



Doctoral Thesis in Information and Communication Technology

# Realizing Low-Latency Packet Processing on Multi-Hundred-Gigabit-Per-Second Commodity Hardware

Exploit Caching to Improve Performance

ALIREZA FARSHIN

# Realizing Low-Latency Packet Processing on Multi-Hundred-Gigabit-Per-Second Commodity Hardware

Exploit Caching to Improve Performance

ALIREZA FARSHIN

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Technology on Monday the 6th March 2023, at 17:00 CET via Zoom and Sal C (Sven-Olof Öhrvik) at Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista, Stockholm, Sweden.

Doctoral Thesis in Information and Communication Technology  
KTH Royal Institute of Technology  
Stockholm, Sweden 2023

© Alireza Farshin

ISBN 978-91-8040-464-8

TRITA-EECS-AVL-2023:9

Printed by: Universitetsservice US-AB, Sweden 2023

## Abstract

By virtue of the recent technological developments in cloud computing, more applications are deployed in the cloud. Among these modern cloud-based applications, many societal applications require bounded and predictable low-latency responses. However, the current cloud infrastructure is unsuitable for these applications since it cannot satisfy these requirements due to many limitations in both hardware and software.

This doctoral dissertation describes our attempts to reduce the latency of Internet services by carefully studying the multi-hundred-gigabit-per-second commodity hardware, optimizing it, and improving its performance. The main focus is to improve the performance of packet processing done by the network functions deployed on commodity hardware, known as network functions virtualization (NFV), which is one of the significant sources of latency for Internet services.

The first contribution of this dissertation takes a step toward *optimizing the cache performance* of time-critical NFV service chains. By doing so, we reduce the tail latencies of such systems running at 100 Gbps. This is an important achievement as it increases the probability of realizing bounded and predictable latency for Internet services.

The second contribution of this dissertation performs whole-stack optimizations on software-based network functions deployed on top of modular packet processing frameworks to further enhance *the effectiveness of cache memories*. We build a system to efficiently handle metadata and produce a customized binary of NFV service chains. Our system improves both throughput & latency of per-core hundred-gigabit-per-second packet processing on commodity hardware.

The third contribution of this dissertation studies the efficiency of I/O security solutions provided by commodity hardware at multi-hundred-gigabit-per-second rates. We characterize the performance of IOMMU & IOTLB (*i.e., I/O virtual address translation cache*) at 200 Gbps and explore the possible opportunities to mitigate its performance overheads in the Linux kernel.

## Keywords

Low-Latency Internet Services, Packet Processing, Network Functions Virtualization, Middle Boxes, Commodity Hardware, Multi-Hundred-Gigabit-Per-Second, Low-Level Optimization



## Sammanfattning

Tack vare den senaste tekniska utvecklingen inom molntjänster används allt fler tillämpningar i molnet. Bland dessa moderna molnbaserade tillämpningar kräver många samhällsorienterade tillämpningar svarstider med låg latens, som är förutsägbara och ligger inom givna gränser. Den nuvarande molninfrastrukturen är dock otillräcklig för sådana tillämpningar eftersom den inte kan uppfylla dessa krav på grund av olika begränsningar i både hårdvara och mjukvara.

I denna doktorsavhandling beskrivs våra försök att minska latenstiden för Internettjänster genom att noggrant studera tillgänglig hårdvara med stöd för flera hundra gigabit per sekund, optimera denna och förbättra dess prestanda. Huvudfokus ligger på att förbättra prestandan för den paketbearbetning som utförs av nätverksfunktioner som installeras på allmänt tillgänglig hårdvara, så kallad nätverksfunktionsvirtualisering (NFV), som är en av de betydande källorna till latens för Internettjänster.

Det första bidraget i den här avhandlingen tar ett steg mot att optimera cache-prestanda för tidskritiska kedjor av NFV-tjänster. Genom att göra detta minskar vi de långa latenstiderna för sådana system som körs vid 100 Gbps. Detta är ett viktigt resultat eftersom det ökar sannolikheten för att uppnå en begränsad och förutsägbar fördröjning hos internettjänster.

Det andra bidraget i den här avhandlingen är optimeringar av hela stacken av mjukvarubaserade nätverksfunktioner som används ovanpå modulära ramverk för paketbearbetning för att ytterligare förbättra effektiviteten hos cacheminnen. Vi bygger ett system för att effektivt hantera metadata och producera anpassade binärversioner av NFV-tjänstekedjor. Vårt system förbättrar både genomströmning och latens för tillgänglig hårdvara där varje CPU-kärna har kapacitet för paketbearbetning i storleksordningen 100 Gbps.

I det tredje bidraget i denna avhandling studeras effektiviteten hos I/O-säkerhetslösningar som tillhandahålls av allmänt tillgänglig hårdvara i hastigheter på flera hundra gigabit per sekund. Vi karakteriserar prestandan hos IOMMU and IOTLB (dvs. "I/O memory management unit" och "I/O virtual address translation cache") vid 200 Gbps och undersöker möjligheterna att minska dess prestanda-overhead i kärnan av operativsystemet Linux.

## Nyckelord

Internettjänster med Låg Fördröjning, Paketbearbetning, Virtualisering av Nätverksfunktioner, Mellanutrustning, Tillgänglig Datorhårdvara, Flera-Hundra-Gigabit-Per-Sekund, Lågnivå-Optimering



## Acknowledgments

This doctoral dissertation is the summary of many discussions, collaborations, and hard work over the past few years. During my doctoral studies, I had the privilege to talk to, work with, and learn from many talented individuals. I want to take this opportunity to give them credit and thank them for their help & support.

First and foremost, I would like to thank my excellent supervisors, Professor Dejan Kostić and Professor Gerald Q. Maguire Jr. I could never be any happier as a doctoral student; they trusted me, even though I had a slightly different background, and they offered me a unique opportunity to learn many exciting things. It has been a great pleasure to be able to work under their supervision, and I will be forever grateful to them for their precious time, tremendous guidance, and endless support.

I want to thank my Google mentors and collaborators, Dr. Khaled Elmeleegy and Professor Luigi Rizzo. I had the opportunity to collaborate with them on the last contribution of this dissertation. I am thankful for all the online meetings & discussions, where I learned many interesting things and practiced being meticulous.

I would like to thank Associate Professor Johan Jansson for reviewing this dissertation and thank Associate Professor Markus Hidell for helping me with the Swedish abstract.

I want to express my appreciation for my friend and colleague, Amir Roozbeh. We shared an office for a few years, collaborated on many projects (including the first contribution of this dissertation), and made a lot of great memories while working & traveling together. I am grateful for all the endless discussions and unforgettable & fun memories.

I had the opportunity to work closely with Assistant Professor Tom Barbette, a postdoctoral researcher at NSLab at the time. We collaborated on a few projects together, had lots of interesting discussions, and shared many beers. The second contribution of this dissertation, PacketMill, is the result of our collaboration.

I also like to thank my colleagues at NSLab: Alexandros Milolidakis, Georgios Katsikas, Giacomo Verardo, Hamid Ghasemirahni, Kirill Bogdanov, Associate Professor Marco Chiesa, Massimo Gironi, Voravit Tanyinyong, and Waleed Reda—and also thank my friends (they know who they are) who made this journey much easier and more enjoyable for me.



The research leading to this dissertation has received funding from (i) Swedish Foundation for Strategic Research (SSF) and (ii) European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889). Furthermore, Google has provided me an unrestricted gift for receiving a 2021-2022 Google PhD Fellowship in Systems and Networking. I am thankful to them for supporting my doctoral studies.



*Last but not least, I also want to thank my family, especially my parents: Mohsen and Mahvash. None of this would have been possible without their continuous support and endless love; I will be forever indebted to them for making me the person I am now.*

Alireza Farshin,  
Stockholm, February 2023

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Why NFV's Performance Matters? . . . . .	2
1.2	Problem: Gap between Processing Capability and Link Speed Growth . . . . .	3
1.3	Research Objectives . . . . .	5
1.4	Research Methodology . . . . .	5
1.5	Research Contributions . . . . .	6
1.5.1	Dissertation Contributions . . . . .	11
1.5.2	Publications . . . . .	11
1.6	Dissertation Hypothesis . . . . .	14
1.7	Research Impact . . . . .	15
1.8	Sustainability, Ethical, and Security Aspects . . . . .	16
1.8.1	Sustainability . . . . .	16
1.8.2	Ethical Aspect . . . . .	17
1.8.3	Security Aspect . . . . .	17
1.9	Dissertation Organization . . . . .	18
<b>2</b>	<b>Improving Hardware Cache Utilization</b>	<b>19</b>
2.1	Cache Architecture . . . . .	20
2.1.1	Last Level Cache (LLC) . . . . .	22
2.1.2	Opportunities . . . . .	23
2.2	CacheDirector . . . . .	24
2.2.1	Data Plane Development Kit (DPDK) . . . . .	26
2.2.2	Design and Implementation . . . . .	28
2.3	Evaluation . . . . .	33
2.3.1	Context of the Evaluation . . . . .	33
2.3.1.1	Testbed . . . . .	33
2.3.1.2	NFV . . . . .	34
2.3.1.3	Measurement Method . . . . .	34

2.3.2	Simple Forwarding . . . . .	34
2.3.2.1	64 B Packets at Low Rate . . . . .	35
2.3.2.2	Mixed-size Packets at 100 Gbps . . . . .	36
2.3.3	Stateful Service Chain . . . . .	38
2.3.3.1	Mixed-size Packets at 100 Gbps . . . . .	38
2.3.3.2	Tail Latency vs. Throughput . . . . .	40
2.3.4	Summary of Evaluation . . . . .	41
2.4	Limitations . . . . .	42
2.4.1	NIC driver support . . . . .	42
2.4.2	Noisy neighbor effect . . . . .	42
2.4.3	Portability . . . . .	43
2.5	Related Works . . . . .	43
2.6	Summary . . . . .	44
<b>3</b>	<b>Optimizing Software-Based Network Functions</b>	<b>45</b>
3.1	Software Packet Processing . . . . .	46
3.1.1	Code Inefficiency . . . . .	47
3.1.2	Patched Metadata Management . . . . .	48
3.2	PacketMill . . . . .	52
3.2.1	Efficient Metadata Management . . . . .	52
3.2.2	Optimized Code . . . . .	56
3.2.2.1	Source Code Modifications . . . . .	56
3.2.2.2	Intermediate Code Modifications . . . . .	58
3.3	Evaluation . . . . .	60
3.3.1	Do PacketMill's Code Optimizations Improve Packet Processing at 100 Gbps? . . . . .	61
3.3.2	How Effective is PacketMill's Model (X-Change) compared to the existing Metadata Management Models? . . . . .	64
3.3.3	How Effective is PacketMill at Different Loads? . . . . .	66
3.3.4	How Does the Workload/Trace Affect PacketMill? . . . . .	66
3.3.5	How about More Sophisticated Network Functions? . . . . .	67
3.3.6	Is PacketMill Useful for Multicore Network Functions? . . . . .	73
3.3.7	How about State-of-the-Art Packet Processing Frameworks? . . . . .	73
3.4	Frequently Asked Questions . . . . .	75
3.5	Related Works . . . . .	77
3.6	Summary . . . . .	78
<b>4</b>	<b>Toward Secure Multi-Hundred-Gigabit-Per-Second Services</b>	<b>79</b>
4.1	Background . . . . .	80
4.1.1	IOMMU Performance Impact on DMA Operations . . . . .	81
4.1.2	Networking Data Path in the Linux Kernel . . . . .	82

4.1.2.1	RX path . . . . .	82
4.1.2.2	TX path . . . . .	83
4.1.2.3	Operation When IOMMU is Enabled . . . . .	83
4.2	IOMMU Performance Characterization . . . . .	84
4.2.1	Data Rate and IOTLB Wall . . . . .	87
4.2.2	Effect of MTU Size . . . . .	91
4.2.3	Large Receive Offload (LRO) . . . . .	92
4.2.4	TX & TCP Segmentation Offload (TSO) . . . . .	93
4.2.5	Different Processors . . . . .	95
4.2.6	Different NICs . . . . .	99
4.2.7	Other Workloads . . . . .	99
4.2.8	Summary of Characterization . . . . .	101
4.3	A Solution to IOTLB Wall . . . . .	101
4.3.1	Challenges and Design Options . . . . .	102
4.3.2	Proposed Solution . . . . .	103
4.3.2.1	Discussion and Future Works . . . . .	104
4.4	Evaluation . . . . .	105
4.4.1	IOTLB Misses and Throughput Drop . . . . .	105
4.4.2	Run-Time Allocation Overhead . . . . .	106
4.4.3	Buffer Shuffling . . . . .	107
4.5	Frequently Asked Questions . . . . .	108
4.6	Related Works . . . . .	110
4.7	Summary . . . . .	110
<b>5</b>	<b>Related Works</b>	<b>111</b>
5.1	Software-Related Works . . . . .	112
5.2	Hardware-Related Works . . . . .	113
<b>6</b>	<b>Conclusion and Future Works</b>	<b>115</b>
6.1	Future Works . . . . .	116
	<b>References</b>	<b>123</b>
<b>A</b>	<b>Network Function Configurations</b>	<b>169</b>
A.1	Simple Forwarder . . . . .	169
A.2	Router . . . . .	169
A.3	NAPT, IDS, and VLAN . . . . .	169
A.4	WorkPackage . . . . .	170

<b>B</b>	<b>IOMMU – Supplementary Information</b>	<b>171</b>
B.1	IOTLB Implementation on Different Architectures . . . . .	171
B.2	Testbed Illustration and Additional Results . . . . .	172
B.3	Page (De)Allocation via Page Pool API . . . . .	173
B.4	Current Implementation of HPA . . . . .	174
B.5	IOTLB Overheads in DPDK . . . . .	176

# List of Figures

1.1	Examples of NF service chains (the upper chain is comprised of a router, NAPT, LB, and another router; while the lower chain is comprised of a router, firewall, DPI, and another router). . . . .	2
1.2	Gap between Ethernet speed evolution . . . . .	4
1.3	Research method used in this doctoral project. . . . .	6
2.1	Memory hierarchy. . . . .	19
2.2	Cache placement policies. Green boxes show the valid locations for a cache line in each policy. . . . .	21
2.3	Physical address mapping within cache hierarchy. . . . .	21
2.4	Non-uniform Cache Architecture in Intel processors. . . . .	22
2.5	Mapping for different cache lines in a 4-KiB page. . . . .	23
2.6	Access time to different LLC slices from core 0 in Xeon-E5-2667 v3 (Haswell). . . . .	24
2.7	DDIO vs. CacheDirector . . . . .	25
2.8	Simplified memory management in DPDK . . . . .	28
2.9	CacheDirector changes to the mbuf structure . . . . .	30
2.10	Distribution of headroom size for $\sim 12.3$ Million Packets + 95 <sup>th</sup> percentile of the distribution. . . . .	30
2.11	Experiment setup . . . . .	33
2.12	End-to-end latency without loopback latency for 64 B packets sent at the rate of 1000 pps. . . . .	35
2.13	End-to-end latency and improvement for a simple forwarding application running on 8 cores with mixed-size packets at 100 Gbps with RSS . . . . .	37
2.14	CDF of end-to-end latency without loopback latency and improvement for a stateful service chain (Router-NAPT-LB) running on 8 cores while sending mixed-size packets at the rate of 100 Gbps with Hardware (H/W) offloading using FlowDirector . . . . .	39

2.15	Tail latency (99 <sup>th</sup> percentile) vs. Throughput for a stateful service chain (Router-NAPT-LB) running on 8 cores while sending mixed-size packets at different rates with H/W offloading using FlowDirector.	40
2.16	Improvement for a stateful service chain (Router-NAPT-LB) running on 8 cores with mixed-size packets at the rate of 100 Gbps with RSS and <i>without</i> any hardware offloading . . . . .	42
3.1	Common metadata management methods in packet processing frameworks (Copying vs. Overlaying) . . . . .	51
3.2	PacketMill Overview . . . . .	53
3.3	Exploiting the information in the router configuration improves throughput and median latency. The server is processing packets with <i>one</i> core running at different frequencies (f in GHz) . . . . .	63
3.4	X-Change makes it possible to forward packets at >100-Gbps rates. The experiments with two NICs report the sum of throughput achieved by one core . . . . .	65
3.5	PacketMill improves per-core packet processing. Overlapped markers show that the performance can be capped despite the increasing offered load . . . . .	66
3.6	For a router running at 2.3 GHz, PacketMill improves the number of processed packets per second for different packet sizes . . . . .	67
3.7	PacketMill improves the performance of a more compute-intensive NF ( <i>i.e.</i> , an IDS+router running at 2.3 GHz), by up to 20% throughput and 17% latency . . . . .	68
3.8	PacketMill is effective for sophisticated network functions. Left and right figures show synthetic NFs that perform one and five memory accesses per packet, respectively . . . . .	70
3.9	Increasing memory intensiveness results in larger number of LLC loads that is inversely proportional to the performance of the synthetic NF . . . . .	72
3.10	PacketMill also improves the performance of multicore NFs. A NAT is running at 2.3 GHz . . . . .	73
3.11	Comparison of packet processing frameworks forwarding fixed-size packets with a single core . . . . .	75
4.1	TCP throughput and receiver CPU utilization with and without IOMMU on the receiver (200-Gbps link, Maximum Transmission Unit (MTU)=1500) . . . . .	88
4.2	Throughput drop and IOTLB misses per MiB versus offered rate, MTU=1500 . . . . .	89

4.3	Throughput drop and IOTLB misses per MiB versus offered rate for MTU=3690 . . . . .	90
4.4	Receiver side average IOTLB misses per MiB for different MTU sizes, in underloaded (100-Gbps) and overloaded (200-Gbps) . . .	91
4.5	Throughput with/without IOMMU for different MTU sizes, in overloaded (200-Gbps) condition . . . . .	92
4.6	Effect of LRO on IOTLB misses and throughput in underloaded and overloaded conditions . . . . .	93
4.7	Using TSO reduces the IOTLB overheads on the Transmit (TX) path due to fewer IOTLB translations . . . . .	94
4.8	Throughput and IOTLB misses on several Xeon processors, operating at 100 Gbps . . . . .	96
4.9	Throughput of IceLake and AMD workstations with and without IOMMU at 200 Gbps . . . . .	98
4.10	IOTLB misses of Intel E810 and NVIDIA/Mellanox ConnectX-6 NICs at 100 Gbps . . . . .	99
4.11	Packet drops put more pressure on IOTLB due to buffer shuffling and poor recycling . . . . .	100
4.12	Memcached (a less I/O insensitive application) has a similar IOTLB pressure to iPerf at 90 Gbps (TPS= $\sim$ 170k) . . . . .	101
4.13	Throughput comparison with IOMMU-off, IOMMU-4-KiB and IOMMU-2-MiB for MTU=1500. The use of 2-MiB mappings recovers the throughput drop introduced by the IOMMU . . . . .	102
4.14	Larger IOMMU mappings on the receiver side significantly reduce throughput drop and IOTLB misses . . . . .	106
4.15	Per 4-KiB page allocation costs are similar with 4-KiB and 2-MiB pages in our experiments . . . . .	107
4.16	Increased IOTLB misses due to the buffer shuffling could cause IOTLB to again become a bottleneck for an iPerf server receiving 1500-B frames with different pool sizes per queue . . . . .	108
5.1	An overview of recent efforts geared toward high-speed networking. . . . .	111
B.1	Our testbed. Green servers represent DuTs. CX5 and CX6 stand for ConnectX-5 and ConnectX-6, respectively . . . . .	172
B.2	IOMMU imposes performance overheads at all MTU sizes, but with a larger absolute throughput drop for MTU sizes larger than $\sim$ 3000 B . . . . .	173
B.3	Page Pool overview. . . . .	174
B.4	DPDK also experiences performance degradation with IOMMU when it uses 4-KiB pages . . . . .	177





# List of Tables

2.1	DPDK memory management libraries. . . . .	26
2.2	Throughput and average improvement in throughput while sending mixed-size packets at the rate of 100 Gbps. . . . .	36
3.1	PacketMill's code optimizations improve microarchitectural metrics by up to $\sim 300\times$ ( <i>i.e.</i> , reducing the number of LLC load misses). . . . .	62
4.1	Processors used for DUT. The entry marked with a star (★) is our primary testbed. . . . .	85



# Listings

2.1	Precalculated headroom sizes for different LLC slices in <code>udata64</code> .	32
3.1	X-Change introduces conversion functions instead of directly writing to the <code>rte_mbuf</code> struct. The code shows an example for setting the VLAN TCI field in the driver. . . . .	54
3.2	X-Change simplifies the metadata management. The code compares the default DPDK and a custom implementation to set the VLAN TCI field . . . . .	55
3.3	A Click's NF configuration file defining a simple forwarder that receives packets from a DPDK-enabled NIC, swaps the Ethernet MAC addresses, and transmits the packets . . . . .	57
3.4	Accessing a metadata field in LLVM Intermediate Representation (IR) bitcode (bottom). The top shows the C++ version of the code.	59



## List of acronyms and abbreviations

AHEAD	Authenticated Encryption with Associated Data
AMX	Advanced Matrix Extension
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASLR	Address Space Layout Randomization
ATS	Address Translation Service
CAPEX	Capital Expenditure
CAT	Cache Allocation Technology
CFG	Control Flow Graph
COTS	Commodity Off-The-Shelf
CPU	Central Processing Unit
CXL	Compute Express Link
DCA	Direct Cache Access
DDIO	Data Direct I/O
DLB	Dynamic Load Balancer
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
DPU	Data Processing Unit
DRAM	Dynamic Random Access Memory
DSA	Data Streaming Accelerator
DSL	Domain-Specific Language
DuT	Device under Test
EAL	Environment Abstraction Layer
eBPF	Extended Berkeley Packet Filter
FCS	Frame Check Sequence
FD.io	Fast Data Project
GEPI	GetElementPtrInst
GPU	Graphics Processing Unit
GRO	Generic Receive Offload
H/W	Hardware

HDD	Hard Disk Drive
HPC	High Performance Computing
I/O	Input/Output
IAA	In-Memory Analytics Accelerators
IACA	Intel® Architecture Code Analyzer
IDS	Intrusion Detection System
IOMMU	I/O Memory Management Unit
IOTLB	I/O Translation Lookaside Buffer (TLB)
IOVA	I/O Virtual Address
IPC	Instructions per Cycle
IR	Intermediate Representation
L1	level one
L2	level two
L3	level three
LB	Load Balancer
LLC	Last Level Cache
LoadGen	Load Generator
LRO	Large Receive Offload
LRU	Least Recently Used
LTO	Link-Time Optimization
MAC	Media Access Control
ML	Machine Learning
MLC	Mid-Level Cache
MMU	Memory Management Unit
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NAPT	Network Address Port Translation
NF	Network Function
NFV	Network Functions Virtualization
NIC	Network Interface Card
NPF	Network Performance Framework
NUCA	Non-Uniform Cache Architecture
NUMA	Non-Uniform Memory Access
NVM	Non-Volatile Memory
OPEX	Operational Expenditure

OS	Operating System
OVS	Open vSwitch
PA	Physical Addresses
PCIe	Peripheral Component Interconnect express
PCM	Performance Counter Monitor
PDE	Page Directory Entry
PGO	Profile-Guided Optimization
PL	Programming Language
PMD	Poll Mode Driver
pos	Plain Orchestration Service
pps	Packet Per Second
PTE	Page Table Entry
QAT	QuickAssist Technology
RDMA	Remote Direct Memory Access
RSS	Receive Side Scaling
RTC	Run-to-Completion
RTT	Round-Trip Time
RX	Receive
SDN	Software-Defined Networking
SIMD	Single Instruction/Multiple Data
SLO	Service Level Objective
SoC	System on Chip
SSD	Solid State Device
SSE	Streaming SIMD Extensions
TLB	Translation Lookaside Buffer
TPU	Tensor Processing Unit
TSO	TCP Segmentation Offload
TTL	Time to Live
TX	Transmit
VM	Virtual Machine
vPMD	Vectorized PMD
VPP	Vector Packet Processing
XDP	eXpress Data Path





# Chapter 1

## Introduction

CLOUD computing has provided opportunities for exploiting logically centralized resources, by which users access different services via the Internet anytime from anywhere. Using the cloud is becoming increasingly popular among companies, and many prefer to use cloud infrastructures to host their services to reduce expenses. In addition, the cloud also provides greater flexibility, scalability, and availability, which can increase their clients' satisfaction. Cloud computing has become even more crucial after a sudden increase in social distancing and the number of work-from-home employees due to the COVID-19 global pandemic. There have been many efforts to increase the throughput of cloud-based Internet services by increasing the number of data centers hosting these services and increasing the bandwidth of links (both within and between data centers – including the introduction of hundred-gigabit-per-second links). However, these efforts *cannot* guarantee low latency for Internet services, making cloud infrastructures unsuitable for time-critical applications [1]. Moreover, higher bandwidth (*e.g.*, 100/200/400 gigabit per second (Gbps)) makes it more challenging to deploy high-performance Internet services on general-purpose hardware, *aka* commodity hardware. To realize these services, it is important to identify those components that have a large *negative* impact on the latency of Internet services and then holistically address these latencies.

