChT} = 4$  for each slot. Here,  $m$  typically exhibits an exponential downward pattern with respect to the length of cycle.

$$m = \begin{cases} 2.5, & \text{cycle length} < 1hr \\ 1.6, & 1hr \leq \text{cycle length} < 3hr \\ 1.05, & 3hr \leq \text{cycle length} < 6hr \\ 0.7, & 6hr \leq \text{cycle length} < 11hr \\ 0.5, & 11hr \leq \text{cycle length} \end{cases}$$

**2. Decision making phase:** After a flag has been raised, the decision making phase decides which  $ChT$  and associated channels are to be replaced. We note that a  $ChT$  is selected for replacement if its contribution in bucket overflow is at least a certain percentage defined as,  $\frac{k_i}{\sum_{i=1}^{ND_{ChT}} k_i}$ . If no such

$ChT$  is found, older events are removed to accommodate new events in the bucket. Next, we find which channel(s) in the  $ChT$ (s) needs to be replaced based on the knowledge from the monitoring phase. The replacement  $ChT$  is placed exactly at the same sequential position of the replaced one in the current list of  $ChT$ s for a slot.

**3. Execution phase:** For a client sending confirmed-up frames, the execution of  $ChT$  update is initiated immediately after the decision making. Whereas, in case of a client sending unconfirmed-up data frames, it is carried out when the client polls the channel from WSDB since this is the only time when the client sends confirmed-up data frames. The execution is initiated by the gateway by sending an update notice in the downlink frame. The client sends a notification of notice reception in the following uplink data frame. To ensure the robustness against frame loss, the update notice (if a confirmed up frame is received) and notification are sent in the following  $N_{ChT}$  slots. The timing of the update is set accordingly in the notice. Note that the execution process is separate for each slot. Furthermore, no extra frame is exchanged for the execution process since all the associated MAC commands are piggybacked in the data frames.

### 6.3 Fallback

Finally, in a rare event of complete communication loss with the gateway, a client calls for the fallback. Although a client sending confirmed-up data frames detects such an event straight away, a client sending unconfirmed-up data frames detects only at the time of channel polling from WSDB. In such an event, the client does not get a downlink packet in any slot and  $ChT$ . As a fallback alternative, the client rejoins the network. It reports the event in the join request so that gateway can assign a different set of  $ChT$ s.

	Avg.	Max	Min
Throughput (in bps)	0.0626	0.0681	0.0601
FDR (in %)	98.4	100	97.1
Latency (in sec)	2.39	2.43	2.37

Table 2: Performance of Whisper in real-world deployment

## 7 Evaluation

In this section, we focus on evaluating the performance of Whisper through a real-world deployment and simulation. We further make a comparison with the ISM band IoT solutions.

### 7.1 Real-world deployment

We have an ongoing deployment of Whisper on 8500 acre dry-land wheat farm in Eastern Washington, which has been operating for over 2.5 months. Here, Whisper is used by the third-party users in collecting sensor data from 17 different fields for multiple data-driven agriculture applications.

#### 7.1.1 Setup

The deployment includes 20 Whisper client radios that communicate with a single gateway. The client radios are retrofitted inside a weatherproof box, powered by solar, and connected to a sensor interface to collect data from five sensors – temperature, humidity, CO<sub>2</sub>, and soil moisture and temperature (Figure 7b). As configured by third-party users, 11 clients report data of all sensors at different periodicity starting from 30 min to 12hr. Similarly, the remaining clients report all sensor data in an event-driven manner based on CO<sub>2</sub> level. All of the data frames are confirmed-up – 51 bytes uplink, 20 bytes downlink. The modulation parameters are configured to have a coding rate (CR) of 4, 62.5 kHz channel bandwidth (ChBW), and preamble length of 8. We tune the spreading factor (SF) based on the distance of the clients from the gateway as shown in Figure 7c.

The gateway is deployed on the farm, and due to a lack of power, is powered by solar power (Figure 7a). Since there is no Internet connectivity in the middle of the farm, we use an Adaptrum TVWS broadband radio at the gateway to create a wireless link to the nearest source of connectivity (farmer’s home) [2]. It in turn enables the evaluation of Whisper in coexistence with off-the-shelf unlicensed TVWS broadband network. We use a Raspberry Pi 3B as the edge device at the gateway. For WSDB, we use Wave DB Connect by RED Technologies [31]. Finally, both BS radio and clients use omnidirectional antennas with approximately 5 dBi gain where the TX power is set to the max according to Section 3.

#### 7.1.2 Results

Thus far, 4766 sensor data points have been collected via Whisper. We evaluate the performance of Whisper in terms of three metrics: throughput, frame delivery ratio (FDR), and

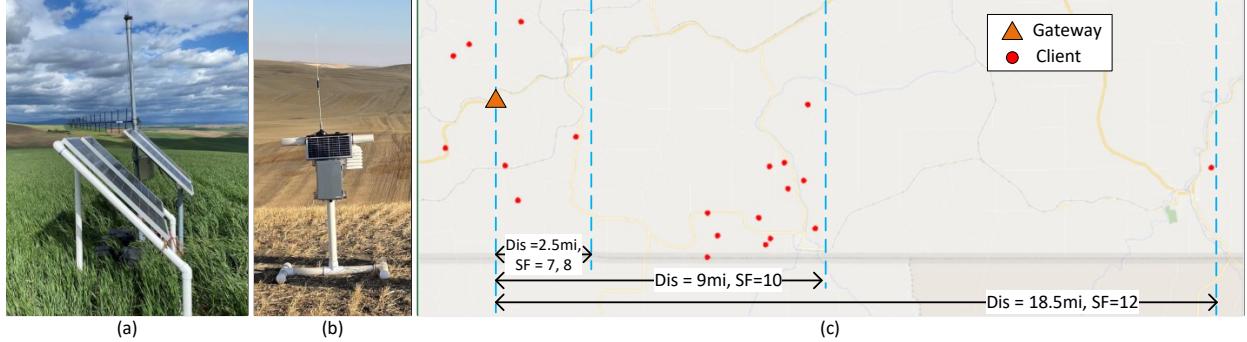


Figure 7: Whisper Deployment. The solar powered IoT Hub shown in (a) includes a broadband TVWS backhaul and Whisper gateway. In (b), Whisper client integrated with sensors and retrofitted inside a weatherproof box. (c) shows the deployment map.

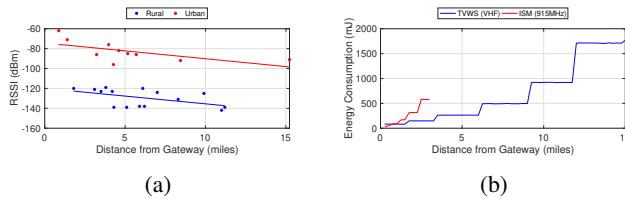


Figure 8: (a) Range of Whisper radio and (b) Comparison of energy consumption between Whisper radio in TVWS VHF band and off-the-shelf LoRa radio in ISM band.

latency. The value of each metric is averaged over a period of 24hr. We do not observe any significant deviation in the average values of these metrics throughout the deployment period. Table 2 summarizes the results. We also observe that Whisper has shown robustness against multiple real-world incidents such as power outages at gateway and clients due to thunderstorms, disrupted Internet connectivity, and WSDB disruption due to maintenance. Furthermore, Whisper shares the spectrum (470 to 488 MHz) with Adaptrum broadband TVWS radio in the deployment as we mentioned above. However, we do not observe any mutual interruption in the communication of Adaptrum radio and Whisper.

### 7.1.3 Range

We perform range experiments to evaluate the sensitivity of the Whisper radio. Range tests are conducted in both urban and rural environments, where both have line-of-site (LOS) and non-line-of-sight (NLOS) connectivity. For the urban settings, we place a BS radio on the rooftop of an industry campus building. Then, we move the client to different locations using a vehicle and record the RSSI. The radios are configured for LoRa modulation having a ChBW of 31.25 kHz, SF of 12, and CR of 4 in the VHF band. Figure 8a shows the RSSI of received frames with respect to distance in miles. We can see that frames are successfully received at 11.3 miles with RSSI of  $-85$  dBm. For the evaluation in rural settings, we use our aforementioned farm deployment. In this scenario, the performance is even better in comparison to the urban settings. We can see that packets are successfully received at

15 miles with a measured RSSI of  $-87$  dBm which is significantly higher than the MDS ( $-149$  dBm) of Whisper radio. These results imply that clients can be deployed as far as 15 miles away from the gateway, and likely even further. For example, in Figure 7c, one of the clients communicates with the gateway from a distance of 18.5 miles. Note that, the range of LoRa in ISM band is found to be 1 - 3 miles in literature and our real-world deployment [4].

### 7.1.4 Energy profile

We next conduct energy profiling of Whisper radio operating in TVWS spectrum and compare it with an off-the-shelf LoRa radio, SX126xMB2xAS [20], operating in the US915 ISM band. The physical setup for the experimentation is similar to the range test. Both radios are configured to send an up-link frame of 51 bytes every hour to the gateway with the same LoRa modulation configurations (CR: 1, ChBW: 62.5 kHz, TX power: 20 dBm). We then vary the distance of the radios from the gateway and tune the SF accordingly for successful communication. In Figure 8b, we show the energy consumption for each frame transmission in TVWS VHF band (174.3 MHz) and ISM band (915 MHz). Note that the energy consumption of the Whisper radio in locking GPS (once in 24hr) is prorated over its transmitted frames. Up to the initial distance of around a mile, the energy consumption in the TVWS spectrum is higher. However, as the distance increases, the time on the air for frame transmitted in the ISM band increases due to the higher spreading factor. Whereas, the lower frequency of the VHF band facilitates longer distance with comparatively lower SF. Consequently, the energy consumption becomes at least 2x higher in the ISM band compared to the TVWS VHF band after a mile of distance.

### 7.1.5 Performance in presence of interference

To evaluate the performance of Whisper under interference in the above-mentioned real-world deployment, we create interference with a separate transmitter. The interference is introduced on both uplink and downlink channels in three different ways - intermittent, continuous, and complete block.

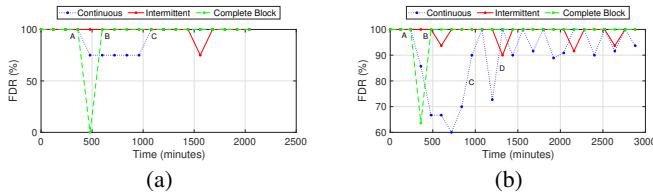


Figure 9: Whisper’s response to different types of interference on (a) downlink and (b) uplink channels in real-world.

In the case of intermittent interference, the interferer hops across the channels of  $N_{ChT} = 4$   $ChT$ s and creates short-term interference for the period equals to the time on air of a frame. Whereas, in continuous interference, the interferer picks one of the channels from the 4  $ChT$ s and creates continuous interference on that channel. Finally, in case of complete block, the interferer is time-synced and aware of the hoping cycle schedule of the target radio. It accordingly creates interference on the active channel for data communication.

We evaluate the impact of these three types of interference on downlink channels. The interferer is placed very close to a client with high transmit power. The interferer is aware of the downlink channels used by the client. We then record the FDR on the downlink channels and the activity of switching channels for the client. In this way, we investigate the impact of interference for the clients, having different periodicity and traffic patterns, in our aforementioned deployment. Since all the clients generating periodic traffic exhibit similar behavior, we only discuss a client having a periodicity of 30 min. Figure 9a shows downlink FDR of the client averaged over a period of 120 min (one hopping cycle). As shown in the figure, the intermittent interference has a very minor impact on the FDR. No  $ChT$  update is initiated in response, whereas, for continuous interference, we observe a significant drop in FDR after the interferer is activated (point A in Figure 9a). As a result, a  $ChT$  update is initiated by the gateway. After the execution of same, the FDR reaches 100% (point C in Figure 9a). Finally, for complete block interference, we see the complete cease in downlink communication for the client. Consequently, the client triggers the fallback mechanism and rejoins the network (point B in Figure 9a). As a new set of  $ChT$ s is assigned after rejoining, the FDR reaches the maximum. We observe a similar impact of intermittent interference and a lesser impact of continuous interference on clients generating event-driven traffic. This is due to the members of the set of 4  $ChT$ s varying with time. Nevertheless, it is difficult to create complete block interference for event-driven traffic.

Next, we evaluate the impact on uplink channels by placing the interferer close to the gateway. We select four uplink channels of the same client for the interferer, however, these channels are used by more than half of the clients in different combinations. To get a precise understanding of the effect of interference, we record uplink frames transmitted by any client on only these four channels. Figure 9b shows corresponding uplink FDR averaged over a period of 120 min. We

observe very minimal effect of the intermittent interference on the network and no  $ChT$  update took place for the same. In case of continuous interference, we see a drop in the FDR after the activation of interferer (point A in Figure 9b). Over the period of two days,  $ChT$ s of two clients are updated followed by increment in FDR (point C and D). Besides, the spectrum sensing unit detects the interference on the channel, and consequently, the channel is not further assigned to any client for sending event-driven traffic in this period by the channel coordinator. Finally, the complete block interference only affects the communication of one client. The client exhibits similar response as described above in case of the downlink.

## 7.2 Simulation

We now focus on evaluating Whisper’s capability in terms of scale. Since deploying a very large-scale network (100s of clients) is challenging, we carry out the simulation. In particular, we evaluate two things: (1) how the larger bandwidth of TVWS spectrum facilitates scaling and (2) how Whisper MAC makes better utilization of this bandwidth compared to mainstream IoT MAC protocols. For this purpose, IoT devices transmitting images is a compelling application scenario (Section 2). We simulate this scenario in three network setups: (S1) Whisper MAC in TVWS spectrum, (S2) Whisper MAC in ISM band, and (S3) pure ALOHA (used by LoRaWAN) in TVWS spectrum. Comparison between S1 and S2 shows the role of bandwidth in our application scenario, whereas S1 and S3 show how Whisper MAC makes better utilization of this larger bandwidth.

### 7.2.1 Setup

We first integrate the LoRa physical layer from FLoRa simulator and the simulation environment of OMNet++ with our protocol [29]. We set up a single-gateway star-topology network where every IoT client device sends a 25 kB image. The gateway has eight BS radios. The number of clients is varied from 5 to 1000 at an increment of 5. The ratio of clients generating periodic traffic ( $N_p$ ) to event-driven traffic ( $N_e$ ) is also adjusted in five levels. We make an even distribution of image generation periodicity starting from 1hr to 24hr among the clients generating periodic traffic. A client with event-driven traffic sends an image once in every 24hr following random distribution. For S1 and S3, the uplink frame size is 255 bytes and downlink ACK frame is 18 bytes. The LoRa modulation parameters are set as following - CR: 1 and ChBW: 62.5 kHz. To simulate the effect of distance and associated data rates, we make an even distribution of six possible SF values (7 – 12) among the clients. Beacon periods are separated by two minutes with a periodicity of  $p_{bcn} = 1$  hr (Appendix B.5). For simulation purposes, since radios are not certified real devices, we use a local proxy of WSDB where each device, including gateway, has 3 TV channels available for transmis-

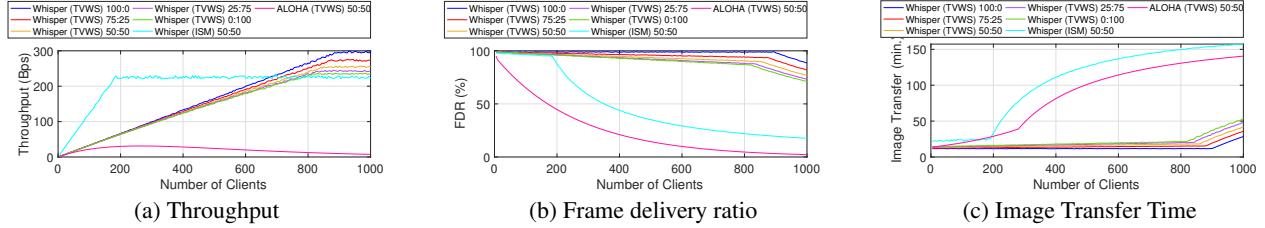


Figure 10: Simulation results showing the role of large TVWS band and how Whisper MAC better utilizes the same in scaling.

sion. For S2, we use the four uplink data rates, DR0-DR3, permitted in the US915 ISM band where corresponding SF ranges from 10 to 7 with the channel bandwidth of 125 kHz. The dwell time restriction limits the uplink payload size (in bytes) for these data rates – DR0:11, DR1:53, DR2:125, and DR3:242. On the downlink side, we use data rates, DR10-DR13, having same SFs as uplink with ChBW of 500 kHz.

## 7.2.2 Results

We analyze the performance in terms of throughput, FDR, and image transfer completion time.

**1. Throughput:** As shown in Figure 10a, we analyze the throughput of the network under a various number of clients and ratios of traffic types. Each point in the graph represents the average throughput of the network in a period of 24hr. Here, we only consider the data frames, not any frames associated with the bootstrapping process. For S1 (Whisper in TVWS), the throughput increases linearly with the number of clients for every ratio of traffic types ( $N_p : N_e$ ) until it reaches the network capacity. As the proportion of event-driven traffic increases in a group of clients, the throughput slightly drops due to the higher collision rate among the randomly triggered events. For S2 (Whisper in ISM band) and S3 (ALOHA in TVWS), we only show the representative results from the traffic ratio of 50 : 50. Although S2 shows similar patterns as S1, there are two distinct factors. First, due to the limited bandwidth in ISM band compared to the TVWS spectrum, S2 reaches the saturation point with  $\sim 5x$  fewer clients than S1. Second, for the same number of clients in the network, S2 exhibits higher throughput as per-channel bandwidth is higher and supported SF is lower in the US915 ISM band compared to the TVWS spectrum. However, the lower value of supported SFs in the ISM band results in a much shorter range. In case of S3, both maximum throughput and number of supported clients are significantly lower than S1. This shows why the existing mainstream IoT MAC protocol is not suited for making better utilization of large TVWS bandwidth.

**2. FDR:** In computing FDR, we consider the frame generated but not successfully transmitted, even due to implied silence by Whisper MAC, as a frame loss. Figure 10b provides similar insights as the throughput. Although the initial throughput is found higher in S2 than S1, FDR is lower in the ISM band due to the larger number of small frame (both uplink and downlink) transmission for sending each image

(Section 2). It, in turn, increases the probability of frame loss.

**3. Image transfer completion time:** We further report the time required to send an image including the re-transmission of frames. The image is dropped after 3hr, if it is not completely transmitted. As shown in Figure 10c, the completion time linearly increases for S1 and S2 until it hits the saturation point, whereas it increases exponentially for S3. Now, although S2 exhibits higher throughput than S1 at the cost of a much shorter range, S2 reports at least 2x higher completion time. Because, in ISM band, a client requires to transmit a larger number of small frames to send an image due to the cap on uplink payload size (Section 2).

## 8 Related Work

Even though unlicensed operations in TVWS spectrum have been extensively studied in literature [5, 6, 21]. Nearly all prior work have focused on broadband scenarios and corresponding FCC regulations. However, there are fundamental differences in NB operation such as power constraint of IoT devices, number of devices in the network, channel occupancy limitation, etc. These make the challenges in unlicensed NB operation non-trivial to solve. There are some prior work on devising suitable signal modulation techniques for NB operation in TVWS spectrum [25, 27, 28]. However, dynamic spectrum access and management in NB operation, compliance with the NB FCC regulations, and corresponding comprehensive MAC protocol design are beyond the scope of these work.

## 9 Conclusion

In this paper, we have presented Whisper, a new solution to enable long-range and wider spectrum communication for IoT by leveraging TVWS spectrum. With Whisper, we get at least 5x longer range compared to LoRa operating in ISM band. Consequently, in our real-world deployment, Whisper has covered 17 fields in a 8500 acre farm with single gateway. Besides, the lower frequency of TVWS spectrum facilitates 2x less energy consumption than ISM band at a range of more than a mile. Note that, for a deployment spanning not more than a mile from the gateway, LoRa in ISM band would be preferable over TVWS spectrum. It also holds for an indoor scenario. Finally, with only 3 white TV channels, Whisper can accommodate 5x more traffic than ISM band.

## References

- [1] Khaled Q Abdelfadeel, Dimitrios Zorbas, Victor Cionca, and Dirk Pesch. *FREE* — Fine-grained scheduling for reliable and energy-efficient data collection in LoRaWAN. *IEEE Internet of Things Journal*, 7(1):669–683, 2019.
- [2] Adaptrum. Adaptrum TVWS broadband radio, 2021. [https://www.adaptrum.com/Content/docs/acrs2\\_datasheet\\_1016.pdf](https://www.adaptrum.com/Content/docs/acrs2_datasheet_1016.pdf).
- [3] Arjan. Airtime calculator for LoRaWAN, 2021. <https://avbentem.github.io/airtime-calculator/ttn/us915-dl/38>.
- [4] A. Augustin, J. Yi, T. Clausen, and W.M. Townsley. A study of LoRa: Long range low power networks for the Internet of Things. *Sensors*, 2016.
- [5] Paramvir Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. White space networking with Wi-Fi like connectivity. *ACM SIGCOMM Computer Communication Review*, 39(4):27–38, 2009.
- [6] Ranveer Chandra, Thomas Moscibroda, Paramvir Bahl, Rohan Murty, George Nychis, and Xiaohui Wang. A campus-wide testbed over the TV White Spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 15(3):2–9, November 2011.
- [7] Tonghao Chen, Derek Eager, and Dwight Makaroff. Efficient image transmission using LoRa technology in agricultural monitoring IoT systems. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 937–944. IEEE, 2019.
- [8] V Chen, S Das, L Zhu, J Malyar, and P McCann. Protocol to access white-space (PAWS) databases. *Internet Engineering Task Force (IETF)*, 2015.
- [9] Federal Communications Commission. FCC rules for unlicensed white space devices. March 2020.
- [10] Federal Communications Commission. Unlicensed white space device operations in the television bands. *FCC ET Docket No. 20-36, Report Order and Further Notice of Proposed Rulemaking*, October 2020.
- [11] LoRa Alliance Technical Committee. LoRaWAN 1.0.3 specification, 2018. <https://lora-alliance.org/sites/default/files/2018-07/lorawan1.0.3.pdf>.
- [12] Semtech Corporation. Semtech SX1262, 2020. <https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1262>.
- [13] Joseph Finnegan, Ronan Farrell, and Stephen Brown. Analysis and enhancement of the LoRaWAN adaptive data rate scheme. *IEEE Internet of Things Journal*, 7(8):7171–7180, 2020.
- [14] Jetmir Haxhibeqiri, Ingrid Moerman, and Jeroen Hoebeke. Low overhead scheduling of LoRa transmissions for improved scalability. *IEEE Internet of Things Journal*, 6(2):3097–3109, 2018.
- [15] Jetmir Haxhibeqiri, Floris Van den Abeele, Ingrid Moerman, and Jeroen Hoebeke. LoRa scalability: A simulation model based on interference measurements. *Sensors*, 17(6):1193, 2017.
- [16] Md Tamzeed Islam, Bashima Islam, and Shahriar Nirjon. Duty-cycle-aware real-time scheduling of wireless links in low power WANs. In *2018 14th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 53–60. IEEE, 2018.
- [17] Akram H Jebril, Aduwati Sali, Alyani Ismail, and Mohd Fadlee A Rasid. Overcoming limitations of LoRa physical layer in image transmission. *Sensors*, 18(10):3257, 2018.
- [18] Mookeun Ji, Juyeon Yoon, Jeongwoo Choo, Minki Jang, and Anthony Smith. LoRa-based visual monitoring scheme for agriculture IoT. In *2019 IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2019.
- [19] Shengyang Li, Usman Raza, and Aftab Khan. How agile is the adaptive data rate mechanism of LoRaWAN? In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 206–212. IEEE, 2018.
- [20] ARM MBED. SX126xMB2xAS, 2021. <https://os.mbed.com/components/SX126xMB2xAS/>.
- [21] Rohan Murty, Ranveer Chandra, Thomas Moscibroda, and Paramvir Bahl. Senseless: A database-driven white spaces network. *IEEE Transactions on Mobile Computing*, 11(2):189–203, 2011.
- [22] The Things Network. Fair use policy explained, 2021. <https://www.thethingsnetwork.org/forum/t/fair-use-policy-explained/1300>.
- [23] The Things Network. Regional parameters, 2021. <https://www.thethingsnetwork.org/docs/lorawan/regional-parameters/>.
- [24] NCSU College of Agriculture and Life Sciences. Low-cost cameras could be sensors to remotely monitor crop stress, 2022. <https://shorturl.at/evxGW>.