From a high-level point of view, the end-to-end latency of Internet services can be divided into two portions: (*i*) network delay and (*ii*) application delay. Network delay, itself, is composed of propagation delay and processing delay; with the latter mostly due to specific Network Functions (NFs) processing packets. Traditionally, propagation delay has been the most significant portion of the end-to-end latency [2]. However, propagation delay is becoming comparable to

the processing delay and the application delay because of the increase in the number of data centers per region [3, 4, 5, 6] and the recent enhancements in networking, such as the wide deployment of fiber infrastructure, the transition toward multi-gigabit-per-second links and switches in data centers [7, 8, 9], and the expected latency reductions in the next generations of mobile communication (5G & 6G) [10, 11, 12]. Therefore, it becomes increasingly important to minimize *both* processing delay and application delay to realize low-latency Internet services.

This doctoral dissertation focuses on minimizing the processing delay of NFs that are deployed on commodity hardware as a part of the transition toward Network Functions Virtualization (NFV) [13]. While our efforts are primarily targeted toward packet processing, the contributions & conclusions of this dissertation could be equally useful for other networking applications requiring low latency at multi-hundred-gigabit-per-second rates. The next section discusses the significance of optimizing NFV performance and the challenges of doing so.

## 1.1 Why NFV's Performance Matters?

Network packets, typically, pass through several middleboxes (*aka* network appliances) that process packets in order to ensure security (*e.g.*, firewall, Intrusion Detection System (IDS), and Deep Packet Inspection (DPI)), connectivity (*e.g.*, router, switch, and Network Address Port Translation (NAPT)), and suitable performance (*e.g.*, Load Balancer (LB)). A set of these NFs that a packet traverses are called a *NF service chain*. Figure 1.1 shows two examples of NF service chains.

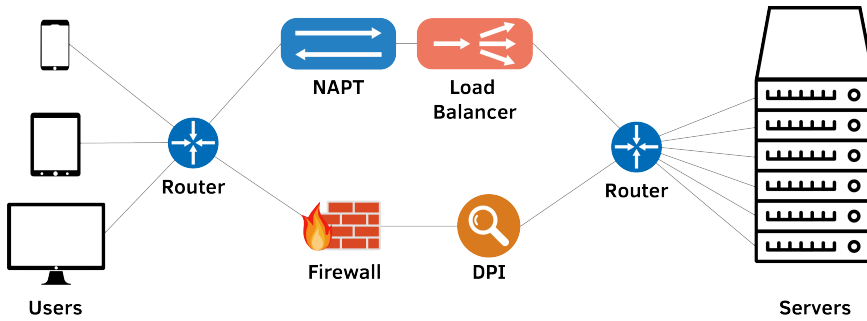


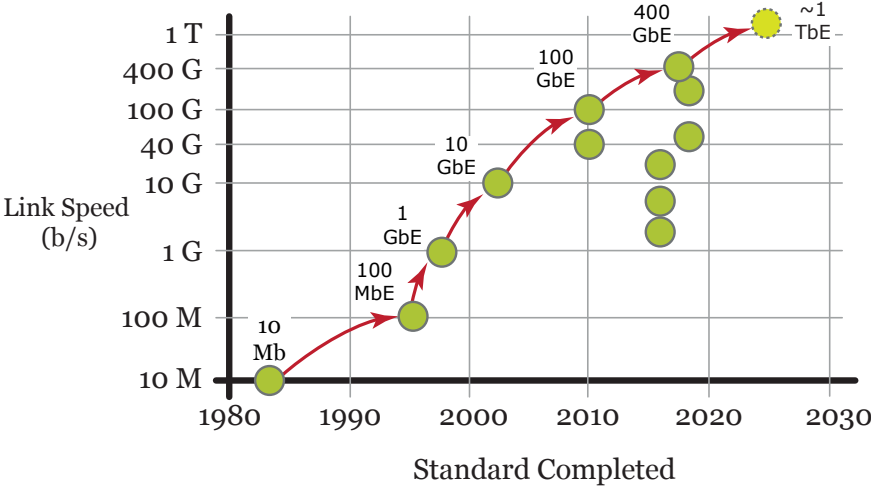
Figure 1.1: Examples of NF service chains (the upper chain is comprised of a router, NAPT, LB, and another router; while the lower chain is comprised of a router, firewall, DPI, and another router).

The number of middleboxes in an enterprise network is comparable to the number of switches & routers [14], making them a large source of the processing delay. Traditionally, middleboxes were implemented on specialized hardware built by large companies (*e.g.*, Cisco and Juniper). However, NFV has pushed network operators/providers to employ software-driven solutions that can be run on Commodity Off-The-Shelf (COTS) servers, *aka* commodity hardware, to (i) avoid proprietary inflexible hardware middleboxes, to (ii) reduce Capital Expenditure (CAPEX) & Operational Expenditure (OPEX), and to (iii) speed up time-to-market. Although this transition reduces the cost to enterprises and offers a higher degree of flexibility & programmability, it induces some challenges for high-performance and latency-critical networking applications, as the general-purpose hardware generally does not achieve the same high throughput and low latency as specialized hardware.

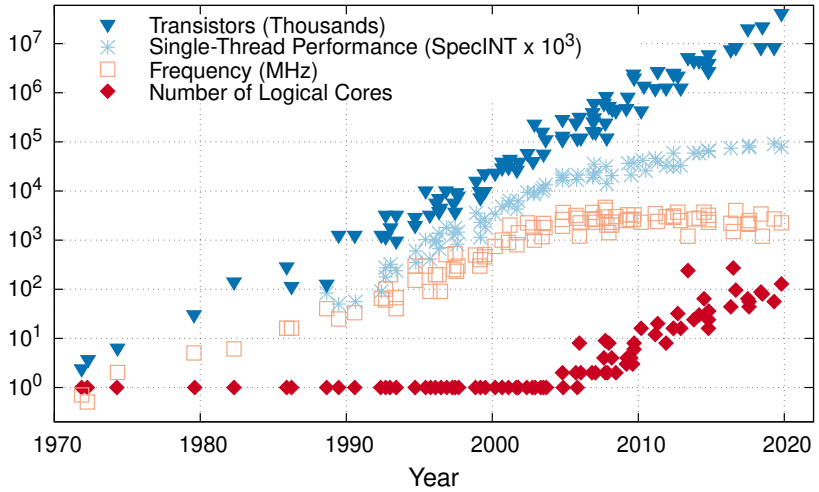
Many have tried to address this challenge by optimizing the performance of *virtual* NFs [15, 16, 17], but their performance is still far from optimal. Moreover, introducing faster link speeds exposes NFs to packets at a higher rate, thus achieving high performance becomes even harder for commodity hardware. Therefore, it is crucial to focus on optimizing the performance of NFV service chains that constitute a *large* portion of the processing delay. Performing such optimization cannot be done simply by using more powerful hardware for a number of reasons. The next section elaborates this problem and mentions the inability of current hardware to cope with the increase in link speeds.

## 1.2 Problem: Gap between Processing Capability and Link Speed Growth

Ideally, the emergence of faster link speeds should lead to low-latency/faster Internet services, but the demise of Dennard scaling and the slowdown of Moore's law puts a cap on commodity hardware performance [18]. Figure 1.2a and Figure 1.2b show the trends of link speed & processor evolution over the last ~40 years. These trends show that Ethernet speed is constantly increasing, whereas the processor's performance is capped. More specifically, processors are being shipped with more Central Processing Unit (CPU) cores, but the per-core performance/frequency is saturating [19].



(a) Ethernet Speed Evolution [20].



(b) Processor Evolution [19].

Figure 1.2: Gap between Ethernet speed evolution (*i.e.*, Nielson’s law [21]) and processors evolution (*i.e.*, Moore’s law [22] & Dennard scaling [18]).

As noted earlier, introducing faster link speeds exposes processing elements to packets at a higher rate. For instance, a server receiving 64-B packets at a link speed of 100 Gbps has only 6.72 ns to process a packet before the next packet arrives, which is a relatively small time budget (*i.e.*, smaller than Last Level Cache (LLC) access latency). Exceeding this time budget causes a tremendous amount of queuing and performance degradation, which is undesirable for low-latency Internet services. To process packets within this time, it is necessary to optimize *both* software and hardware to reduce the processing latency. Networking software needs to mitigate the unnecessary inefficiencies (*e.g.*, inter-core communication [16], inefficient memory accesses [23], and costly kernel stack operations [24]) in order to operate in level one (L1) & level two (L2) caches, *i.e.*, minimizing even LLC accesses, whereas networking hardware should provide (*i*) cache-efficient methods to interact with software and (*ii*) processing capabilities to reduce processing latency.

### 1.3 Research Objectives

The goal of this dissertation is to realize low-latency Internet services on multi-hundred-gigabit-per-second commodity hardware by optimizing the performance of NFV service chains processing packets. Since the processing time budget is shrinking to *nanoseconds*, it is essential to take into account every nanosecond and make the most out of available resources, especially cache memories due to their fast access latency given the small time budget when processing packets at multi-hundred-gigabit-per-second rates. Our goal can be split into two main objectives:

**Objective 1** *Study & evaluate the effectiveness of existing computer systems in handling high link-rate network traffic.*

**Objective 2** *Identify and exploit potential opportunities to optimize packet processing\* to facilitate realizing low-latency Internet services.*

### 1.4 Research Methodology

This doctoral project uses a quantitative research approach [25]. Using this approach, we followed a closed-loop method, which starts by studying both software and hardware in computer systems to (*i*) evaluate their effectiveness when handling multi-hundred-gigabit-per-second traffic and (*ii*) find optimization opportunities in order to get the maximum performance out of the current hardware.

---

\*Our efforts are primarily geared toward kernel-bypass frameworks, but we briefly examine the Linux kernel in Chapter 4.

After identifying opportunities, we experimentally evaluated their effectiveness at multi-hundred-gigabit-per-second networking & potentially apply them to networking applications. Next, we designed/re-designed networked systems (mainly NFV systems) according to what we have learned in our study; then we implemented our proposed design; and we finished this process by empirically evaluating the effectiveness of our solution. Furthermore, we continuously used the observations, made during the implementation, and experimental results, gathered throughout this project, to improve our proposed design. Figure 1.3 depicts our method to conduct this project.

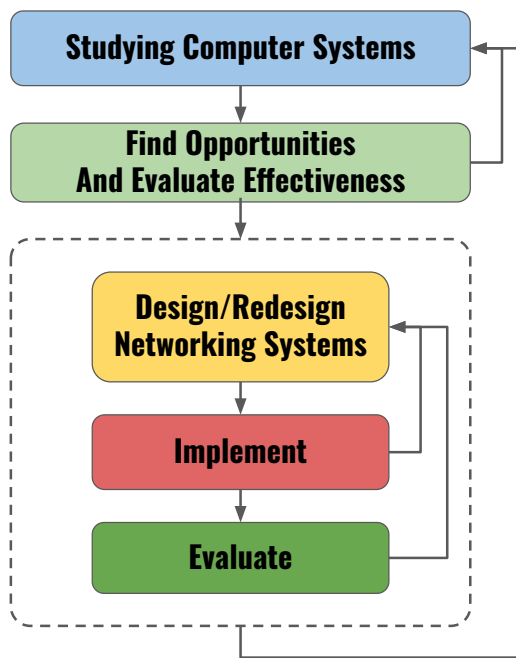




Figure 1.3: Research method used in this doctoral project.

## 1.5 Research Contributions

The objectives were realized via four main tasks (numbered A to D below). This section elaborates each task and outlines the specific contributions of this doctoral project in each task.

**Task A: Improving hardware cache utilization.** Realizing multi-hundred-gigabit-per-second, low-latency Internet services on commodity hardware is only possible with careful cache management, thus software needs to utilize higher cache levels (*i.e.*, L1 & L2). The goal is to study the current implementation of cache management techniques & technologies available in commodity hardware, such as Direct Cache Access (DCA) & Cache Allocation Technology (CAT), and then examine their significance with respect to the performance of multi-hundred-gigabit-per-second networking. In particular, this task (*i*) identifies the limitations and proposes modifications & improvements for future generations of commodity hardware and (*ii*) proposes software improvement techniques to make fast & low-latency networking possible.

**Contributions.** This task resulted in two conference papers presented at EuroSys'19 [26] & ATC'20 [27]; 9 patent applications [28, 29, 30, 31, 32, 33, 34, 35, 36, 37] filed by Ericsson; and two poster presentations presented at EuroSys'19 & EuroSys'20 [38]. The main contributions of this task were:


- A.1** We studied the Intel LLC organization, quantified its characteristics for two generations of Intel processors, and measured the access time to different LLC slices from each of the cores. In addition, we introduced a new memory management scheme, called slice-aware memory management [26, 39, 40], which allocates memory by taking into account the characteristics of LLC slices. Evaluation results showed that our proposed scheme could boost the performance of applications.
- A.2** Using our proposed memory management scheme, we developed a network Input/Output (I/O) solution, called CacheDirector [26], which extends Data Direct I/O (DDIO) and sends the packet header to the appropriate LLC slice, *i.e.*, closest to the processing core. We demonstrated that CacheDirector could be used to cut tail latencies of time-critical NFV service chains running at 100 Gbps.
- A.3** We examined the current implementation of DCA in Intel processors, called DDIO technology [27, 40]. We empirically unveiled the undocumented details of DDIO implementation and demonstrated the importance of tuning/optimizing it appropriately for a given Internet service to achieve high performance, especially with the introduction of multi-hundred-gigabit-per-second networks.
- A.4** The source code for these works are available at:
  -  aliireza/slice-aware
  -  aliireza/ddio-bench




**Individual Contribution.** A.1 and A.3 were done in collaboration with Amir Roozbeh. We both contributed equally to this part of the project. Additionally, I made all of the open-source contributions.

**Task B: Optimizing software-based NFs.** Today, many NFs are deployed on commodity hardware, via *unspecialized* modular software with software-based packet processing (for example, by Ericsson, Cisco, and Intel [41, 42, 43]). Software-based NFs are typically built via modular frameworks to increase flexibility and to simplify the composition of complex network services, by customizing and connecting simple monolithic elements. However, these frameworks usually produce a general-purpose binary based on an input configuration file, which results in many inefficiencies, such as virtual calls, dead code, and unordered basic blocks. In contrast, we mitigate these code inefficiencies and produce an optimized binary while maintaining high-level modularity and flexibility, as opposed to relying on handwritten assembly code [44]. We believe that efficient packet processing at multi-hundred-gigabit-per-second rates calls for holistic system optimizations, specifically *milling* the entire software stack to *squeeze* every bit of performance from the network & system hardware.

**Contributions.** This task has resulted in a conference paper presented at ASPLOS’21 [45, 46] and 2 patent applications [35, 47] filed by Ericsson.

**B.1** We presented a system, called  PacketMill, which (i) introduces a new model to efficiently manage packet metadata and (ii) employs code-optimization techniques to better utilize commodity hardware. PacketMill grinds the whole packet processing stack, from the high-level network function configuration file to the low-level userspace network (specifically Data Plane Development Kit (DPDK)) drivers, to mitigate inefficiencies and produce a customized binary for a given network function.

**B.2** The source code of this work is available at:  [aliireza/packetmill](https://github.com/aliireza/packetmill)

**Individual Contribution.** B.1 and B.2 were done in collaboration with Tom Barbette. We both contributed equally to this project.

**Task C: Toward efficient & secure multi-hundred-gigabit-per-second services.** The introduction of multi-hundred-gigabit-per-second network interfaces (100/200/400 Gbps) has motivated many researchers to study networked systems & their underlying hardware to understand the challenges of achieving suitable performance at higher rates. These studies explored various topics, such as kernel network stack [48], Peripheral Component Interconnect express (PCIe) [49], Remote Direct Memory

Access (RDMA) [50], DDIO & LLC [27, 26], and smart Network Interface Card (NIC) [51, 52]. However, most studies were primarily interested in improving performance, while paying less attention to critical real-world requirements such as security, privacy, and efficiency. Consequently, these studies often *disable* many software & hardware features to maximize the performance of networking applications. We examined some of the missing features that are essential for large service providers and data center companies to realize secure & efficient Internet services. More specifically, we target I/O Memory Management Unit (IOMMU) in server-grade processors, which ensures privacy in multi-tenant environments and protects services from being targeted by Direct Memory Access (DMA) attacks. Another important feature is power/frequency management (*e.g.*, c & p states) that provide the foundation for efficiently utilizing the processing power, which is crucial for deploying power-efficient & environmental-friendly services.

**Contributions.** This task resulted in an under-submission PeerJ Computer Science journal article [53, 54, 55, 56]. The main contributions of this task are:

- C.1** We characterized I/O TLB (IOTLB) behavior and its effects on recent Intel Xeon Scalable & AMD EPYC processors at 200 Gbps, by analyzing the impact of different factors contributing to IOTLB misses & causing throughput drop.
- C.2** We discussed and analyzed possible mitigations, including proposals and evaluation of a practical hugepage-aware memory allocator for the network device drivers to employ hugepage IOTLB entries in the Linux kernel.
- C.3** The source code of this work is available at: [🔗 aliireza/iommu-bench](https://github.com/aliireza/iommu-bench)

**Individual Contribution.** I have been the main driver & contributor in this project.

**Task D: Utilizing new technologies.** One way to close the gap between network link rates and the maximum performance of processors is to utilize available resources more efficiently and employ new features of networking equipment (*e.g.*, smart NICs, programmable switches, RDMA). There have already been some efforts along these lines. For instance, Metron [16] utilizes Software-Defined Networking (SDN) as well as flow steering features in modern NICs (*e.g.*, FlowDirector [57]) to send requests directly to the CPU core responsible for processing a given packet in an NFV service chain; while MICA [58] uses a similar approach to partition keys among different CPU cores and tag packets at the client. Both approaches improve the performance of these applications as there are fewer inter-core communications. There are many similar optimizations that can

be done to improve the performance of current systems, but applying them requires a careful study of hardware, software, and their interaction. More specifically, we *(i)* study & examine the offloading capabilities of modern network equipment and *(ii)* employ them to reduce the latency of multi-hundred-gigabit-per-second Internet services while addressing the challenges of doing so, *e.g.*, finding a suitable trade-off between cost, flexibility, and performance. For instance, in the case of smart NICs equipped with System on Chip (SoC), *i.e.*, low-frequency ARM cores and Application Specific Integrated Circuit (ASIC)-based packet processors, it is essential to achieve optimal resource allocation while addressing the performance limitations (*e.g.*, communication overhead imposed by PCIe) to make the most out of offloading capabilities, *i.e.*, offloading the *right* amount of stateless & costly operations to ARM cores & ASIC on the NIC vs. keeping the stateful & complex operations on powerful x86 cores on the host [59, 60]. Additionally, both offloaded sections (*i.e.*, running on the smart NIC) & non-offloaded sections (*i.e.*, running on the x86 CPU) should be optimized to achieve the maximum performance from the hardware. Our ultimate goal is not only to utilize these technologies but also to identify the need for *(i)* specific hardware features to support them and *(ii)* new offloading/processing capabilities required by high-speed networking applications.

**Contributions.** This task resulted in a conference paper presented at NSDI’22 [61], an under-submission work, and 7 patent applications [62, 63, 64, 47, 65, 66, 67] filed by Ericsson. The main contributions of this task are:

- D.1** We systematically studied the impact of temporal and spatial traffic locality on the performance of commodity servers equipped with high-speed network interfaces.
- D.2** We built a software solution, called Reframer, which deliberately delays packets and reorders them to increase traffic locality. Reframer can be deployed as an independent NF either on the same server, a smart NIC, or potentially a programmable switch.
- D.3** We studied novel packet processing architectures that receive external “hints” about which packets are soon to arrive, thus enabling prefetching into fast cache memories of the state needed to process them, just-in-time for the packets’ arrival.
- D.4** We explored possible approaches to *(i)* obtain prefetching hints either from network devices or the end hosts in the communication and *(ii)* use these hints to better utilize cache memories.

**Individual Contribution.** Amir Roozbeh and I were responsible for analyzing a 28-minute KTH campus trace for **D.1**. Additionally, I was actively involved in

(i) writing and editing both NSDI'22 & an under-submission work and (ii) the research discussions. It is worth noting that Hamid Ghasemirahni is the first author of both papers, and he has made all the open-source contributions.

### 1.5.1 Dissertation Contributions

This doctoral dissertation mainly focuses on elaborating the research contributions of the EuroSys'19 [26] & ASPLOS'21 [45] conference papers and an under-submission PeerJ Computer Science journal article [53], which contributed to tasks **A**, **B**, and **C** discussed above. All other research contributions of this doctoral research project (e.g., publications and conference presentation videos) are open access, and an interested reader can access them online for free.

### 1.5.2 Publications

This doctoral research project resulted in the following publications: [26, 27, 45, 46, 61, 53, 38, 68, 28, 29, 30, 31, 32, 47, 33, 34, 35, 36, 69, 70, 37, 62, 63, 64, 65, 66, 67]. These were part of tasks **A**, **B**, **C**, and **D** (discussed above). The works below marked with ★ are *included* in this doctoral dissertation; the rest of the publications are *not* discussed further in this dissertation.

#### Conference Papers

- C1** ★ **Alireza Farshin**, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. “Make the Most out of Last Level Cache in Intel Processors,” in *Proceedings of the Fourteenth European Conference on Computer Systems (EuroSys '19)*. ACM, March 2019. doi:10.1145/3302424.3303977. (**A**)
- C2** **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 *2020 USENIX Annual Technical Conference (ATC '20)*. USENIX Association, July 2020. ISBN 978-1-939133-14-4. (**A**)
- C3** ★ **Alireza Farshin**, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. “PacketMill: Toward per-core 100-Gbps Networking,” in *Proceedings of the 26<sup>th</sup> ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, April 2021. doi:10.1145/3445814.3446724. (**B**)
- C4** Hamid Ghasemirahni, Tom Barbette, Georgios Katsikas, **Alireza Farshin**, Amir Roozbeh, Massimo Gironi, Marco Chiesa, Gerald Q. Maguire Jr., and

Dejan Kostić. “Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets,” in *19<sup>th</sup> USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. USENIX Association, April 2022. ISBN 978-1-939133-27-4. 🏆 **Community Award Winner! (D)**

## Journal Articles

- J1** ★ **Alireza Farshin**, Luigi Rizzo, Khaled Elmeleegy, and Dejan Kostić. “Overcoming the IOTLB Wall for Multi-100-Gbps Linux-based Networking,” in *PeerJ Computer Science*. (Under Submission) (C)

## Licentiate Thesis and Master’s Thesis Supervisions

- T1** **Alireza Farshin**. 2019. “Realizing low-latency Internet Services via low- level Optimization of NFV Service Chains.” TRITA-EECS-AVL-2019:41 ISBN 978-91-7873-175-6. Licentiate Thesis. KTH. Available at: urn:nbn:se:kth:diva-249664. (A)
- T2** Omar Giordano. 2021. “Design and Implementation of an Architecture-aware In-memory Key-Value Store.” Masters Thesis. KTH. Available at: urn:nbn:se:kth:diva-291213. Academic supervisor: **Alireza Farshin**. (A)

## Patent Applications

- P1** Amir Roozbeh, **Alireza Farshin**, Dejan Kostić, and Gerald Q. Maguire Jr. “Methods and Nodes for Handling Memory,” PCT Application PCT/SE2018/051311, Filed 2018-12-13. (A)
- P2** Amir Roozbeh, Dejan Kostić, Gerald Q. Maguire Jr., and **Alireza Farshin**. “Memory Allocation in a Hierarchical Memory System,” PCT Application PCT/SE2019/050596, Filed 2019-06-20. (A)
- P3** Amir Roozbeh, **Alireza Farshin**, Dejan Kostić, and Gerald Q. Maguire Jr. “Methods and Devices for Controlling Memory Handling,” PCT Application PCT/SE2020/050161, Filed 2020-02-13. (A)
- P4** Amir Roozbeh, **Alireza Farshin**, Dejan Kostić, and Gerald Q. Maguire Jr. “Entities, System and Methods Performed Therein for Handling Memory Operations of an Application in a Computer Environment,” PCT Application PCT/SE2019/050948, Filed 2019-10-02. (A)
- P5** Chakri Padala, Amir Roozbeh, **Alireza Farshin**, Dejan Kostić, and Gerald Q. Maguire Jr. “Efficient Loading of Code Portions to a Cache,” PCT Application PCT/SE2020/050527, Filed 2020-05-22. (A)

- P6** Amir Roozbeh, **Alireza Farshin**, Dejan Kostić, Gerald Q. Maguire Jr., Hamid Ghasemirahni, and Tom Barbette. “Reordering and Reframing Packets,” PCT Application PCT/IB2020/054991, Filed 2020-05-26. **(B) & (D)**
- P7** Amir Roozbeh, **Alireza Farshin**, Dejan Kostić, and Gerald Q. Maguire Jr. “Method and System for Efficient Input/Output Transfer in Network Devices,” PCT Application PCT/SE2020/051107 & PCT/SE2020/051108, Filed 2020-11-20. **(A)**
- P8** Amir Roozbeh, **Alireza Farshin**, Tom Barbette, Dejan Kostić, and Gerald Q. Maguire Jr. “Methods and Systems for Efficient Metadata and Data Delivery between a Network Interface and Applications,” PCT Application PCT/IB2021/052976, Filed in 2021-04-09. **(B)**
- P9** Amir Roozbeh, **Alireza Farshin**, Chakri Padala, Dejan Kostić, and Gerald Q. Maguire Jr. “System, Method, and Apparatus for Fine-grained Control of I/O Data Placement in Memory Subsystem,” PCT Application PCT/SE2021/050803, Filed in 2021-08-17. **(A)**
- P10** Amir Roozbeh, Chakri Padala, and **Alireza Farshin**. “System and Method for Cache pooling and Efficient Usage and I/O Transfer in Disaggregated and Multi-Processor Architectures via Processor Interconnect,” PCT Application PCT/SE2021/051016, Filed in October 2021. **(A)**
- P11** Amir Roozbeh, **Alireza Farshin**, Marco Chiesa, and Fabio Luciano Verdi. “System and Method for Accurate Traffic Monitoring on Multi-Pipeline Switches,” PCT Application PCT/EP2021/084572, Filed in December 2021. **(D)**
- P12** Amir Roozbeh, **Alireza Farshin**, and Dejan Kostić. “System and Method for Organizing Physical Queues into Virtual Queues,” PCT Application PCT/EP2022/051103, Filed in January 2022. **(D)**
- P13** Amir Roozbeh, **Alireza Farshin**, Marco Chiesa, Tom Barbette, and Dejan Kostić. “Apparatus, System, and Methods for Sliced Accelerated Packet Processing at Terabit-per-second Networking.,” US Provisional Application, Filed in May 2022. **(D)**
- P14** Amir Roozbeh, Chakri Padala, **Alireza Farshin**, Dejan Kostić, and Gerald Q. Maguire Jr. “Processing Unit, Packet Handling Unit, Arrangement and Methods for Handling Packets,” PCT Application PCT/SE2022/050710, Filed in July 2022. **(D)**

**P15** Amir Roozbeh, **Alireza Farshin**, Marco Chiesa, Dejan Kostić, and Hamid Ghasemirahni. “Hint Entity, Receiver Node, System and Methods Performed Therein for Handling Data in a Computer Environment,” PCT Application PCT/SE2022/051036, Filed in November 2022. (D)

**P16** Amir Roozbeh, **Alireza Farshin**, and Marco Chiesa. “Entity and Method Performed Therein for Handling Packets in a Computer Environment,” US Provisional Application, Filed in November 2022. (D)

## 1.6 Dissertation Hypothesis

The scope of this doctoral research project is beyond that of a single doctoral dissertation, as it focuses on many different tasks & domains. This doctoral dissertation contains only a subset of the work done during this broader project. More specifically, this dissertation focuses on three main publications (*i.e.*, **C1**, **C3**, and **J1** that are marked with ★ in Section 1.5.2). The outcome of these publications tries to verify a unified hypothesis as follows:

*“If security researchers could exploit cache memories & their characteristics (e.g., access time differences) to perform side-channel attacks & gather information, could we use their characteristics to understand how to do packet processing faster?”*

Caching instructions & data have been heavily studied to improve the performance of numerical computations, where the knowledge about the access patterns to vectors and instructions is used for cache prefetching. However, the same techniques are not entirely valid for network I/O. Therefore, this dissertation exploits this opportunity to examine caching in the context of network I/O where network speeds were increasing at a more rapid pace than processors and memories. The outcomes of this doctoral dissertation successfully verify our hypothesis in three different domains as follows:

- The first paper [26] digs into the details of cache organization and introduces cache-slice awareness to exploit access time differences to different portions of LLC in Intel processors in order to improve the tail latency of high-speed NFs;
- The second paper [45] shows that one can use modern compilation techniques coupled with a cache-aware layout of (meta)data to improve networking performance using kernel-bypass frameworks; and

- The third paper [53] shows that understanding the IOMMU cache (*aka* IOTLB) can improve networking performance (while maintaining security) even when processing TCP traffic using the Linux kernel (rather than relying on the kernel-bypass used in the first & second papers). Additionally, it brings out a clear error in the way that memory is being allocated for network I/O by the existing Linux kernels, *i.e.*, they still use 4-KiB pages decades after the first kernel while the network interfaces have become almost  $10^5 \times$  faster; see Section 1.2.

To verify our hypothesis, we use a unified method to (*i*) make careful measurements & understand the details and then (*ii*) craft a cache-related solution that exploits the knowledge to improve packet processing.

## 1.7 Research Impact

This research project aims to realize low-latency Internet services on multi-hundred-gigabit-per-second commodity hardware by optimizing packet processing. There are three aspects of my research: (*i*) professional impact, (*ii*) personal impact, and (*iii*) sustainable impact. This section briefly summarizes the first two aspects and the next section elaborates on the sustainable impacts of this project.

**Professional impact.** Although I mainly focused on one specific subdomain, *i.e.*, the performance of virtually deployed NFs, my contributions (*i.e.*, results, findings, and techniques) could be equally useful in other subdomains of the (networked) system’s research. This generality occurs because my research goal was to use available resources more efficiently and *squeeze* every bit of performance from the existing network & system hardware; therefore, these contributions are useful for other performance-sensitive contexts interested in nanosecond- and microsecond-level improvements. In addition to the previously enumerated publications (see Section 1.5.2), my doctoral research proposal led to a Google Ph.D. Fellowship (2021) in Systems and Networking. Moreover, this research project has been featured in the following news articles:

- N1 “Maximizing COTS hardware performance with CPU memory management,” Ericsson Blog, 22 March 2019.
- N2 “Four researchers are making Intel processors more efficient,” KTH, 28 March 2019.
- N3 “Researchers in Sweden mine cache of Intel processors to speed up data packet processing,” AlphaGalileo, 7 May 2019.



- N4 “A positive side effect of cache monitoring – doing good rather than evil,” KTH EECS, 19 June 2019.
- N5 “Researchers mine cache of Intel processors to speed up data packet processing,” Tech Xplore, 8 July 2019.
- N6 “How to deploy multi-hundred-gigabit networking on commodity hardware,” Ericsson Blog, 16 April 2021.
- N7 “KTH researchers behind breakthrough set to halve data center energy consumption,” KTH, 14 June 2022.
- N8 “Optimerar cacheminnet för snabbare internetjänster,” Framtidens Forskning, 17 June 2022.
- N9 “Packet reordering: The key to efficient high-speed packet processing” Ericsson Blog, 18 August 2022.

**Personal impact.** During my doctoral studies, I have studied high-performance networking hardware & software, looked for performance inefficiencies, and proposed techniques to optimize their performance for latency-critical network services. I chose to work on this topic because of its multidisciplinary nature that exploits the synergy of networked systems, computer architecture, and compiler optimizations. This has leveraged my background in Electrical Engineering while allowing me to gain experience in new domains, which is intellectually satisfying.

## 1.8 Sustainability, Ethical, and Security Aspects

This section discusses the contributions of this doctoral research project to sustainable development and it also elaborates on ethical as well as security aspects related to this work.

### 1.8.1 Sustainability

We believe that this research will have a positive impact on sustainability as defined in the 1987 Brundtland Report:

*“Development that meets our needs of the present without compromising the ability of future generations to meet their own needs.”*

— Brundtland Report, 1987 [71]

The contributions of this work contribute to three different domains of sustainability; these are: environmental, societal, and economic sustainability. These contributions are described below.

**Environmental Sustainability.** One of the objectives of this doctoral project is to achieve the maximum performance of currently available hardware. By accomplishing this objective, Internet services, especially NFV service chains, can be implemented on fewer CPU cores (or fewer servers), which will lessen power consumption, consequently decrease carbon dioxide emission, hence reduce the ecological footprint of these services.

**Societal Sustainability.** In this work, we try to use different optimization techniques to realize low-latency Internet service. Doing so will directly affect people's lives, which can increase their productivity and their quality of life. Additionally, improving the performance of existing hardware also increases the processing capability of existing devices. Consequently, many services and products might become cheaper and more affordable, thereby leading to greater equity in society.

**Economic Sustainability.** By employing the contributions of this research, cloud & service providers can use their resources more efficiently. Hence, they can serve more users with their current deployed infrastructure, which not only postpones the need for expansion but also reduces their CAPEX and OPEX.

## 1.8.2 Ethical Aspect

In the first part of this work, we exploit knowledge (*i.e.*, Intel's Complex Addressing) that has been shown by others to be usable for malicious purposes, such as performing different types of cache attacks, building precise covert channels, and accessing users' valuable information. In contrast, we use this knowledge for benevolent reasons, that is, to improve the performance of processors.

## 1.8.3 Security Aspect

This work improves the efficiency and performance of existing hardware. Consequently, Internet services can be deployed on fewer servers. Therefore, these services might become more susceptible to cyberattacks. Additionally, a more in-depth study of Intel's Complex Addressing, done in the first part of this dissertation, might lead to the development of new cache attacks. Furthermore, our IOMMU-related work enhances I/O security by offering a higher-performance solution for the Linux kernel.

## 1.9 Dissertation Organization

This section outlines this doctoral dissertation's structure, where it discusses three main publications\* (see Section 1.5.1), each in a separate chapter. Each chapter is standalone and provides its necessary background and related works. The rest of this dissertation is organized as follows:

- **Chapter 2** exploits the cache architecture in recent Intel processors to improve the performance of I/O intensive networking applications.
- **Chapter 3** employs low-level optimization techniques to produce a whole-stack optimized binary for NFs.
- **Chapter 4** studies the performance implications of IOMMU at 200 Gbps and explores potential solutions in the Linux kernel to maximize IOTLB hits.
- **Chapter 5** highlights the most relevant works related to this dissertation.
- **Chapter 6** concludes this dissertation and proposes some directions for future research.

---

\*Some parts of the text are adapted from EuroSys'19 [26] & ASPLOS'21 [45] conference papers, an under-submission PeerJ Computer Science journal article [53], and my previously published licentiate thesis [69]. The authors of these articles retained the copyright and have given their joint approval for the parts that appear in this dissertation.

## Chapter 2

# Improving Hardware Cache Utilization

A computer system is typically comprised of several CPU cores connected to a memory hierarchy. The memory hierarchy is composed of progressively slower, bigger, and cheaper memories, see Figure 2.1. This hierarchy begins with CPU registers (*i.e.*, fastest); followed by cache memories, located on the processor's die, and then continues with the main memory (*i.e.*, Dynamic Random Access Memory (DRAM)) and secondary memories such as Non-Volatile Memory (NVM) and storage (*e.g.*, Solid State Device (SSD) or Hard Disk Drive (HDD)).

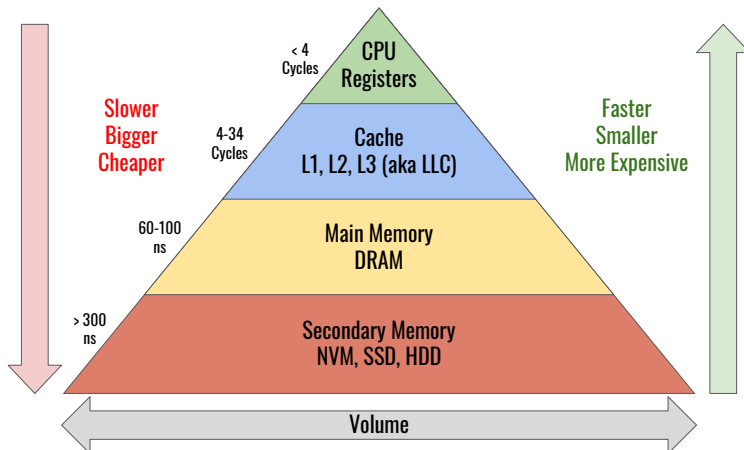


Figure 2.1: Memory hierarchy.

As mentioned previously, the time budget for processing packets at the link rate of a hundred gigabits per second can be as small as 6.72 ns, which makes the cache a valuable resource since every access to the DRAM or secondary memories is quite expensive in terms of latency (as quantified in Figure 2.1). This motivated us to investigate the architecture of cache memory in current processors to identify opportunities to better exploit caching. This chapter starts by explaining cache architecture. Next, it scrutinizes the Non-Uniform Cache Architecture (NUCA) of LLC in Intel processors. Section 2.4 describes how the identified opportunities were used to improve the performance of packet processing in time-critical NFV service chains. In Section 2.3, these performance improvements are quantified. Section 2.4 describes some of the limitations of the proposed improvements. Section 2.5 summaries related work. The chapter ends with a summary in Section 2.6.

## 2.1 Cache Architecture

Modern processors implement a hierarchical cache as a L1, L2, and level three (L3), also known as the LLC. L1 and L2 caches are typically private to each core, while LLC is shared among all CPU cores on a chip. The L1 cache is typically based on the Harvard architecture, *i.e.*, data and instructions are stored in separate storage. However, other cache levels (*i.e.*, L2 and L3) are usually designed according to the von Neumann architecture, *i.e.*, data and instructions are stored in the same memory. Processors organize caches with a minimum unit of a *cache line*. Moreover, processors use different cache placement policies: (i) direct mapped, (ii) n-way set associative, and (iii) fully set associative. Figure 2.2 depicts these policies, in which from left to right the efficiency increases, but they become slower and more expensive.

Modern processors typically have a 64-B cache line. In addition, they are n-way set associative, which means “n” cache lines form one set. The relationship between the CPU’s cache size, cache associativity, and the number of cache sets is:

$$\text{Number of Sets} = \frac{\text{Cache Size}}{\text{n-way Associativity} \times \text{Cache Line Size}}$$

When a CPU needs to access a specific memory address, it checks the different cache levels to determine whether a cache line containing the target address is available. If the data is available in any cache level (*aka* a cache hit), the memory access will be served from that level of cache. Otherwise, a cache miss occurs and the next level in the cache hierarchy will be examined for the target address. If the target address is unavailable in the LLC, then the CPU requests this data from

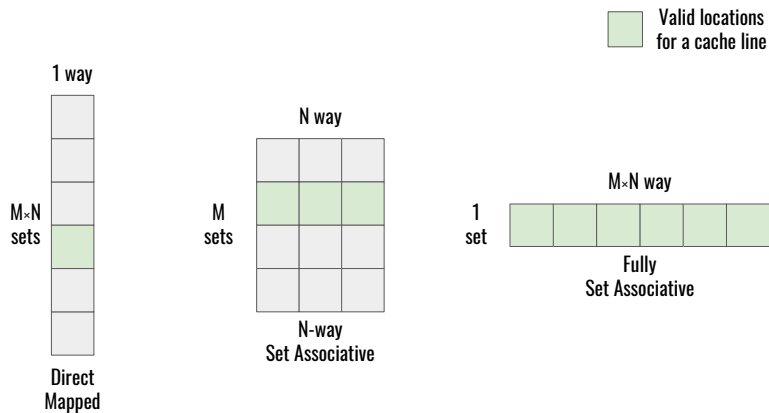


Figure 2.2: Cache placement policies. Green boxes show the valid locations for a cache line in each policy.

the main memory. A CPU can implement different cache replacement policies (e.g., Least Recently Used (LRU)) to evict cache lines in order to make room for subsequent requests.

The physical memory addresses are logically divided into different portions (based upon an offset, set index, and tag, see Figure 2.3). The set index defines which set in the cache can hold the data corresponding to a given address. By concurrently comparing the tag portion of a given address with the tag portion of the address of the cache lines available in one set, we can determine whether the data corresponding to that address is present in the cache or not.

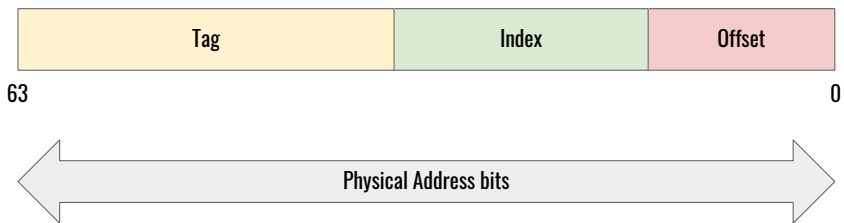


Figure 2.3: Physical address mapping within cache hierarchy.

### 2.1.1 Last Level Cache (LLC)

Since the introduction of the Sandy Bridge architecture in Intel processors, LLC is not unified, which is known as NUCA [72, 73]. Intel re-designed the LLC by dividing the LLC into multiple slices [74]. In the first version of this re-design the CPU cores and all LLC slices are interconnected by a bi-directional ring bus, see Figure 2.4.

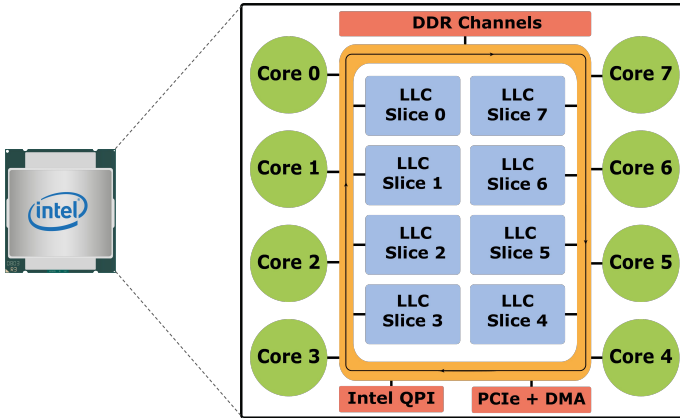


Figure 2.4: Non-uniform Cache Architecture in Intel processors.