- [25] Mahbubur Rahman, Dali Ismail, Venkata P Modekurthy, and Abusayeed Saifullah. LPWAN in the TV White Spaces: A practical implementation and deployment experiences. *arXiv preprint arXiv:2102.00302*, 2021.
- [26] Brecht Reynders, Qing Wang, Pere Tuset-Peiro, Xavier Vilajosana, and Sofie Pollin. Improving reliability and scalability of LoRaWANs through lightweight scheduling. *IEEE Internet of Things Journal*, 5(3):1830–1842, 2018.
- [27] Abusayeed Saifullah, Mahbubur Rahman, Dali Ismail, Chenyang Lu, Ranveer Chandra, and Jie Liu. SNOW: Sensor network over white spaces. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 272–285, 2016.
- [28] Abusayeed Saifullah, Mahbubur Rahman, Dali Ismail, Chenyang Lu, Jie Liu, and Ranveer Chandra. Low-power wide-area network over white spaces. *IEEE/ACM Transactions on Networking*, 26(4):1893–1906, 2018.
- [29] Mariusz Slabicki, Gopika Premankar, and Mario Di Francesco. Adaptive configuration of LoRa networks for dense IoT deployments. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.
- [30] STMicroelectronics. STM32L151RE, 2020. <https://www.st.com/en/microcontrollers-microprocessors/stm32l151re.html>.
- [31] RED Technologies. Wave DB Connect forTVWS, 2021. <https://www.redtechnologies.fr/sas-technology-copy>.
- [32] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT platform for data-driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, 2017.

## A Compliance of Emission

We conduct laboratory measurements to ensure that the emission generated by Whisper radio complies with the FCC leakage regulations. Our evaluation is twofold. First, we look at the changes in the desired to undesired (D/U) signal ratio on the first adjacent channel of select DTV receivers when the source of the undesired signal is changed from a broadband white space device (WSD) to a NB WSD (Whisper radio). Second, we evaluate the changes in the D/U ratio on the first adjacent channel of select DTV receivers in response to the airtime of Whisper radio.

## A.1 Setup

As the bandwidth of a NB TVWS channel is limited to 100 kHz, we select the lowest and highest possible bandwidths supported by the Whisper radio within the limit. Recall that airtime in LoRa modulation is determined by the spreading factor (SF). In this case, we use the lowest (SF7) and highest (SF12) possible spreading factors to change the airtime from low to high, respectively. The duty cycle of the Whisper IoT radio was set at 78% (ON time 780 ms, OFF time 220 ms for each second transmission). Note that the FCC proposed duty cycle is considerably less: 1%.

As shown in Table 3, a set of DTVR - RX1, RX4, RX5, RX10, and RX12 - are identified for this evaluation to diversify over different price ranges, resolutions, dimensions, and form factors. Channel 9 in the high-VHF band (center frequency: 189 MHz) and Channel 16 in the UHF band (center frequency: 485 MHz) are selected to provide the desired DTV signal. The D/U ratio for the Whisper radio is average across four desired signal levels – moderately strong (−43 dBm), moderate (−53 dBm), moderately weak (−65 dBm), and very weak (−80 dBm) at ±3 MHz and ±6 MHz from the edge of the broadcast DTV channel. These are the same desired signal levels used in the broadband WSD laboratory testing. In this way, the two sets of measurements for the D/U ratio on the first adjacent channel can be compared. The video loop used in the D/U measurements for the broadband WSD measurements is used for this test. For each measurement, the undesired signal power is increased until artifacts are observed.

## A.2 Results

Table 4 and 5 summarize the results of D/U ratio evaluation for different DTVR receivers.

For the ATSC 1.0 DTVs, (RX1, RX4, RX5, and RX10), the D/U ratio on the first adjacent channel for the NB TVWS IoT radio indicates the receivers are even more selective (i.e., the value of the D/U ratio more negative) with respect to an undesired NB WSD than an undesired broadband WSD. Note that the D/U ratio at ±6 MHz from the broadcast channel’s edge is usually a few dB better (more selective) than the D/U ratio at ±3 MHz from the broadcast channel’s edge.

For the ATSC 3.0 receiver (RX12) tests at a modulation level of 256QAM, the D/U ratio for NB and broadband WSDs for moderate and weak desired signals are about the same, within error. There is a 5–6 dB difference for moderately strong and very weak desired signal, where the NB WSD is more selective than the broadband WSD for moderately strong signals and the NB WSD is less selective than the broadband WSD for very weak signals. In both instances, the D/U ratio represents very high ATSC 3.0 receiver selectivity. As RX12 is tested only at 256 QAM, it could be a function of the higher threshold at this modulation level.

DTVR ID	Manufacturer	Model	Standard	Profile	Resolution	Display size
RX1	Samsung	UN65NU8000	ATSC1.0	TV	4K	65in
RX4	Samsung	UN32N5300AFXZA	ATSC1.0	TV	1080p	32in
RX5	Insignia	NS-24DF310NA19	ATSC1.0	TV	720p	32in
RX10	Mediasonic	HOMEWORX HW130STB	ATSC1.0	Set-up box	1080p	-
RX12	RedZone	TVXPLORER BUNDLE	ATSC3.0	USB dongle	HD	-

Table 3: Information of DTVR receivers

DTV center frequency = 189MHz (Channel 9)				
	Change in D/U when SF is increased from SF7 to SF12		Change in D/U when b/w is increased from 7.8kHz to 62.5 kHz	
DTV Receiver	7.8kHz	62.5kHz	SF7	SF12
RX1	-0.1	1.4	1.8	0.3
RX4	0.9	1.4	0.8	0.3
RX5	0.1	1.2	1.5	0.4
RX10	0.1	8.1	8.0	0.0
RX12	0.5	2.5	3.1	0.2

Table 4: Change in the D/U Ratio of DTV Receivers for the DTV Transmitter Operating on Channel 9

DTV center frequency = 485MHz (Channel 16)				
	Change in D/U when SF is increased from SF7 to SF12		Change in D/U when b/w is increased from 7.8kHz to 62.5 kHz	
DTV Receiver	7.8kHz	62.5kHz	SF7	SF12
RX1	-0.2	1.7	1.9	0.0
RX4	0.6	1.0	0.6	0.2
RX5	0.1	0.7	1.0	0.2
RX10	0.1	4.7	5.0	0.2
RX12	0.3	1.1	1.3	0.0

Table 5: Change in the D/U Ratio of DTV Receivers for the DTV Transmitter Operating on Channel 16

In general, the NB WSD operating at 62.5 kHz bandwidth with a spreading factor of SF12 displays higher impact on DTVR operation compared to other configurations. For the traditional ATSC 1.0 DTV receivers (RX1, RX4, and RX5), there is negligible change in the D/U ratio for the NB WSD operating at bandwidths of 7.8 kHz (at spreading factors SF7 and SF12) and 62.5 kHz (at spreading factors SF7 and SF12), for measurements in both the UHF and high-VHF bands. For the low-cost digital-to-digital converter (RX10), there is significant change in the D/U ratio with the 62.5 kHz bandwidth and SF12, with a greater change observed in the high-VHF band than the UHF band. For the ATSC 3.0 receiver, there was a 2 – 3 dB change in the D/U ratio observed in the high-VHF frequency band for the 62.5 kHz bandwidth and SF12. The change in the UHF frequency band was minimal: 1.1 – 1.3 dB.

## B MAC Protocol

### B.1 Handling collision in quantum selection

If two assignments –  $A_1$  and  $A_2$  – of a scheduling quantum are colliding, we can make these collision free by ensuring that one of these keeps silence in the colliding hour. For example, if  $A_2$  keeps silence, then we represent collision free

form of it as  $A_2 < h_{s2}, p_2, \text{Silent} [ < h_{s1}, p_1 > ] >$ . Inside  $\text{Silent} []$ , we can keep all the assignments colliding with  $A_2$ . In this way, the new assignment is made collision free with respect to the existing assignments of a quantum. Now, the question arises, how to measure the effect of silencing? Here, we introduce a new metric, frequency of silence ( $\omega_q$ ), representing how frequently the new assignment requires to be silent. We can get the value of  $\omega_q$  from the generic form of solution to Diophantine equation. In the aforementioned example,  $\omega_q$  for  $A_2$  is quotient of  $p_1$  divided by  $\text{gcd}(p_1, p_2)$ . If  $\omega_q = 1$ , then the new assignment requires to be silent in every occurrence of it to avoid collision. Consequently, the quantum is of no use for the new assignment. Based on it, if the quantum has existing assignments (mutually non-colliding by definition) that collide with the new one, we use  $\omega_{q\%} = \frac{\text{number of assignments impelling } \omega_q \text{ frequency of silence}}{\omega}$  for quantum selection. If  $\omega_{q\%} \geq 1$ , then the new assignment requires to be silent in every occurrence of it. Note that the value of  $\omega_{q\%}$  is clamped at 1 for any further calculation.

### B.2 Construction of channel occupancy table

In Figure 4,  $A_{c0}$  is the first assignment for the channel with the occupancy of  $\tau = 6$ . The first table in the figure represents the entry of  $A_{c0}$ . The first entry in the table with no assignment is to facilitate future entries.  $A_{c1}$  is the next assignment with the occupancy of  $\tau = 9$  and does not collide with  $A_{c0}$ . Now, there are  $2^2$  possible combination of these two assignments as shown in the second table which is basically a binary counter of two digits. However,  $A_{c0}$  and  $A_{c1}$  do not collide, and thus the last entry in the second table has  $\tau = 0$ . Consequently, any combination in future having these two assignments will not have a common colliding hour. So we can remove this entry and get the third table.  $A_{c2}$  is the next assignment with the occupancy of  $\tau = 12$  and collides with both  $A_{c0}$  and  $A_{c1}$ . Using the method of binary counting we derive the fourth table from the third one. Finally, if the value of  $\tau$  crosses the limit of 36sec in an entry while adding a new assignment, the assignment is not valid for the channel.

### B.3 Pseudo code of slot allocation algorithm

In Algorithm 1, we show how possible combinations of slots are prepared. We first get the dictionary of collision free scheduling quanta for all base station radios using scheduling quantum selection as mentioned above. The output of

---

**Algorithm 1:** Pseudo code: Generate possible combination of slots for slot allocation algorithm

---

```

Function Get_PossibleSlotCombinations( $p, \sigma,$ 
 $\alpha_{min}$ ) :
    GET  $qDictionary$  FROM Quantum_Selection( $p$ );
    INIT  $possibleSlotCmbnts[]$  TO empty ;
    while  $qDictionary \neq empty$  do
        POP  $entry <A, collisionFreeq[]>$  FROM
             $qDictionary$ ;
        for  $\alpha = \alpha_{max}$  TO  $\alpha_{min}$  do
            if  $collisionFreeq \neq empty$  then
                INIT  $listOfGrpdq[]$  TO (SELECT * FROM
                    (SELECT * FROM  $collisionFreeq$  GROUPBY
                     ( $continuity = \alpha$ ) ORDERBY (AVG( $\rho_q$ )));
                INIT  $cntGrpdq$  TO Count( $listOfGrpdq$ );
                INIT  $numOfSlotRqrd$  TO  $\lceil \frac{\sigma}{\alpha} \rceil$ ;
                INIT  $cntSlotGotChnl$  TO zero;
                INIT  $slotsWithChnl[]$  TO empty;
                for  $i = 0$  TO  $cntGrpdq - 1$  do
                    INIT  $A_c$  TO  $<A, \tau, listOfGrpdq[i]>$ ;
                    GET  $channel$  FROM Channel_Selection ( $A_c$ );
                    if  $channel \neq null$  then
                        ADD  $<listOfGrpdq[i], channel>$  TO
                             $slotsWithChnl[]$ ;
                        INCREASE  $cntSlotGotChnl$  BY 1;
                        if  $cntSlotGotChnl == numOfSlotRqrd$  then
                            ADD  $<slotsWithChnl[], \alpha, Avg(\rho_q), \rho_c, A>$ 
                                TO  $possibleSlotCmbnts[]$ ;
                            break;
            
```

---

scheduling quantum selection is a dictionary having different possible (with/out) modification for silencing) assignments ( $A < h_s, p, Silent[] >$ ) as the key and corresponding list of collision free scheduling quanta as the value. For each entry in the dictionary, we then vary the continuity ( $\alpha$ ) of the scheduling quantum from max to given min. For each continuity value, we group the  $\alpha$  consecutive scheduling quanta. Groups are ordered by the average  $\rho_q$  of member scheduling quanta in each group. Here, to serve the requirement of  $\sigma$  scheduling quanta, we take  $\lceil \frac{\sigma}{\alpha} \rceil$  groups of scheduling quantum. Next, we select downlink channel for each group using the channel selection algorithm. Now, each group is an individual slot, and the combination of  $\lceil \frac{\sigma}{\alpha} \rceil$  slots together satisfy the requirement from the client. In this way, we make a list of the possible combinations of slots. Next, to select a combination from the list, we use the above mentioned weight function.

## B.4 Cases where $p$ is out of boundary

Now, what if the traffic has a periodicity of  $p_{low}$  seconds that is less than 1hr? We convert it as  $\lceil \frac{p_{low}}{3600} \rceil$  requirements hav-

ing a periodicity of 1hr. During the slot allocation, we select slot combinations for each requirement having a time gap of no more than  $p_{low}$  seconds from the selected slot combinations for earlier requirement. If a client generates periodic traffic with  $p > 24hr$ , it freshly joins the network before data transmission. Because as mentioned in Section 5.4, a static client with periodic traffic requires to poll channel status from WSDB once in every 24 hours with the recent GPS reading. The poll request is piggybacked as a MAC command in a confirmed up data frame. As a client generating traffic at a periodicity of more than 24hr, it does not send any data frame within 24hr of the previous one where the poll request can be piggybacked. Besides, sending a frame just for channel polling would be power inefficient. Thus, it it freshly joins the network every time when it has data frame to transmit. It in turn saves the power of polling channel in every 24 hours.

## B.5 Client bootstrapping

### B.5.1 Beacon

In one hour, there are  $N_{bcnprd}$  pre-specified beacon periods, each having  $N_{bcnslot}$  of the same length. Every beacon slot has identical periodicity of  $p_{bcn}$ . For example, in Figure 5, each beacon period has  $N_{bcnslot} = 2$  beacon slots with a periodicity of  $p_{bcn} = 1hr$ . A NB channel for downlink transmission is associated with each beacon slot. These NB beacon channels are distinctly picked from the TV channels registered for the gateway which increases robustness against noisy link and interference. The beacon slot structure along with channels are pre-specified and pre-loaded in the client radio. Here, the question arises what would happen if the WSDB in future ceases transmission on a particular beacon channel for the gateway? Although it is very infrequent, it needs to be addressed to comply with the regulation. In such a case, the gateway goes silent in the beacon slot associated with ceased channel. However, the clients continue to listen on that channel without violating the regulation. Finally,  $N_{bcnprd}$ ,  $N_{bcnslot}$ , and  $p_{bcn}$  are adjusted based on application and expected traffic pattern. For example, for a deployment expecting high event-driven traffic,  $N_{bcnprd}$  is set to a higher value with lower  $p_{bcn}$ .

### B.5.2 Join

On reboot, a client radio first locks the GPS and retrieves the current location and time with a precision of one second. It then hops across the beacon channels according to the previously specified beacon slot structure and listens for the beacon. Since the client is already time synced using GPS and knows beacon schedule, it requires minimal hopping depending on the channel quality. Each beacon frame embeds join slot information as MAC command in an encrypted LoRaWAN multicast frame. To be specific, each beacon frame contains the scheduling info of  $N_{jnslot}$  join slots between the

current and next beacon period. Hence, the info received in a beacon is only valid till the next beacon period. For example, in Figure 5, each beacon contains the scheduling info of following  $N_{jnslot} = 3$  join slots. Here, the gateway books the join slots as an event-driven traffic. The benefit is twofold. First,  $N_{jnslot}$  can be adjusted depending on the expected join requests in a deployment over the time. Second, it utilizes no or low-utilized channels in that hour assigned for periodic assignments. It in turn impels variation in the channels used in the join slots with the time which increases robustness against noisy channels and long-term interference. Note that the uplink and downlink channels are same in the join slot.

Upon receiving the beacon, a client selects one of the join slots from the info embedded in the beacon. Since multiple clients may attempt to send join request at the same time, we need to ensure that clients are selecting join slots in a distributed manner to reduce the collision. Here, the client leverages dynamic quadratic hash function where the unique device id is used as the key. The gateway decides the coefficients of the hash function equation. To do so, it estimates the set of clients likely to send join request using set the difference between the provisioned clients for the deployment and clients already joined the network for periodic traffic. The coefficients chosen based on this estimated set are embedded in the beacon. For example, in Figure 5,  $Client_0$  and  $Client_1$  select two different join slots.

In the join slot, a client transmits the join request along with its location and slot requirement for the data communication. Upon receiving the join request, the gateway sends the client's location to the channel coordinator for the registration on the WSDB (see Figure 2). Once the registration of the client is done, the Whisper MAC manager gets the available channel for it and allocates the slots for data communication. The gateway then sends a join response incorporating the allocated slots and expiry of the associated channels.



# Learning to Communicate Effectively Between Battery-free Devices

Kai Geissdoerfer  
*TU Dresden*

Marco Zimmerling  
*TU Dresden*

## Abstract

Successful wireless communication requires that sender and receiver are operational at the same time. This requirement is difficult to satisfy in battery-free networks, where the energy harvested from ambient sources varies across time and space and is often too weak to continuously power the devices. We present Bonito, the first connection protocol for battery-free systems that enables reliable and efficient bi-directional communication between intermittently powered nodes. We collect and analyze real-world energy-harvesting traces from five diverse scenarios involving solar panels and piezoelectric harvesters, and find that the nodes' charging times approximately follow well-known distributions. Bonito learns a model of these distributions online and adapts the nodes' wake-up times so that sender and receiver are operational at the same time, enabling successful communication. Experiments with battery-free prototype nodes built from off-the-shelf hardware components demonstrate that our design improves the average throughput by 10–80× compared with the state of the art.

## 1 Introduction

The last few years have seen rapid innovation in battery-free systems [40], culminating in a number of real-world applications [1, 12, 27]. These systems pave the way toward a more sustainable Internet of Things (IoT) [7] by enabling small, cheap, and lightweight devices to perform complex tasks (e.g., DNN inference [20]) off ambient energy while using tiny, environmentally friendly capacitors as energy storage [40]. However, to replace today's trillions of battery-powered IoT nodes, battery-free devices must learn to communicate.

**Challenge.** The power that can be harvested from solar, vibrations, or radio signals is typically insufficient to continuously operate a device. A traditional energy-neutral device buffers harvested energy in a rechargeable battery and can *freely control* its average duty cycle to avoid power failures. Instead, a battery-free device cannot avoid power failures, and has *very limited control* over when the power failures begin and end. Fig. 1 illustrates this so-called *intermittent operation*. After executing for a short time, a battery-free device is forced to

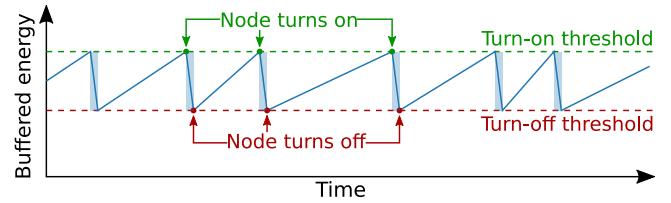
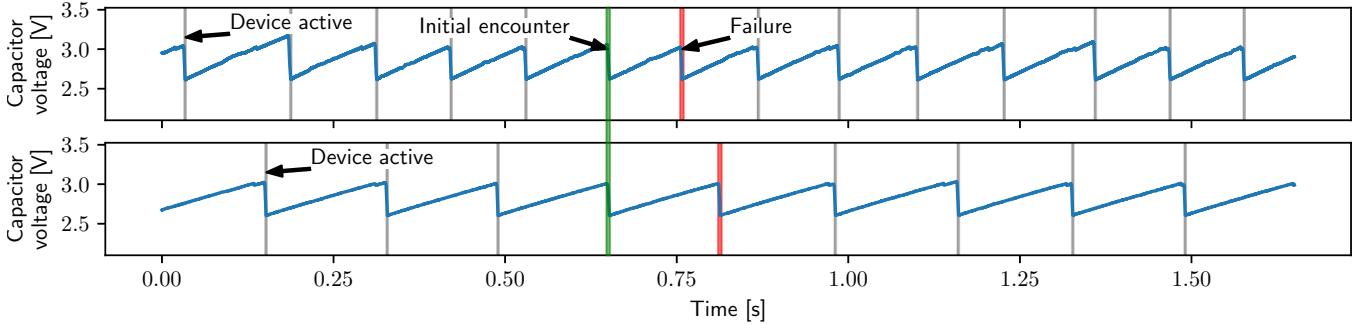


Figure 1: Because ambient power is often weak, a battery-free node must buffer energy before it can wake up and operate for a short time period. This is known as intermittent operation.

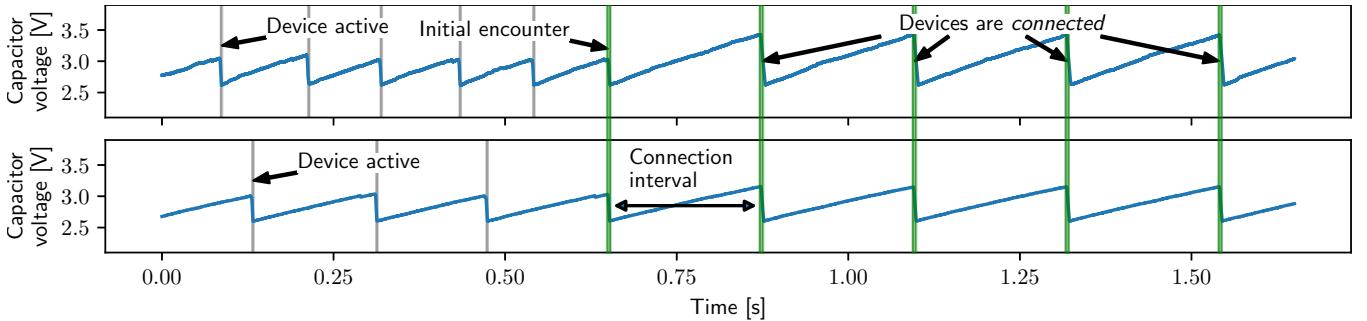
become inactive and wait for a long, fluctuating time until its capacitor is sufficiently charged again. For example, when harvesting energy from indoor light, our prototype battery-free nodes need to stay off and recharge, on average, for hundreds of milliseconds before they can operate for at most 1 ms.

Many techniques have been developed to deal with intermittency on a *single* battery-free device [6, 11, 34], but how to communicate *between* intermittently powered devices is one of the most pressing problems yet to be solved [22, 28, 48]. This is due to the fact that device-to-device communication is a fundamental building block for a variety of network and system services, including optimal clock synchronization [26], ranging and localization [9, 21], sensor calibration [41], distribution and coordination of sensing and computing tasks [32], collaborative learning [47], and efficient and reliable wireless networking [25]. Realizing these services across battery-free devices has the potential to enable novel and more sustainable IoT and sensor network applications, from automatic contact tracing to planetary-scale environmental monitoring.

To be able to communicate, sender and receiver must be active and have enough energy for at least one complete packet transmission *at the same time*. However, since the nodes' activity phases are generally interleaved and short compared to their charging times, as visible from the real-world trace in Fig. 2a, it often takes thousands of wake-ups until two nodes encounter each other and communication becomes possible [19]. Moreover, after an encounter, the nodes quickly get out of sync if they become active immediately after a



(a) Because of their short and interleaved activity phases, battery-free devices often need a long time with hundreds of wake-ups until they encounter each other. Even after an initial encounter, the devices quickly get out of sync, rendering communication inefficient and unreliable.



(b) With Bonito, devices learn and exchange statistical models of their charging times and agree on a connection interval that ensures that both devices have sufficient energy at the same time. Maintaining a connection over multiple encounters enables efficient and timely communication.

Figure 2: The challenge of efficient battery-free device-to-device communication in (a) and our proposed protocol in (b).

recharge, as stipulated by the state of the art [8, 33] and apparent in Fig. 2a. This is because ambient energy varies across time and space [3], which leads to fluctuating and different charging times between the nodes.

Besides establishing a first encounter [19], active radio communication has been considered too demanding for battery-free devices [36]. Conversely, work on backscatter communication has focused on physical-layer issues, such as improving range and throughput, purposely considering high-energy environments, batteries, or cables to continuously power the devices in the experiments to avoid intermittency [29, 35, 38, 49]. However, when running off ambient energy, duty cycling of the backscatter transceivers becomes necessary [14, 29, 43]—and, without a battery, the intermittency problem occurs.

**Contribution.** This paper presents Bonito, the first connection protocol for battery-free wireless networks. Bonito provides reliable and efficient bi-directional communication despite the time-varying intermittency of battery-free devices.

The real-world trace in Fig. 2b illustrates the high-level protocol operation. Unlike the state of the art, Bonito enables two battery-free nodes, after an initial encounter, to maintain a *connection* across multiple consecutive encounters. To this end, Bonito continually adapts the *connection interval*, which is the time between the end of an encounter and the beginning of the next encounter. A shorter connection interval provides

more communication opportunities in the long run. However, a connection interval that is shorter than any of the nodes’ charging times breaks the connection and requires the nodes to wait for a long time until they encounter each other again. Thus, the challenge is to keep the connection interval as short as possible without losing the connection, which is difficult in the face of time-varying charging times.

One of our key insights is that, depending on the scenario and energy-harvesting modality, the charging time of a battery-free node approximately follows well-known probability distributions. We leverage this insight in Bonito by letting each node *continuously learn and track* the parameters of a model that approximates the distribution of locally observed charging times against non-stationary effects (e.g., changes in mean or variance). Then, to maintain an efficient and reliable connection, the nodes exchange at every encounter their current model parameters and jointly adapt the connection interval.

We implement Bonito on a custom-designed ultra low-power battery-free node. Our prototype is built from off-the-shelf components, including an ARM Cortex-M4 microcontroller that features a 2.4 GHz Bluetooth Low Energy (BLE) radio. The node harvests energy from a solar panel or a piezoelectric harvester, using a 47  $\mu$ F capacitor as energy storage.

To evaluate Bonito through testbed experiments and fairly compare it against two baselines, we use up to 6 Shepherd ob-

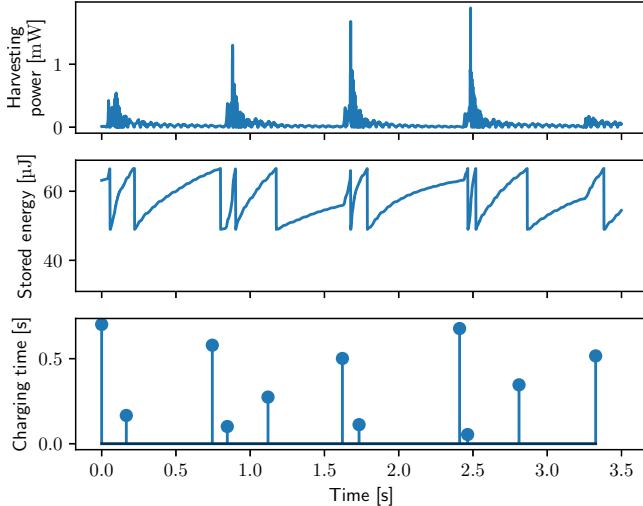


Figure 3: The top plot shows an example trace of real kinetic harvesting power during jogging (see picture in Fig. 4a). The middle and bottom plots show the corresponding energy stored in the capacitor and the resulting charging times of a simulated battery-free device.

servers [18] to record and replay real-world energy-harvesting traces from 5 diverse scenarios. Our results show, for example, that Bonito maintains connections for hundreds of consecutive encounters, and that it outperforms the state of the art by 10–80× in terms of throughput. We also conduct a case study that demonstrates the utility of Bonito for accurate and timely occupancy monitoring in homes and commercial buildings.

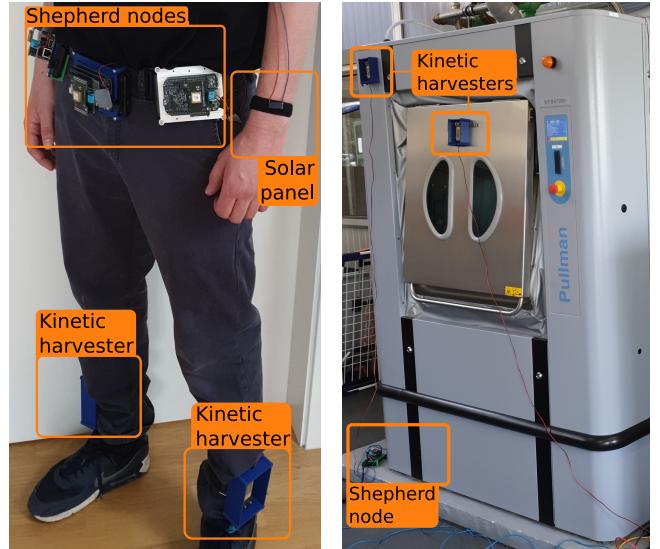
Overall, this paper contributes the following:

- We collect 32 h of energy-harvesting traces from 5 different scenarios. Our analysis of these traces provides new insights into spatio-temporal intermittency patterns.
- We design the Bonito protocol. Bonito enables, for the first time, reliable and efficient communication between intermittently powered battery-free devices.
- We demonstrate an efficient implementation of Bonito on a prototype node with a  $3.1 \text{ mm}^3$  ceramic capacitor.
- Results from testbed experiments and an occupancy monitoring case study provide evidence that Bonito performs well under a diverse range of real-world conditions.

## 2 Motivation

While previous work on intermittency has focused on individual battery-free devices [6, 11, 33] or discovery of neighboring devices [19], reliable and efficient device-to-device communication is still an open challenge. By *device-to-device communication* we mean the regular exchange of application data between two battery-free devices after they have successfully discovered each other through a first encounter [19].

To motivate the need for our work, we consider the scenario of battery-free wearables. Fig. 3 shows real-world data from a piezoelectric energy harvester that is attached to the ankle of a person (see Fig. 4a). The upper plot shows the harvest-



(a) Runner with full measurement setup for the *jogging* dataset.

(b) Washing machine with partial setup for the *washer* dataset.

Figure 4: Pictures from two of the five scenarios in which we use synchronized Shepherd nodes [18] to record energy-harvesting traces.

ing power while the person is jogging, recorded by a Shepherd node. Shepherd is a measurement tool that records time-synchronized voltage and current traces from one or more energy-harvesting nodes with high rate and resolution [18]. The power spikes correspond to when the foot strikes the ground, with significantly lower harvesting power during the rest of the stride cycle. Based on trace-driven simulations, the middle plot shows the corresponding amount of harvested energy stored in an ideal  $17 \mu\text{F}$  capacitor powering a battery-free device that turns on when the capacitor voltage exceeds 3 V and turns off when the capacitor voltage falls below 2 V. We see that when the device powers up, the stored energy is quickly consumed, forcing it to turn off already after about 1 ms. While powered off the harvesting power exceeds the standby power, so energy is accumulated and the capacitor voltage rises again. Compared to the short activity phases, the time needed to charge the capacitor, shown in the bottom plot of Fig. 3, is much longer and varies significantly over time.

The variability of a node’s charging time is a function of its location and the associated energy environment, that is, how much power the harvester delivers at any given time. Thus, two battery-free devices, even when they are physically close to each other, have a different energy environment and therefore experience different charging times.

As an example, Fig. 5 plots the charging times of two devices during jogging over one hour. One device is powered by a piezoelectric harvester attached to the left ankle of a person, while the other device is powered by the same type of harvester attached to the right ankle of the person (see Fig. 4a). Each point in Fig. 5 indicates the charging times of both de-

Dataset	Energy Source	Harvester Part Number	Duration	#Devices	#Links	#Wake-ups	Model
Jogging	Human motion	MIDE S128-J1FR-1808YB	1 h	3	10	13252	Exponential
	Outdoor solar	IXYS KXOB25-05X3F		2		119127	Normal
Stairs	Outdoor solar	IXYS KXOB25-05X3F	1 h	6	15	359002	Normal
Office	Indoor light	IXYS SM141K06L	1 h	5	10	98324	Gaussian mixture
Cars	Car vibrations	MIDE S128-J1FR-1808YB	2 h	6	15	8517	Exponential
Washer	Machine vibrations	MIDE S128-J1FR-1808YB	45 min	5	10	22224	Normal

Table 1: Overview of energy-harvesting datasets we record in a variety of scenarios.

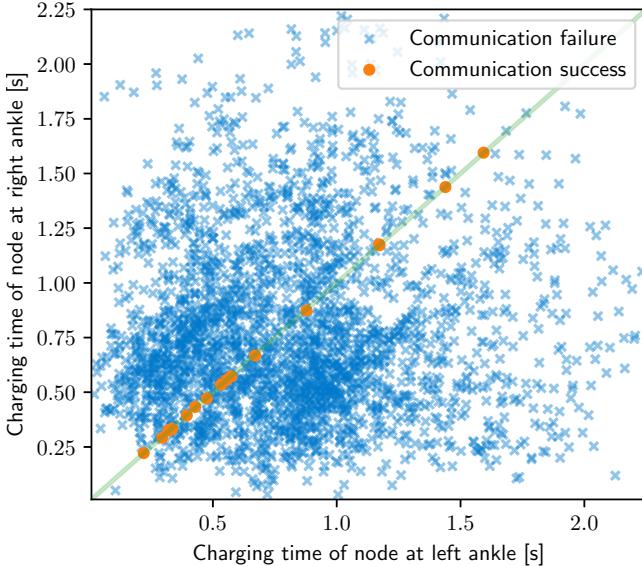


Figure 5: Charging times of two battery-free devices powered by kinetic harvesters attached to a jogger’s ankles (see Fig. 4a). Using the greedy approach, the devices communicate successfully only in 0.04 % of the cases in which the charging times are almost identical.

vices when they begin to charge their  $17\ \mu\text{F}$  capacitors at the same time from the same initial charge. We observe that in many instances the two nodes have vastly different charging times. This means that if nodes become active as soon as they reach the turn-on threshold, which is the state-of-the-art approach, called *greedy* and illustrated in Fig. 2a, the nodes often wake up with an offset that prevents communication, despite a successful encounter at the previous wake-up. Indeed, the success rate for the two nodes in Fig. 5 is less than 0.04 %. This leads to poor communication reliability and efficiency as the nodes more often than not fail to exchange their data.

To assess the generality of these observations, we record distributed energy-harvesting traces in diverse scenarios using multiple Shepherd nodes [18]. Table 1 lists the main characteristics of the five datasets we collected:

- The full *jogging* dataset comprises traces from two participants, each equipped with two piezoelectric harvester at the ankles and a solar panel at the left wrist (see Fig. 4a). The two participants run together for an hour in a public

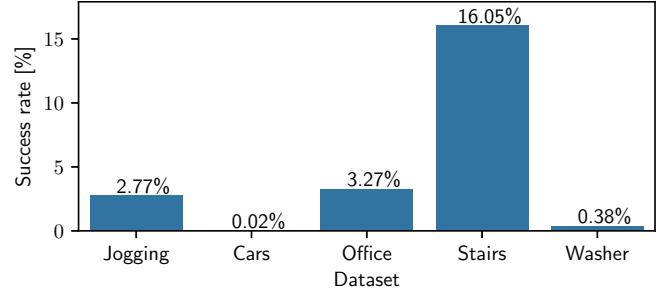


Figure 6: Success rate of greedy approach in trace-driven simulations, averaged across all pairs of devices (i.e., links) in a given dataset.

park, including short walking and standing breaks.

- For the *stairs* dataset, we recorded traces from six solar panels that are embedded into the surface of an outdoor stair in front of a lecture hall. Over the course of one hour, numerous students pass the stairs, leading to temporary shadowing effects on some or all of the solar panels.
- The *office* dataset comprises traces from five solar panels mounted on the doorframe and walls of an office with fluorescent lights. During the one-hour recording, people enter and leave the office and operate the lights.
- The *cars* dataset contains traces from two cars. Each car is equipped with three piezoelectric harvesters mounted on the windshield, the dashboard, and in the trunk. The cars drive for two hours in convoy over a variety of roads.
- The *washer* dataset includes five traces from piezoelectric harvesters mounted on a WPB4700H industrial washing machine, as shown in Fig. 4b, while the machine runs a washing program with maximum load for 45 min.

Fig. 6 plots for each dataset the average success rate across all pairs of devices (i.e., communication links) in the scenario. Even in the most favorable scenario, *stairs*, where the solar panels receive a fairly constant and similar energy input from natural sunlight, we find that the greedy approach succeeds in only 16 % of the cases. In all other scenarios, the success rate ranges below 3.5 %. Our experiments on real battery-free nodes in Sec. 5 confirm these trace-driven simulation results.

### 3 The Bonito Protocol

This section describes the Bonito protocol. The Bonito protocol enables two battery-free devices to stay connected after a first encounter, which can happen either coincidentally or with the support of a neighbor discovery protocol [19].

#### 3.1 Overview

Bonito aims to make nodes repeatedly encounter each other so they can exchange application data reliably and efficiently, as shown in Fig. 2b. To ensure that nodes wake-up with a time offset small enough for a successful encounter, they agree at every encounter on a new *connection interval*  $T_C$ . This is the time between the end of the current encounter and the beginning of the next (i.e., planned) encounter.

**Main idea and approach.** For two nodes  $i$  and  $j$  with known charging times  $c_i$  and  $c_j$ , the shortest possible connection interval  $T_C^*$  is simply the maximum of their charging times

$$T_C^* = \max(c_i, c_j) \quad (1)$$

If a shorter connection interval  $T_C < T_C^*$  is used, then one node does not reach the required energy level to become active by  $T_C$ . Thus, the encounter fails, preventing the nodes from agreeing on the next connection interval—the connection is *lost*. A lost connection entails that the nodes often need to wait for a long time until they encounter each other again to resume communication. However, choosing a longer connection interval  $T_C > T_C^*$  to mitigate the risk of a lost connection adds unnecessary delay as nodes, after having reached the required energy level, are forced to wait before they wake up at  $T_C$ .

The key challenge is to determine the connection interval  $T_C$  such that both nodes have enough energy while introducing only minimal delay. This is difficult as the charging times  $c_i$  and  $c_j$  are unknown and time-varying, as discussed in Sec. 2.

Using a probabilistic approach, we address this problem as follows. Let  $p$  be the probability that nodes  $i$  and  $j$  have sufficient energy to become active after a connection interval  $T_C$ . This corresponds to the probability that the nodes' charging times,  $c_i$  and  $c_j$ , are shorter than the connection interval  $T_C$ . Modeling  $c_i$  and  $c_j$  as random variables with a strictly monotonically increasing joint cumulative distribution function (cdf)  $F_{i,j}$ , this translates into

$$p = F_{i,j}(c_i = T_C, c_j = T_C) \quad (2)$$

Solving for  $T_C$  yields the minimum connection interval that guarantees, with a user-defined probability  $p$ , a successful encounter of the two nodes at their next wake-up

$$T_C = F_{i,j}^{-1}(p) \quad (3)$$

where  $F_{i,j}^{-1}$  is the inverse joint cdf of  $c_i$  and  $c_j$ .

**Base protocol.** In practice, the joint cdf  $F_{i,j}$  is rarely known a priori. Moreover,  $F_{i,j}$  can only be estimated online by the nodes based on full knowledge of each other's charging times.

Unfortunately, this requires frequent communication between battery-free nodes—precisely what Bonito intends to enable.

To circumvent this chicken-and-egg problem, we assume that the charging times,  $c_i$  and  $c_j$ , are statistically independent. In this case, the joint cdf  $F_{i,j}$  is the product of the marginal cdfs  $F_i$  and  $F_j$ . The marginal cdfs can be estimated locally by each node from observations of their own charging times.

Based on these insights, we propose the following main steps of the Bonito protocol:

1. Each node  $i$  continuously estimates the marginal cdf  $F_i$  of its charging time based on local measurements.
2. When two nodes  $i$  and  $j$  encounter each other, they exchange their current estimates of  $F_i$  and  $F_j$ .
3. Using the same inputs (i.e., the marginal cdfs  $F_i$  and  $F_j$  and the user-defined probability  $p$ ), both nodes compute the same new connection interval  $T_C$  according to (3).
4. Both nodes become active and communicate after the new connection interval  $T_C$ , and continue with step 2.

In this way, Bonito adapts the connection interval to changes in the energy environment, effectively enabling battery-free nodes to stay connected across several hundreds of subsequent encounters, as demonstrated by our experiments in Sec. 5.

To achieve this performance, we first need to answer the following key questions in our design of Bonito:

- How to compactly represent and exchange the marginal cdfs  $F_i$  and  $F_j$  in the face of limited energy (Sec. 3.2)?
- How to learn and track online an accurate estimate of  $F_i$  against a changing energy environment? (Sec. 3.3)
- How to efficiently compute the inverse joint cdf  $F_{i,j}^{-1}(p)$  to obtain the connection interval  $T_C$ ? (Sec. 3.4)

#### 3.2 Modeling Charging Time Distributions

Because of the small energy storage, battery-free devices can only exchange a limited amount of data during an encounter. Thus, the marginal cdfs  $F_i$  and  $F_j$  must be represented in a compact form in order to be able to exchange them.

Unlike the common belief that the duration of a recharge is completely random [11, 31], we make the empirical observation that, in the scenarios we considered, the nodes' charging times can be faithfully modeled by well-known distributions. The rightmost column of Table 1 lists the models we use for each dataset. To illustrate, Fig. 7 plots representative charging time distributions and the corresponding models for the stairs, cars, and office datasets. Non-stationary effects like a time-varying mean are removed in the plots as these are effectively handled by our online learning approach detailed in Sec. 3.3.

We observe in Fig. 7a that when harvesting energy from outdoor solar with a constant harvesting voltage, the charging time can be modeled by a normally distributed random variable. The intuition is that temporary environmental effects, such as shadowing and change in incidence angle, let the charging time vary around a certain value. Fig. 7b shows that an exponential distribution is often a good fit when harvesting kinetic energy. This can be explained by the decaying

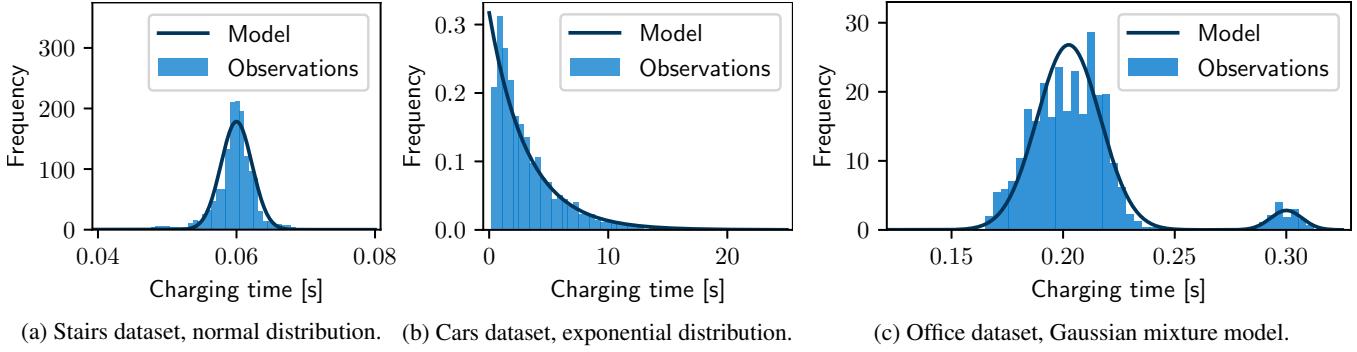


Figure 7: Charging time distributions of individual nodes. The nodes’ charging times can be modeled by well-known distributions.

response of a piezoelectric harvester to the distinct impulses of a car during driving (e.g., acceleration, breaking, bumps) or a person during jogging (see Fig. 3). In the washer scenario, instead, we find that the continuous shaking of the industrial washing machine over long periods induces approximately normally distributed charging times. Looking at Fig. 7c, we see that in the office scenario the charging times are mostly distributed around a certain value. However, the maximum power point tracking (MPPT) of the DC-DC converter used in this scenario, which periodically disconnects the charger for a short time, leads to a second peak. We approximate this distribution with a Gaussian mixture model (GMM).

These observations motivate us to model the marginal cdf  $F_i$  of a node’s charging time in the scenarios we considered through the parameters of a normal distribution (2 parameters), an exponential distribution (1 parameter), or a GMM (6 parameters for two Gaussians and two weights). The last column of Table 1 lists the corresponding model for each of the datasets. The jogging dataset contains traces from different types of harvesters: We use an exponential distribution to model the charging times of kinetic harvesting nodes and a normal distribution for the solar harvesting nodes. During an encounter, a node only needs to share the type of model and the current estimates of the model parameters.

### 3.3 Learning Distribution Parameters Online

We now turn to the problem of estimating the parameters of a given charging time distribution based on local observations. Given a sample of  $n$  independent and identically distributed observations, the log-likelihood  $\mathcal{L}(\theta | x)$  and the corresponding maximum likelihood estimator  $\hat{\theta}$  are given by

$$\mathcal{L}(\theta | x) = \ln \left( \prod_{i=1}^n f_\theta(x_i) \right) = \sum_{i=1}^n \ln f_\theta(x_i) \quad (4)$$

$$\hat{\theta} = \arg \max_{\theta} \mathcal{L}(\theta | x) \quad (5)$$

where  $f_\theta(x_i)$  is the conditional probability to observe  $x_i$  if the underlying distribution is parameterized with  $\theta$ .

Unfortunately, vanilla maximum likelihood estimation is not viable in our setting. First, the observations of the charg-

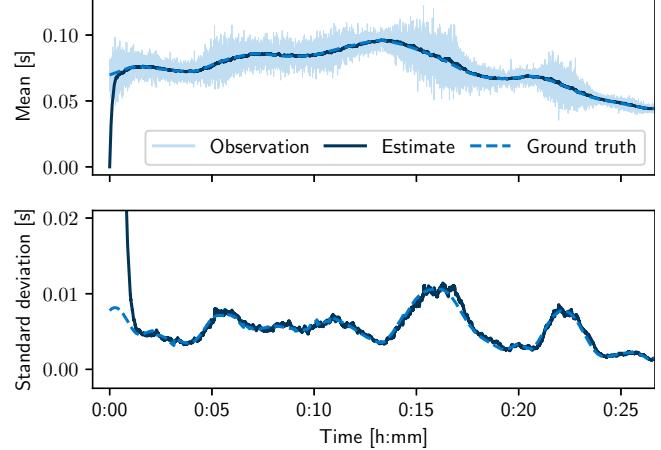


Figure 8: Varying mean and standard deviation over a moving window of one of the traces from the stairs dataset reveal non-stationarity. Using SGD, the changing distribution parameters are tracked online.

ing time become available only one by one at runtime, yet the nodes do not have enough memory and energy to recompute the estimator with every new observation. Further, the charging time distributions are non-stationary. For instance, the dashed lines in Fig. 8 reveal trends in the mean and standard deviation of a node’s charging time from the stairs dataset. Thus, an approach is needed that dynamically adjusts the parameter estimates to changing energy harvesting conditions.

To address these problems, Bonito learns the distribution parameters online using stochastic gradient descent (SGD), which has become a popular method for training a wide range of machine learning models [5]. Compared to a sliding window based approach, SGD is less computationally demanding as the parameter update is only computed for the current observation rather than for a set of past observations that also have to be kept in memory. If the gradient of  $\mathcal{L}(\theta | x)$  is known, one solution to (5) is to iteratively adjust  $\hat{\theta}$  along the gradient, known as gradient descent

$$\nabla \mathcal{L}(\theta | x) = \nabla \sum_{i=1}^n \ln f_\theta(x_i) \quad (6)$$

$$\hat{\theta} = \hat{\theta} + \eta \cdot \nabla \mathcal{L}(\hat{\theta} | x) \quad (7)$$

By pulling the  $\nabla$  operator in (6) into the sum, the update step in (7) can be split into a series of updates for every individual observation  $x_i$ . This yields the update equation of SGD

$$\hat{\theta}_i = \hat{\theta}_{i-1} + \eta \cdot \nabla \mathcal{L}(\hat{\theta} | x_i) \quad (8)$$

Sec. A derives the gradient equations required to solve (8) for the normal, exponential, and Gaussian mixture models. By keeping the learning rate  $\eta$  constant, Bonito implicitly reduces the weight of old observations relative to more recent observations. This way, devices dynamically learn changing properties of the charging time distribution locally, without information exchange with other devices.

**Example.** Fig. 8 illustrates how Bonito learns and tracks mean and standard deviation of a non-stationary normal distribution. To obtain ground truth, we sample charging times (i.e., observations) from a known normal distribution, whose mean and variance change dynamically over time. We extract these changes from one of the traces in the stairs dataset using a 2 min moving average filter. We can see in Fig. 8 that the parameter estimates of Bonito converge from their initial values (zero mean and unit standard deviation) to the true ground truth parameters within less than a minute. Then the estimates closely follow the changes of the underlying distribution.

### 3.4 Computing Inverse Joint CDF Efficiently

Having shared the type of model and the current estimates of the model parameters during an encounter, Bonito needs to compute the new connection interval  $T_C$  from the inverse joint cdf  $F_{i,j}^{-1}$  for a user-defined probability  $p$ . This is difficult since there exists no closed-form solution for most bivariate distributions, let alone for joint cdfs of different distribution families (e.g., when a solar and a kinetic energy harvesting node in the jogging scenario want to communicate). Instead, we have to solve (3) numerically, while taking into account the energy and compute constraints of battery-free devices.

We are interested in the connection interval  $T_C$  where the joint cdf is equal to the user-defined target probability, that is,  $F_{i,j}(T_C) = p$ . This yields the following objective function

$$f(T_C) = F_{i,j}(T_C) - p = F_i(T_C) \cdot F_j(T_C) - p = 0 \quad (9)$$

Note that  $f(T_C)$  has a single root—the sought solution—as  $F_{i,j}$  is strictly monotonically increasing. Bonito solves this problem using the well-known bisection method, which iteratively finds the root of any continuous function that has its root inside a bracket (i.e., search interval). Indeed, we can derive such a bracket based on the inverse cdfs of our marginal distributions, which either have a closed form solution (exponential and normal) or are easy to approximate (GMM).