Intel uses an undocumented hash-based scheme, called Complex Addressing, that receives the physical address as an input and determines which slice should be associated with that particular address. There have been many attempts to find the slice mapping and reverse-engineer Intel's Complex Addressing [75, 76, 74, 77, 78, 79]. For instance, Cl  mentine Maurice, *et al.*, [79] use the uncore performance monitoring unit (*e.g.*, C-Box counters) to find the mapping. We employed the same technique to find the mapping for a CPU with the Haswell architecture. The results show that the mapping changes for every 64-B cache line (see Figure 2.5).

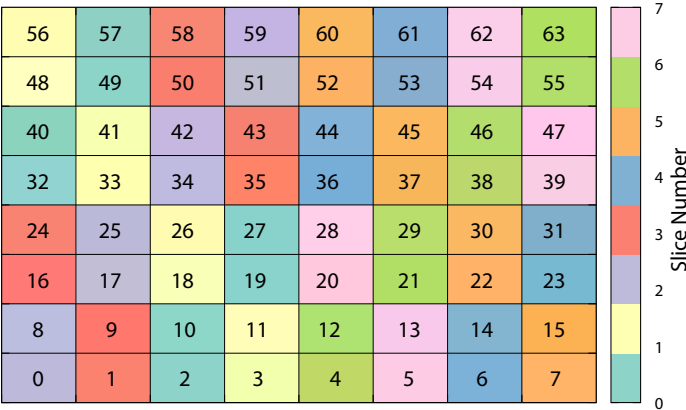


Figure 2.5: Mapping for different cache lines in a 4-KiB page.

### 2.1.2 Opportunities

Due to the difference in paths from each core to the different slices in the LLC, we expected to experience a difference in access times. To verify this hypothesis, we designed a test application to measure the number of cycles needed to access cache lines residing in different slices of LLC from a single core. All of these measurements were made on a system running Ubuntu 16.04 (Linux kernel-4.4.0-104) with 128 GiB of DRAM and two Intel Xeon-E5-2667-v3 processors (*i.e.*, Haswell architecture) running at 3.2 GHz.

Figure 2.6 shows the results for core 0 when the cache lines are read from different LLC slices. These results suggest that LLC access times are *bimodal* since the caches are located on a physical ring bus, *e.g.*, accessing slices 0, 2, 4, and 6 require fewer CPU cycles. Additionally, these results show that reading data from the appropriate slice (that is closest to the CPU core) can save up to  $\sim 20$  cycles in each access to LLC, which for the given clock speed is equal to 6.25 ns. This saving could be aggregated, as cache misses in higher levels are inevitable for some real-world applications. The aggregated savings can be used to execute useful instructions instead of stalling, *i.e.*, waiting for data to be available to the CPU. Furthermore, the amount of saving is comparable with the time budget for processing small packets being sent at 100 Gbps (*i.e.*, 6.72 ns).



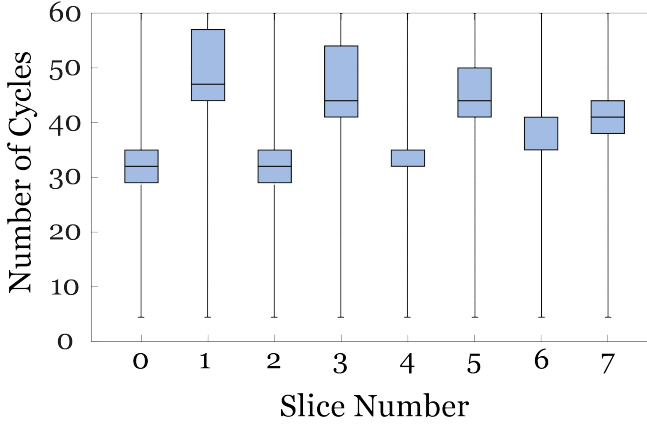


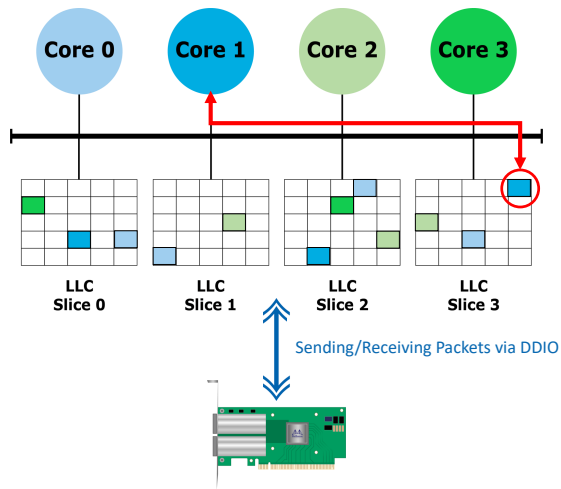
Figure 2.6: Access time to different LLC slices from core 0 in Xeon-E5-2667 v3 (Haswell).

We consider the NUCA characteristics of LLC as an opportunity to improve cache performance/utilization. Next, we will discuss how the difference in the access time can be exploited to improve the performance of packet processing done by the virtual NFs.

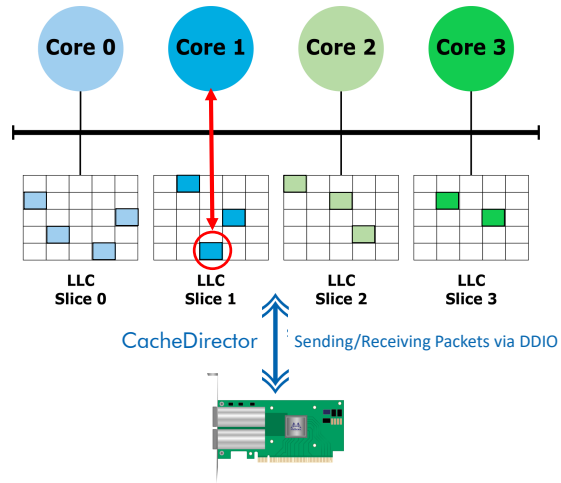
## 2.2 CacheDirector

This section advances state-of-the-art networking solutions by exploiting the NUCA characteristic of Intel’s LLC design in userspace packet processing. We proposed CacheDirector [26], a network I/O solution that extends DDIO [27] and sends each packet’s header directly to the appropriate slice in the LLC; hence, the CPU core that is responsible for processing a packet can access the packet header in fewer CPU cycles. To show the benefits of CacheDirector, we implement this solution as an extension to the DPDK [80]. Note that the concept behind CacheDirector could be applied to other packet processing frameworks. We used DPDK as it was easier to prototype CacheDirector in userspace, but the same approach could be used for kernel network stack optimizations (*e.g.*, eXpress Data Path (XDP) [81]).

Figure 2.7 shows the difference between DDIO and CacheDirector. DDIO does not take into account the complex addressing and it places packets randomly in different slices, which results in inefficient cache accesses as the packet might be placed in the furthest slice, see Figure 2.7a.



(a) DDIO.



(b) CacheDirector.

Figure 2.7: DDIO vs. CacheDirector. Each box represents a packet, the color of which is associated with the core that is responsible for processing it. The red lines highlight the distance between the packets and their processing cores.

CacheDirector improves cache performance by making DDIO slice-aware and placing the packet's header in the appropriate LLC slice. By doing so, when the packet's header is unavailable in higher cache levels (*i.e.*, L1 and L2) but the packet's header exists in LLC, then it can be accessed from the closest slice, see Figure 2.7b.

This section continues with some background about DPDK & its memory management libraries and then elaborates the design principles & implementation of CacheDirector.

## 2.2.1 Data Plane Development Kit (DPDK)

DPDK is a userspace network I/O framework, first developed by Intel. DPDK enables direct communication between applications and network devices without involving the Linux network stack. Additionally, DPDK offers a set of components and libraries through its Environment Abstraction Layer (EAL) that can be used by DPDK-based applications for packet processing. Table 2.1 shows the DPDK core components responsible for memory management.

Table 2.1: DPDK memory management libraries.

Component	Description
Ring Manager ( <i>librte_ring</i> )	Provides a ring data structure to manage any sort of queue. The ring manager is used by the Memory Pool Manager to store buffers of network packets and it can be used to realize a general communication mechanism between cores or between applications.
Memory Pool Manager ( <i>librte_mempool</i> )	Allocates pools of objects as a memory pool from a specific amount of memory.
Network Packet Buffer Management ( <i>librte_mbuf</i> )	Manages memory buffers (mbufs). These mbufs are created at DPDK initialization time, stored in a mempool, and then used by the DPDK application to hold data ( <i>e.g.</i> , network packets).

During DPDK initialization, the NIC is unbound from the Linux kernel (*e.g.*, Intel NICs) or it uses bifurcated drivers (*e.g.*, NVIDIA/Mellanox drivers) to make

userspace interaction with the NIC possible. After initialization, one or more memory pools are allocated from hugepage(s) in memory. These memory pools (*aka* mempools) include fixed-size elements (objects), created by the *librte\_mempool* library. DPDK's memory management is Non-Uniform Memory Access (NUMA) aware and it applies memory alignment techniques to improve performance. In DPDK, network packets are represented by packet buffers (mbufs) through the *rte\_mbuf* structure. Buffer management allocates and initializes mbufs from available elements in mempools. Each mbuf contains metadata, a fixed-size headroom, and a data segment (used to store the actual network packet), see Figure 2.8. The metadata includes a message type, length, starting address of the data segment, and userdata. It also contains a pointer to the next buffer. This pointer is needed when using multiple mbufs to handle packets whose size is larger than the data area of a single mbuf. After initializing a driver for all of the receiving and transmitting ports, one or more queues are configured for receiving/transmitting network packets from/to the NIC. These queues are implemented as ring buffers from the available mbufs in mempools. Finally, the receiving ports are set with specific Media Access Control (MAC) addresses or to promiscuous mode and then DPDK is ready to send and receive network packets.

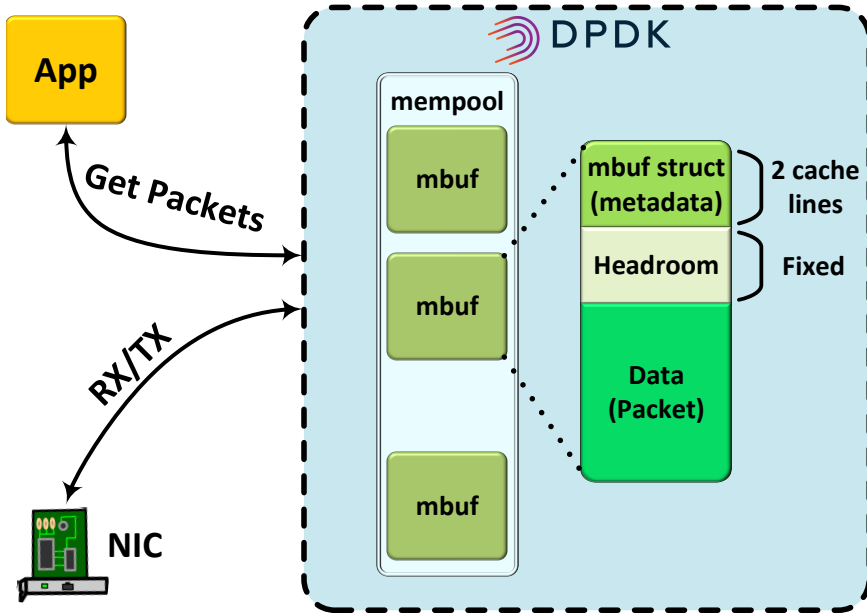


Figure 2.8: Simplified memory management in DPDK: the size of the mbuf struct is equal to two cache lines (*i.e.*, 128 B) and the headroom size is fixed (default value: 128 B) [26].

Communication between an application and NIC is managed in DPDK through a Poll Mode Driver (PMD). PMD provides Application Programming Interfaces (APIs) and uses polling to eliminate the overhead of interrupts. PMD enables DPDK to directly access the NIC's descriptors for both receiving and transmitting packets. To receive packets, DPDK fetches packet(s) from the NIC's Receive (RX) descriptor into its receiving queues when the application periodically checks for new incoming packets. To send packets, the application places the packets into transmitting queues from which DPDK takes packet(s) and pushes them into the NIC's TX descriptor, see Figure 2.8.

## 2.2.2 Design and Implementation

The main objective of CacheDirector is to bring awareness of Intel's LLC Complex Addressing to DPDK. More specifically, incoming packets are placed into the appropriate LLC slice, thus the core responsible for processing these packets can access them faster. To achieve this goal, the buffer and memory pool manager in DPDK initialize the mbufs so that they will be mapped to the appropriate slice.

However, implementing this idea faces some challenges. These challenges and the ways CacheDirector tackles them are described below.

**Small chunks.** Since Intel's LLC Complex Addressing maps almost every cache line (64 B) to a different LLC slice, it is impossible to send large packets to the appropriate LLC slice without packet fragmentation. To deal with this challenge, CacheDirector ensures that at least the *first* 64 B of each packet, containing the packet's header, are mapped to the appropriate LLC slice by introducing *dynamic* headroom to the mbufs. As there are some applications that might access a different part of the packet more frequently (*e.g.*, Virtual Extensible LAN and DPI), CacheDirector can be configured to map any other 64-B portion of the packet to the appropriate LLC slice.

**Dynamic headroom.** CacheDirector can dynamically change the amount of headroom such that the starting address of the data area of a mbuf is at an address which is mapped to the desired LLC slice for each CPU core using that mbuf at run-time, see Figure 2.9. However, since DPDK assumes that the headroom is fixed (*e.g.*, 128 B), setting the headroom size to values greater than this will result in a reduction of the data area of mbufs (the default size is 2 KiB). If the remaining data area is less than the packet's size, then DPDK uses multiple mbufs for one packet, which might be an expensive operation as an application needs to traverse a linked list to access the whole packet. To tackle this, we must find the maximum amount of headroom required for mbufs in order to ensure that no adverse shrinkage of the data area will happen. Therefore, we performed an experiment in which ~12.3 million packets from a campus trace were sent to a server and then calculated the distribution of the dynamic mbufs' headroom sizes. Figure 2.10 shows the histogram of these values. The median of the distribution is 256 B; 95% of the values are less than 512 B; and the maximum needed headroom size is 832 B.

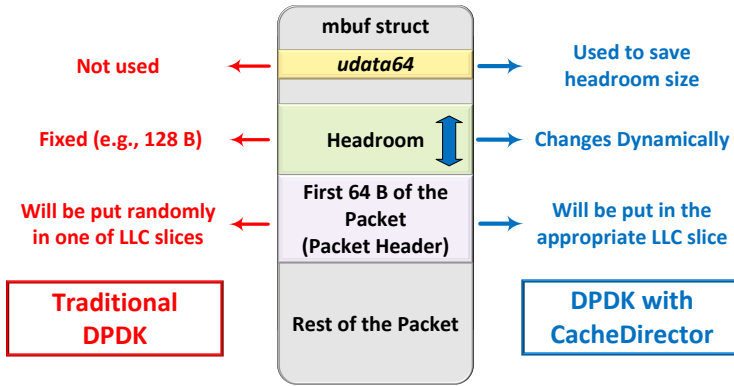


Figure 2.9: CacheDirector changes to the mbuf structure [26].

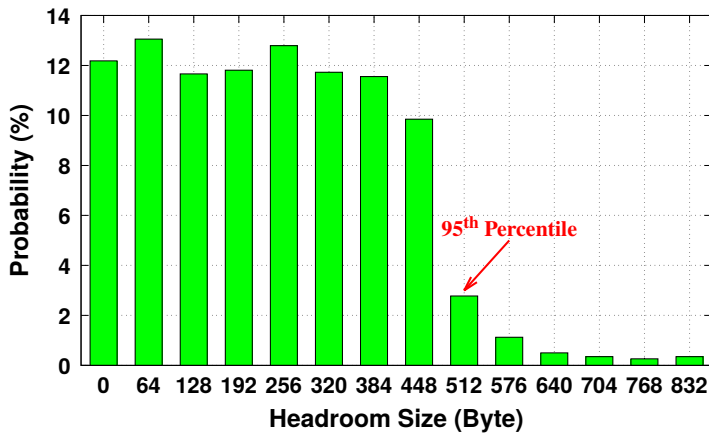


Figure 2.10: Distribution of headroom size for ~12.3 Million Packets + 95<sup>th</sup> percentile of the distribution.

Examining this distribution, we set the default headroom size to 832 B to ensure that the maximum desired data area is available - but this is at the cost of extra memory usage. Note that extra memory usage does *not* affect performance (*e.g.*, does not increase TLB misses), as memory allocation is done by using hugepages. The distribution of dynamic headroom size might vary in different micro-architectures. However, differences in the distribution and memory wastage are not a big concern, as they can be eliminated by handling the mbuf allocation at the application level (*e.g.*, in FastClick [24]). For instance, an application can allocate one large mempool containing mbufs. Then, it can sort mbufs across multiple mempools, each of which is dedicated to one CPU core, based on their LLC slice mappings. However, we implemented CacheDirector in DPDK as an application-agnostic solution.

**Ensuring the appropriate headroom size.** Since a mbuf can be used by multiple cores, CacheDirector must ensure that the headroom size is set to the appropriate value so that the first 64 B of the data segment is mapped to the appropriate LLC slice for the CPU core that will be fetching a packet from the NIC. Therefore, at run-time, CacheDirector sets the actual headroom size just before giving the address to the NIC for DMA-ing packets. We implemented this as a part of userspace NIC drivers in DPDK. For example, when CPU core 5 wants to fetch packets from a NIC, the NIC driver calculates the headroom such that the data segment of each mbuf is in slice 5. It is worth noting that this step is unnecessary when mbufs are sorted at the application level.

**Mitigating calculation overhead.** To avoid unnecessary run-time overhead, we calculate the headroom needed to place the data segment of each mbuf into specific LLC slices during DPDK's initialization phase. These values are saved in the userdata part (*i.e.*, *udata64*) of the mbuf structure (metadata), see Listing 2.1.

Later, the NIC driver sets the actual headroom size based on the CPU core that will be fetching a packet from the NIC by using these saved values. For example, when CPU core 2 wants to fetch data from the NIC, the NIC driver looks into the userdata part of each mbuf and sets its headroom according to the pre-calculated value for slice 2. It is worth mentioning that we save the number of cache lines instead of the actual headroom size and since 832 (the maximum required headroom size) is 13 cache lines, 4 bits is sufficient for each core. Therefore, our solution would be scalable for up to 16 cores on one CPU, as *udata64* is 64 bits in size.



**Listing 2.1** Precalculated headroom sizes for different LLC slices in *udata64*.

```
union {  
    /* < Can be used for external metadata */  
    void *userdata;  
    /* < Allow 8-byte userdata */  
    uint64_t udata64;  
    struct {  
        uint8_t slice0; /*< Save headroom for slice0*/  
        uint8_t slice1; /*< Save headroom for slice1*/  
        uint8_t slice2; /*< Save headroom for slice2*/  
        uint8_t slice3; /*< Save headroom for slice3*/  
        uint8_t slice4; /*< Save headroom for slice4*/  
        uint8_t slice5; /*< Save headroom for slice5*/  
        uint8_t slice6; /*< Save headroom for slice6*/  
        uint8_t slice7; /*< Save headroom for slice7*/  
    } slice_off;  
};
```

## 2.3 Evaluation

In this section, we demonstrate CacheDirector’s effectiveness by evaluating the performance of DPDK with and without CacheDirector functionality for two different types of applications in NFV systems: a simple forwarding (Section 2.3.2) and a stateful service chain (Section 2.3.3).

### 2.3.1 Context of the Evaluation

The evaluation will use the testbed described in Section 2.3.1.1 to evaluate two applications using the data described in Section 2.3.1.2 and the measurement method described in Section 2.3.1.3.

#### 2.3.1.1 Testbed

A testbed, based on the design described in [82], is used in our experiments. In this setup, we use a simple desktop machine as a Plain Orchestration Service (pos) for deploying, configuring, and running experiments as well as collecting and analyzing the data (see Figure 2.11). In addition, we have connected two identical servers, one as Load Generator (LoadGen) and another one as Device under Test (DuT) which is running a virtual NF. These two machines each have dual Intel Xeon E5-2667 v3 processors, 128 GiB of DRAM, and a NVIDIA/Mellanox ConnectX-4 MT27700 card. The LoadGen has a dual port NVIDIA/Mellanox NIC. In all of the experiments on DuT, hyper-threading is disabled, and one CPU socket (including 8 CPU cores) on which we run experiments is isolated. The Operating System (OS) is Ubuntu 16.04.4 with Linux kernel v4.4.0-104. In order to implement the CacheDirector functionality in DPDK, we extended DPDK v18.05 and we disabled Vectorized PMD (vPMD).

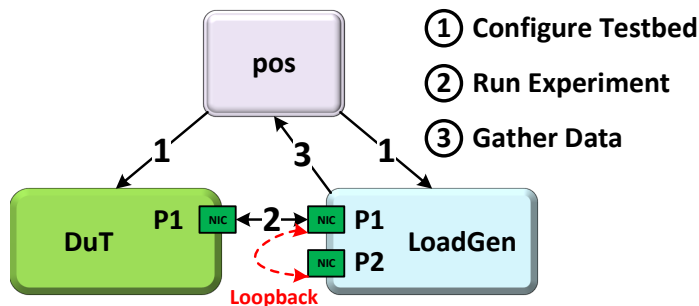


Figure 2.11: Experiment setup [26].

### 2.3.1.2 NFV

To see the impact of CacheDirector on NFV service chains, we evaluate the performance of Metron [16], a state-of-the-art platform for NFV, in the presence of CacheDirector. We implemented two different applications, a simple forwarding and a stateful service chain, using Metron’s extension of FastClick [24]. In our experiments, we use a campus trace\*, in which 26.9 % of frames are smaller than 100 B; 11.8 % are between 100 & 500 B; and the remaining frames are more than 500 B.

### 2.3.1.3 Measurement Method

For measuring end-to-end latency, we follow the black-box approach explained in [82], where data is collected on the egress/ingress port of the LoadGen to measure throughput and latency. To do so, the LoadGen writes a timestamp in each packet’s payload and sends the packet to the DuT which is running a virtual NF. After processing the packets, DuT sends the packets back to the LoadGen. Upon receiving each packet, the LoadGen reads the saved timestamp inside each packet’s payload and calculates throughput and the end-to-end latency for each packet. This latency is composed of three parts: queuing delay & processing time at LoadGen; link delay; and queuing delay & processing time at DuT. CacheDirector only affects the processing time of packets at the DuT and consequently the queuing delay on that side. To assess the delays *not due to the DuT*, we run a loopback experiment in which the two ports of LoadGen were interconnected back to back (P1 and P2 in Figure 2.11), *i.e.*, traffic sent from one port of LoadGen is received by the other port without any additional processing. By doing so, we are able to measure and characterize the effect of the link latency and extra overheads of the LoadGen, such as queuing and timestamping costs. From this point on, we refer to this portion of the end-to-end latency as “*loopback*” latency. We measure this latency for all configurations and we removed the minimum value of the loopback latency from the end-to-end latency in the measurements (unless stated otherwise).