To derive a lower bracket, we first note that  $F(x) < 1$  for any cdf  $F$ . It follows that  $F_{i,j}(x=z, y=z) = F_i(x=z) \cdot F_j(y=z) < \min(F_i(x=z), F_j(y=z))$  and therefore the lower bracket

$$F_{i,j}^{-1}(p) > \max(F_i^{-1}(p), F_j^{-1}(p)) \quad (10)$$

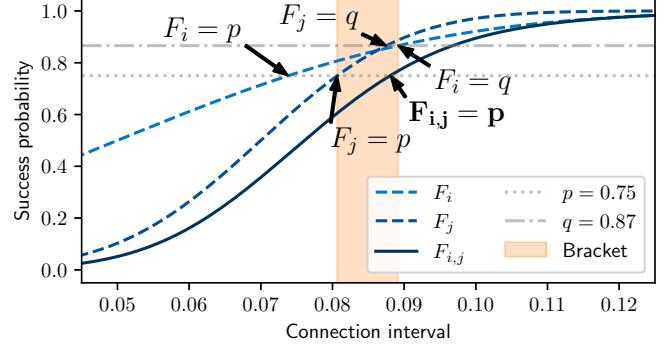


Figure 9: Bracketing the inverse joint cdf based on the inverse cdf of the marginal distributions enables efficient computation of the connection interval on resource-constrained battery-free devices.

To derive an upper bracket, we introduce  $q = \sqrt{p}$  and  $c = \max(F_i^{-1}(q), F_j^{-1}(q))$ . Let  $F_m$  be the marginal cdf (i.e., either  $F_i$  or  $F_j$ ) such that  $F_m^{-1}(q) = c$ , that is, the marginal cdf that reaches  $q$  later. Let  $F_n$  be the other marginal cdf that reaches  $q$  sooner. From  $F_n(c) \geq F_m(c)$  follows  $F_{i,j}(c) = F_m(c) \cdot F_n(c) \geq F_m(c) \cdot F_m(c) = q^2 = p$ . Finally, because  $F_{i,j}(c)$  is monotonically increasing, we obtain the upper bracket

$$F_{i,j}^{-1}(p) \leq \max(F_i^{-1}(q), F_j^{-1}(q)) \quad (11)$$

**Example.** Fig. 9 shows an example, where (10) and (11) are used to determine an initial bracket for  $F_{i,j}^{-1}(p = 0.75)$ . The resulting bracket  $[0.61, 0.77]$  is already relatively tight, and therefore we find the solution  $F_{i,j}^{-1}(p = 0.75) = 0.88$  with a tolerance of 0.01 after only three bisection steps.

### 3.5 Impact of Target Probability

The target probability  $p$  is a key parameter of the Bonito protocol that must be set by the user. It specifies the probability that both devices have accumulated enough energy in their capacitors to become active after a connection interval  $T_C$ . A high target probability requires a long connection interval  $T_C$ , increasing communication delay and lowering throughput.

To illustrate how the choice of  $p$  impacts communication reliability and efficiency, we run trace-driven simulations as detailed in Sec. 2 on the traces from the datasets in Table 1. We use two metrics to quantify the performance of Bonito: As a proxy for communication reliability, we define the *success rate* as the ratio of successful encounters with Bonito to the total number of wake-ups. As a proxy for communication efficiency, we consider the *relative delay* as the median of all successful connection intervals with Bonito divided by the median of the optimal clairvoyant solutions according to (1).

Fig. 10 plots for each dataset success rate and relative delay averaged across all links. We can observe the following:

- A higher target probability  $p$  leads to a higher success rate, which demonstrates the plausibility of our approach.

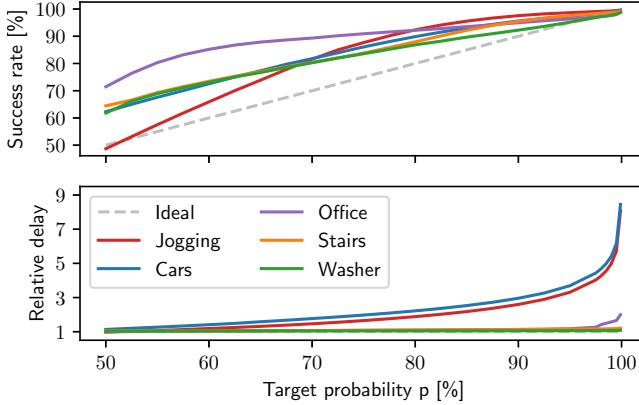


Figure 10: Trace-driven simulations reveal that the rate of successfully arranged encounters matches the user-defined target probability. The price to pay in terms of latency depends on the model of the underlying charging time distribution.

In most cases, the success rate is even slightly higher than requested, presumably due to small model errors.

- Since connection losses are costly, a higher target probability is preferable in practice. Fig. 10 shows that the price to pay in terms of a higher relative delay depends on the scenario. For the cars and jogging datasets, where most or all links include at least one node with approximately exponentially distributed charging times, the relative delay increases exponentially with  $p$ , due to the heavy tail of the distribution. For GMM (office), the increase is moderate, whereas it is hardly noticeable for the normal distribution (washer and stairs).

## 4 Implementation

In this section, we describe the hardware and software components of our prototype implementation.

### 4.1 Hardware

We design a ultra low-power battery-free node based on the popular Nordic Semiconductor nRF52805 microcontroller (MCU). This particular MCU features a 2.4 GHz BLE radio and a state-of-the-art 32-bit 64 MHz ARM Cortex-M4, which is powerful enough to complete also more demanding computations in a short time, benefitting overall system efficiency. To enable low-power timekeeping between wake-ups, the MCU is equipped with a 32 kHz crystal with  $\pm 20$  ppm frequency tolerance. A TI BQ25504 DC-DC step-up converter charges a 2 mm  $\times$  1.25 mm  $\times$  1.25 mm 47  $\mu$ F multilayer ceramic capacitor (MLCC) from a connected solar panel or a piezoelectric energy harvester. Once the capacitor voltage reaches a hardware-programmable threshold of 3.3 V, the BQ25504 sets one of its pins high. This pin is wired to a TI TS5A23166 analog switch that connects the MCU to the capacitor-buffered supply voltage.

Due to its DC bias characteristics, the capacitor has an ef-

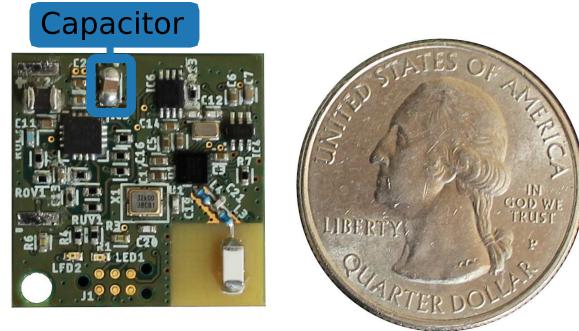


Figure 11: Prototype battery-free node based on the nRF52805 MCU. A sustainable 3.1 mm<sup>3</sup> ceramic capacitor is used as energy storage.

Bonito data					
Preamble	Address	Model type	Model parameters	Application data	CRC
2B	3B	1B	4..24B	$\geq 0B$	1B

Figure 12: Packet format. Using Bonito, nodes exchange between 5 B and 25 B carrying model type and parameters during an encounter.

fective capacitance of only 17  $\mu$ F at 3.3 V. This allows for a maximum active time of around 1 ms per wake-up. A larger capacitance would increase the active time per wake-up and the charging time between wake-ups. To minimize the physical dimensions and the price of the node, we choose the minimum capacitance that allows the nodes to remain active for long enough to compensate for clock drift accumulated over a connection interval of 5 s (see Sec. 4.2).

The node also integrates a circuit to measure the current flow from the harvester, which can be used as a sensing signal [39]. The two-layer printed circuit board (PCB) shown in Fig. 11 measures 20 mm  $\times$  20 mm. The total cost of all components is \$8.73.

### 4.2 Software

We implement Bonito and the Find neighbor discovery protocol [19] on our battery-free nodes. Find is used to establish an initial encounter after a connection loss or a power failure.

**Bonito protocol settings.** We use the 2 Mbit/s BLE mode and the frame structure depicted in Fig. 12. Depending on the model type, encoded by one byte, a packet carries 1, 2, or 6 model parameters represented by 32-bit floating point values.

To jointly agree on the next connection interval, Bonito requires nodes to exchange messages bi-directionally during an encounter. The exact sequence of packet exchanges is subject to application requirements and can be flexibly configured. We implement the packet sequence shown in Fig. 13. When two nodes encounter each other using Find, one of the nodes receives the first beacon and replies with an acknowledgement. At all following encounters, the node that received the first beacon starts to listen at the time agreed on using Bonito.

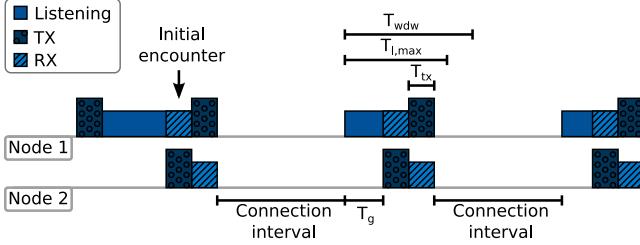


Figure 13: After the initial encounter, nodes use Bonito to agree on a connection interval. At the next encounter, one of the nodes starts to listen and the other node transmits its packet after a grace period to account for clock drift due to the long charging times. After receiving this packet, the listening nodes replies with its own packet.

Due to the small energy buffer, a node can keep the radio on for at most  $T_{wdw} = 1$  ms. Thus, the maximum listening time is  $T_{l,max} = T_{wdw} - T_{tx} - T_{ta} = 820\mu s$ , where  $T_{ta} \approx 40\mu s$  is the time it takes to switch from receive to transmit mode and  $T_{tx} = 140\mu s$  is the airtime of a packet with 6 model parameters and 4 B of application data. To increase the robustness to clock drift in the face of long charging times and hence long connection intervals, we let the node that sends first transmit its packet after a grace period of  $T_g = 0.5 \cdot T_{l,max} - 1.5 \cdot T_{tx} = 200\mu s$ . We can thus tolerate an offset of up to  $\pm 200\mu s$  between the clocks of the two nodes, which corresponds to a maximum connection interval of 5 s when taking into account the frequency tolerance of the 32 kHz crystal oscillator. Upon receiving the packet, the other node switches to transmit mode and sends its own packet.

In our current implementation of Bonito, the devices consider a connection as lost whenever a planned packet exchange fails, for example, due to fading, external interference, or when one of the two devices does not reach the turn-on threshold by the end of the connection interval. In this case, they return to discovery mode and use Find to re-establish the connection.

**Runtime support.** In addition to Bonito and Find, we implement an efficient *soft intermittency* runtime, where the MCU is gracefully suspended to an ultra low-power mode before an impeding power failure [19]. This reduces the costs associated with a cold start after a hardware reset and allows to keep track of time between consecutive wake-up events using the built-in real-time clock (RTC). To this end, a node periodically samples the capacitor voltage during charging with the built-in analog-to-digital converter (ADC) until the capacitor voltage reaches a software-defined turn-on threshold. Then the node executes protocol and application code until it is interrupted by the power-fail comparator, upon which it immediately transitions back into low-power mode to replenish its energy buffer.

Although our runtime tries to prevent hardware resets, after multiple seconds without any energy input, the sleep current drains the remaining charge from the capacitor and the node eventually powers off. While powered off, the on-board static

random access memory (SRAM) is subject to decay, that is, bits that were set to one may flip and become zero after some time. To still retain the trained model of a node’s charging time distribution across short power failures, we store it in a dedicated section of the SRAM. After every model update, we compute a checksum over this section and store it next to the model parameters. If the recomputed checksum after a hardware reset does not match the checksum stored in memory, we conclude that the memory is corrupted and restart training the model with the initial parameters.

## 5 Evaluation

This section uses testbed experiments to evaluate Bonito on real battery-free nodes under realistic, repeatable conditions. We start by showing in Sec. 5.2 how Bonito dynamically adjusts the connection interval to changes in the nodes’ charging times to maintain long-running connections. In Sec. 5.3, we compare Bonito against two baseline approaches. Finally, in Sec. 5.4, we quantify the runtime overhead of Bonito. Our experiments reveal the following key findings:

- Bonito establishes connections that outlast on average hundreds of consecutive encounters even between nodes that harvest from different types of energy sources.
- Bonito improves the throughput by 10–80× compared with the current state of the art. It achieves this by consciously keeping the connection interval as short as possible while maintaining a high success rate that agrees to within 1 % of the requested target probability.
- Depending on the distribution model, Bonito consumes between 4 % and 25 % of the energy available per wake-up on our nodes. The energy cost of losing a connection is 1000× higher than the energy overhead of Bonito.

### 5.1 Testbed and Settings

We connect two battery-free nodes (see Fig. 11) to two Shepherd observers [18]. In addition to recording spatio-temporal harvesting traces (see Sec. 2), Shepherd can also replay previously recorded traces and monitor the behavior of connected battery-free devices. The observers synchronously replay for all 60 links in our datasets (see Table 1) the two corresponding energy-harvesting traces. At the same time, the observers log the serial output and GPIO events of the attached nodes, which we use to compute performance metrics. In total, we collect measurements from 218 hours of testbed experiments.

For the stairs, office, and washer scenarios, we replay the recorded energy-harvesting traces as is. When using the original traces from the cars and jogging scenarios, however, we were not able to collect sufficient data points. The reason is that the piezoelectric harvesters were selected and tuned for the frequency and amplitude of the washer scenario, which led to a relatively low harvesting power in the cars and jogging scenarios, as evident from the small number of wake-ups in Table 1. Because it can take thousands of wake-ups until two nodes encounter each other, we had to scale the cars and jogging traces by a factor of five to allow for a meaningful

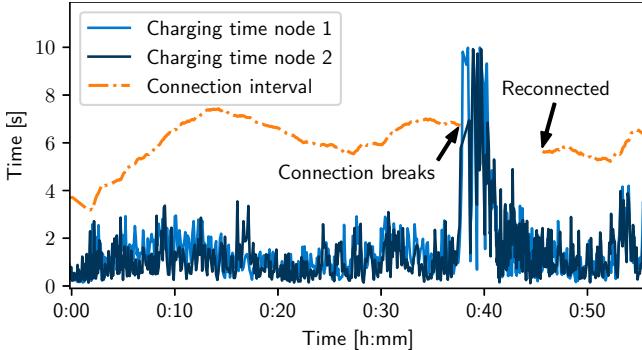


Figure 14: Real-world trace from testbed experiments showing the charging times of two nodes from the cars dataset. By dynamically adjusting the connection interval, Bonito maintains a connection for 37 min until the cars leave the highway and enter stop-and-go traffic; the charging times increase dramatically and the connection breaks.

evaluation. Note that this does not change the dynamics and shape of the charging time distributions, nor does it affect relative performance when comparing different approaches.

In all experiments, we configure Bonito with a target probability of  $p = 0.99$ . We use a learning rate of  $\eta = 0.01$  for the normal and exponential models and  $\eta = 0.001$  for GMM, which we found to perform well in a wide range of scenarios.

## 5.2 Maintaining Long-running Connections

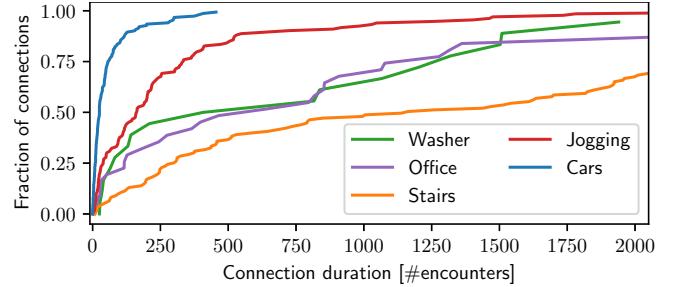
We begin by looking at how well Bonito can maintain a connection between battery-free devices. As an illustrative example, Fig. 14 shows the charging times of two nodes from the cars dataset and the connection interval determined by Bonito over the course of 55 min. Bonito successfully maintains the connection for more than half an hour by dynamically adjusting the connection interval based on the continuously updated models of the nodes’ charging time distributions. Then, after around 37 min, the two cars driving in convoy exit the highway and enter stop-and-go traffic. As a result, the charging times increase suddenly and exceed the connection interval—the connection is lost. At this point, the nodes switch over to executing the Find neighbor discovery protocol and successfully reconnect after roughly 10 min. Afterward, Bonito takes over and again maintains a connection for several minutes.

Fig. 15a plots for all datasets the cdf of the connection duration in terms of the number of encounters, while Fig. 15b plots it in terms of time for the unscaled datasets (see Sec. 5.1). Overall, we find that in 90 % of the cases, the nodes stay connected for at least 30 consecutive encounters, and 40 % of the connections last for 800 encounters or more. This demonstrates that Bonito enables, for the first time, reliable and efficient communication between intermittently powered nodes.

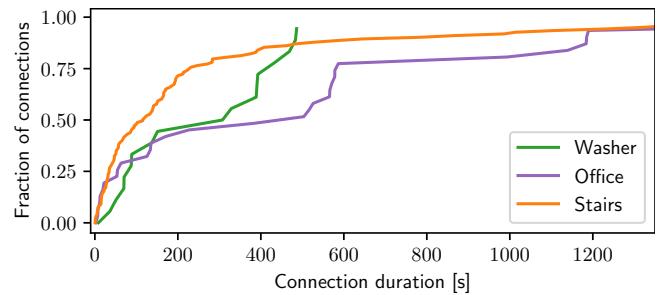
## 5.3 Bonito versus Baseline Approaches

We now compare Bonito against two baseline approaches:

- **Greedy:** This is the current state of the art. Using Greedy, nodes wake up and attempt to communicate as soon as



(a) cdf of connection duration in terms of number of encounters.



(b) cdf of connection duration in terms of time.

Figure 15: Bonito maintains connections over hundreds of encounters even in challenging scenarios with different types of energy sources.

they reach the minimum required energy level. Greedy is the prevalent execution model in the intermittent computing literature [8, 33] as it maximizes the effective duty cycle of a battery-free device.

- **Modest:** As a complementary approach to Greedy, we design Modest. Using Modest, each node keeps track of the maximum observed charging time  $c_{max}$ . During an encounter, two nodes  $i$  and  $j$  share their current maximum charging times  $c_{max,i}$  and  $c_{max,j}$ , and agree to meet again after a connection interval of  $T_C = \max(c_{max,i}, c_{max,j})$ .

Our comparison uses two end-to-end metrics that also account for periods where Find runs to establish a first encounter after a connection loss or power failure. *Throughput* is the number of packets delivered from one node to another node per time unit. Note that traffic is always bi-directional, that is, the same number of packets is also delivered in the other direction (see Fig. 13). *Latency* is the time between two consecutive packet exchanges. We also consider *success rate*, which is the ratio of successfully arranged encounters to the total number of trials when using Greedy, Modest, or Bonito.

Fig. 16 plots for each dataset the throughput gains of Bonito and Modest over Greedy. We see that Bonito improves the throughput by 10–80×. For example, for the stairs dataset, Bonito achieves a throughput of 15.18 pkt/s versus 0.33 pkt/s with Greedy. Modest outperforms Greedy across the board, too, but often falls far short of Bonito’s throughput.

To understand the reasons for the significant performance differences among the different approaches, we plot in Fig. 17 success rate and latency for the stairs dataset. As the charging

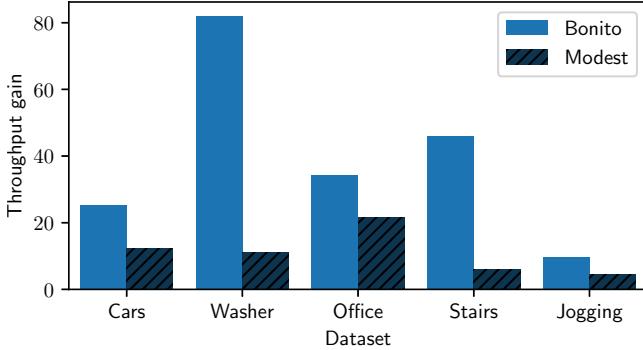


Figure 16: Throughput improvement over Greedy. By maintaining connections over many wake-ups, the average number of encounters with Bonito is at least an order of magnitude higher than with Greedy.

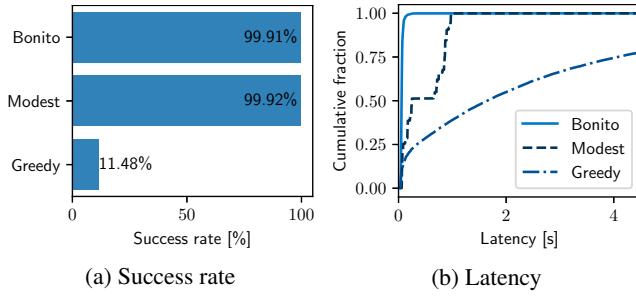


Figure 17: Detailed comparison of performance metrics for the stairs scenario. Bonito achieves a high success rate that is on a par with the Modest approach, while providing a significantly lower latency.

times vary across time and space, Greedy achieves a low success rate of only 11.48 % (see Fig. 17a). This means that in 9 out of 10 cases the nodes loses the connection right after the first encounter. Every time the connection is lost, the nodes cannot communicate until they reconnect, causing excessively long latencies as visible in Fig. 17b. Instead, Modest chooses the connection interval highly conservatively, which leads to a high success rate of 99.92 % but also long latencies. Bonito provides much shorter latencies at almost the same high success rate, which agrees to within 1 % of the requested target probability. By aiming to keep the connection interval short and to avoid the latency associated with reconnecting after a connection loss, Bonito significantly increases the end-to-end throughput compared with the two baseline approaches.

#### 5.4 Bonito’s Runtime Overhead

Next, we evaluate the runtime overhead of Bonito based on the logs from the testbed experiments. The overhead can be broken down into three components: (i) updating the model parameters using SGD, (ii) exchange of the model parameters over wireless during an encounter, and (iii) computing the inverse joint cdf to obtain the connection interval.

The time required to update the model is constant: 1.3  $\mu$ s for exponential, 3.2  $\mu$ s for normal, and 28.8  $\mu$ s for GMM. This constitutes up to 2.8 % of the around 1 ms active time per wake-up. Similarly, the airtime to exchange 4, 8, or 24 bytes

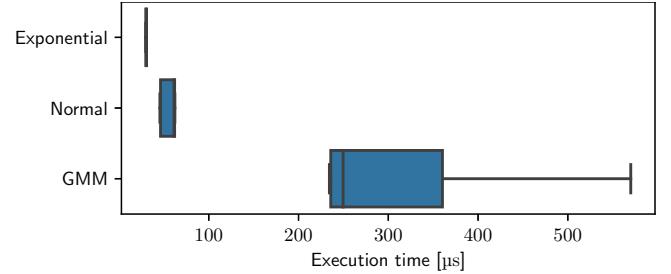


Figure 18: Distribution of execution times on our battery-free node when computing the inverse joint cdf. The execution time depends on the number of model parameters and varies with the number of bisection steps needed to satisfy the required tolerance.

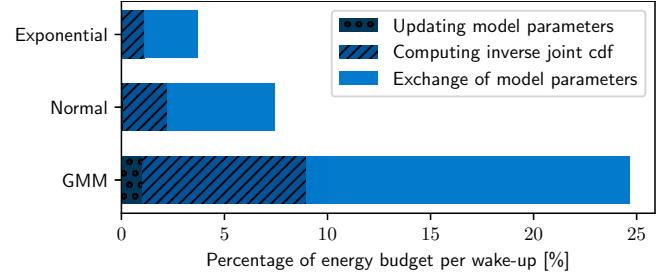


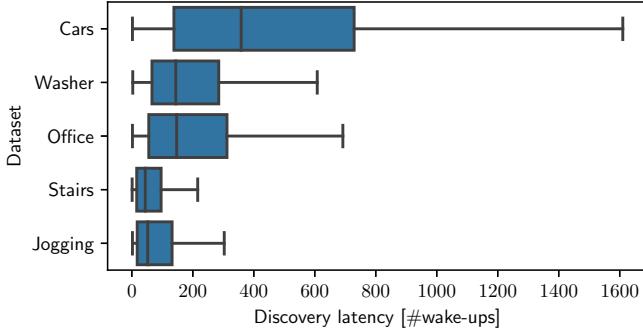
Figure 19: The energy overhead of Bonito ranges between 4 % and 25 % of the energy available per wake-up on our nodes. In absolute terms, the cost to recover from a lost connection is 1000 $\times$  higher.

of model parameters is fixed and determined by the bitrate of the BLE radio. By contrast, Fig. 18 shows that the time to compute the inverse joint cdf varies depending on the number of bisection steps required to reach the desired tolerance.

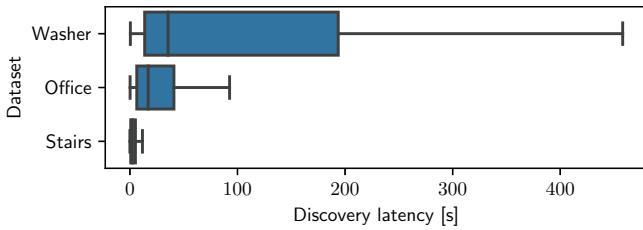
In terms of energy, our battery-free nodes have an energy budget of 27.5  $\mu$ J per wake-up. Fig. 19 shows for each model the median percentage of energy budget spent by Bonito. We can see that the required energy mainly depends on the number of model parameters and the computational complexity of evaluating the inverse joint cdf. In the worst case, for GMM, Bonito consumes 7.1  $\mu$ J, which amounts to about 25 % of the available energy per wake-up. To set this into perspective, Fig. 20a plots for all datasets the time it takes for two nodes to synchronize with the Find neighbor discovery protocol [19] in terms of the number of wake-ups, while Fig. 20b plots it in terms of time for the unscaled datasets (see Sec. 5.1). On average it takes 283 wake-ups, or 7782.5  $\mu$ J, to synchronize after a lost connection—1000 $\times$  more than the energy required by Bonito to maintain a connection. This demonstrates that, overall, the absolute energy costs of Bonito are well spent.

## 6 Case Study: Occupancy Monitoring

Occupancy monitoring is essential to save energy in homes and commercial buildings [10, 16]. Recently, it has also become an important tool to manage the spread of infectious diseases, such as SARS-CoV2 [37]. To assess the potential of Bonito for real-world battery-free applications, we conduct an



(a) Discovery latency in terms of number of wake-ups.



(b) Discovery latency in terms of time.

Figure 20: Synchronizing two devices with the Find neighbor discovery protocol takes a long time and consumes significant energy. Using Bonito, devices can establish long-running connections to periodically exchange data without the need to resynchronize.

occupancy monitoring case study with our prototype nodes.

**Occupancy sensor.** To efficiently count the number of people in a room, we use the solar panel as a sensor [23, 39] to detect when a person enters or leaves the room. Fig. 21 shows the solar panel current of two nodes mounted next to each other on a doorframe (see Fig. 22), when a person enters the room in Fig. 21a and when a person leaves the room in Fig. 21b. To detect the direction of movement, the nodes record the time when they detect the onset of the shadowing by the person. Then the nodes exchange the recorded times and compute the time difference  $\tau$ . The sign of  $\tau$  indicates the direction.

**Setup.** We mount two battery-free nodes equipped with IXYS SM141K06L solar panels next to each other on the doorframe at the entrance of an office room, as shown in Fig. 22. The nodes sample the solar panel current with a sampling rate of 1 kHz, and record the time when the solar panel current falls below 87.5 % of its average value. The nodes run Bonito and insert the timestamp of detected events into the packets. Together with logging information (charging time, connection interval, etc.) every packet carries 26 B of application data.

Because the clocks of the two nodes are not synchronized, timestamps are transmitted relative to the start of the corresponding packet. To this end, nodes measure the time between the detected event and the start of the transmission and insert the result into the packet. The receiving node timestamps the reception of the packet and converts the contained relative timestamp to its local clock. Finally, by relating a received

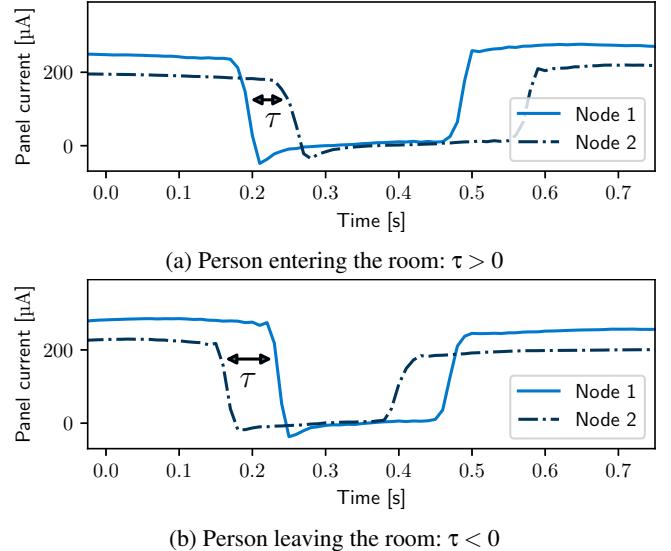


Figure 21: The shadow of a person passing ambient light harvesting devices on a doorframe causes a distinct temporal pattern in the solar panel current. By comparing the times of the onset of the shadowing on the two nodes, we can determine the direction of movement.

timestamp to the timestamp of the corresponding event that was recorded locally, the nodes compute the time difference  $\tau$ .

The nodes transmit the result over wireless to an nRF52840 development board that serves as a base station. We configure the base station to timestamp the reception of packets containing a detected event and button presses of two on-board push buttons, one for each direction. Four participants randomly enter and leave the room one by one. Another person records ground truth by pressing the corresponding button on the nRF52840 board precisely when a person passes through the doorframe.

**Results.** The confusion matrix in Table 2 shows that the system correctly classified 60 out of 61 events, corresponding to an accuracy of 96 %. It missed just one in-event, and falsely reported an in-event and an out-event for a single in-event.

Fig. 23 plots the latency in terms of the time between a



Figure 22: Two of our battery-free nodes are attached to the doorframe and harvest energy from ambient light. Thanks to Bonito, the nodes can communicate in a timely and reliable fashion, allowing them to count the number of people entering and leaving the room.

		Ground truth		
		In	Out	No event
Recorded	In	30	0	1
	Out	0	31	0
	No event	1	0	0

Table 2: By collaborating, the battery-free nodes classified people entering and leaving the room with an average accuracy of 96.83 %.

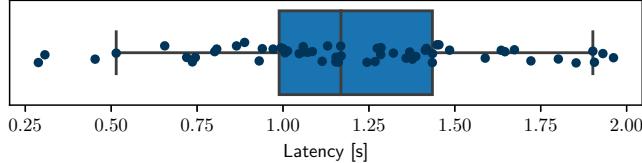


Figure 23: Due to the low communication latency provided by Bonito, the system reported detected events both timely and accurately.

button press and the reception of the detected event at the base station. The median latency was 1.2 s and all events were reported within less than 2 s. Over the course of the experiment, the two nodes successfully exchanged 10.56 kB of application data for an application-level throughput of 28.38 B/s.

Fig. 24 shows a ten-second excerpt from the experiment. The markers indicate the charging times of the nodes. Solid vertical lines indicate button presses (ground truth); dashed vertical lines indicate when an event was received at the base station. We can observe that, right after the received out-event, node 1 reports an exceptionally high charging time of 210 ms. This happens when the shadowing by a person occurs while a node charges its capacitor: The shadowing reduces the energy input for a short time, which prolongs the recharge. Nevertheless, by keeping the connection interval at around 700 ms, Bonito provides a stable connection despite such dynamics.

## 7 Discussion

Bonito is the first connection protocol for battery-free devices. It enables two devices to communicate efficiently and reliably by dynamically adapting the connection interval to changes in the devices’ energy availability. In this section, we discuss limitations and opportunities for extending Bonito.

**From connections to networks.** The ability to efficiently and reliably exchange data between two devices is the fundamental building block required to form large wireless networks consisting of multiple battery-free devices. A number of trade-offs and challenges arise from each of the possible approaches to move from the two-node setting to larger networks, which could be explored by future work. For example, devices may sequentially connect with their neighbors or devices may try to establish Bonito connections with one common connection interval between multiple devices.

**Communication with battery-powered devices.** While we focus on communication between two battery-free devices, Bonito is also useful for effective communication from battery-

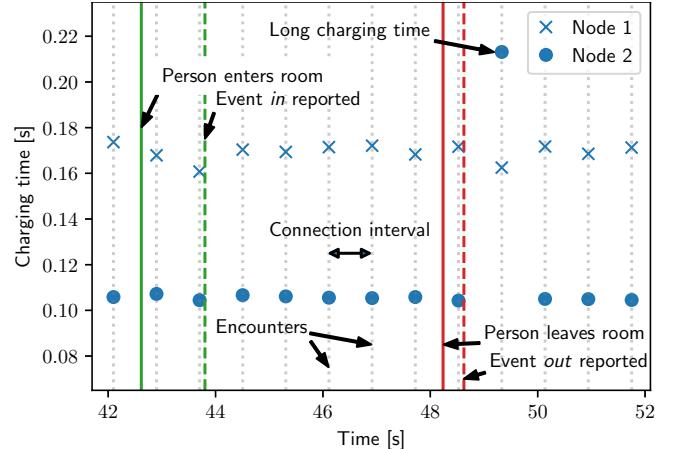


Figure 24: Example trace from the occupancy monitoring case study. The system correctly classifies and reports events to the base station. With Bonito, the connection interval is chosen large enough to sustain outliers of the charging time in response to transient shadowing.

free to battery-powered devices. For example, a battery-free tag may want to transmit data to a user’s smartphone or to a battery-powered gateway in a wireless sensor network. Because battery-powered devices are in control of their wake-up times, any connection interval works for them. Thus, instead of computing the inverse joint cdf of the charging time distribution of both devices, it is sufficient to compute the inverse cdf of the charging time of the battery-free device in order to determine a connection interval that works for both devices.

**Model accuracy.** The goodness-of-fit of the learned charging time model critically affects the performance of Bonito. With perfect knowledge of the underlying distribution, Bonito would compute the minimum connection interval feasible for the requested target probability. Overestimating the real distribution leads to increased delay, while underestimation reduces reliability. If the distribution is so complex that a large number of model parameters or a non-parametric model (e.g., a deep neural network) would be required to accurately capture this complexity, then the limited resources on a battery-free device may not be sufficient to learn the model online.

**Exploiting statistical dependence.** In the current implementation, Bonito assumes statistical independence of the nodes’ charging time distributions in order to compute a connection interval without prior knowledge of the statistical properties of the joint charging time distribution. After establishing a connection, the devices can record observations of the joint distribution and could attempt to exploit statistical dependence between their charging times, possibly improving communication efficiency and reliability.

## 8 Related Work

**Intermittent computing.** The thriving research area of intermittent computing has made great strides in recent years, including the first real deployments of battery-free sensors [1].

This achievement rests upon techniques that ensure forward progress [34], consistent peripheral state [6], and a reliable notion of time [11] despite frequent and random power failures. This line of research is highly relevant but completely orthogonal to our work as it deals exclusively with intermittency issues on *individual* devices and, if at all, considers communication with continuously powered base stations [42].

**Battery-free device-to-device communication.** Prior work on battery-free wireless device-to-device communication is mainly theoretical [24, 30, 50], studying the energy trade-offs for different scheduling, transmission, and decoding policies. Recent work discusses middleware and applications for networks of intermittently powered devices, yet explicitly leaves the question of how to communicate between the devices as an open problem [28, 48]. A simulative study also acknowledges the sheer difficulty of synchronizing the wake-up times of intermittently powered devices and proposes to communicate an energy state via an always-on backscatter radio, without demonstrating a real implementation or experiments [45]. Similar to Bonito, a recent theoretical work proposes to let nodes agree on a future point in time when they become active to increase communication throughput [46]. This time is computed based on a moving average of previous charging times, whereas Bonito lets the user explicitly trade reliability against delay by taking into account the charging time distributions.

In terms of practical work, tag-to-tag backscatter communication has mainly focused on physical-layer issues and considers intermittency an orthogonal problem [29, 35, 38, 49]. Instead, the Find neighbor discovery protocol explicitly addresses the intermittency problem and shows that by delaying wake-ups by a random time battery-free nodes can encounter each other faster [19]. We use Find to bootstrap efficient and reliable device-to-device communication with Bonito. Concurrently to our work, a protocol was proposed and implemented that lets devices “die early” when no packet is received to preserve energy and maximize the number of wake-ups [13].

**Delay-tolerant networking (DTN).** DTN studies networks that are only intermittently connected because of, for example, node failures, mobile users, and power outages [4, 17]. Both DTN and Bonito have the same high-level goal: effective communication in intermittently connected networks. However, DTN and Bonito address orthogonal problems toward the same end goal. While DTN is concerned with forwarding, routing, naming, in-network storage, and optimization of node trajectories to generate encounters in the spatial domain, Bonito aims to generate encounters in the time domain between nodes that are spatially close to each other. Whether concepts from the DTN literature could be applied on top of Bonito is an interesting question for future research.

**Energy-aware MAC protocols.** Numerous MAC protocols have been proposed for ad-hoc and sensor networks [15]. These protocols turn the radio off most of the time, and power it up only to send or receive a packet. The goal is to achieve

a desired network lifetime by maintaining a certain average duty cycle. A fundamental assumption of these protocols is that the radio can be powered up *at any point in time*, which is exploited to reduce idle listening by flexibly scheduling communication among nodes. This is, however, not possible in a battery-free system, where devices are unavailable whenever the capacitor voltage is below a certain threshold, which renders existing energy-aware MAC protocols ineffective.

## 9 Conclusions

We have presented Bonito, a connection protocol for wireless battery-free devices. By adapting the connection interval to the different and time-varying charging times of intermittently powered nodes, Bonito maintains long-running connections that provide significantly better throughput, latency, and reliability than the state of the art. We have evaluated Bonito by implementing it on a battery-free prototype, conducting testbed experiments with real energy-harvesting traces from diverse scenarios, and demonstrating its utility in an occupancy monitoring case study. With Bonito, we contribute a prime communication primitive, device-to-device unicast, that brings the capabilities of battery-free systems one step closer to those known from today’s battery-supported systems.

## Availability

The data described in Sec. 2 and a Python implementation of the Bonito protocol from Sec. 3 are available under a permissive MIT license at <https://bonito.nes-lab.org/>.

## Acknowledgments

We thank Sarah Nollau, Ingmar Splitt, Friedrich Schmidt, Jus-  
tus Paulick, and Lebenshilfe Altenburg e. V. for supporting  
the data collection campaign, and all participants of the occu-  
pancy monitoring case study. Thanks also to the anonymous  
reviewers, and to our shepherd, Shyam Gollakota. This work  
was supported by the German Research Foundation (DFG)  
within the Emmy Noether project NextIoT (grant ZI 1635/2-1)  
and the Center for Advancing Electronics Dresden (cfaed).

## A Appendix: Gradient Equations

**Exponential distribution.** The derivative of the log-likelihood function is given by:

$$\mathcal{L}(\lambda) = \log(\lambda \cdot \exp(-\lambda x)) = \log \lambda - \lambda x_i \quad (12)$$

$$\nabla \mathcal{L}(\lambda) = \frac{1}{\lambda} - x_i \quad (13)$$

Calculating the *natural gradient* by defining the step size in terms of the Kullback-Leibler divergence in the distribution space has been shown to speed up convergence in many cases [2]. We obtain the natural gradient by multiplying the regular gradient from (13) with the inverse of the Fisher Information Matrix of the exponential distribution  $M_{exp}$ :

$$M_{exp} = \lambda^{-2} \quad (14)$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = [M_{exp}]^{-1} \cdot \frac{1}{\lambda} - x_i = \lambda - \lambda^2 \cdot x_i \quad (15)$$

**Gaussian mixture model.** We adopt the gradient equations from [44]: Let  $f(x_i, \mu, \sigma^2)$  be the probability density function of the standard normal distribution. The responsibility function  $r(x_i, k)$  quantifies the contribution of the  $k$ -th component to the model:

$$r(x_i, k) = \frac{\rho_k \cdot f(x_i, \mu_k, \sigma_k^2)}{\sum_l^K (\rho_l \cdot f(x_i, \mu_l, \sigma_l^2))} \quad (16)$$

The update equations for the model parameters for the  $k$ -th component are then:

$$\frac{\partial \mathcal{L}}{\partial \rho_k} = r(x_i, k) - \rho_k \quad (17)$$

$$\frac{\partial \mathcal{L}}{\partial \mu_k} = \frac{1}{\rho_k} \cdot r(x_i, k) \cdot (x_i - \mu_k) \quad (18)$$

$$\frac{\partial \mathcal{L}}{\partial \sigma_k^2} = \frac{1}{\rho_k} \cdot r(x_i, k) \cdot (x_i - \mu_k)^2 - \sigma_k^2 \quad (19)$$

**Normal distribution.** We consider the special case of a gaussian mixture model with a single component and also use the equations from [44]:

$$\frac{\partial \mathcal{L}}{\partial \mu} = (x_i - \mu) \quad (20)$$

$$\frac{\partial \mathcal{L}}{\partial \sigma^2} = (x_i - \mu)^2 - \sigma^2 \quad (21)$$

## References

- [1] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campana, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Battery-less zero-maintenance embedded sensing at the mithräum of circus maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys)*, 2020.
- [2] S. Amari and S.C. Douglas. Why natural gradient? In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1998.
- [3] Abu Bakar and Josiah Hester. Making sense of intermittent energy harvesting. In *Proceedings of the 6th ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSSys)*, 2018.
- [4] Sanjit Biswas and Robert Morris. ExOR: opportunistic multi-hop routing for wireless networks. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2005.
- [5] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade: Second Edition*. Springer Berlin Heidelberg, 2012.
- [6] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.
- [7] Albert Cohen, Xipeng Shen, Josep Torrellas, James Tuck, Yuanyuan Zhou, Sarita Adve, Ismail Akturk, Saurabh Bagchi, Rajeev Balasubramonian, Rajkishore Barik, Micah Beck, Ras Bodik, Ali Butt, Luis Ceze, Haibo Chen, Yiran Chen, Trishul Chilimbi, Mihai Christodorescu, John Criswell, Chen Ding, Yufei Ding, Sandhya Dwarkadas, Erik Elmroth, Phil Gibbons, Xiaochen Guo, Rajesh Gupta, Gernot Heiser, Hank Hoffman, Jian Huang, Hillery Hunter, John Kim, Sam King, James Larus, Chen Liu, Shan Lu, Brandon Lucia, Saeed Maleki, Somnath Mazumdar, Julian Neamtiu, Keshav Pingali, Paolo Rech, Michael Scott, Yan Solihin, Dawn Song, Jakub Szefer, Dan Tsafrir, Bhuvan Urgaonkar, Marilyn Wolf, Yuan Xie, Jishen Zhao, Lin Zhong, and Yuhao Zhu. Inter-disciplinary research challenges in computer systems for the 2020s. Technical report, 2018.
- [8] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
- [9] Pablo Corbalán and Gian Pietro Picco. Concurrent ranging in ultra-wideband radios: Experimental evidence, challenges, and opportunities. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2018.
- [10] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. BOSS: Building operating system services. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [11] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the 25th ACM International Conference on*

- Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [12] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3), 2020.
  - [13] Vishal Deep, Mathew L. Wymore, Alexis A. Aurandt, Vishak Narayanan, Shen Fu, Henry Duwe, and Daji Qiao. Experimental Study of Lifecycle Management Protocols for Batteryless Intermittent Communication. In *Proceedings of the 18th IEEE International Conference on Mobile Ad Hoc and Smart Systems (MASS)*, 2021.
  - [14] Farzan Dehbashi, Ali Abedi, Tim Brecht, and Omid Abari. Verification: can wifi backscatter replace RFID? In *Proceedings of the 27th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2021.
  - [15] I. Demirkol, C. Ersoy, and F. Alagoz. Mac protocols for wireless sensor networks: a survey. *IEEE Communications Magazine*, 44(4), 2006.
  - [16] Varick L. Erickson, Miguel Á. Carreira-Perpiñán, and Alberto E. Cerpa. Occupancy Modeling and Prediction for Building Energy Management. *ACM Transactions on Sensor Networks*, 10(3), 2014.
  - [17] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2003.
  - [18] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless IoT. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.
  - [19] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.
  - [20] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
  - [21] Bernhard Großwindhager, Michael Stocker, Michael Rath, Carlo Alberto Boano, and Kay Römer. Snaploc: An ultra-fast uwb-based indoor localization system for an unlimited number of tags. In *Proceedings of the 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2019.
  - [22] Josiah Hester and Jacob Sorber. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2017.
  - [23] Sara Khalifa, Mahbub Hassan, Aruna Seneviratne, and Sajal K. Das. Energy-Harvesting Wearables for Activity-Aware Services. *IEEE Internet Computing*, 19(5), 2015.
  - [24] Meng-Lin Ku, Wei Li, Yan Chen, and K. J. Ray Liu. Advances in energy harvesting communications: Past, present, and future challenges. *IEEE Communications Surveys & Tutorials*, 18(2), 2016.
  - [25] J.N. Laneman, D.N.C. Tse, and G.W. Wornell. Cooperative diversity in wireless networks: Efficient protocols and outage behavior. *IEEE Transactions on Information Theory*, 50(12), 2004.
  - [26] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.
  - [27] Tianxing Li and Xia Zhou. Battery-free eye tracker on glasses. In *Proceedings of the 24th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.
  - [28] Gaosheng Liu and Lin Wang. Self-Sustainable Cyber-Physical Systems with Collaborative Intermittent Computing. In *Proceedings of the 12th ACM International Conference on Future Energy Systems*, 2021.
  - [29] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Golakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2013.
  - [30] Edward Longman, Oktay Cetinkaya, Mohammed El-Hajjar, and Geoff V. Merrett. Wake-up Radio-enabled Intermittently-powered Devices for Mesh Networking: A Power Analysis. In *Proceedings of the 18th IEEE Annual Consumer Communications Networking Conference (CCNC)*, 2021.
  - [31] Brandon Lucia, Vignesh Balaj, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*, 2017.

- [32] Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. Computational nanosatellite constellations: Opportunities and challenges. *ACM GetMobile: Mobile Computing and Communications*, 25(1), 2021.
- [33] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [34] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic Task-based Intermittent Execution for Energy-harvesting Devices. *ACM Transactions on Sensor Networks*, 16(1), 2020.
- [35] Amjad Yousef Majid, Michel Jansen, Guillermo Ortas Delgado, Kasim Sinan Yildirim, and Przemysław Pawełczak. Multi-hop backscatter tag-to-tag networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2019.
- [36] Amjad Yousef Majid, Patrick Schilder, and Koen Langendoen. Continuous sensing on intermittent power. In *Proceedings of the 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2020.
- [37] Giulia Pazzaglia, Marco Mameli, Luca Rossi, Marina Paolanti, Adriano Mancini, Primo Zingaretti, and Emanuele Frontoni. People Counting on Low Cost Embedded Hardware During the SARS-CoV-2 Pandemic. In *Proceedings of Pattern Recognition. ICPR International Workshops and Challenges*, 2021.
- [38] Jihoon Ryoo, Yasha Karimi, Akshay Athalye, Milutin Stanaćević, Samir R. Das, and Petar Djurić. Barnet: Towards activity recognition using passive backscattering tag-to-tag network. In *Proceedings of the 16th ACM Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.
- [39] Muhammad Moid Sandhu, Sara Khalifa, Kai Geissdoerfer, Raja Jurdak, and Marius Portmann. SolAR: Energy positive human activity recognition using solar cells. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021.
- [40] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th ACM International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.
- [41] Olga Saukh, David Hasenfratz, and Lothar Thiele. Reducing multi-hop calibration errors in large-scale mobile sensor networks. In *Proceedings of the 14th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2015.
- [42] Lukas Sigrist, Rehan Ahmed, Andres Gomez, and Lothar Thiele. Harvesting-Aware Optimal Communication Scheme for Infrastructure-Less Sensing. *ACM Transactions on Internet of Things*, 1(4), 2020.
- [43] Vamsi Talla, Joshua Smith, and Shyamnath Gollakota. Advances and Open Problems in Backscatter Networking. *ACM GetMobile: Mobile Computing and Communications*, 24(4), 2021.
- [44] D. M. Titterington. Recursive Parameter Estimation Using Incomplete Data. *Journal of the Royal Statistical Society: Series B (Methodological)*, 46(2), 1984.
- [45] Alessandro Torrisi, Kasim Sinan Yildirim, and Davide Brunelli. Enabling Transiently-Powered Communication via Backscattering Energy State Information. In *Applications in Electronics Pervading Industry, Environment and Society*. Springer International Publishing, 2021.
- [46] Mathew L. Wymore, Vishal Deep, Vishak Narayanan, Henry Duwe, and Daji Qiao. Lifecycle Management Protocols for Batteryless, Intermittent Sensor Nodes. In *Proceedings of the 39th IEEE International Performance Computing and Communications Conference (IPCCC)*, 2020.
- [47] Xun Xian, Xinran Wang, Jie Ding, and Reza Ghanadan. Assisted learning: A framework for multi-organization learning. In *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*, 2020.
- [48] Kasim Sinan Yildirim and Przemyslaw Pawelczak. On Distributed Sensor Fusion in Batteryless Intermittent Networks. In *Proceedings of the 15th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2019.
- [49] Jia Zhao, Wei Gong, and Jiangchuan Liu. X-tandem: Towards multi-hop backscatter communication with commodity wifi. In *Proceedings of the 24th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.
- [50] Tongxin Zhu, Jianzhong Li, Hong Gao, and Yingshu Li. Broadcast scheduling in battery-free wireless sensor networks. *ACM Transactions on Sensor Networks*, 15(4), 2019.