## 2.3.2 Simple Forwarding

The simple forwarding application swaps the sending and receiving MAC addresses of the incoming packets and sends them back to LoadGen. This application assesses the impact of CacheDirector on stateless or low-processing network functions. We ran this application for different numbers of cores and different sets of traffic. Here we discuss only the results for two sets of traffic while using 8 cores on one CPU socket: (i) five thousand 64-B packets generated by the FastClick *RatedSource*

---

\*The same trace that was used in [16].

module at the rate of 1000 Packet Per Second (pps) and (ii) mixed-size packets from the real trace at 100 Gbps.

### 2.3.2.1 64 B Packets at Low Rate

CacheDirector only affects the processing time of packets at the DuT and consequently the queuing delay on that side. Therefore, to minimize the queuing effect and to see the pure impact of CacheDirector we send five thousand 64-B packets at a low rate (*i.e.*, 1000 pps). Figure 2.12 shows the variation of the higher percentiles of end-to-end latency for 50 such runs. This figure shows that CacheDirector reduces the higher percentile latencies by around  $\sim 20\%$  which is equal to  $1\ \mu\text{s}$  improvement per packet on the DuT side. It is important to note that even improvements of  $1\ \mu\text{s}$  must not be ignored since  $1\ \mu\text{s}$  is equal to 3200 CPU cycles for a processor running at 3.2 GHz, which could be utilized to process packets instead of stalling. This becomes even more critical for 100 Gbps links, as a server has only 6.72 ns (*i.e.*,  $\sim 17$  cycles) to process a 64-B packets before receiving a new packet.

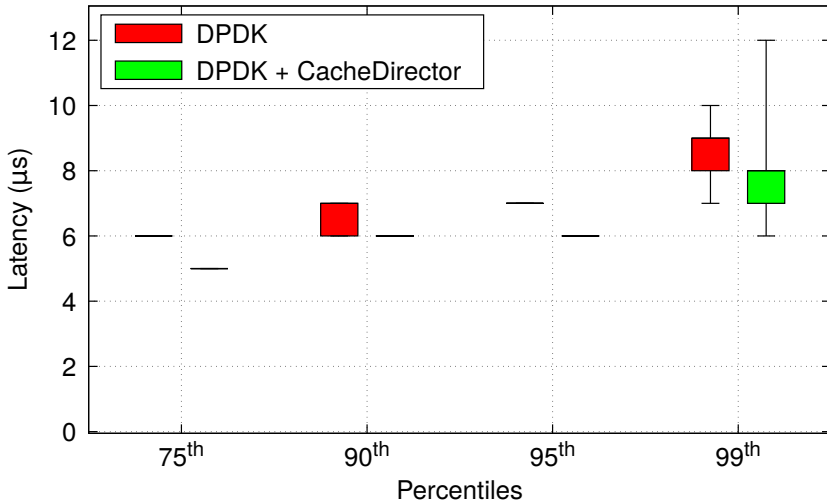


Figure 2.12: End-to-end latency without loopback latency for 64 B packets sent at the rate of 1000 pps. At each percentile, the left box refers to DPDK and the right one DPDK with CacheDirector. The minimum loopback latency is  $9\ \mu\text{s}$  [26].

### 2.3.2.2 Mixed-size Packets at 100 Gbps

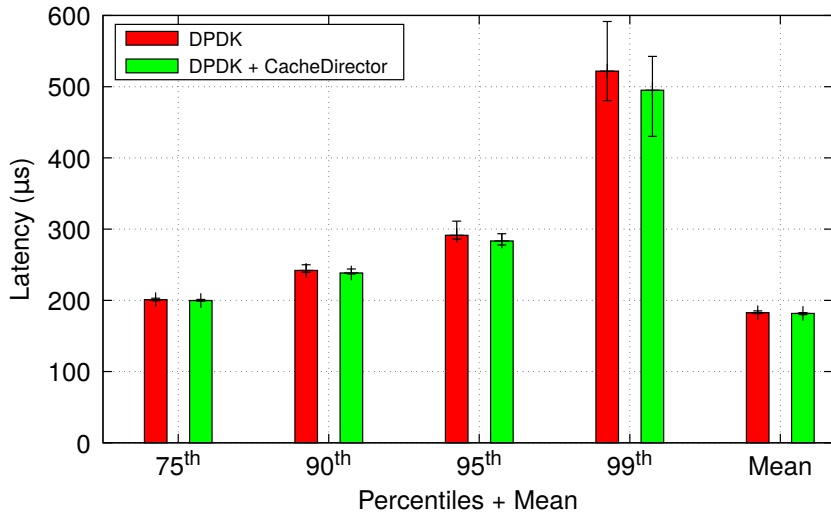
To assess CacheDirector’s impact at gigabit-per-second link speeds, we send packets from the campus trace with mixed-size packets at 100 Gbps. Figure 2.13 shows the results of 50 runs, in which we use Receive Side Scaling (RSS) [83] to distribute packets among 8 cores. The improvement in tail latencies for mixed-size packets at this rate is even greater than for 64 B packets. The top row of Table 2.2 shows the measured throughput for this experiment. The  $\sim 76$  Gbps limit for the forwarding application is due to the NVIDIA/Mellanox NIC’s limitation for packets smaller than 512 B [84] and other architectural limitations such as PCIe [49] and DDIO\*.

Table 2.2: Throughput and average improvement in throughput while sending mixed-size packets at the rate of 100 Gbps.

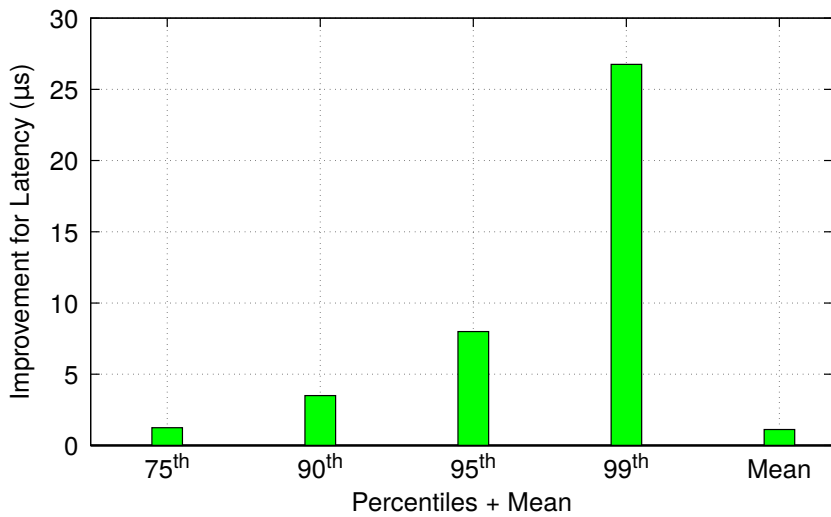
Scenario	Throughput (Gbps)	Improvement (%)
Simple Forwarding	76.58	0.0407
Router-NAPT-LB (FlowDirector with H/W offloading)	75.94	0.0359
Router-NAPT-LB (RSS w/o H/W offloading)	42.77	0.2367

---

\*DDIO uses a limited number of ways in LLC for I/O. The default number of ways is 2, which is equal to 10 % in our CPU that has 20 ways in LLC [85, 27].



(a) End-to-end latency without loopback latency.



(b) Latency improvement.

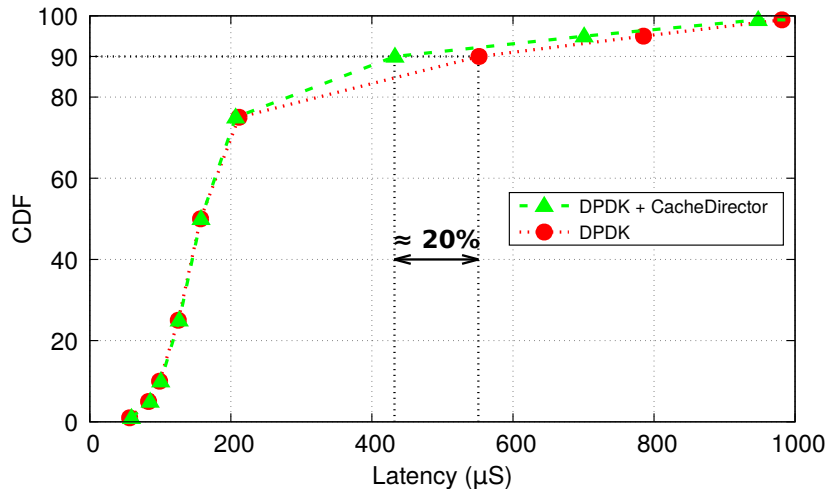
Figure 2.13: End-to-end latency and improvement for a simple forwarding application running on 8 cores with mixed-size packets at 100 Gbps with RSS. The minimum loopback latency is 495  $\mu\text{s}$ . The values show the median of 50 runs. Error bars represent 1<sup>st</sup> and 3<sup>rd</sup> quartiles [26].

### 2.3.3 Stateful Service Chain

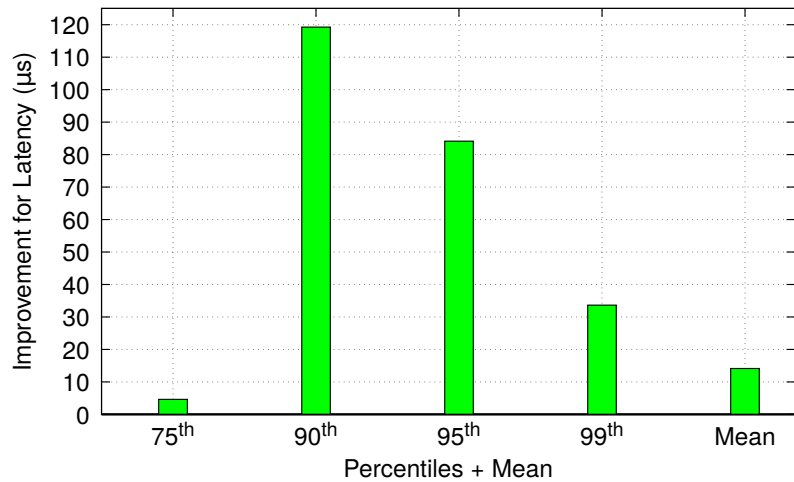
To show the practicality and benefits of CacheDirector, we ran Metron [16] to evaluate the performance of a stateful NFV service chain built from three network functions: a router, a NAPT, and LB using a flow-based Round-Robin policy. For the router, we followed Metron’s approach, in which the routing table of the router with 3120 entries is offloaded to the NVIDIA/Mellanox NIC by using its flow steering capabilities (*aka* FlowDirector technology [57]), while the remainder of the router’s functionalities is handled in software.

#### 2.3.3.1 Mixed-size Packets at 100 Gbps

For this evaluation, packets were generated using the campus trace, and the results from 50 runs are shown in Figure 2.14. The second row of Table 2.2 (on page 36) showed the throughput for this experiment. Since this service chain is more memory-intensive compared to the simple forwarding application, the gain in increased throughput becomes more tangible for the Router-NAPT-LB. Note that using FlowDirector changes the trend in latency improvements (compare Figure 2.13 and Figure 2.14). The improvements are increasing for RSS, *i.e.*, the improvement for the 99<sup>th</sup> percentile is higher than for the 90<sup>th</sup> percentile. However, the improvements for FlowDirector behave in the opposite way (*i.e.*, the performance gain is decreasing). We observed that FlowDirector reduces contention in each slice by performing better load balancing compared to RSS for the campus trace that was used. Moreover, we believe that the reason for this behavior may also be related to DDIO’s 10 % limit [85] and the slice imbalance incurred by RSS.



(a) End-to-end latency without loopback latency.



(b) Latency improvement.

Figure 2.14: CDF of end-to-end latency without loopback latency and improvement for a stateful service chain (Router-NAPT-LB) running on 8 cores while sending mixed-size packets at the rate of 100 Gbps with H/W offloading using FlowDirector. The minimum loopback latency is 495  $\mu\text{S}$ . The values show the median of 50 runs [26].



### 2.3.3.2 Tail Latency vs. Throughput

To see the impact of CacheDirector on a *not* fully-loaded system, we measured the performance of Metron with and without CacheDirector for different loads. Figure 2.15 illustrates the data points and fitted curves for this experiment. The fitted curves are defined as piecewise functions, wherein the lower (Throughput less than 37 Gbps) and higher (Throughput  $\geq 37$  Gbps) parts of data points are fitted to linear and quadratic functions, respectively. The results show that our technique slightly shifts the knee of the tail latency vs. throughput curve, which means CacheDirector would still be beneficial while the system experiences a moderate load (*i.e.*, around 50 Gbps) and before the tail latency starts growing dramatically. Equation (2.1) and Equation (2.2) show the fitted curves where  $X$  is the throughput in Gbps.

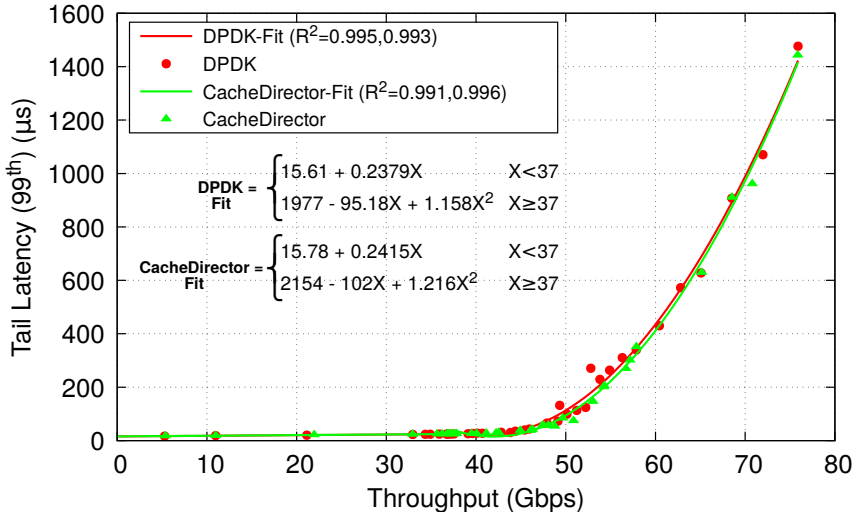


Figure 2.15: Tail latency (99<sup>th</sup> percentile) vs. Throughput for a stateful service chain (Router-NAPT-LB) running on 8 cores while sending mixed-size packets at different rates with H/W offloading using FlowDirector. Note that the values of tail latency *include* loopback cost. The data points show the median of 50 runs. The solid lines represent the fitted curves (*i.e.*, piecewise functions) to the measurement points [26].

$$\text{DPDK}_{\text{fit}}(x) = \begin{cases} 15.61 + 0.2379x & x < 37 \\ 1977 - 95.18x + 1.158x^2 & x \geq 37 \end{cases} \quad (2.1)$$

$$\text{CacheDirector}_{\text{fit}}(x) = \begin{cases} 15.78 + 0.2415x & x < 37 \\ 2154 - 102x + 1.216x^2 & x \geq 37 \end{cases} \quad (2.2)$$

### 2.3.4 Summary of Evaluation

In this section, we showed that using CacheDirector brings slice awareness to packet processing. Doing so can reduce the average latency by up to  $\sim 6\%$  ( $14\mu\text{s}$ ) and more importantly tail latencies (90-99<sup>th</sup> percentiles) by up to  $\sim 21.5\%$  ( $119\mu\text{s}$ ) for NFV systems. By doing so, we improved the performance of a highly tuned NFV platform that works at the speed of the underlying networking hardware.

The reason for this improvement is that CacheDirector places the packet header into the appropriate LLC slice. As a result, any time the CPU requires the content of the packet header when it is not present in the L1 and L2 caches but available in LLC, the CPU stalls for fewer cycles waiting for the packet header to be brought into the higher cache levels; therefore, the CPU can process packets faster, which results in more frequent fetching of enqueued packets. Hence, the queuing delay is reduced. CacheDirector offers NFV service providers a tangible gain as they can utilize their system's capacity more efficiently while providing a more predictable response time to their customers and reducing their Service Level Objective (SLO) violations due to reduced tail latencies [86].

Despite CacheDirector improvements, we observed that sending packets to the appropriate slice is *not* always beneficial and some situations show only a small gain. For instance, running the stateful NFV service chain (Router-NAPT-LB) while using CacheDirector *without* offloading the routing table into the NIC *decreases* the improvement for the 99<sup>th</sup> percentile latency from  $34\mu\text{s}$  to  $4\mu\text{s}$ , see Figure 2.16 and the bottom row of Table 2.2 (on page 36). The reason for this behavior is due to the router lookups being performed in software, which not only reduces CacheDirector's effectiveness but also degrades the *overall* performance of the service chain as throughput is reduced from  $\sim 76\text{ Gbps}$  to  $\sim 42\text{ Gbps}$ .

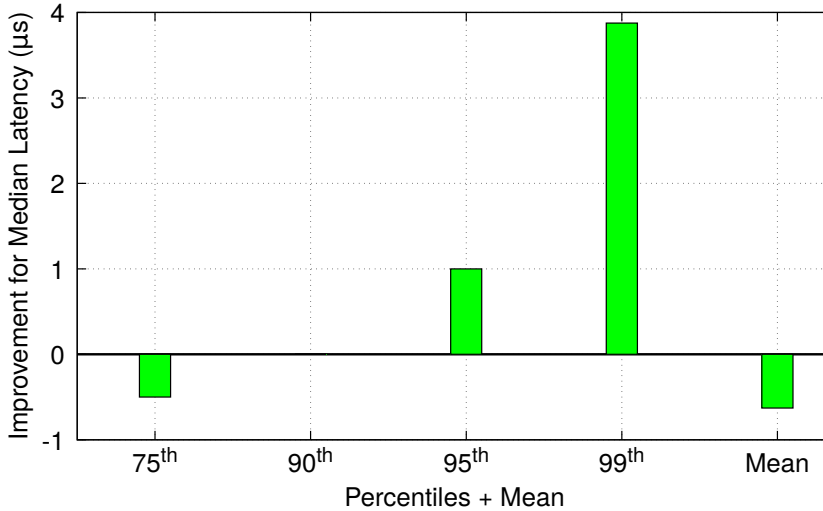


Figure 2.16: Improvement for a stateful service chain (Router-NAPT-LB) running on 8 cores with mixed-size packets at the rate of 100 Gbps with RSS and *without* any hardware offloading. This experiment was run 100 times. The maximum improvement is  $\sim 1\%$  for 99<sup>th</sup> percentile.

## 2.4 Limitations

This section elaborates limitations and other aspects of CacheDirector. Section 2.4.1 describes the status of the NIC drivers currently supported. Section 2.4.2 describes the effect of noisy neighbors on performance. Finally, Section 2.4.3 questions the applicability of CacheDirector to newer CPU architectures.

### 2.4.1 NIC driver support

CacheDirector is implemented in DPDK for the MLX5 driver (which supports NVIDIA/Mellanox ConnectX-4, ConnectX-5, and ConnectX-6 network adapters) and does not presently support vPMD.

### 2.4.2 Noisy neighbor effect

Since LLC is shared among the different cores, having a noisy neighbor degrades performance. In CacheDirector, we force part of our data to a single LLC slice; hence the degradation caused by a noisy neighbor may be more visible than when we are not slice-aware. The noisy neighbor effect is not limited to another

application running on a different core. For instance, when DuT is receiving large packets (*i.e.*, 1500 B), DDIO technology loads the whole packet (of  $\sim 24$  cache lines) into *different* LLC slices, hence previously enqueued packets can be evicted from the LLC when LoadGen sends at 100 Gbps, despite the fact that packets were sent to the LLC by DDIO. This can also happen without CacheDirector, but the probability of eviction in LLC when using CacheDirector is proportional to  $\frac{1}{\text{Number of LLC slice}}$  which is greater than the normal case, *i.e.*,  $\frac{1}{(\text{Number of LLC slice})^2}$ . In practice, one can use multiple slices for memory allocation as Section 2.1.2 showed that LLC access times are bimodal, which can result in a lower probability of LLC eviction.

### 2.4.3 Portability

It has been shown in [26] that the access time differences in LLC still exist in newer CPU architectures (*e.g.*, Skylake). However, the benefits of CacheDirector might decrease as newer architectures implement a larger L2 cache with a non-inclusive policy.

## 2.5 Related Works

This section discusses other efforts relevant to fast packet processing.

NFV is a transition toward deploying network functions on general-purpose hardware as opposed to using specialized physical boxes. To achieve high throughput and low latency with commodity hardware, NFV applications mostly employ userspace packet processing frameworks to eliminate the costly traditional kernel-based network stack [87, 80, 88, 89]. In addition to userspace I/O frameworks, there have also been efforts to optimize the kernel network stack [81, 90, 91]. Additionally, several efforts have been made to improve NFV application performance when running over userspace I/O frameworks [16, 92, 17, 93, 94, 95, 96]. Furthermore, there are a limited number of works that employed CAT to mitigate the noisy neighbor effect and improve NFV performance [97, 98, 99]. Our work can be seen as complementary to these works since none of them considered NUCA characteristics of LLC to improve performance.

## 2.6 Summary

In this chapter, we studied the cache architecture in recent Intel processors and identified an opportunity for improving cache performance. Then, we proposed CacheDirector, a network I/O solution that places the packets' header into the appropriate LLC slices. The result of our experiment showed that CacheDirector could cut the tail latency (90<sup>th</sup> - 99<sup>th</sup> percentiles) of time-critical NFV service chains by up to  $\sim 21.5\%$ . The main takeaway of this chapter is that performing efficient packet processing at multi-hundred-gigabit-per-second requires a better understanding of the underlying hardware and making the most out of cache memories.

## Chapter 3

# Optimizing Software-Based Network Functions

**N**ETWORKING has shifted from inflexible, proprietary, and specialized hardware toward SDN and NFV. Today, many network appliances are realized using commodity hardware and the network functions are increasingly software-driven. The flexibility and programmability of such platforms has led to the emergence of many software networking solutions. Unfortunately, the introduction of multi-hundred-gigabit-per-second network equipment and dramatic increases in the telecommunication bandwidth strain the performance of commodity hardware [100]; see Section 1.2. While many try to introduce in-network processing via modern hardware (*e.g.*, P4 architecture [101] and modern & programmable NICs) to address the performance limitations of commodity hardware [102], many network functions are deployed on commodity hardware, via *unspecialized* modular software, as software-based packet processing is being promoted by Ericsson, Cisco, and Intel [41, 42, 43]. Unfortunately, software-based networking solutions have come at the price of lower performance.