# Saiyan: Design and Implementation of a Low-power Demodulator for LoRa Backscatter Systems

Xizhen Guo  
*Tsinghua University*      Longfei Shangguan  
*University of Pittsburgh & Microsoft*      Yuan He  
*Tsinghua University*  
Nan Jing      Jiacheng Zhang      Haotian Jiang      Yunhao Liu  
*Yanshan University*      *Tsinghua University*      *Tsinghua University*      *Tsinghua University*

## Abstract

The radio range of backscatter systems continues growing as new wireless communication primitives are continuously invented. Nevertheless, both the bit error rate and the packet loss rate of backscatter signals increase rapidly with the radio range, thereby necessitating the cooperation between the access point and the backscatter tags through a feedback loop. Unfortunately, the low-power nature of backscatter tags limits their ability to demodulate feedback signals from a remote access point and scales down to such circumstances.

This paper presents Saiyan, an ultra-low-power demodulator for long-range LoRa backscatter systems. With Saiyan, a backscatter tag can demodulate feedback signals from a remote access point with moderate power consumption and then perform an immediate packet re-transmission in the presence of packet loss. Moreover, Saiyan enables rate adaption and channel hopping – two PHY-layer operations that are important to channel efficiency yet unavailable on long-range backscatter systems. We prototype Saiyan on a two-layer PCB board and evaluate its performance in different environments. Results show that Saiyan achieves  $3.5\text{--}5\times$  gain on the demodulation range, compared with state-of-the-art systems. Our ASIC simulation shows that the power consumption of Saiyan is around  $93.2\ \mu\text{W}$ . Code and hardware schematics can be found at: <https://github.com/ZangJac/Saiyan>.

## 1 Introduction

Backscatter radios have emerged as an ultra-low-power and economical alternative to active radios. The ability to communicate over long distances is critical to the practical deployment of backscatter systems, particularly in outdoor scenarios (*e.g.*, smart farm) where backscatter tags need to deliver their data to a remote access point regularly. Conventional RFID technology [12] functions within only a few meters

Yuan He is the corresponding author.

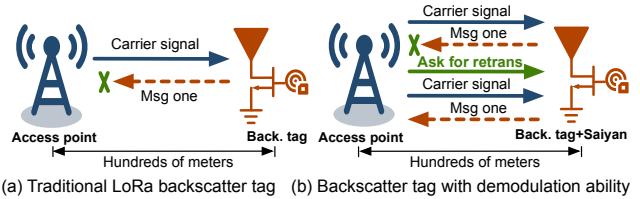


Figure 1: Saiyan empowers the LoRa backscatter tag to demodulate feedback signals from a remote access point.

and is not well suited for outdoor scenarios. To this end, the research community has proposed long-range backscatter approaches [23, 40, 47] that leverage Chirp Spreading Spectrum (CSS) modulation on LoRa [5] to improve the signal resilience to noise, thereby extending the communication range. For instance, LoRa backscatter [47] allows tags to communicate with a source and a receiver separated by 475 m.

However, existing long-range LoRa backscatter systems present a new challenge on packet delivery. The backscatter signals travel twice the link distance and suffer drastic attenuation as the link distance scales. They become very weak after traveling long distances, thereby causing severe bit errors and packet losses. Figure 2 shows the Bit Error Rate (BER) of PLoRa [40] and Aloba [23], two representative long-range LoRa backscatter systems. Evidently, the BER of both systems rises rapidly from less than 1% to over 50% as the tag is moved away from the transmitter (Tx). The receiver is almost unable to demodulate any backscatter signal once the tag is placed 20 m away from the transmitter. Considering that the backscatter tags are unaware of packet loss, each packet must be transmitted blindly for multiple times to lift the packet delivery ratio, which inevitably wastes precious energy and wireless spectrum and cause interference to other radios that work on the same frequency band.

To address these issues, we expect a *downlink* from the access point to the backscatter tag, through which the feedback signals (*e.g.*, asking for a packet re-transmission) can be delivered, thereby forming a *feedback loop*. We envision that such a feedback loop will bring opportunities to bridge

the gap between long-range backscatter communication and the growing packet loss rate, as reflected on the following aspects:

- *Making on-demand re-transmissions in the presence of packet loss.* The backscatter tag demodulates feedback signals from an access point and makes a re-transmission only if it is asked to do so. This reactive packet re-transmission can mitigate packet loss while improving power and channel efficiency.
- *Scheduling channel hopping to minimize interference.* The unlicensed band where the LoRa resides in is already over crowded. The access point monitors the wireless spectrum and notifies the backscatter tag to switch channels in the presence of in-band interference. As such, the channel utilization and packet delivery ratio can be improved effectively.
- *Adapting data rate to link condition.* The condition of backscatter links varies over time. The access point assesses each backscatter link and keeps the backscatter tag updated through the feedback loop. Each tag then adapts its data rate proactively to utilize the wireless link better.

In addition, such a feedback loop empowers the network administrator to turn on/off sensors on backscatter tags remotely, thereby avoiding labor-intensive and time-consuming physical access to the devices.

To enjoy these benefits, the primary hurdle to overcome is the packet demodulation on LoRa backscatter tags. LoRa is based on frequency modulation. To demodulate a LoRa symbol, the commercial LoRa receiver operates by down-converting the incident LoRa chirp to the baseband, sampling it at twice the chirp bandwidth (BW), and then converting the signal samples from the time domain to the frequency domain using Fast Fourier Transformation (FFT). These operations consume over 40 mW power altogether [6]. Considering a miniaturized energy harvester equipped with a palm-sized solar panel, this harvester merely generates 1 mW power every 25.4 seconds in a bright day [3]. In other words, to support the standard LoRa demodulation, a backscatter tag needs to wait for 17 minutes until it accumulates enough power. Although the envelope detector has been used for packet demodulation on many backscatter systems [38, 46, 52], it is ill-suited for LoRa demodulation because the envelope of a LoRa signal is a constant.

In this paper, we propose Saiyan, a low-power demodulator for long-range LoRa backscatter systems. Saiyan is based on an observation that *a frequency-modulated chirp signal can be transformed into an amplitude-modulated signal using a differential circuit*. The amplitude of this transformed signal scales with the frequency of the incident chirp signal, thereby allowing us to demodulate a LoRa chirp by tracking the peak amplitude on its transformed counterpart without using power-intensive hardware, such as a down-converter and an ADC. To put this high-level idea into practice, the challenges in design

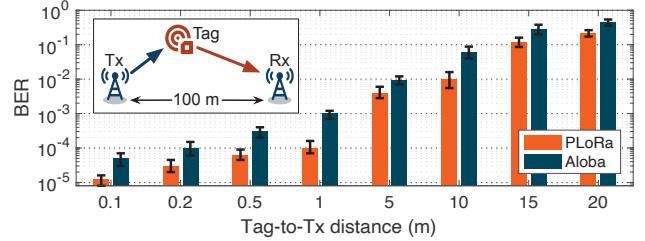


Figure 2: BER of PLoRa [40] and Aloba [23] in different tag-to-transmitter settings. The BER of both systems rises dramatically with the increasing distance between the transmitter and the tag. We re-implement both systems on PCB.

and implementation must be addressed, as summarized below.

**Frequency-amplitude transformation.** The low-power nature of backscatter tags requires the differential circuit to be extremely low-power. Moreover, to support higher data rate, such a differential circuit should also be hyper-sensitive to the frequency variation of LoRa signals. However, the narrow bandwidth of LoRa signals (*e.g.*, 125/250/500 KHz) renders the conventional detuning circuits, such as RLC resonant circuit, inapplicable. In Saiyan, we instead repurpose the Surface Acoustic Wave (SAW) filter as a signal converter by leveraging its sharp frequency response (§2.1). To minimize the power consumption on demodulation, we further replace the power-consuming ADC with a well-designed double-threshold based comparator (§2.2) coupled by a proactive voltage sampler (§2.3).

**Improving the demodulation sensitivity.** Although the aforementioned vanilla Saiyan can demodulate LoRa signals with the minimum power consumption, its communication range is limited to 55 m because of the Signal-to-Noise Ratio (SNR) losses in both SAW filter and envelope detector. To extend the communication range, we introduce a low-power cyclic-frequency shifting circuit coupled with an Intermediate Frequency (IF) amplifier to simultaneously remove the noise while magnifying the signal power. This low-power circuit brings 11 dB SNR gain and doubles the demodulation range (§3.1). Furthermore, a low-power correlator is leveraged to extend the demodulation range further to 148 m (§3.2).

**Implementation.** We implement Saiyan on a 25 mm × 20 mm two-layer Printed Circuit Board (PCB) using analog circuit components and an ultra-low power Apollo2 MCU [13]. The Application Specific Integrated Circuit (ASIC) simulation shows that the power consumption can be reduced to 93.2 μW, which is affordable on an energy harvesting tag. The main contributions of this paper are summarized as follows:

- We simplify the standard LoRa demodulation from energy perspective and design the first-of-its-kind low-power LoRa demodulator that can run on an energy harvesting tag.
- We design a set of simple but effective circuits and algorithms, prototyping them on PCB board for system evaluation.

- We demonstrate that Saiyan outperforms the state-of-the-arts on power consumption, communication range, and throughput.

The remainder of this paper is structured as follows. We present the design of vanilla Saiyan in Section 2, followed by super Saiyan in Section 3. Section 4 describes the implementation details. The experiment (§5) follows. We review related works in Section 6 and conclude in Section 7.

## 2 Vanilla Saiyan

A LoRa symbol is represented by a chirp whose frequency grows linearly over time, as formulated below.

$$s(t) = A \sin(2\pi f(t)t) \quad (1)$$

where  $A$  is the signal amplitude;  $f(t) = F_0 + kt$  describes how the frequency of this chirp signal varies over time;  $F_0$  is the initial frequency offset;  $k$  is the frequency changing rate. The frequency of a LoRa chirp wraps to 0 right after peaking  $BW$ —the bandwidth of LoRa. Different LoRa chirps peak the frequency  $BW$  at different time due to the difference in their initial frequency offset, as shown in Figure 3(a). Applying a differential to a LoRa chirp, we have:

$$\begin{aligned} s'(t) &= \frac{ds(t)}{dt} = A \cos(2\pi f(t)t) [2\pi \frac{df(t)}{dt} t + 2\pi f(t)] \quad (2) \\ &= \underbrace{2\pi A(F_0 + 2kt)}_{\text{Amplitude}} \cos(2\pi f(t)t) \end{aligned}$$

The above equation indicates that the amplitude of the transformed signal  $s'(t)$  is proportional to the frequency of the input LoRa chirp  $s(t)$ , as shown in Figure 3(b). This frequency-amplitude correlation allows us to demodulate the frequency-modulated (FM) chirp signal by tracking the peak amplitude of its transformed amplitude-modulated (AM) counterpart.

### 2.1 Frequency-amplitude Transformation

To realize the differential operation [10], an intuitive solution is using RLC resonant circuit. However, the narrow bandwidth of LoRa (e.g., 125/250/500 KHz) renders this idea infeasible (see Appendix A.1 for details). In Saiyan we instead exploit the sharp frequency response of the Surface Acoustic Wave (SAW) filter to transform LoRa chirps into amplitude-modulated signals.

**SAW filter primer.** A SAW filter consists of two interdigital transducers (shown in Figure 4). The input interdigital transducer transforms electrical signals into acoustic waves; the output interdigital transducer then transforms acoustic waves back into electrical signals. This two-stage signal transformation introduces 6 dB insertion loss to the incident signal [4].

**Re-purposing SAW filter as a signal converter.** Our design is based on the observation that the frequency response of a

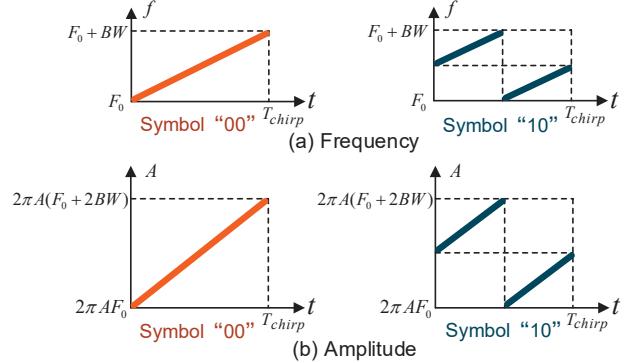


Figure 3: LoRa symbols before and after frequency-amplitude transformation. (a) Different LoRa symbols in the frequency domain. (b) The amplitude of LoRa symbols after frequency-amplitude transformation.

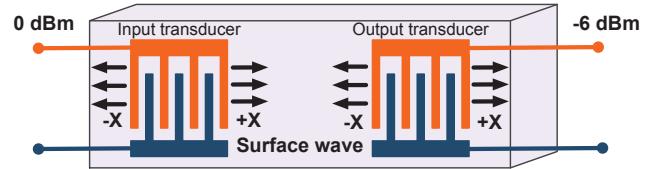


Figure 4: Diagram of the SAW filter. The SAW filter converts electrical signal into acoustic signal and then back through two inter-digital transducers.

SAW filter grows monotonically within a certain frequency band (termed as *critical band*). After passing through the critical band, the chirp signal will be transformed into an AM signal whose amplitude scales with the frequency of this input FM chirp. This allows us to demodulate LoRa chirp by simply tracking the peak amplitude of the AM signal. On the other hand, since SAW filter by its own design is battery-free, such frequency-amplitude transformation doesn't incur extra power consumption to backscatter tags.

In Saiyan, we take into account the working frequency and bandwidth of LoRa signals and select a general-purpose Qualcomm B3790 [1] SAW filter as the signal converter. Figure 5 shows its frequency response. The signal amplitude grows by 25 dB as the frequency of the incident signal scales from 433.5 MHz to 434 MHz (500 KHz bandwidth). To validate this 25 dB amplitude gap is strong enough for differentiating LoRa chirps, we feed four different chirp symbols into this SAW filter and plot the output in Figure 6. Evidently, these symbols peak their amplitude at clearly different time points, confirming the effectiveness of the SAW filter.

### 2.2 Demodulation

The transformed symbols are down-converted to the baseband through an envelope detector. Before demodulation, the

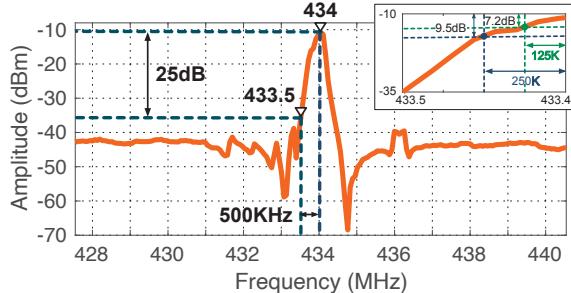


Figure 5: The amplitude-frequency response of the SAW filter adopted by Saiyan. The central frequency is 434 MHz. The measured insertion loss of this SAW filter is 10 dB. We observe 25 dB, 9.5 dB, and 7.2 dB amplitude variation as the frequency of an incident signal grows from 433.5 MHz, 433.75 MHz, and 433.875 MHz to 434 MHz, respectively.

standard LoRa receiver first digitizes these baseband signals using an ADC, which is power intensive. To save power, an intuitive solution is to replace the ADC with a low-power voltage comparator [37, 46]. The threshold of this comparator is set to a value slightly lower than the peak amplitude of the basband signal. This allows us to locate the peak amplitude by checking the comparator’s output. Unfortunately, due to the in-band interference and hardware noise, the transformed AM signal may experience multiple amplitude peaks and valleys that may confuse the comparator.

**Double-threshold based comparator.** In Saiyan we instead adopt a double-threshold based comparator to stabilize the output binary sequence. Let  $U_H$  and  $U_L$  denote the high-voltage and low-voltage threshold defined in this comparator. When the amplitude of an input signal is sufficiently higher than  $U_H$ , the comparator outputs a high voltage. When the amplitude of this signal is equivalent to  $U_H$  or above, no chattering occurs since the output will not respond unless the input falls below  $U_L$ . Following this idea, the output voltage  $B_i$  can be formulated as follows:

$$B_i = \begin{cases} \text{low}, & \text{if } A_i < U_H \quad \& \quad B_{i-1} = \text{low} \\ \text{high}, & \text{if } A_i \geq U_H \quad \& \quad B_{i-1} = \text{low} \\ \text{low}, & \text{if } A_i < U_L \quad \& \quad B_{i-1} = \text{high} \\ \text{high}, & \text{if } A_i \geq U_L \quad \& \quad B_{i-1} = \text{high} \end{cases} \quad (3)$$

where  $A_i$  represents the amplitude of the  $i^{\text{th}}$  signal sample. This double-threshold comparator takes into account the amplitude samples both in the past and present. It nulls out the chattering caused by the amplitude oscillation. The threshold setup is detailed in system implementation (§4).

To show the effectiveness of this double-threshold based comparator, we apply it to a LoRa chirp and plot the output in Figure 7. For comparison, we also plot the output of two single-threshold based comparators ( $U_H$  alone and  $U_L$  alone, respectively). We can see that using a high threshold  $U_H$

Table 1: The required sampling rate (KHz) in theory/practice to achieve 99.9% decoding accuracy.

	SF=7	SF=8	SF=9	SF=10	SF=11	SF=12
<b>K=1</b>	15.6/20	7.8/12	3.9/5.5	1.95/2.6	0.98/1.2	0.49/0.6
<b>K=2</b>	31.2/40	15.6/20	7.8/12	3.9/5.5	1.95/2.6	0.98/1.2
<b>K=3</b>	62.5/85	31.2/40	15.6/20	7.8/12	3.9/5.5	1.95/2.6
<b>K=4</b>	125/180	62.5/85	31.2/40	15.6/20	7.8/12	3.9/5.5
<b>K=5</b>	250/400	125/180	62.5/85	31.2/40	15.6/20	7.8/12

alone fails to detect the amplitude peak due to the valleys emerging on signal amplitude (*i.e.*,  $t \in [t_E, t_F]$  in Figure 7(b)). Using a low threshold  $U_L$  alone causes false positives due to the misleading peak emerging on the signal amplitude ( $t \in [t_A, t_B]$  in Figure 7(d)). In contrast, the double-threshold based comparator produces a series of stable binary voltages that can guide us to locate the peak amplitude at the correct position (*i.e.*, at the tail of the high voltage samples  $t_F$  shown in Figure 7(e)).

### 2.3 Low-power Voltage Sampler

The comparator quantizes chirp samples into binary voltages which are stored in a counter of micro-controller (MCU). The sampling rate tradeoffs the power consumption and the demodulation performance and thus cannot be set arbitrarily. A higher sampling rate supports a higher link throughput. It however consumes more power. Suppose a LoRa chirp encodes  $K$  bits data. The data rate equals  $K \cdot BW / 2^{SF}$ , where  $BW$  and  $SF$  respectively represent bandwidth and spreading factor. According to the Nyquist sampling theorem, the sampling rate should be not lower than  $2 \cdot BW / 2^{SF-K}$ .

However, in reality, using the theoretical minimum sampling rate exacerbates bit errors. We conduct a benchmark experiment to measure the practical sampling rate required to achieve 99.9% decoding accuracy. Table 1 lists the results with different settings of spreading factor and coding rate. We find that the required sampling rate in practice is slightly higher than the theoretical minimum sampling rate. Suggested by this result, we conservatively set the sampling rate to  $3.2 \cdot BW / 2^{SF-K}$ , which guarantees the demodulation performance.

**Decoding.** After quantization, the low-power MCU decodes each LoRa chirp by localizing the bit ‘1’ within each LoRa symbol, as shown in Figure 8. The LoRa preamble contains ten identical up-chirps. Upon detecting the LoRa preamble, Saiyan waits for 2.25 symbol times (sync. symbols) and operates demodulation on the payload hereafter.

**Remarks.** The vanilla Saiyan demodulates LoRa signals with the minimum power consumption. However, its demodulation sensitivity is limited due to the signal attenuation in the SAW

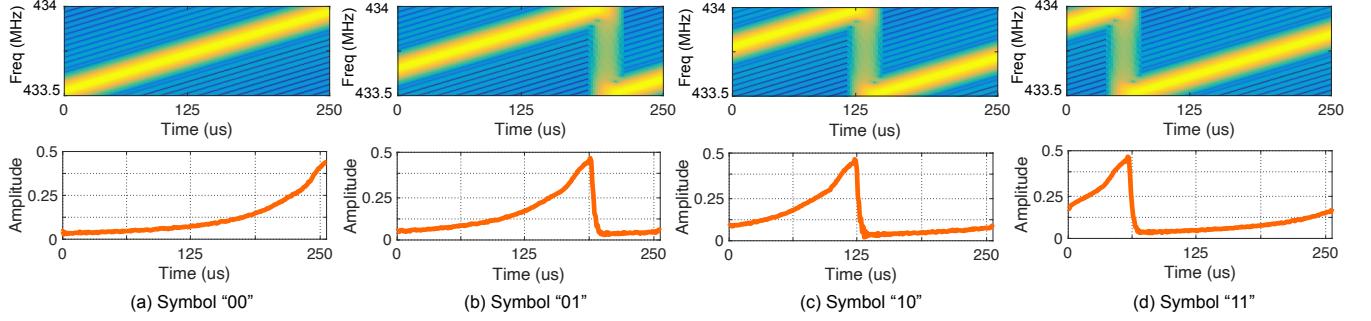


Figure 6: The input (top) and output (bottom) signals of the SAW filter. The amplitude of the output signal scales with the frequency of the input signal. They reach the maximal value simultaneously.

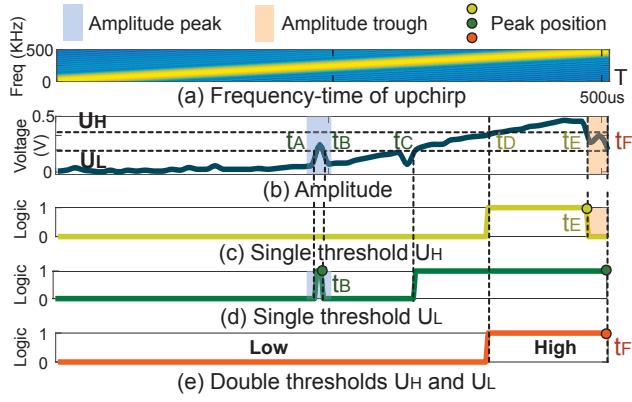


Figure 7: Comparing the output of different voltage comparators. (a): the incident LoRa chirp. (b): the output of an envelope detector. (c)-(d): the output of the single-threshold based comparator that uses  $U_H$  or  $U_L$  as the cut-off amplitude. (e) the output of the double-threshold based comparator that uses  $U_H$  and  $U_L$  simultaneously as the cut-off amplitudes.

filter and the noise added by the envelope detector. Next, we introduce super Saiyan to improve the sensitivity.

### 3 Super Saiyan

Super Saiyan takes the following actions to consistently improve the demodulation sensitivity: *i*) improving the SNR of baseband chirp signals with a cyclic-frequency shifting circuit, and *ii*) improving the sensitivity of demodulator with correlation.

#### 3.1 Cyclic-frequency Shifting

**Understanding the principle of envelope detector.** The envelope detector has been widely adopted by low-power RF devices to down-convert the incident signal. However, due to the inherent non-linearity caused by the squaring operation of

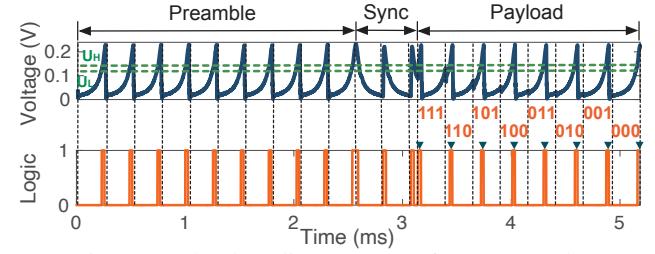


Figure 8: The decoding process of a LoRa packet

CMOS devices [27], both the targeted signal (*i.e.*, feedback signals from the LoRa access point) and the RF noises will be down-converted to the baseband. Consequently, the targeted signal becomes even weaker after down-conversion. We explicate this phenomenon using the following example. Let  $S_{in}$  be the incident signal:  $S_{in} = S_t + S_n$ , where  $S_t$  and  $S_n$  denote the targeted signal and RF noises, respectively. The output signal  $S_{out}$  of this envelope detector can be represented by:

$$\begin{aligned} S_{out} &= kS_{in}^2 = k(S_t + S_n)^2 \\ &= kS_t^2 + 2kS_t \cdot S_n + kS_n^2 \end{aligned} \quad (4)$$

where  $k$  represents the attenuation factor. The first term  $S_t^2$  on the right side of this equation manifests that the targeted signal  $S_t$  is shifted to the baseband through self-mixing. The second and the third terms both indicate the RF noises are shifted to the baseband after mixed with the targeted signal and the noises themselves, respectively, causing strong interference on the baseband.

**Cyclic-frequency shifting.** In Saiyan we design a low-power circuit to mitigate the SNR loss brought by the envelope detector. The circuit is realized by two RF mixers and two clock signals. Its operation is detailed as follows.

- **Step 1.** The micro-controller first generates a clock signal  $CLK_{in}(\Delta f)$  and mixes it with the incident signal  $S(F)$ , resulting in two sideband signals  $S(F - \Delta f)$  and  $S(F + \Delta f)$ , as shown in Figure 9(a)-(b). The sideband signals and the

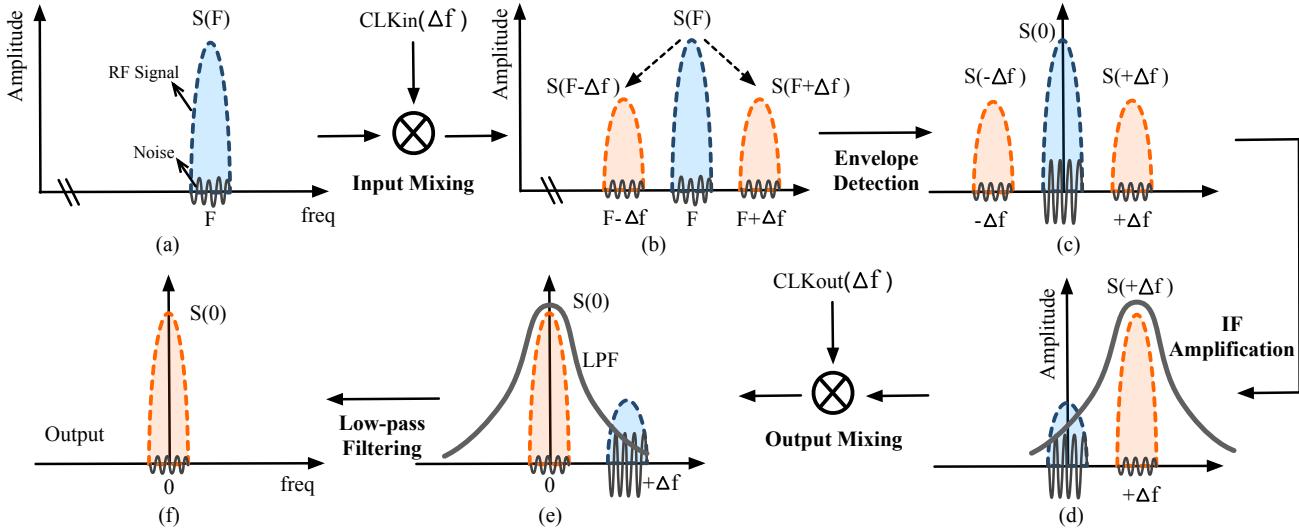


Figure 9: The illustration of the cyclic-frequency shifting. (a) The input signal  $S(F)$ . (b)  $S(F)$  is first mixed with the clock signal, resulting in two sideband signals  $S(F - \Delta f)$  and  $S(F + \Delta f)$ . (c) The envelope detector extracts the envelope of those three signals and down-converts them to the the baseband. (d) The IF amplifier boosts the power of  $S(\Delta f)$  and attenuates the power at other frequency bands. (e) The desired signal  $S(\Delta f)$  with significantly lower noises is shifted back to the baseband. (f) The output signal  $S(0)$ .

incident signal are then down-converted to the intermediate frequency (IF) band (denoted by  $S(-\Delta f)$  and  $S(\Delta f)$ ) and the baseband (denoted by  $S(0)$ ) respectively with an envelope detector (Figure 9(c)).

- **Step 2.** Since RF noises are not down-converted to the IF band by the envelope detector, we amplify the unpolluted IF signal  $S(\Delta f)$  using a low-power IF amplifier. The frequency selectivity of this IF amplifier filters out signals at other frequencies (e.g.,  $S(0)$ ), as shown in Figure 9(d).
- **Step 3.** The power-amplified IF signal  $S(\Delta f)$ , mixed with another clock signal  $CLK_{out}(\Delta f)$ , is shifted back to the baseband, as shown in Figure 9(e). At the same time, the noisy baseband signal  $S(0)$  will be shifted to the IF band and then filtered by a low-pass filter (Figure 9(f)).

In a nutshell, this circuit first moves the targeted signal to an intermittent frequency band (step 1) to avoid the RF noise contamination introduced by the envelope detector. This also leaves us an opportunity to remedy the SNR loss in down-conversion (step 2). Finally, the targeted signal is moved back to the baseband for demodulation. At the same time the DC offset, flicker and other noises are moved to the IF band and removed by a low-pass filter (step 3).

Figure 10 shows the spectrums before and after feeding the chirp signal into the cyclic frequency shifting circuit. Evidently, both the inband and out-of-band RF noises have been cleaned by the circuit, ensuring the decodability of chirp signals. Our quantitative measurement shows that the cyclic-frequency shifting circuit brings in 11 dB SNR gain.

**Clock signal generation.** The above circuit design relies on two clock signals  $CLK_{in}(\Delta f)$  and  $CLK_{out}(\Delta f)$ . To save power,

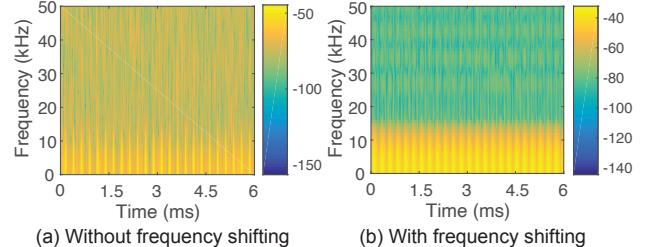


Figure 10: The spectrum of an incident LoRa signal when being down-converted into the baseband with an envelope detector. (a) Without cyclic-frequency shifting. (b) With cyclic-frequency shifting. The LoRa signal contains 24 LoRa chirps ( $BW=500\text{kHz}$ ,  $SF=8$ ).

we program the MCU to generate  $CLK_{in}(\Delta f)$  signal and then leverage a delay line to copy  $CLK_{in}(\Delta f)$  as  $CLK_{out}(\Delta f)$ :

$$CLK_{out}(\Delta f) = CLK_{in}(\Delta f + \Delta\phi) \quad (5)$$

where  $\Delta\phi$  is the phase shift caused by the delay line. We tune the length of this delay line to ensure  $\cos(\Delta\phi) \approx 1$  so that  $CLK_{out}(\Delta f)$  equals  $CLK_{in}(\Delta f)$ .

**Circuit integration.** We integrate this cyclic-frequency shifting circuit into the envelope detector. Figure 11 shows the schematic of this design. It consists of an input mixer, an output mixer, an envelope detector, an IF amplifier, a low-pass filter (LPF), an oscillator, and a transmission line. Specifically, The base clock signal is provided by a micro-power precision oscillator LTC6907 [11]. A low-power transistor 2N222 [8] is adopted as the IF amplifier.

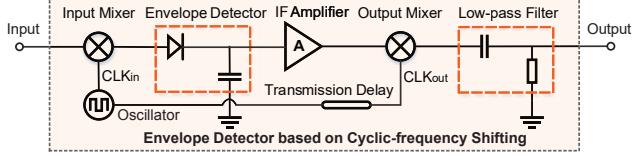


Figure 11: The schematic of cyclic-frequency shifting.

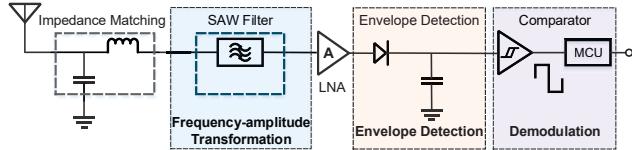


Figure 12: The high-level circuit schematic of Saiyan.

### 3.2 Correlation

While the above cyclic-frequency shifting circuit successfully improves the SNR of the incident signal, the demodulation accuracy still suffers degradation when the incident signal is too weak, *e.g.*, close to the noise floor. We thus employ correlation — a mainstream approach that has been largely adopted for packet detection to further improve the demodulation sensitivity. It operates by correlating signal samples with a local chirp template. An energy peak shows up as long as the incident signal matches the template. The receiver then tracks the energy peak and demodulates the incident signal.

## 4 Implementation

We describe the system implementation in this section.

### 4.1 Backscatter Tag

We implement Saiyan on a  $25\text{ mm} \times 20\text{ mm}$  two-layer PCB using commercial off-the-shelf analog components and an ultra-low power Apollo2 ( $10\text{ }\mu\text{A/MHz}$ ) [13] MCU. We determine its size through a mixed analytical and experimental approach, striking a balance between the form factor and circuit interference. Figure 13 shows the hardware prototype. Saiyan functions with an omni-directional antenna [2] with  $3\text{ dBi}$  gain.

**Architecture and workflow.** Figure 12 shows the architecture of Saiyan. The incident signal passes through a passive SAW chip B39431B3790Z810 [1] and is transformed into an amplitude-modulated signal. We place a common-gate low-noise amplifier (CGLNA) [17] between the SAW filter and the customized envelope detector to amplify the transformed signal. The amplified signal is then down-converted to the baseband through the envelope detector. Finally, a low-power voltage comparator NCS2202 [9] is leveraged to quantize the output signal from the envelope detector.

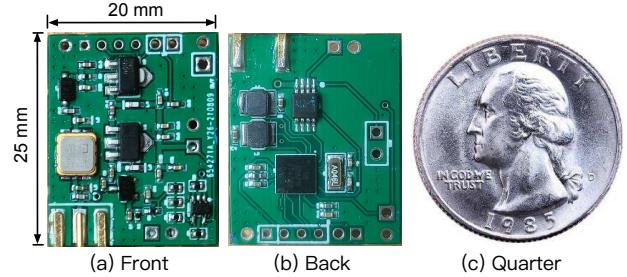


Figure 13: The hardware prototype of Saiyan. The quarter next to Saiyan demonstrates the form factor.

**Plug-and-play.** As an ultra-low-power peripheral, Saiyan can be integrated into the existing long-range LoRa backscatter systems [23, 40] with ignorable engineering efforts. Taking PLoRa [40] as an example, we replace its packet detection module with Saiyan and retained all the remaining functional units the same. This simple replacement allows PLoRa tag to demodulate the feedback signals while retaining the modulation capability at the same time. On the software side, we replicate the sampling rate control logic to facilitate the demodulation.

**Power management.** The energy harvester on Saiyan comprises of a palm-sized photovoltaic panel and a high-efficiency step-up DC/DC converter LTC3105 [3]. It generates  $1\text{ mW}$  power every  $25.4$  seconds in a bright day. The power management module provides a constant  $3.3\text{V}$  output voltage to the MCU. The power consumption of this power management module in working mode is approximately  $24\text{ }\mu\text{W}$ .

**Determining the voltage thresholds  $U_H$  and  $U_L$ .** Ideally,  $U_H$  should be slightly lower than the peak amplitude of the input signal  $A_{max}$ . Let  $G$  be the gap between  $A_{max}$  and the voltage threshold  $U_H$ . We have:  $G = 20\lg(A_{max}/U_H)$ . Thus,  $U_H$  can be estimated on the basis of the following equation:  $U_H = A_{max}/10^{\frac{G}{20}}$ . The threshold voltage  $U_L$  is set to  $U_H - U_F$ , where  $U_F$  represents the amplitude of the envelope detector's output. The thresholds  $U_H$  and  $U_L$  are tuned by two adjustable on-board resistors. In practice, considering that  $A_{max}$  and  $U_F$  both vary with the link distance, we measure these two values offline under different link distance settings and store a mapping table on each tag to facilitate the configuration of  $U_H$  and  $U_L$ . To alleviate this manual configuration overhead, one could leverage an Automatic Gain Control (AGC) [42, 43] to adapt the power gain automatically. We leave it for future work.

### 4.2 LoRa Transmitter and Receiver

**LoRa transmitter.** We use two types of LoRa transmitters in the evaluation: *i*) a LoRa transmitter implemented on a software-defined radio platform USRP N210, and *ii*) a com-



Figure 14: Outdoor experiment field.

mercial off-the-shelf LoRa node equipped with a Semtech SX1276RF1JAS [7] chip. Both platforms use a single omnidirectional antenna with 3 dBi gain. The transmission power is set to 20 dBm.

**LoRa receiver.** The LoRa receiver is implemented on a software-defined radio platform USRP N210. We set the sampling rate to 10 MHz, thereby allowing the receiver to monitor six LoRa channels simultaneously.

### 4.3 ASIC Simulation

We simulate the Application Specific Integrated Circuit (ASIC) of Saiyan based on the TSMC 65-nm CMOS process. The active area of on-chip Integrated Circuits (IC) is  $0.217 \text{ mm}^2$ . The ASIC simulation shows that the power consumption of Saiyan is  $93.2 \mu\text{W}$ . Specifically, the power consumption of LNA, oscillator, and digital circuit is  $68.4 \mu\text{W}$  and  $22.8 \mu\text{W}$ , and  $2 \mu\text{W}$ , respectively. Once Saiyan demodulated the feedback signals, the MCU starts preparing data for packet re-transmissions, which consumes extremely low power (*i.e.*, the power consumption of the ultra-low power Apollo2 [13] in Saiyan is merely  $19.6 \mu\text{W}$ ).

### 4.4 MAC-layer for Multi-tag Coexistence

We briefly discuss MAC-layer in this section. The downlink packets can be divided into three groups: unicast packet, multicast packet, and broadcast packet. In unicast, all backscatter tags within the radio range will receive and demodulate this unicast packet from the access point. However, only the targeted tag will response (*e.g.*, re-transmit the lost packet). Hence, no collision occurs. However, in multicast and broadcast, collision happens as long as more than one backscatter tag replies at the same time. For instance, the access point sends a downlink packet (*e.g.*, turn off the humidity sensor), while multiple tags acknowledge the reception of this downlink packet simultaneously. In this case, the access point can leverage slotted ALOHA [22] protocol to coordinate tags and minimize collisions. We take Figure 15 as an example to illustrate the MAC-layer operation. Suppose three tags are sending an acknowledgement to the access point to confirm the reception of a downlink packet. Each tag will randomly select a time slot and store it in its local counter. Upon the

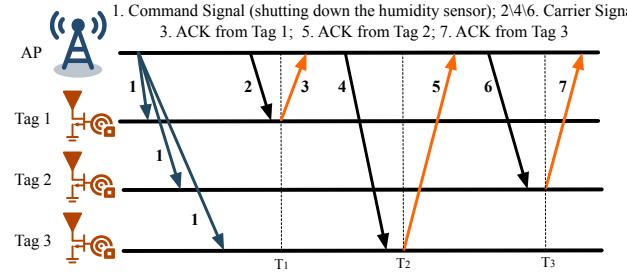


Figure 15: The illustration of MAC-layer operations in Saiyan. Each tag randomly selects a slot to transmit. The access point (AP) signals the beginning of each slot with a carrier signal.

detection of a carrier signal from the access point, each tag decreases the slot number by one and transmits as soon as the slot number goes zero. The randomness in slot selection minimizes the interference among tags.

## 5 Evaluation

In this section, we present the evaluation results of field studies (§5.1) and micro-benchmarks (§5.2). Two case studies follow (§5.3). Unless otherwise posted, the transmitter and the receiver are collocated throughout the experiment.

**Setups.** The LoRa transmitter works on the 433.5 MHz frequency band. The spreading factor and the bandwidth are set to 7 and 500 KHz, respectively. The payload of each LoRa packet contains 32 chirp symbols. In each experiment, we let the transmitter transmit 1,000 LoRa packets and then repeat the experiment for 100 times to ensure the statistical validity. We adopt *BER*, *throughput*, and *demodulation range* as the key metrics to assess Saiyan’s performance.

- **BER** refers to the ratio of error bits to the total number of bits received by Saiyan.
- **Throughput** measures the amount of received data correctly decoded by Saiyan within one second.
- **Demodulation range** refers to the maximum distance between the tag and the LoRa transmitter when the BER is maintained below 1%.

### 5.1 Field Studies

We conduct field studies both indoors and outdoors to assess the impact of coding rate (CR), spreading factor (SF), and bandwidth (BW) on BER, demodulation range, and throughput, which are three key evaluation metrics.

#### 5.1.1 Outdoor experiments

**Impact of coding rate.** We place a Saiyan tag 10 m, 20 m, 50 m, 100 m, and 150 m away from a LoRa transmitter. Under

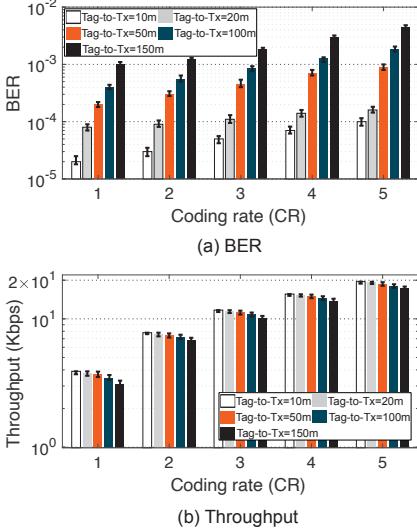


Figure 16: BER and throughput in different coding rate settings.

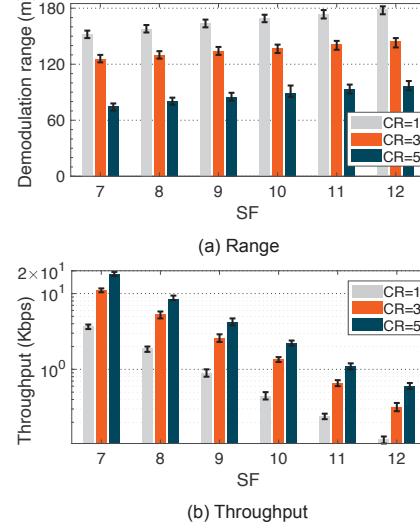


Figure 17: Demodulation range and throughput in different SF settings.

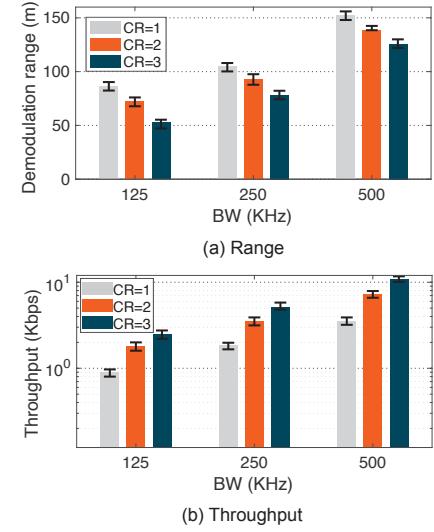


Figure 18: Demodulation range and throughput in different BW settings.

each distance setting, we vary the coding rate of LoRa signals and measure BER and throughput. We have three observations based on the results shown in Figure 16.

First, the BER grows with the coding rate. As shown in Figure 16(a), the BER under the highest coding rate setting (*i.e.*, 5) is 2.4–5.2× higher than the BER under the lowest coding rate setting (*i.e.*, 1) across all different Tx-to-tag distances. For instance, when the Tx-to-tag distance is 100 m, Saiyan achieves a BER of 1.85% under the highest coding rate setting. The BER then drops to 0.4% under the same Tx-to-tag distance setting when we change the coding rate to 1. This is expected since the Saiyan tag has to differentiate more types of LoRa chirps under the high coding rate setting.

Second, the throughput grows linearly with the coding rate (Figure 16(b)). For example, when the Tx-to-tag distance is 100 m, the achievable throughput at CR=5 (18.12 Kbps) is around 5.1× higher than the throughput at a coding rate of 1 (3.57 Kbps).

Third, both the BER and the throughput get exacerbated with the growing Tx-to-tag distance. For instance, when CR=5, the BER grows dramatically from 0.1% to 4.4% as the Tx-to-tag distance grows from 10 m to 150 m. The throughput, on the other hand, declines from 19.6 Kbps to 17.2 Kbps. This is expected since Saiyan relies on the signal power to demodulate the incident LoRa signal.

**Impact of spreading factor.** Next, we vary the spreading factor from 7 to 12 and assess Saiyan’s demodulation range and throughput under each setting. The results are shown in Figure 17. We observe that the demodulation range grows with the increasing spreading factor. The throughput, on the contrary, declines with the increasing spreading factor. For instance, the demodulation range under the highest spreading factor setting (*i.e.*, SF=12) is 1.1–1.3× longer than the

demodulation range under the lowest spreading factor setting (*i.e.*, SF=7) across three different coding rate settings. The throughput drops by 30.3–35.1× as we decrease the SF from 12 to 7. This is expected since a higher spreading factor enhances the anti-noise capability of LoRa signals; thus the demodulation range grows. On the other hand, the symbol time grows with the increasing spreading factor, resulting in a lower throughput.

**Impact of bandwidth.** We set the spreading factor to 7 and assess the impact of LoRa bandwidth on the demodulation range and throughput. The results are shown in Figure 18. We observe that the demodulation range and the throughput both grow with the LoRa bandwidth. Specifically, given the coding rate of 2, the demodulation range grows from 72.2 m to 138.6 m as we increase the bandwidth from 125 kHz to 500 kHz. On the other hand, since the LoRa symbol time is inversely proportional to the bandwidth, we observe the throughput drops around 4× from 7.2 Kbps to 1.8 Kbps as we decrease the bandwidth from 500 kHz to 125 kHz.

### 5.1.2 Indoor experiments

We repeat the above experiments in an indoor environment where the LoRa signals have to penetrate one or multiple concrete walls to arrive at the backscatter tag.

**Penetrating one concrete wall.** Similar to the trend shown in the outdoor scenario, the throughput measured in the indoor scenario also grows with the increase of the coding rate (Figure 19). For example, the throughput grows from 3.7 Kbps to 18.7 Kbps when the coding rate varies from 1 to 5. The demodulation range, on the other hand, declines from 48.8 m to 26.2 m as we increase the coding rate from 1 to 5.

**Penetrating two concrete walls.** The LoRa signal experi-

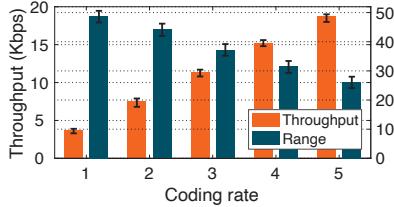


Figure 19: Throughput and downlink range in the presence of one concrete wall.

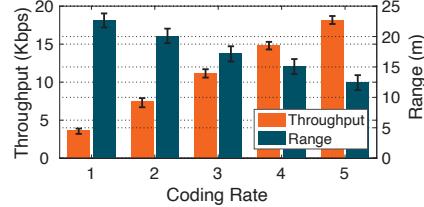


Figure 20: Throughput and downlink range in the presence of two concrete walls.

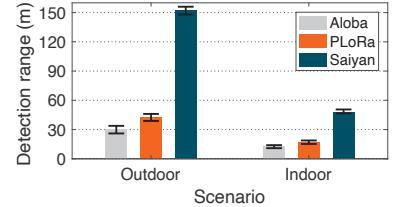


Figure 21: Comparison of Saiyan, Aloba, and PLoRa on the detection range.

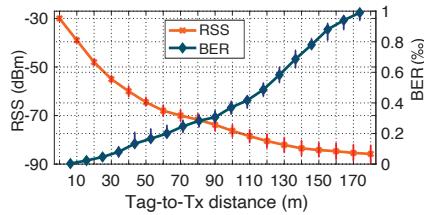


Figure 22: RSS and BER over distance.

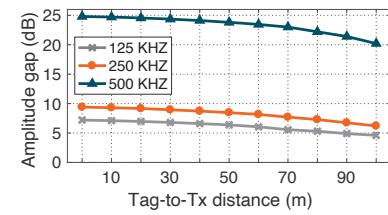


Figure 23: The amplitude gap of the output signal after SAW filter

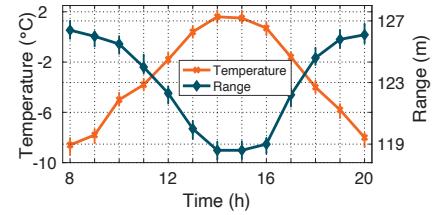


Figure 24: Demodulation range under different temperatures

ences stronger attenuation when penetrating two concrete walls. Accordingly, we observe the demodulation range and the throughput decline by  $2.21\text{-}2.09\times$  and  $1.01\text{-}1.05\times$  compared to those under the single concrete wall settings (Figure 20).

### 5.1.3 Comparison with state-of-the-art systems

We further compare Saiyan with two state-of-the-art systems, namely, Aloba [23] and PLoRa [40] in both outdoor and indoor environments. PLoRa operates cross-correlation to detect a LoRa packet. Aloba feeds the incident signal into a moving average filter and then leverages the unique RSSI pattern of the LoRa preamble to detect a LoRa packet. They both cannot demodulate the payload. Therefore, we compare them with Saiyan in terms of the packet detection range.

Figure 21 shows the experiment result. In the outdoor line-of-sight settings, Saiyan achieves a packet detection range of 148.6 m, outperforming ALoBa (30.6m) and PLoRa (42.4m) by  $4.52\times$  and  $3.26\times$ , respectively. In an indoor none-line-of-sight environment, although the packet detection range of Saiyan declines to 44.2 m, it still outperforms Aloba (12.4 m) and PLoRa (16.8 m) by  $3.56\times$  and  $2.63\times$ , respectively.