This chapter focuses on optimizing the performance of software packet processing frameworks. To do so, we (*i*) introduce a new model to efficiently manage packet metadata *and* (*ii*) employ code-optimization techniques to better utilize commodity hardware. Our optimizations make it possible to better utilize both instruction & data cache memories, which is aligned with our previous optimizations described in Chapter 2. This chapter starts by briefly describing the existing packet processing frameworks and their critical hindrances to performance. Later, we present our system, called PacketMill, which grinds the whole packet processing stack, from the high-level network function configuration file to the low-

level userspace network (specifically DPDK) drivers, to mitigate inefficiencies and produce a customized binary for a given network function.

## 3.1 Software Packet Processing

Many network operators & providers have shifted toward pure software solutions that can be run on COTS servers, *aka* commodity hardware, to (i) avoid proprietary inflexible hardware middleboxes and to (ii) reduce CAPEX & OPEX. These efforts can be classified into three main categories [103, 104]:

1. Low-level building blocks for realizing I/O frameworks (*e.g.*, DPDK [80], PF\_RING ZC [105], netmap [87], and XDP [81]).
2. Specialized virtual NF as a unified piece of software (*e.g.*, Open vSwitch (OVS) [106], ESwitch [44], PacketShader [89], and DPDKStat [107], and FlowMon-DPDK [108]).
3. Modular frameworks for composing network functions (*e.g.*, Click [109], FastClick [24], Vector Packet Processing (VPP) [110], the Snabb NFV project [111], and BESS [112, 113]).

This chapter focuses on the third category, *i.e.*, modular network function composition frameworks. We briefly discuss some of the popular frameworks for packet processing.

**Click** introduced one of the first modular architectures for building software routers [109]. The building blocks of Click are called elements. These elements can be connected together to compose a graph defining a complex network function. Each element implements a simple function (*e.g.*, packet classification, queuing, and decrementing Time to Live (TTL)). During the initialization phase, Click parses the input processing graph, provided by the user, and *virtually* builds the control flow graph. Later, Click executes the elements while traversing the graph for every packet. FastClick [24] is a high-speed variant of Click that leverages different acceleration techniques (*e.g.*, linked-list batching) and integrates kernel-bypass networking frameworks (*i.e.*, DPDK and netmap) into Click.

**VPP** or Vector Packet Processing framework is a software router, developed by Cisco, which focuses on L2–L4 packet processing based on vector processing, allowing users to configure the forwarding graph and process packets in batches. In VPP, each traversed node in the dataflow graph processes all packets in a batch (called a vector) before moving to the next node. The VPP design applies a number of optimization techniques and supports interrupt mode when using native drivers. VPP is part of Fast Data Project (FD.io), *i.e.*, a collaborative open-source project

aimed at establishing a high-performance I/O services framework for dynamic compute environments, and it has teamed up with Intel to take advantage of Single Instruction/Multiple Data (SIMD) instructions (*e.g.*, SSE & AVX) as much as possible.

**BESS** or Berkeley Extensible Software Switch (*aka* SoftNIC [112]) is another modular framework that was inspired by Click but simplifies & extends Click's design choices. BESS [113] introduces a set of built-in modules that can be connected and fed to a daemon process. The demon process deals with packet scheduling (enabling traffic prioritization) and processing. BESS is the base for a programmable platform called SoftNIC that augments NIC functionalities in software and claims to provide comparable performance to hardware with more flexibility [112]. BESS was designed with an eye toward utilizing new hardware NIC features and kernel-bypass technologies (*e.g.*, DPDK), thereby achieving better performance compared to Click [109] by leaving out the unused & old implementation.

Next, we discuss two important factors imposing performance limitations to process packets at multi-hundred-gigabit-per-second rates, *i.e.*, (i) code inefficiency and (ii) patched\* metadata management.

### 3.1.1 Code Inefficiency

Packet processing frameworks are built based on a modular design to bring a higher degree of flexibility and to simplify the composition of complex network services, by customizing and connecting simple monolithic elements. These frameworks usually adapt a general-purpose binary based on an input configuration file, which results in many inefficiencies, such as virtual calls, dead code, and unordered basic blocks. More specifically, the binary dynamically creates the control flow graph based on the input file and then executes it. All of the above frameworks utilize different acceleration techniques, such as kernel-bypass techniques and batch processing to achieve performance comparable to custom hardware appliances. However, as general-purpose processors were not optimized for packet processing, they do not provide the same performance as specialized hardware. Moreover, the modular & flexible design of these frameworks prevents them from achieving the performance of the underlying hardware.

**Optimization Efforts.** Two relevant attempts to overcome code inefficiencies in packet processing frameworks are:

- ① One way to mitigate code inefficiencies is to employ compiler optimization techniques, such as static class analysis, dead code elimination, instruction selection,

---

\*We use *patch* since packet processing frameworks have tried to adapt themselves to work with DPDK.



and inlining. Kohler *et al.*, [114] introduced a Click optimization toolkit to eliminate modular inefficiencies in Click and improve its performance. This toolkit is mainly a source-to-source tool that scans a Click-language file and employs optimization techniques, resembling general compiler optimizations, to transform the code into a more efficient version. More specifically, the Click optimization toolkit reads a Click configuration file, builds a graph of elements, analyzes & transforms the graph, and finally produces a more optimized configuration file and/or source code. The Click optimization toolkit includes a number of tools: `click-fastclassier`, `click-devirtualize`, `click-dxform`, and `click-undead`. The most relevant tool to our work is `click-devirtualize`, which is a static class analysis tool that devirtualizes function calls, *i.e.*, replacing virtual function calls during graph traversal with direct calls extracted from the graph analysis. `click-fastclassifier` is a dynamic code generation tool that generates new source code for classifiers based on the specific decision tree and replaces the generic classification elements with more specialized ones. `click-xform` tool resembles instruction selection or peephole optimization. It reads a Click configuration file to identify occurrences of arbitrary patterns and replacement subgraphs. Then, for each pattern's occurrence - it replaces in the source code a collection of general-purpose elements with optimized combined elements with lower overhead. By doing so, the number of elements in the forwarding path is potentially reduced, which lowers the cost of virtual calls. Finally, `click-undead` optimizer, similar to dead code elimination, removes unused elements from a Click configuration file.

② Protocol space mismatch can occur between development and deployment phases, leading to redundant logic. Bangwen *et al.*, [115] proposed a tool, called NFReducer, which employs classic compiler optimization techniques to eliminate redundant logic from a configured NFV instance. NFReducer has been developed using LLVM and it utilizes a symbolic execution engine (*i.e.*, KLEE [116]) to filter out infeasible paths of NFs. This tool performs three main optimizations based on the NF configuration: (i) excluding unrelated logic from NFV code; (ii) applying constant propagation, constant folding, and dead code elimination; and (iii) eliminating cross-NF redundancy when multiple NFs are chained.

### 3.1.2 Patched Metadata Management

Packet processing usually requires additional information beyond the raw packets (*i.e.*, bits received from the wire). The extra information is divided into two categories: (i) metadata and (ii) packet annotations (*aka* user/application metadata). The former contains additional details on the raw packet/buffer itself, such as its length, timestamp, checksum, and pointers to different protocol stacks in the packet,

which are required to operate on the raw packets. The latter is the information used during the packet processing, *i.e.*, the information that has to be calculated/extracted from the packet at one place and used in *another* place [109], such as VLAN ID, MPLS label, source & destination IP addresses & ports, statistics, and Wi-Fi association. The metadata is usually defined by the driver and the NIC, whereas the (packet) annotations are derived and used by the application. Next, we briefly explain the evolution of metadata management, starting from the Linux kernel.

The Linux kernel uses `sk_buff` data structures, *aka* socket buffers, to manage & handle network packets\*. An `sk_buff` contains many metadata fields (the size/number of which depends on the protocol standards) to facilitate manipulation of packets, see `linux/skbuff.h`. Each `sk_buff` also provides a *fixed* 48-B free space, *aka* control buffer (cb), which can be used for application-specific annotations [117, 118].

Since Click started as a kernel-space packet processing framework, it developed a metadata class, called `Packet`, for handling packets, inspired by `sk_buff`. `Packet` had pointers to different protocol headers (*e.g.*, network layer and transport layer). Additionally, it likewise reserved 48 B to be used by Click elements for storing packet annotations. As 48-B space may not be enough, developers had to carefully prevent collisions.

Modern packet processing frameworks (*e.g.*, FastClick, BESS, and VPP) utilize kernel-bypass libraries (*e.g.*, DPDK and netmap) to achieve zero-copy packet transfers and eliminate Linux kernel stack costs. In the rest of this chapter, we focus on DPDK-based packet processing.

DPDK uses mbufs to carry network packets/buffers. Each mbuf has three sections: (i) a `rte_mbuf` data structure containing the metadata, (ii) a fixed-size headroom reserved for prepending/appending data, and (iii) a data segment used for storing the raw packet [119]. Each `rte_mbuf` struct is only two cache lines<sup>†</sup> (*i.e.*, 128 B) to keep it as small as possible, leaving the management of packet annotations to the application (*i.e.*, the packet processing framework). DPDK provides userspace NIC drivers, *aka* PMD, which enables the *direct* interaction of an application with the NIC. DPDK allocates mbufs (metadata + headroom + data) in its initialization phase. Subsequently, PMD uses these pre-allocated mbufs to receive/transmit packets at run-time. To receive packets, PMD passes the mbuf data address & its driver-specific descriptors to the NIC so that the NIC can DMA the received packets and their metadata to these addresses. Later, when the PMD detects a DMA completion via polling, PMD copies the relevant information from the driver descriptors to the mbuf metadata (*i.e.*, `rte_mbuf` struct). To transmit packets, PMD performs a similar operation

---

\*BSD buffers are called `mbufs`.

<sup>†</sup>The most frequently used fields are defined to be in the first cache line.

(i.e., updating driver-specific descriptors) before passing the mbuf data address & driver descriptors to the NIC. Unfortunately, integrating DPDK with packet processing frameworks causes metadata management to become a bottleneck, thereby realizing *suboptimal* performance at multi-hundred-gigabit-per-second rates. This performance degradation happens for three reasons:

**First**, modern packet processing frameworks typically employ batch processing to improve cache locality, i.e., an application receives a batch of packets, processes them, and then asks for another batch. Therefore, the number of packets' *metadata* required at any given time is equal to the batch size (i.e., the number of packets received from the PMD). However, DPDK uses a *distinct* `rte_mbuf` for every packet, which *reduces* the probability of the metadata data structures remaining in the cache. More specifically, warm cache lines containing recently processed packets' metadata may be evicted to make room for the newly arrived packets' metadata. An optimal solution would use a limited number of metadata data structures (e.g., `rte_mbuf`) and keep them in the cache.

**Second**, since the `rte_mbuf` struct does not provide enough space for storing/keeping (packet) annotations, packet processing frameworks have to allocate larger data structures to enable/facilitate packet processing. Figure 3.1 compares the two common ways to extend the `rte_mbuf` metadata, which we refer to as “Copying” vs. “Overlaying”.

*Copying.* This method is mainly used by Click & FastClick. They *handpick*, via copying or converting, the information useful for packet processing from the `rte_mbuf` struct into their own data structure (i.e., `Packet` class [120]) that contains a 48-B space for annotations, see ❶ in Figure 3.1. Unfortunately, this method is inefficient because it involves *two* copy/conversion operations: (i) driver descriptors to `rte_mbuf` struct and (ii) `rte_mbuf` struct to `Packet` object.

*Overlaying.* Some packet processing frameworks (e.g., BESS) *overlay* the beginning of their data structures on the `rte_mbuf` and cast it to avoid copying & conversion, see ❷ in Figure 3.1. They insert their dynamic metadata or annotations *after* the `rte_mbuf` struct and before the headroom. More specifically, BESS uses the `sn_buff` struct (recently renamed to `Packet` [121]), where they provide a 384-B space for storing the `rte_mbuf` struct, 64-B for the immutable fields (e.g., packet address and socket ID), 128-B static/dynamic metadata fields, and 64-B space for a module's & driver's internal use [122].

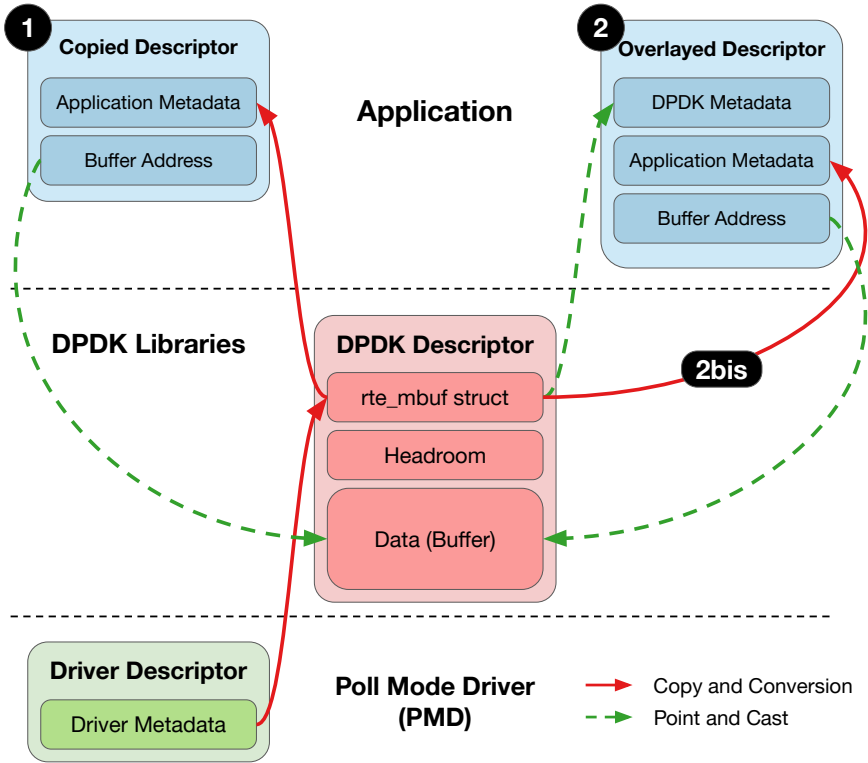


Figure 3.1: Common metadata management methods in packet processing frameworks (Copying vs. Overlaying) [45].

VPP follows a similar approach, where they use `vlib_buffer_t` to store the buffers' metadata. `vlib_buffer_t` is also known as the primary buffer metadata used by the vector library (vlib). VPP overlays the beginning of its data structure with the `rte_mbuf` struct, but does *not* use it. Instead, it copies/converts some fields from the DPDK data structure into the `vlib_buffer_t`, as it needs to make the metadata format fit for Streaming SIMD Extensions (SSE) instructions, see **2bis** in Figure 3.1.

FastClick also supports Overlaying, which should be enabled at compile time. It casts every `rte_mbuf` into a `Packet` object and then (similar to BESS) inserts its annotations after it [120].

Although overlaying mitigates the cost of copying, it is still inefficient since packet processing frameworks have to adapt their format to the DPDK format (*e.g.*, BESS) and/or do a transformation (*e.g.*, VPP), which often results in carrying unnecessary fields while processing packets, thereby reducing cache locality.

**Third**, since different NFs require different information for processing a packet, using *one* standard data structure to keep the metadata & packet annotations is *non-optimal*, as it could spread the required information over multiple cache lines, thereby increasing cache occupancy and increasing the number of memory accesses needed to process a packet. A performant design should change the type and/or order of variables used to keep the metadata & packet annotations based on the functionality of a given NF.

The next section introduces our system, called PacketMill, which tries to mitigate the code & metadata inefficiencies discussed in this section.

## 3.2 PacketMill

This section explains PacketMill, our proposed system for optimizing the performance of modular software packet processing frameworks. Our goal is to mitigate the inefficiencies discussed in Sections 3.1.1 & 3.1.2. To do so, PacketMill introduces a new metadata management model, called X-Change, to enable metadata customization while improving cache locality. Additionally, it modifies the code based on the input configuration file – as this contains information that assists in the compilation process. Figure 3.2 shows our proposed pipeline to produce a specialized binary for a given NF configuration, where numbered (green) shapes are proposed by us. We start by explaining the metadata management model and then continue with our efforts to mitigate code inefficiencies.

### 3.2.1 Efficient Metadata Management

Section 3.1.2 discussed the problems associated with the current ways of managing metadata in DPDK-enabled packet processing frameworks. Current packet processing frameworks rely on the generic `rte_mbuf` to store metadata, which requires adaptation & extra overhead/effort to process packets efficiently. PacketMill introduces a new metadata management model (X-Change) that enables frameworks developed on top of DPDK to *exchange* their customized/specialized metadata buffers with the userspace DPDK drivers (*i.e.*, PMD) and to bypass `rte_mbuf`, thus addressing the second problem in Section 3.1.2. Additionally, PacketMill’s metadata management model makes it possible to use a limited number of metadata buffers (*e.g.*, 32) to improve cache locality and prevent unnecessary cache evictions, thus solving the first problem in Section 3.1.2. It aims to provide efficiency

while ensuring backward compatibility for previously developed DPDK-based applications. To achieve our goals, we developed an API within DPDK, which required some changes to DPDK’s PMDs, see ① in Figure 3.2.

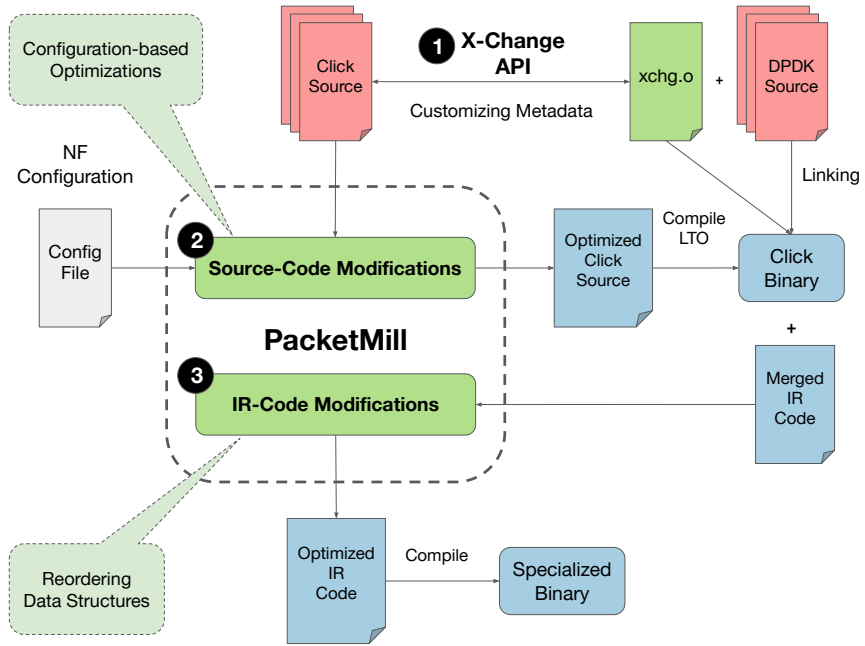


Figure 3.2: PacketMill Overview [45].

**Implementation.** To showcase X-Change, we modified MLX5\* PMD (used by NVIDIA/Mellanox NICs). We add a header file (.h) and a source file (.c) to the MLX5 source code. The header file defines conversion functions for both RX & TX paths to assign/copy different metadata fields, as opposed to the current typical DPDK implementation where PMD *directly* assigns the metadata to a `rte_mbuf` struct. Note that these functions will eventually get inlined, as we use Link-Time Optimization (LTO). Listing 3.1 compares our proposed approach and the default DPDK behavior with a simple example.

\*Our current prototype does not support vectorized PMD, so we have disabled it in all of our experiments, except those described in Section 3.3.1.

---

**Listing 3.1** X-Change introduces conversion functions instead of directly writing to the `rte_mbuf` struct. The code shows an example for setting the VLAN TCI field in the driver.

---

```
/* Default DPDK */  
pkt->vlan_tci = rte_be_to_cpu_16(cqe->vlan_info);
```

---

```
/* X-Change */  
xchg_set_vlan_tci(pkt, rte_be_to_cpu_16(cqe->vlan_info));
```

---

The source file implements the standard behavior of DPDK (*i.e.*, using `rte_mbuf`). Consequently, since DPDK compiles our pre-implemented source file by default, X-Change enables full backward compatibility to metadata-agnostic applications. However, it enables developers to re-implement the conversion functions and customize DPDK's metadata to their needs, thereby enabling the PMD to write the metadata directly to their application's data structures. Moreover, X-Change makes it possible to *easily* try out (or switch between) different metadata management models with low overhead (*i.e.*, simply by linking to a different object file implementing the conversion functions). Listing 3.2 shows an example re-implementation of a conversion function.

**Listing 3.2** X-Change simplifies the metadata management. The code compares the default DPDK and a custom implementation to set the VLAN TCI field.

```
/* X-Change Implementation of Default DPDK */
void
xchg_set_vlan_tci(struct xchg* pkt, uint16_t vlan_tci) {
    ((struct rte_mbuf*)pkt)->vlan_tci = vlan_tci;
}

/* X-Change Implementation for Custom Buffers */
void
xchg_set_vlan_tci(struct xchg* pkt, uint16_t vlan_tci) {
    SET_VLAN_ANNO((Packet*)pkt, vlan_tci);
}
```

PacketMill's model also proposes a new way to interact with PMD. The default DPDK implementation asks applications to provide an empty array of pointers in order to receive packets. Later, PMD fills this array with the address of received packet buffers (*i.e.*, the metadata and the raw packet), and DPDK provides new *untouched* packet buffers to the PMD to replace the *cached* buffers used for the received packets. However, X-Change enables applications to provide their own packet buffers to the PMD. By doing so, the PMD can directly write into the application's metadata and “exchange” the used buffers (containing received packets) with the newly received ones from the application. X-Change uses a similar workflow for the TX path. After processing the received packets, the application passes the processed buffers to the PMD. Subsequently, the PMD copies/converts the application's metadata to the NIC descriptors and does a *swap* of the previously sent buffers, sitting in the transmit ring, with to-be-sent buffers; thus, the application has as many empty buffers as it has sent (which can be exchanged again in the RX path).

**Summary.** X-Change is an optimization to DPDK that provides custom buffers to drivers; thus, metadata can be directly written into the applications' buffers rather



than using an intermediate DPDK metadata (*i.e.*, `rte_mbuf`). X-Change uses conversion functions instead of direct assignment to set the metadata fields. The X-Change implementation (*i.e.*, the definition of different conversion functions) is dependent on the driver's features and descriptors. A recent work called TinyNF, done by Pirelli *et al.*, [123], proposes a simple and *formally verifiable* driver model (for Intel 82599 NIC) that removes the need for dynamic packet metadata. However, it prevents buffering of packets, such as switching packets between cores, reordering packets, and stream processing, which introduces lots of drawbacks for some network functions. In contrast, X-Change also reduces the number of metadata buffers, but *without* imposing those restrictions. Moreover, X-Change is more generic (as opposed to TinyNF) since it pushes programmability into the driver, making it possible to implement buffer exchanging, or even TinyNF, without re-compiling DPDK. X-Change results in the following improvements:

- Enables applications to use their tailored metadata and to bypass the generic `rte_mbuf`, thus avoiding unnecessary copy/transform operations and cache evictions;
- Pushes down part of the application's RX/TX loops to initialize packet annotations into the PMD, thereby simplifying the application's processing path;
- Limits the amount of metadata used to the application's requirement (*i.e.*, proportional to the RX burst size + the number of packets enqueued in software), keeping metadata cache lines warmer and making the most out of DDIO [27];
- Skips buffer allocation/release operations through DPDK buffer pools, which are inefficient due to supporting/maintaining many (unnecessary) features; and
- Makes it possible for the application to easily use different packet chaining models (*e.g.*, vector, linked list, or a combination of both) to better fit their needs.

### 3.2.2 Optimized Code

PacketMill performs two main types of code optimizations: (*i*) source-code modifications and (*ii*) IR code modifications. The former embeds & modifies the source code, as *early* as possible, based on the information provided by the NF configuration file. Informing the compiler of this known information should enable many optimizations, as compilers have become much smarter [124]. The latter exploits the LLVM toolchain to modify the final IR bitcode produced by LTO, making it possible to perform optimizations, as late as possible, *i.e.*, when the *whole* program's IR bitcode is available.

#### 3.2.2.1 Source Code Modifications

Our first step to producing a more specialized binary is to perform configuration-based optimization, which gets rid of unnecessary pointers to already-known data

structures/variables while removing unused code. To do so, we *embed* some of the already-known information about the NF into the source code, see ② in Figure 3.2. More specifically, we use (i) the packet processing graph and (ii) constant element parameters defined in the NF configuration file, see Listing 3.3.

**Listing 3.3** A Click’s NF configuration file defining a simple forwarder that receives packets from a DPDK-enabled NIC, swaps the Ethernet MAC addresses, and transmits the packets. The graph contains three elements: (i) `FromDPDKDevice` *alias* input, (ii) `EtherMirror`, and (iii) `ToDPDKDevice` *alias* output, chained together sequentially.

```
// Elements Definition
input :: FromDPDKDevice(PORT 0, N_QUEUES 1, BURST 32);
output :: ToDPDKDevice(PORT 0, BURST 32);
// Processing Graph
input -> EtherMirror -> output
```

Embedding the packet processing graph, *i.e.*, the processing elements and their connections, informs the compiler about the NF’s Control Flow Graph (CFG), resulting in better code layout. It also makes it possible to declare the processing elements statically in the source code, *i.e.*, allocating them in a *static* .data or .bss segment (or stack) rather than the heap\*, *potentially* resulting in a less fragmented access pattern and fewer TLB misses. Moreover, using statically defined elements *and* defining the CFG enables us to perform full devirtualization, *i.e.*, inlining the virtual calls, as opposed to `click-devirtualize` that only defines the type of the function pointer *rather than* the actual object reference.

Constant element parameters define the characteristics of processing elements. For example, an element receiving packets (*e.g.*, `FromDPDKDevice` in Listing 3.3) should define the maximum number of packets fetched from the I/O device (*i.e.*, a BURST size). Embedding these values in the source code enables the compiler to perform constant propagation & constant folding while removing/eliminating dead code & unrolling loops, thereby improving cache locality.

**Implementation.** To benefit from the `click-devirtualize` toolchain [114] and reduce the implementation overhead, we resurrected & adopted `click-devirtualize` to work with FastClick [24] and then implemented additional optimization on top of it. These optimizations are similar to NFVReducer [115], as both share the same goal, *i.e.*, removing redundant and/or unnecessary code. However, NFVReducer focuses on optimizing the performance of popular IDSs, whereas our techniques are applicable to *any* kind of NFs composed by modular packet processing frameworks. Despite these differences,

\*We define element objects in the source code and re-initialize them properly after executing the binary.

PacketMill can potentially be combined with NFVReducer to filter out infeasible paths via symbolic execution. Finally, it is important to highlight that our proposed optimizations are *not* limited to Click-based frameworks; they could be useful for other software packet processing frameworks.

### 3.2.2.2 Intermediate Code Modifications

Modern compilers (*e.g.*, gcc and clang/LLVM) support LTO [125, 126, 127], making it possible to perform inter-procedural optimizations during the linking phase where the *whole* program is visible to the linker/optimizer. When LTO is enabled, compilers typically produce IR code rather than regular object files (containing machine code), so that whole-program analysis and optimization can be done during linking. LTO can realize better code layout and smaller binaries, as it is easier for the compiler to collect & use information about symbols, variables, functions, and the callgraph to eliminate dead code and reorder functions. Additionally, since the whole program is available, developers can potentially implement customized optimization passes to optimize the executable even further.

PacketMill exploits LLVM's LTO to address the third problem in Section 3.1.2, *i.e.*, lack of per-NF data structure speciality. Our goal is to specialize/customize the *one* standard metadata used in a packet processing framework for a given NF. To do so, we develop an optimization pass (via LLVM) that reorders the variables/fields of a metadata structure based on the access pattern of a given NF, see ③ in Figure 3.2 on page 53. By doing so, the more frequently accessed fields will be placed at the beginning of a data structure (*i.e.*, the first cache line(s)), avoiding extra accesses to multiple cache lines. More specifically, our pass finds the references (done by the NF) to different variables & fields of a metadata structure within a *module* and then sorts these variables based on the *estimated*<sup>\*</sup> number of accesses to the variables. Later, the pass fixes these references so that LLVM's `GetElementPtrInst` (GEPI)<sup>†</sup> instructions perform the correct accesses. Listing 3.4 shows an example LLVM IR bitcode for accessing a variable of a C++ object. The current version of our pass only sorts the variables, but one could also *remove* unused variables/fields. Additionally, it is possible to extend our pass to consider other sorting criteria (*e.g.*, order of access), which remains as our future work. To examine the full potential of LTO, we extend DPDK's build system to work with clang and produce LLVM IR bitcode<sup>‡</sup>. It is worth mentioning that using LTO increases the compilation time, which could be reduced by using a scalable variant of LTO (*e.g.*, ThinLTO [128]).

---

<sup>\*</sup>The *real* number of accesses depends on the received workload.

<sup>†</sup>It calculates the address of a sub-element of an aggregate data structure.

<sup>‡</sup>Note that DPDK currently only supports LTO for gcc and icc that produce `fat-lto-objects` containing both machine and IR codes.

**Listing 3.4** Accessing a metadata field in LLVM IR bitcode (bottom). The top shows the C++ version of the code.

```

1  /* A Simple Metadata Class */
2  class Packet {
3  public:
4      long unusedlong;
5      void *unusedptr;
6      void *data;
7      char unusedchar;
8      int length; // <-- accessed
9  };
10 /* Access Example */
11 Packet p;
12 p.length = 100;

; Class Declaration (line 2-9)
@class.Packet = type { i64, i8*, i8*, i8, i32 }
; Object Definition + Initialization (line 11-12)
%1 = alloca @class.Packet, align 8 ; Allocate p
%2 = getelementptr inbounds @class.Packet,
    @class.Packet* %1, i32 0, i32 4 ; Get addr. of length
store i32 100, i32* %2, align 4 ; Store 100

```

**Challenges.** While some compilers (*e.g.*, Rust) support structure reordering [129], C & C++ compilers are forbidden to reorder data structures (*e.g.*, struct or class) [130], which makes reordering variables/fields of a data structure at IR level challenging. When compilers make some assumptions about a data structure’s order [130, 131, 132], reordering cannot be done for *any* data structures *without* careful consideration. Particularly, it requires deep knowledge of the workflow of the code and the relationship between different data structures. For instance, reordering the data structures that exchange data with hardware could break the program’s correctness. Two common scenarios where this problem can occur are: ① when a piece of code relies on the order of the variables in a data structure (*e.g.*, (i) using vector instruction to initialize or process a data structure and (ii) interacting with hardware) and ② when dynamically linked libraries access the data structure. Note that the references in statically linked libraries can be fixed (repaired) if we apply the reordering when the whole program’s IR bitcode is available. Moreover, in the case of aggregation/composition, it is important to fix the references to the container class/struct. Our pass currently does not perform any verification, but it is possible to verify the correctness of the NF when reordering the metadata, see Section 3.4.

**Implementation.** To mitigate these challenges, our pass reorders the metadata structure (*i.e.*, Packet in FastClick) *only* when the metadata management model

is set to use the Copying model. However, it is possible to apply a similar approach for the Overlaying model by extending our pass to reorder `rte_mbuf` & PMD's descriptors. To fix the references, our pass takes into account references done by `class.WritablePacket` and `class.PacketBatch`, which are dependent classes of `class.Packet`. We apply our pass via LLVM's `opt` on the pre-code generation output of LTO (*i.e.*, `click.0.5.precodegen.bc`) [133]. However, it would be possible to develop a custom LTO pass and then extend clang's C++ frontend to define a keyword for our proposed optimization.

To the best of our knowledge, PINstruct [134] is the only *published* work that has considered reordering data structures for a C program. They traced memory accesses via MemPin (a tool that uses the Intel Pin tool [135]) and reordered the OpenMPI data structures manually. However, they neither automated the data structure reordering nor evaluated its benefits.

### 3.3 Evaluation

This section demonstrates the effectiveness of PacketMill in improving the performance of packet processing. To better understand each optimization done by PacketMill, this section discusses them individually and then evaluates their impact on microarchitectural & application-level metrics. Later, we apply all of the optimizations together and demonstrate their combined impact.

**NF configurations.** We focus on five network functions: (*i*) a simple forwarder, (*ii*) a router, (*iii*) an IDS followed by the router, (*iv*) a NAT, and (*v*) a synthetic memory- & compute-intensive NF, see Appendix A for details. The simple forwarder & the router represent scenarios where a NF is relatively *I/O-bound* rather than *CPU-bound*, whereas the IDS+router, the NAT, and the synthetic NF demonstrate more sophisticated network functions that require more processing.

**Testbed.** We use a testbed with two (Skylake) servers equipped with NVIDIA/Mellanox ConnectX-5 VPI and interconnected via a 100-Gbps link. One server acts as a packet generator, and generates & sinks packets and measures the end-to-end latency & throughput, while the other server acts as a DuT and processes packets based on the given input NF configuration file. The packet generator and the DuT are equipped with  $2 \times 8$ -core Xeon Gold 6134 @ 3.2 GHz and  $2 \times 18$ -core Xeon Gold 6140 @ 2.3 GHz (nominal frequency), respectively. Both servers run Ubuntu 18.04.4 (Linux kernel 4.15.0-112). We use `perf` to measure microarchitectural metrics. Additionally, we isolate the DuT's CPU socket on which we run the experiment to increase our measurement accuracy. We use LLVM 10.0.0 (trunk 375507) to compile and optimize FastClick [24] and DPDK (v20.02). To prevent Intel DDIO from becoming a bottleneck in our measurements, we change the `IO LLC WAYS` register's value to `0x7F8` (*i.e.*, 8 set bits) [27]. Additionally, we set the uncore frequency to 2.4 GHz (*i.e.*, the maximum frequency in our testbed) to minimize DRAM and LLC latency [136, 137]. We use the Copying model (*i.e.*, the default metadata management model in FastClick) unless stated otherwise. We use the Network Performance Framework (NPF) tool [138] to facilitate reproducibility of our tests.

**Generated traffic.** We use two types of traffic in our evaluation: (i) a 28-min campus trace and (ii) synthetically generated traces with fixed-size packets (see Sections 3.3.4 and 3.3.7). The campus trace has 799 M packets with an average size of 981 B. In each run, we replay the first two million packets of the trace 25 times. We repeat each test five times and report the median values when there are no error bars. Note that the achieved throughput is proportional to processed packets  $\text{pps} \times \text{packet size}$ . Therefore, replaying a trace with a smaller average packet size could result in lower throughput, but the same improvements (*i.e.*, more pps).

### 3.3.1 Do PacketMill's Code Optimizations Improve Packet Processing at 100 Gbps?

We evaluate the router's performance when processing the repeated portions of the campus trace at different clock frequencies. We change the processor's clock frequency of the DuT to assess the impact of PacketMill on different classes of processors, *i.e.*, more cores with lower frequency vs. fewer cores with higher frequency. Additionally, reducing the clock frequency somewhat emulates the situation where the processor receives traffic at a higher rate (than the injected rate, *e.g.*, >100 Gbps) or when the NF is more CPU-bound.

**Configuration-based optimizations.** Table 3.1 & Figure 3.3 show the results of our experiments when applying different source-code optimization techniques: (i) devirtualization (done by `click-devirtualize`), (ii) constant embedding, and (iii) static graph (*i.e.*, defining the elements statically and their connections in the source code). These results demonstrate that all techniques & their combination have positive impact on the number of cache misses, throughput, and median latency. More specifically, using a static graph rather than a dynamic one improves throughput by up to 20% (or 14.8 Gbps) and dramatically reduces the *LLC misses* (up to  $\sim 300\times$ ), see the second row of Table 3.1. Note that 10-Gbps-throughput improvements are significant, as they would translate to supporting more 10-Gbps links, thus reducing the number of NFs in the network.

Table 3.1: PacketMill’s code optimizations improve microarchitectural metrics by up to  $\sim 300\times$  (*i.e.*, reducing the number of LLC load misses). We measure cache misses & IPC with `perf` every 100 ms and report the average measured during the experiment, performed at 3 GHz.

Configuration Generation	Vanilla	Devirtualization	Constant Embedding	Static Graph	All
LLC kilo loads	1097	1159	1176	24	26
LLC kilo load-misses	803	841	845	0.98	2.58
Instructions per Cycle (IPC)	2.24	2.30	2.28	2.58	2.59
Million packets per second (Mpps)	8.66	9.05	9.12	10.16	10.41

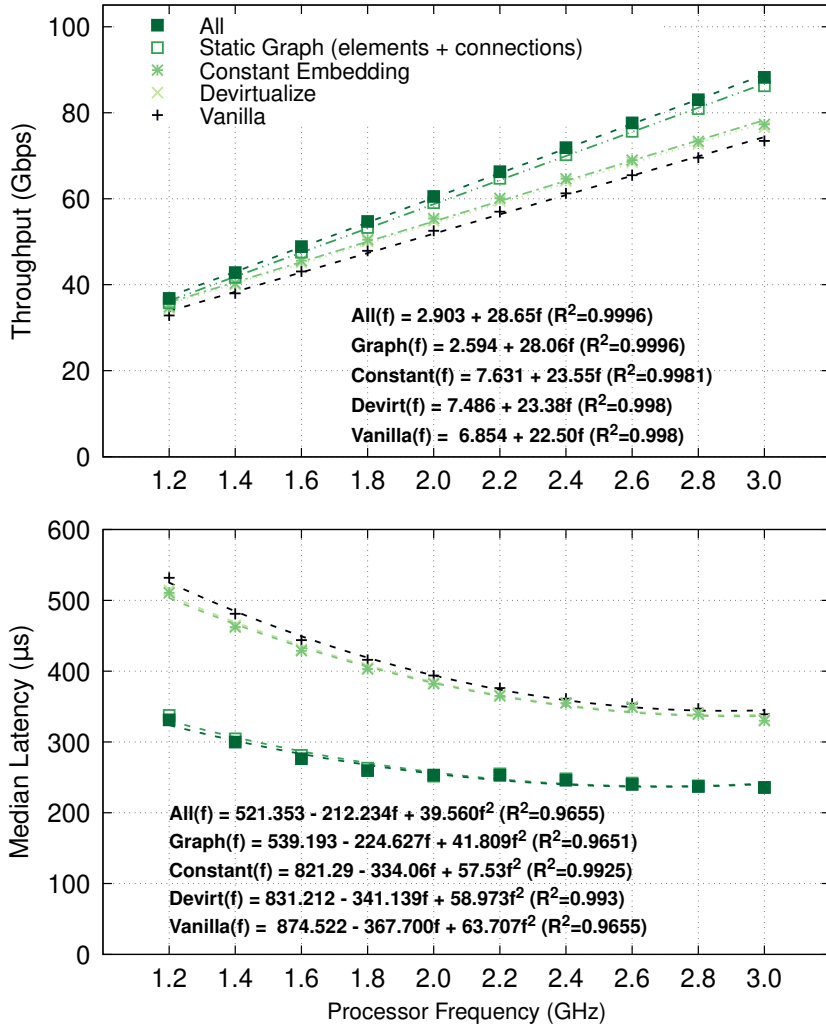


Figure 3.3: Exploiting the information in the router configuration improves throughput and median latency. The server is processing packets with *one* core running at different frequencies ( $f$  in GHz) [45].



**LTO & structure reordering.** Applying LTO and reordering `Packet` class of FastClick for the router configuration (running at 3 GHz) increases throughput and decreases median latency, at no additional cost, by up to 5.4 Gbps (6.8%) and 13  $\mu$ s ( $\sim$ 3.8%), respectively. Reordering contributes to one-third of the improvements. As mentioned earlier, these sub-ten-percent improvements should not be ignored, as these are essential for cost-effectively deploying Internet services, facilitating service providers meeting their SLOs. Note that other frequencies also result in similar improvements.

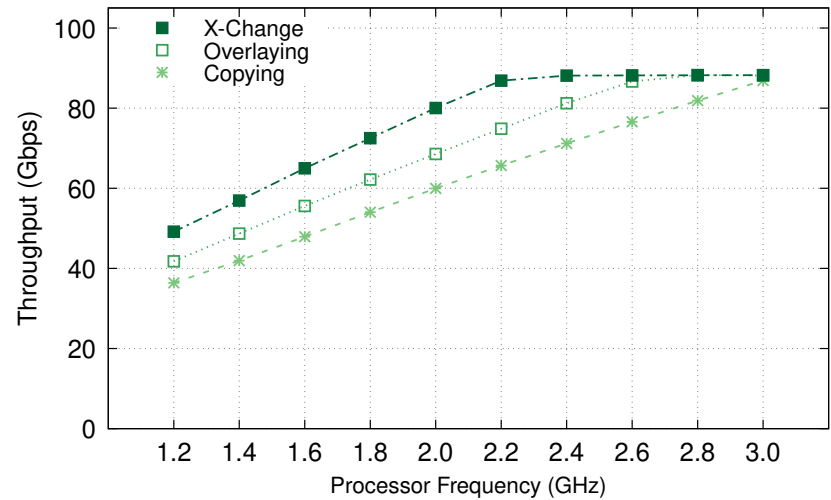
### 3.3.2 How Effective is PacketMill’s Model (X-Change) compared to the existing Metadata Management Models?

We use FastClick to compare all three methods (*i.e.*, Copying, Overlaying, and X-Change) for a simple forwarding configuration. We disable PacketMill’s code optimizations to examine the impact of metadata management alone. We enable LTO in all scenarios to have the best achievable performance of each model. Note that disabling LTO could underestimate the benefits of X-Change, due to not inlined conversion calls. Moreover, DPDK could be recompiled with X-Change’s source file included as a header file to achieve a similar result *without* LTO.

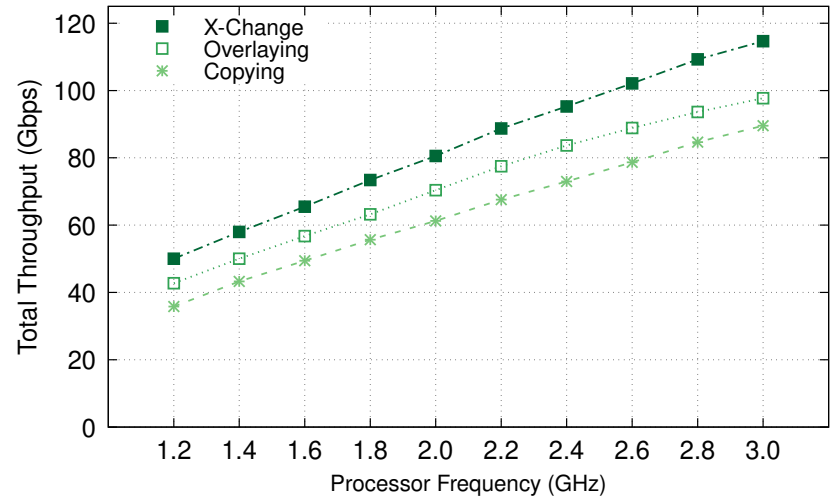
Figure 3.4a demonstrates that PacketMill’s metadata management model (X-Change) improves throughput significantly by mitigating inefficiencies of the other two models. These results show that increasing the processor’s frequency does not improve the throughput of X-Change & Overlaying models after a certain frequency (*i.e.*, 2.2 GHz & 2.6 GHz, respectively), as there may be other bottlenecks in the system (*e.g.*, using one RX/TX queue or other NIC-related issues [51]). To investigate the full potential of X-Change, we set up a 200-Gbps testbed, where two servers generate traffic toward the DuT equipped with two NICs (connected to the *same* CPU socket). DuT forwards the received packets from the two generators via only *one* core.

Figure 3.4b shows that X-Change is the only metadata management model which enables a *single* core to forward packets at >100 Gbps. Additionally, both Figures 3.4a and 3.4b show that Overlaying model performs better than Copying, as Copying involves double copy/transform operations. Moreover, our measurements show that X-Change significantly reduces the number of LLC load misses; for instance, the X-Change-enabled forwarder running at 3 GHz *only* results in  $\sim$ 200 misses per 100 ms, whereas Copying and Overlaying cause  $\sim$ 3000 and  $\sim$ 6000 misses per 100 ms, respectively. We also observed that the performance of Copying+Overlaying method (used by VPP) is similar to Copying model. In summary, an inefficient metadata management model prevents the system from

processing packets at a higher rate, *i.e.*, degrades throughput by >10 Gbps (*i.e.*, a typical data center’s link speed).



(a) One NIC via one core.



(b) Two NICs via one core.

Figure 3.4: X-Change makes it possible to forward packets at >100-Gbps rates. The experiments with two NICs report the sum of throughput achieved by one core [45].