## 5.2 Micro-benchmarks

To better understand the performance of each design component in Saiyan, we run micro-benchmarks to assess the receiver sensitivity, the SAW filter, as well as the power consumption and the system cost.

### 5.2.1 Receiver sensitivity

We define the receiver sensitivity as the minimum Received Signal Strength (RSS) of an incident signal that can be detected by Saiyan. To assess the receiver sensitivity, we measure the BER and the Received Signal Strength (RSS) under different Tx-to-tag distance settings. As expected, the BER grows gradually with the increase of the Tx-to-tag distance, as shown in Figure 22. Nevertheless, Saiyan can still detect the incident signal when the tag is 180 m away from the transmitter. As we increase the tag-to-Tx further, the signal strength is too weak to be detected by Saiyan. The above experiment demonstrates an -85.8 dBm receiver sensitivity, outperforming the conventional envelope detector by 30 dBm [27].

### 5.2.2 Performance of the SAW filter

**Frequency-amplitude response.** Saiyan relies on the frequency-amplitude response of the SAW filter to demodulate LoRa signals. A sharp frequency-amplitude response (*e.g.*, a small frequency variation leads to a large amplitude gap) is desirable as it allows the Saiyan tag to detect the minute frequency variation on the incident signal.

We feed LoRa signals with different bandwidth into the SAW filter and measure the amplitude variation of the output signal. The results are shown in Figure 23. As expected, the amplitude variation of the output signal (*a.k.a.*, amplitude gap) tends to be less significant with the decreasing chirp bandwidth. For instance, when the Tx-to-tag distance is 10 m, the amplitude gap drops from 24.7 dBm to 9.3 dBm, and further to 7.1 dBm as we decrease the chirp bandwidth from 500 KHz to 250 KHz, and further to 125 KHz, respectively. A similar trend shows up as we increase the Tx-to-tag distance.

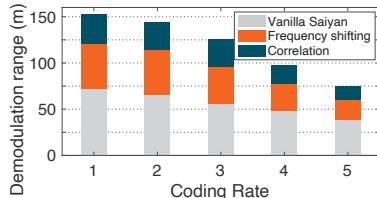


Figure 25: Ablation study of Saiyan.

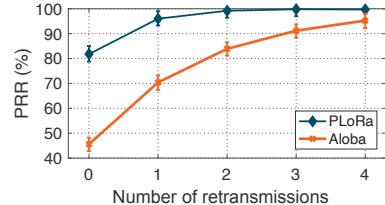


Figure 26: PRR in different settings.

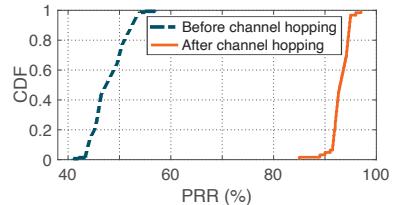


Figure 27: PRR lifts with Saiyan.

Table 2: Energy consumption (under 1% duty cycling) and cost of each component in Saiyan tag.

Component	SAW Filter	LNA	OSC Clock	Envelope Detector	Comparator	MCU	Total
Energy ( $\mu$ W)	0	248.5	86.8	0	14.45	19.6	369.4
Cost (\$)	3.87	4.15	1.25	1.20	1.26	15.43	27.2

For instance, when the signal bandwidth is 500 KHz, the amplitude gap of the output signal drops from 24.7 dBm to 20.2 dBm as the Tx-to-tag distance increases from 10 m to 100 m.

**The impact of temperature.** The frequency selectivity of the SAW filter is affected by the ambient temperature [36]. We thus run an experiment to assess the impact of temperature on the demodulation range. The experiment is conducted outdoors on a sunny day from 8 a.m. to 8 p.m.. Figure 24 shows the result. We observe that the demodulation range in general is insensitive to the temperature. For instance, when the temperature rises from the lowest -8.6 °C at 8 a.m. to the highest 1.6 °C at 2 p.m., the demodulation range merely drops from 126.4 m to 118.6 m.

### 5.2.3 Ablation study

We conduct an ablation study to assess the effectiveness of each design component of Saiyan. In this experiment, we set the spreading factor and the bandwidth to 7 and 500 KHz respectively and measure the maximum demodulation range under different coding rate settings. The results are shown in Figure 25. We find that the vanilla Saiyan achieves a relatively short demodulation range (38.4 m—72.6 m) across five different coding rate settings. The demodulation range then grows by 1.56×–1.73× with the help of the cyclic frequency shifting module. The cross-correlation further improves the demodulation range by 1.94×–2.25×.

### 5.2.4 Power consumption & system cost

Table 2 summarizes the power consumption (under 1% duty cycling as in LoRa [22]) and cost of each component in Saiyan. Among these hardware components, the most power-hungry parts are LNA and oscillator (OSC) clock, which account for 67.3% and 23.5% of the total power consumption, respectively. As we demonstrate in §4.3, the power consumption can be effectively reduced by 74.8% when implementing Saiyan

on ASIC. The hardware cost of Saiyan, on the other hand, is around 27.2 USD, which can be also reduced sharply after ASIC fabrication.

## 5.3 Case Studies

Next, we run two real-world case studies to showcase packet re-transmission (§5.3.1) and frequency hopping (§5.3.2).

### 5.3.1 Packet re-transmission through the ACK mechanism

**Setups.** We integrate Saiyan into PLoRa and Aloba tags, which allows the tags to demodulate the feedback signals from the receiver and make an immediate packet re-transmission if needed. The link distance is set to 100 m.

**Results.** As shown in Figure 26, PLoRa and Aloba achieve 81.8% and 45.6% packet reception ratio (PRR) without packet re-transmission. The PRR of Aloba grows drastically from 45.6% to 70.1% when the Aloba tag is allowed to re-transmit the lost packet only once. The PRR then grows to 83.3% and further to 95.5% when the Aloba tag re-transmits the lost packet twice and three times, respectively. The PRR of PLoRa shows the similar trend. These results demonstrate that Saiyan effectively improves the packet reception ratio for long-range LoRa backscatter systems.

### 5.3.2 Interference avoidance through channel hopping

As an ultra-low-power tag working on the ISM band, both PLoRa and Aloba are likely to bear strong in-band interference from other legacy RF devices working on the same band. We show that with Saiyan, these backscatter tags can demodulate the feedback signals from the receiver and switch to other channels to avoid interference.

**Setups.** We use PLoRa to demonstrate the feasibility of channel hopping. The PLoRa tag communicates with the receiver at the 434 MHz frequency band. It switches to the 434.5 MHz frequency band upon detecting the feedback signal from the receiver. We put a software-defined radio three meters away from the receiver to jam the channel at the 433 MHz frequency band.

**Results.** Figure 27 shows the CDF of PRR before and after the channel hopping. We can see the PRR is very low when the USRP jams the channel (dotted line). As the receiver

initiates a channel hopping command to the backscatter tag, we witness a significant lift on the PRR. In particular, the median PRR grows from 47% to 92% once PLoRa switches to another channel. This result clearly demonstrates that Saiyan can support better channel utilization through remote control.

## 6 Related Work

We review research topics relevant to Saiyan in this section.

**RFID system.** A passive RFID tag modulates sinusoidal tone from an RFID reader to transmit data [52, 58]. It can also demodulate amplitude-modulated (AM) signals from a nearby RFID reader [16, 26, 51, 53]. Specifically, the RFID tag down-converts the incident signal to the baseband and accumulates the signal power through an integrator circuit. Subsequently, it compares the accumulated power to a threshold to demodulate incident signals. Saiyan differs from passive RFID tags in two aspects. First, Saiyan demodulates frequency-modulated signal as opposed to amplitude-modulated signal. Second, Saiyan is designed for long-range backscatter systems whereas the passive RFID tag functions within only a few meters.

**Ambient backscatter systems.** Ambient backscatter systems empower backscatter tags to take the ambient wireless traffic as the carrier signals [14, 15, 18, 20, 23, 29–33, 35, 37, 39, 40, 47–50, 55–57, 60]. For example, WiFi backscatter [33] reuses WiFi signals as the carrier, thereby allowing for the backscatter tag to communicate with a commercial WiFi receiver. Interscatter [29] enables backscatter tags to modulate Bluetooth signals into WiFi signals. LoRa backscatter [47] allows backscatter tags to communicate over long distances by taking advantage of the noise resilience of LoRa symbols. These pioneer works have remarkably improved the throughput and the communication range of backscatter systems. Some recent works [37, 44, 55, 56, 59, 60] support a few types of downlink functionalities such as carrier sensing [37, 44, 55, 56, 59, 60] and packet detection [23, 40] at the packet level. For example, WiFi backscatter [33], Passive-WiFi [34], Interscatter [29], LoRa backscatter [47], and Netscatter [24] use the presence and absence of carrier packets to convey downlink data. However, they cannot demodulate downlink packets at the symbol level, particularly under long-range settings. Saiyan can serve as an important building block to the existing long-range backscatter systems, where the on-demand retransmission is needed due to the drastic packet loss.

**Low-power demodulator.** With the growth of low-power IoT market, the research community has shifted the focus to the design and implementation of low-power RF receivers, *e.g.*, by replacing the active components with their passive counterparts, or by offloading the power-intensive functions to external devices. Ensworth et al. [19] proposed a 2.4 GHz low-power BLE receiver that offloads the RF local oscillator to an external device. Carlos et al. [41] proposed a low-power 802.15.4 receiver that could demodulate phase-modulated

ZigBee signals at orders of magnitude lower power consumption compared with the standard 802.15.4 receiver. However, the working range of this low-power receiver is limited to tens of centimeters, which sets a strong barrier towards the practical deployment. Turbo charging [39] designs a multi-antenna cancellation circuit to facilitate the signal demodulation on backscatter tags. Similarly, full-duplex backscatter [38] enables a backscatter tag to demodulate the instantaneous feedback signal from another backscatter tag. Saiyan differs from these systems in two aspects. First, Saiyan is designed for demodulating frequency-modulated signals as opposed to phase or amplitude modulated signals. Second, Saiyan can support up to 180 m demodulation range, whereas all the aforementioned systems function within only tens of centimeters.

**SAW filter.** The SAW filter has been widely adopted by wireless communication systems such as telecommunications [25], radar [54], and aerospace communications [45], *etc.* These systems leverage the low-distortion and minimal passband variation of the SAW filter to filter out noise and interference signals. Furthermore, medical devices transform a SAW filter into a sensor for in-situ detection (*e.g.*, detecting chemical gas concentration) [21, 28]. Different from all the above applications, Saiyan exploits the sharp frequency response of the SAW filter to demodulate frequency-modulated signal.

## 7 Conclusion

We have presented the design, implementation, and evaluation of Saiyan, the first-of-its-kind low-power demodulator for LoRa backscatter systems. Saiyan allows LoRa backscatter tags to demodulate the command or feedback signals from a remote access point that is hundreds of meters away. With such capability, the backscatter tag can realize a plethora of networking functionalities, such as packet re-transmission, channel hopping, and rate adaptation. Field study shows that Saiyan outperforms state-of-the-art systems by 3.5–5× in terms of demodulation range. The ASIC simulation shows that the power consumption of Saiyan is around 93.2  $\mu\text{W}$ .

## Acknowledgment

We thank our shepherd Fadel Adib and the anonymous reviewers for their insightful comments. We are also very grateful to Dr. Lu Li from University of Electronic Science and Technology of China for his constructive feedback. This work is supported in part by National Key R&D Program of China No. 2017YFB1003000, National Science Fund of China under grant No. 61772306, and the R&D Project of Key Core Technology and Generic Technology in Shanxi Province (2020XXX007).

## References

- [1] B39431-B3790-Z810 by Qualcomm-RF360 SAW filters. [Webpage](#).
- [2] 3 dBi omni-directional antenna in 433 MHz. [Webpage](#).
- [3] Energy harvesting chip LTC3105. [Webpage](#).
- [4] Introduction to SAW filter theory & design techniques. [Webpage](#).
- [5] LoRa Alliance. [Webpage](#).
- [6] LoRa receiver. [Webpage](#).
- [7] LoRa transceivers SX1276RF1JAS in 433 MHz. [Webpage](#).
- [8] Low-power amplifier transistors 2N222. [Webpage](#).
- [9] Low-power comparator NCS2202. [Webpage](#).
- [10] Realization of Differential Circuit. [Webpage](#).
- [11] Silicon oscillators LTC6907. [Webpage](#).
- [12] The RF in RFID. [Webpage](#).
- [13] Ultra-low power microcontroller Apollo2 Blue. [Webpage](#).
- [14] Mohamed R. Abdelhamid, Ruicong Chen, Joonhyuk Cho, Anantha P. Chandrakasan, and Fadel Adib. Self-reconfigurable micro-implants for cross-tissue wireless and batteryless connectivity. In *Proceedings of ACM MobiCom, Virtual event, September 21-25, 2020*.
- [15] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. BackFi: High throughput WiFi backscatter. In *Proceedings of ACM SIGCOMM, Budapest, Hungary, August 20-25, 2018*.
- [16] Binbin Chen, Ziling Zhou, and Haifeng Yu. Understanding RFID counting protocols. In *Proceedings of ACM MobiCom, Miami, Florida, USA, September 20-October 4, 2013*.
- [17] Chunyuan Chiu, Zhencheng Zhang, and Tsung Hsien Lin. Design of a 0.6-V, 429-MHz FSK transceiver using Q-enhanced and direct power transfer techniques in 90-nm CMOS. *IEEE Journal of Solid-State Circuit*, 55(1):3024–3035, 2020.
- [18] Farzan Dehbashi, Ali Abedi, Tim Brecht, and Omid Abari. Verification: Can WiFi backscatter replace RFID? In *Proceedings of ACM MobiCom, New Orleans, USA, October 25-29, 2021*.
- [19] Joshua F. Ensworth, Alexander T. Hoang, and Matthew S. Reynolds. A low power 2.4 GHz super-heterodyne receiver architecture with external LO for wirelessly powered backscatter tags and sensors. In *Proceedings of IEEE RFID, Phoenix, AZ, May 9-11, 2017*.
- [20] Joshua F. Ensworth and Matthew S. Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with Bluetooth 4.0 Low Energy (BLE) devices. In *Proceedings of IEEE RFID, San Diego, CA, USA, April 15-17, 2015*.
- [21] Fahim, Mainuddin, U. Mittal, Jitender Kumar, A. T. Nimal, and M. U. Sharma. Single chip readout electronics for SAW based gas sensor systems. In *Proceedings of IEEE SENSORS, Glasgow, UK, October 29- November 1, 2012*.
- [22] Amalinda Gamage, Jansen Christian Liando, Chaojie Gu, Tan Rui, and Mo Li. LMAC: Efficient carrier-sense multiple access for LoRa. In *Proceedings of ACM MobiCom, Virtual event, September 21-25, 2020*.
- [23] Xiuzhen Guo, Longfei Shangguan, Yuan He, Jia Zhang, Haotian Jiang, Awais Ahmad Siddiqi, and Yunhao Liu. Aloba: Rethinking on-off keying modulation for ambient LoRa backscatter. In *Proceedings of ACM SenSys, Virtual event, November 16-19, 2020*.
- [24] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *Proceedings of USENIX NSDI, Santa Clara, CA, March 16-18, 2016*.
- [25] Tzuhsuan Hsu, Fengchieh Su, Kuanju Tseng, and Minghuang Li. Low loss and wideband surface acoustic wave devices in thin film Lithium Niobate on Insulator (LNOI) platform. In *Proceedings of 34th International Conference on Micro Electro Mechanical Systems (MEMS), Gainesville, FL, USA, January 25-29, 2021*.
- [26] Pan Hu, Pengyu Zhang, and Deepak Ganesan. Laissez-faire: Fully asymmetric backscatter communication. In *Proceedings of ACM SIGCOMM, London, United Kingdom, August 17-21, 2015*.
- [27] Xiongchuan Huang, Guido Dolmans, Harmke de Groot, and John R. Long. Noise and sensitivity in RF envelope detection receivers. *IEEE Transactions on Circuit and Systems*, 60(10):1549–7747, 2013.
- [28] Tarikul Islam, Upendra Mittal, A T Nimal, and M U Sharma. Surface Acoustic Wave (SAW) vapour sensor using 70 MHz SAW oscillator. In *Proceedings of 6th International Conference on Sensing Technology (ICST), Kolkata, India, December 18-21, 2012*.

- [29] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of ACM SIGCOMM, Salvador, Brazil, August 22-26, 2016*.
- [30] Junsu Jang and Fadel Adib. Underwater backscatter networking. In *Proceedings of ACM SIGCOMM, Beijing, China, August 19-24, 2019*.
- [31] Zhang Jianhui, Zheng Siwen, Zhang Tianhao, Wang Mengmeng, and Li Zhi. Charge-aware duty cycling methods for wireless systems under energy harvesting heterogeneity. *ACM Transactions on Sensor Networks*, 16(15):1–23, 2020.
- [32] Mohamad Katanbaf, Anthony Weinand, and Vamsi Talla. Simplifying backscatter deployment: Full-duplex LoRa backscatter. In *Proceedings of USENIX NSDI, virtual, April 12-14, 2021*.
- [33] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. WiFi backscatter: Internet connectivity for RF-powered devices. In *Proceedings of ACM SIGCOMM, Chicago, USA, August 17-22, 2014*.
- [34] Bryce Kellogg, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakota. Passive WiFi: Bringing low power to WiFi transmissions. In *Proceedings of USENIX NSDI, Santa Clara, CA, March 16-18, 2018*.
- [35] Songfan Li, Chong Zhang, Yihang Song, Hui Zheng, Lu Liu, Li Lu, and Mo Li. Internet-of-microchips: Direct radio-to-bus communication with SPI backscatter. In *Proceedings of ACM MobiCom, Virtual event, September 21-25, 2020*.
- [36] Alexei N. Liashuk, Sergey A. Zavyalov, Aleksandr N. Lepetaev, Anatoliy V. Kosykh, and Igor V. Khomenko. Digitally temperature compensated SAW oscillator based on the new excitation circuit. In *Proceedings of IEEE International Frequency Control Symposium & the European Frequency and Time Forum, Denver, CO, USA, April 12-16, 2015*.
- [37] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of ACM SIGCOMM, Hong Kong, China, August 12-16, 2013*.
- [38] Vincent Liu, Vamsi Talla, and Shyamnath Gollakota. Enabling instantaneous feedback with full-duplex backscatter. In *Proceedings of ACM MobiCom, Maui, Hawaii, USA, September 7-11, 2014*.
- [39] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of ACM SIGCOMM, Chicago, USA, August 17-22, 2014*.
- [40] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianchang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. PLoRa: A passive long-range data network from ambient LoRa transmissions. In *Proceedings of ACM SIGCOMM, Budapest, Hungary, August 20-25, 2018*.
- [41] Carlos Perez-Penichet, Claro Noda, Ambuj Varshney, and Thiemo Voigt. Battery-free 802.15.4 receiver. In *Proceedings of IEEE/ACM IPSN, Porto, Portugal, April 11-13, 2018*.
- [42] Brecht Reynders, Franco Minucci, Erma Perenda, Hazem Sallouha, , and Roberto Calvo Palomino. Fast-settling feedforward automatic gain control based on a new gain control approach. *IEEE Transactions on Circuits and Systems*, 61(9):651–655, 2014.
- [43] Brecht Reynders, Franco Minucci, Erma Perenda, Hazem Sallouha, Roberto Calvo, Yago Lizarribar, Markus Fuchs, Matthias Schafer, Markus Engel, Bertold Van den Bergh, Sofie Pollin, Domenico Giustiniano, Gerome Bovet, and Vincent Lenders. SkySense: Terrestrial and aerial spectrum use analysed using lightweight sensing technology with weather balloons. In *Proceedings of ACM MobiSys, Online, June 16-19, 2020*.
- [44] Mohammad Rostami, Karthik Sundaresan, Eugene Chai, Sampath Rangarajan, and Deepak Ganesan. Redefining passive in backscattering with commodity devices. In *Proceedings of ACM MobiCom, Virtual event, September 21-25, 2020*.
- [45] Franz Seifert, Helmut Stocker, and Otto Franz. The first SAW based IFF system and its operation in Austrian aerospace defence. In *Proceedings of IEEE History of Telecommunications Conference, Paris, France, eptember 11- 12, 2008*.
- [46] Joshua R. Smith, Alanson P. Sample, Pauline S. Powledge, Sumit Roy, and Alexander V. Mamishev. A wirelessly-powered platform for sensing and computation. In *Proceedings of ACM UbiComp, Orange County, California, September 17-21, 2006*.
- [47] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyamnath Gollakota. LoRa backscatter: Enabling the vision of ubiquitous connectivity. In *Proceedings of ACM UbiComp, Maui, HI, USA, September 11-15, 2017*.

- [48] Ambuj Varshney and Lorenzo Corneo. Tunnel emitter: Tunnel diode based low-power carrier emitters for backscatter tags. In *Proceedings of ACM MobiCom, Virtual event, September 21-25, 2020*.
- [49] Ambuj Varshney, Oliver Harms, Carlos Perez Penichet, Christian Rohner, and Thiem Voigt Frederik Hermans. LoReA: A backscatter architecture that achieves a long communication range. In *Proceedings of ACM SenSys, Delft, Netherlands, November 06-08, 2017*.
- [50] Anran Wang, Vikram Iyer, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakota. FM backscatter: Enabling connected cities and smart fabrics. In *Proceedings of USENIX NSDI, Boston, MA, USA, March 27-29, 2017*.
- [51] Ju Wang, Liqiong Chang, Shourya Aggarwal, Omid Abari, and Srinivasan Keshav. Soil moisture sensing with commodity RFID systems. In *Proceedings of ACM MobiSys, Toronto, Ontario, Canada, June 16-19, 2020*.
- [52] Jue Wang, Haitham Hassanieh, Dina Katabi, and Piotr Indyk. Efficient and reliable low-power backscatter networks. In *Proceedings of ACM SIGCOMM, Helsinki, Finland, August 13-17, 2012*.
- [53] Davide Zanetti, Boris Danev, and Srdjan Apkun. Physical-layer identification of UHF RFID tags. In *Proceedings of ACM MobiCom, Chicago, Illinois, USA, September 20-24, 2010*.
- [54] Peng Zhang, Houjun Wang, Li Li, Lianping Guo, and Ping Wang. FPGA based echo delay control method for pulse radar testing. In *Proceedings of 13th IEEE International Conference on Electronic Measurement and Instruments (ICEMI), Yangzhou, China, October 20-22, 2017*.
- [55] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. HitchHike: Practical backscatter using commodity WiFi. In *Proceedings of ACM SenSys, Stanford, CA, USA, November 14-16, 2016*.
- [56] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. FreeRider: Backscatter communication using commodity radios. In *Proceedings of ACM CONEXT, Incheon, Republic of Korea, December 12-15, 2017*.
- [57] Pengyu Zhang, Mohammad Rostami, Pan Hu, and Deepak Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of ACM SIGCOMM, Salvador, Brazil, August 22-26, 2016*.
- [58] Yufan Zhang, Ertao Li, and Yihua Zhu. Energy-efficient Dual-codebook-based backscatter communications for wireless powered networks. *ACM Transactions on Sensor Networks*, 17(9):1–20, 2021.
- [59] Jia Zhao, Wei Gong, and Jiangchuan Liu. Towards scalable backscatter sensor mesh with decodable relay and distributed excitation. In *Proceedings of ACM MobiSys, Virtual event, June 16-19, 2020*.
- [60] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. OFDMA-enabled WiFi backscatter. In *Proceedings of ACM MobiCom, Los Cabos, Mexico, October 21-25, 2019*.

## A Appendix

In this section, we prove the infeasibility of RLC resonant circuit to realize LoRa frequency-amplitude transformation.

### A.1 The Infeasibility of RLC Resonant Circuit

The center frequency  $\omega_0$ , the passband  $\Delta\omega$ , and the quality factor  $Q$  of a resonant circuit satisfy that:

$$Q = \frac{\omega_0}{\Delta\omega} \quad (6)$$

A higher  $Q$  value leads to a narrower passband width. Taking a step further, the quality factor  $Q$  is determined by the resistance  $R$ , inductance  $L$ , and capacitance  $C$  of this circuit following the equation:

$$Q = \sqrt{L}/(R \cdot \sqrt{C}) \quad (7)$$

Given a constant center frequency of  $\omega_0 = 1/(2\pi\sqrt{LC})$ , we can deduce the capacitance  $C$  satisfy that:

$$C = \frac{1}{Q\omega_0 R} = \frac{\Delta\omega}{\omega_0^2 R} \quad (8)$$

Generally, the equivalent  $R$  of RF circuit is  $50 \Omega$ . Taking LoRa signals working on 433 MHz frequency band (with 500 KHz bandwidth) as an example, this requires  $C$  to be as low as  $5.2 \times 10^{-14} pF$ .