### 3.3.3 How Effective is PacketMill at Different Loads?

To fully understand the combined benefits of PacketMill, we measured the tail latency of a single-core router while injecting packets to it at different loads. Figure 3.5 demonstrates that PacketMill improves the packet processing at 100 Gbps when a router forwards packets using a *single core* running at 2.3 GHz. More specifically, our proposed model & techniques shift the knee of the tail latency vs. throughput curve, *i.e.*, achieving lower latency *even* when the load is higher.

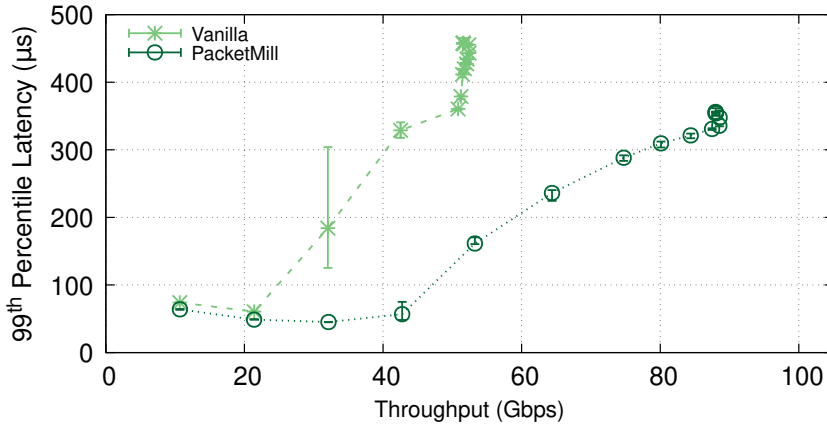


Figure 3.5: PacketMill improves per-core packet processing. Overlapped markers show that the performance can be capped despite the increasing offered load [45].

### 3.3.4 How Does the Workload/Trace Affect PacketMill?

We have performed most of our experiments with the campus trace, as we thought using fixed-size packets could increase the effect of measurement/layout bias. Additionally, we reported throughput in bits per second since we were targeting 100-Gbps networking. In this section, we use FastClick to generate fixed-size packets to show that PacketMill's improvements are not trace-dependent. Figure 3.6 reports the throughput in both bits per second and pps for a router running at 2.3 GHz while receiving fixed-size packets. It shows that PacketMill's improvements are consistent for different packet sizes, as long as there are no other bottlenecks in the system. Note that increasing the packet size after a certain point (*e.g.*, ~800 B) reduces the number of processed packets per second, due to the PCIe bandwidth [27, 49].

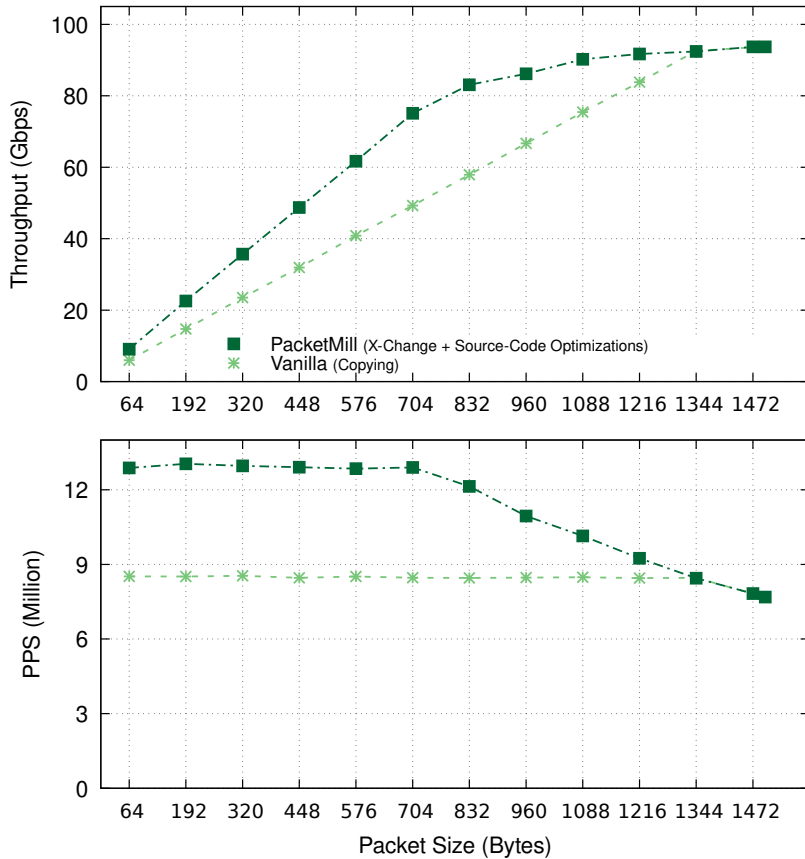


Figure 3.6: For a router running at 2.3 GHz, PacketMill improves the number of processed packets per second for different packet sizes [45].

### 3.3.5 How about More Sophisticated Network Functions?

So far, we have shown the individual (Section 3.3.1 & Section 3.3.2) and combined\* benefits (Section 3.3.3 & Section 3.3.4) of using different optimizations proposed by PacketMill on the router and the simple forwarder configuration. We showed that using PacketMill improves the system's performance when performing relatively lightweight processing. This section investigates the impact of PacketMill on more sophisticated NFs. We start by applying PacketMill to an IDS followed by a router, which requires more processing to check the correctness of TCP, UDP, and ICMP

\*Combined impact does not take into account data structure reordering.

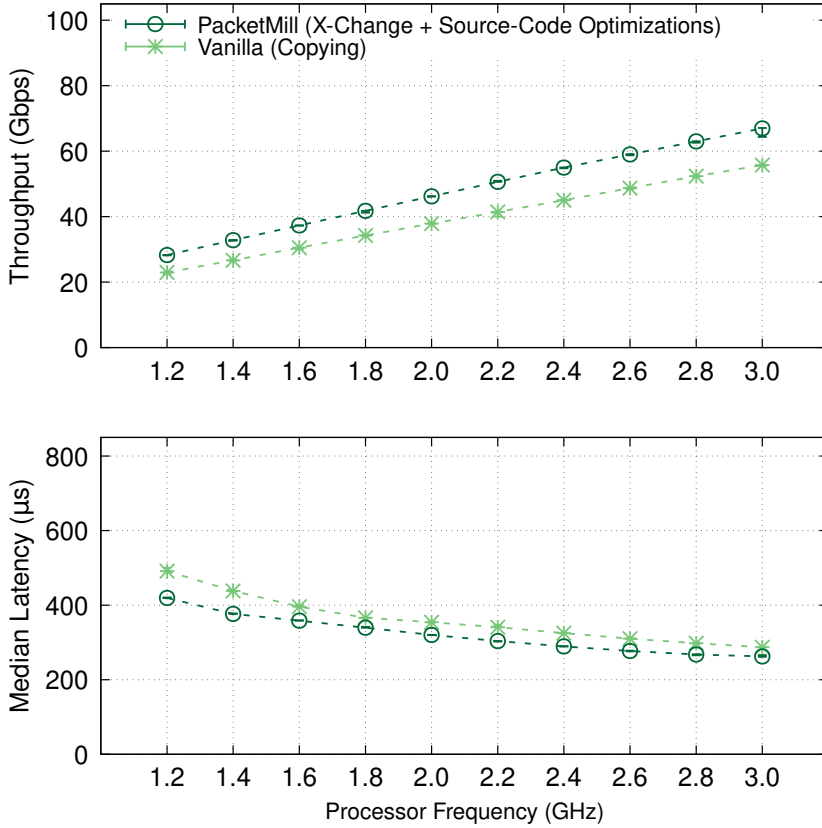


Figure 3.7: PacketMill improves the performance of a more compute-intensive NF (*i.e.*, an IDS+router running at 2.3 GHz), by up to 20% throughput and 17% latency [45].

headers and encapsulating the packet in a VLAN header. Figure 3.7 shows the throughput & median latency vs. frequency curve for the IDS+router. These results demonstrate that PacketMill is also beneficial for more CPU-demanding NFs.

To generalize this notion to more sophisticated NFs, we use FastClick’s `WorkPackage` [139] element to emulate the behavior of more memory- and compute-intensive functions. This element generates  $N$  ( $N$  is 1 and 5 in our configuration) random accesses (per-packet) to a static memory of  $S$  MB. Additionally, it generates  $W$  pseudo-random numbers to simulate more CPU-bound workloads. Figures 3.8a and 3.8b show 3D colormaps of improvements

for different values of  $W$  and  $S$ , when a core is performing different numbers of random accesses per packet (*i.e.*,  $N$ ). In these figures,  $X$  &  $Y$  axes represent  $W$  &  $S$  while the  $Z$  axis & colormap show the PacketMill's improvements & Vanilla's throughput, respectively.

While it is difficult to come up with a unified benchmark that succinctly captures a wide variety of applications, these results demonstrate that PacketMill is beneficial for a wide range of CPU- and memory-bound NFs. PacketMill's gain reduces when the application becomes less I/O intensive; in other words when its throughput decreases, *i.e.*, the lighter the color, the lower the  $Z$ . Additionally, comparing Figures 3.8a and 3.8b (*i.e.*,  $N = 1$  vs.  $N = 5$ ) shows that increasing the number of accesses per packet amplifies the impact of increasing memory intensiveness, reducing Vanilla's throughput and PacketMill's improvements. The key behind these gains is PacketMill's highly efficient use of the underlying hardware. It is worth mentioning that the results of this section could underestimate PacketMill's improvements for real NFs, as the emulated NFs presented in this section do not contain a complicated processing graph (as opposed to real NF chains).

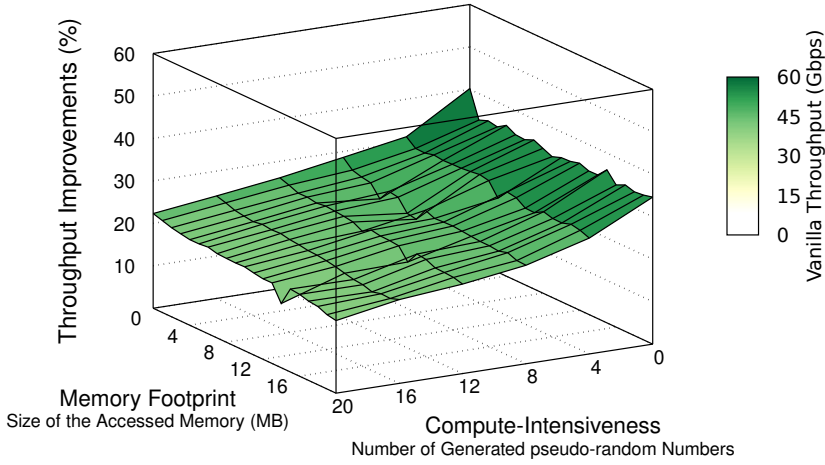
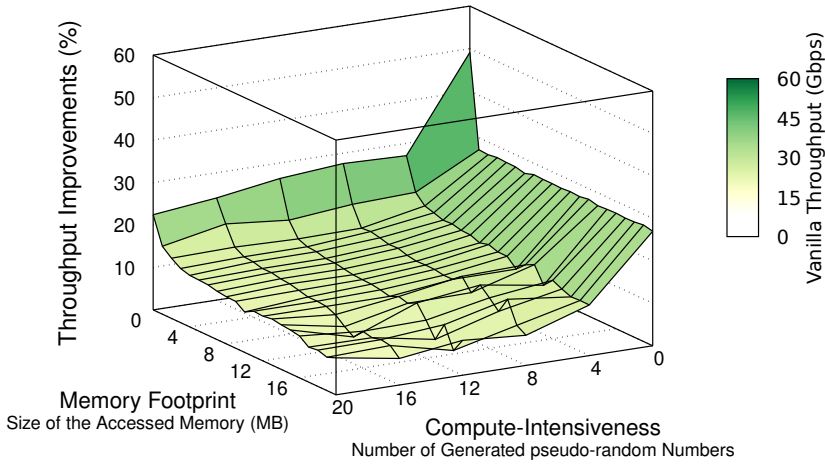
(a) 1 access per packet ( $N = 1$ ).(b) 5 accesses per packet ( $N = 5$ ).

Figure 3.8: PacketMill is effective for sophisticated network functions. Left and right figures show synthetic NFs that perform one and five memory accesses per packet, respectively. The improvements reduce when memory footprint, compute-intensiveness, and/or number of accesses per packet of an NF increases. A WorkPackage+router is running at 2.3 GHz [45].

**Impact of memory intensiveness.** To have a more detailed understanding of memory intensiveness, we zoom into a slice of Figure 3.8a where an NF performs a single memory access per packet ( $N = 1$ ) and generates four random numbers ( $W = 4$ ), *i.e.*, doing light-weight processing\*. Figure 3.9 shows the impact of changing memory footprint on throughput, LLC load misses, and LLC kilo loads. Comparing the three sub-figures, we can make the following observations:

1. We notice that Vanilla's throughput is inversely proportional to Vanilla's LLC loads—PacketMill also shows similar behavior.
2. The number of LLC loads gets saturated when the size of the accessed memory is increased to  $\sim 3$  MB, which suggests the threshold where almost all of the memory accesses are being done through LLC; see the bottom sub-figure in Figure 3.9.
3. The number of LLC loads is never zero, even for accessed memory sizes smaller than 1 MB. This observation implies that the application is still not L1/L2 bound, as there is always a considerable amount of LLC accesses, most probably due to the application footprint and DDIO [27].
4. The percentage of LLC load misses increases after  $\sim 14$  MB, highlighting the point where the application starts accessing the main memory (*i.e.*, DRAM), see the middle sub-figure in Figure 3.9. However, the performance is not substantially affected, as a large fraction (*i.e.*,  $\sim 90\%$ ) of LLC loads are still hitting LLC.
5. PacketMill results in more LLC loads and LLC load misses per 100 ms, as PacketMill is processing more packets.
6. PacketMill's improvements are consistent for this specific NF that performs one memory access per packet.

---

\*This specific slice can represent an emulated simple Key-Value Store (KVS) with variable memory footprints.



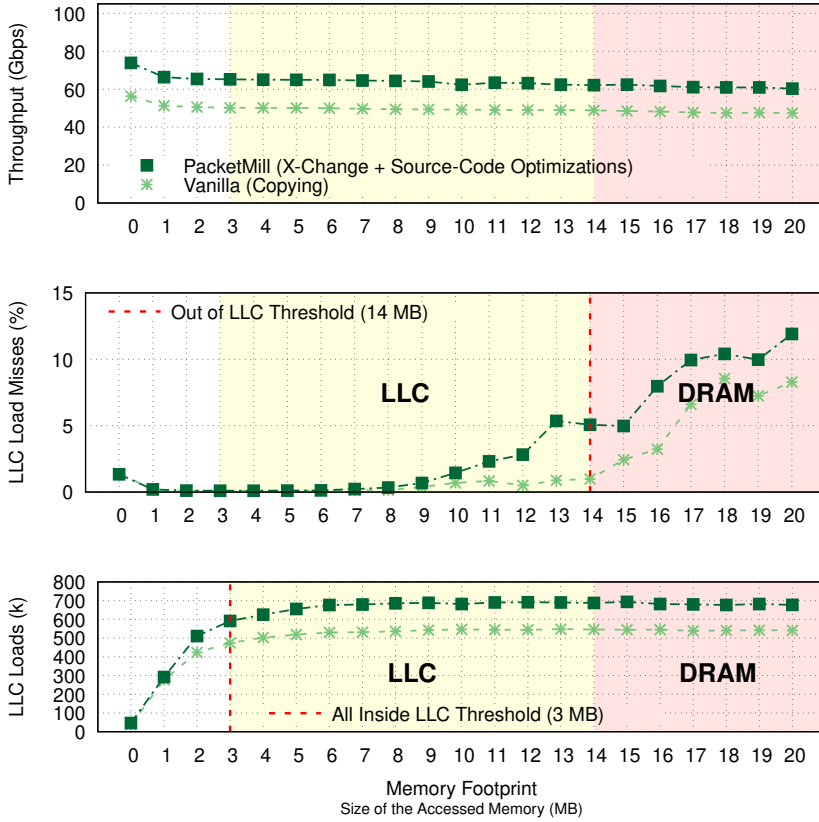


Figure 3.9: Increasing memory intensiveness results in larger number of LLC loads that is inversely proportional to the performance of the synthetic NF. From top to bottom, the figures show throughput, LLC load misses, and LLC loads, respectively [45].

### 3.3.6 Is PacketMill Useful for Multicore Network Functions?

The evaluation above mainly focused on single-core performance to highlight the gains achieved by our approach. Figure 3.10 shows that applying PacketMill to multicore NFs is also beneficial; in this case, for a NAT with different numbers of cores\*. These results demonstrate that the benefits of applying PacketMill to multicore NFs is comparable to the single-core improvements shown in Figure 3.8 on page 70.

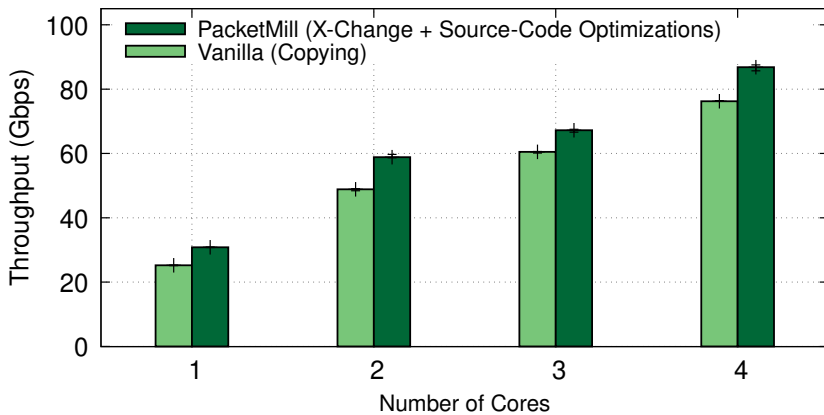


Figure 3.10: PacketMill also improves the performance of multicore NFs. A NAT is running at 2.3 GHz [45].

### 3.3.7 How about State-of-the-Art Packet Processing Frameworks?

A fair comparison between PacketMill and state-of-the-art packet processing frameworks (*e.g.*, BESS [113, 112] & VPP [110, 140]) requires (i) developing a high-performance NF in pure DPDK, (ii) modifying those other frameworks to enable the case for source-to-source optimizations & LLVM pass that reorders metadata, and (ii) devising scenarios/experiments to avoid any incorrect conclusions, which is beyond the scope of this dissertation (such a comparison was done previously by Tianzhu Zhanget *al.*, [141]). However, we have performed a simple comparison to, specifically, show the full potential of X-Change. This section compares the performance of a simple forwarding application running via FastClick, PacketMill, DPDK, DPDK+X-Change, BESS, and VPP.

\*We use RSS to distribute packet among different cores.

For the DPDK+X-Change case, we developed a sample application, called `l2fwd-xchg`, for DPDK to support X-Change, which is a modified version of the L2 forwarding sample application (`l2fwd`). In this example, the metadata is reduced to *two* simple fields (*i.e.*, the buffer address and packet length) instead of the 128-B `rte_mbuf`. This application\* can also serve as a template for developers to write their own applications, benefiting from X-Change. Note that since X-Change currently does not support vectorized PMD, we disabled it for all experiments. However, this should not affect the improvement trend, as a full vectorized implementation of X-Change would still result in the same benefits, addressing the inefficiencies of current metadata management models. Extending X-Change to support vectorized PMD remains as our future work.

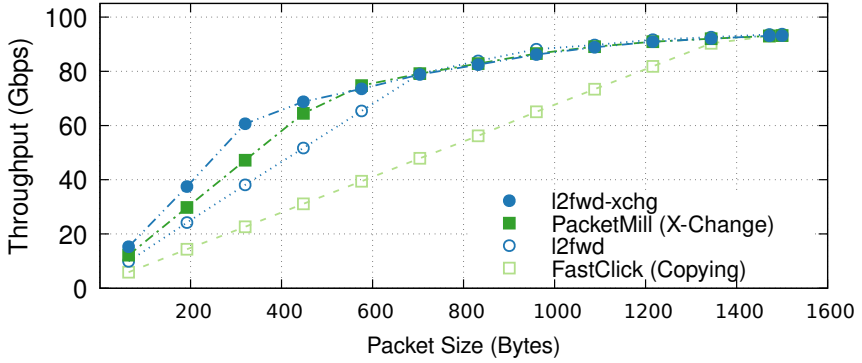
Figure 3.11 shows the results of our experiments when different frameworks/applications are forwarding fixed-size packets while a single core is running at 1.2 GHz. Increasing the frequency would eventually result in the same behavior as Figure 3.4a, as these applications perform simple forwarding operations, hiding the full potential of X-Change due to other bottlenecks (*e.g.*, using one RX/TX queues).

**FastClick vs. DPDK.** Figure 3.11a compares the performance of DPDK-based forwarding applications with default FastClick (*i.e.*, it uses Copying model) and PacketMill (*i.e.*, it uses X-Change). These results shows that PacketMill enables FastClick to process packets faster than `l2fwd` that is a simple forwarding application with minimal features & footprint and limited metadata. Additionally, it shows that applying X-Change to even simple DPDK-based applications could result in significant improvements, as `l2fwd-xchg` is forwarding packets up to ~59% faster than `l2fwd`.

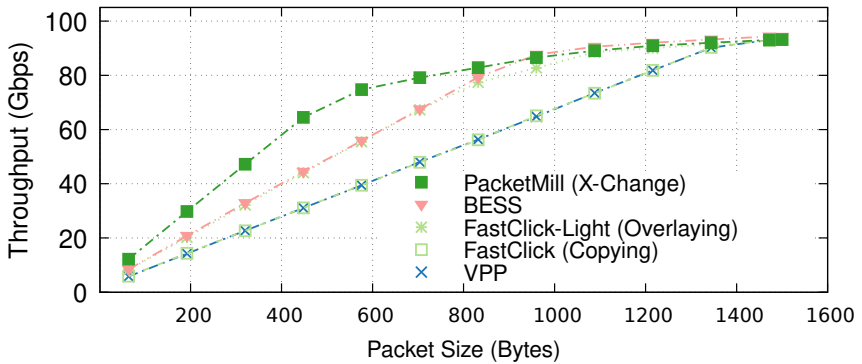
**FastClick vs. BESS vs. VPP.** Figure 3.11b compares the achieved throughput of different modular packet processing frameworks when they *only* perform simple forwarding. These results show that PacketMill achieves the best overall performance. Additionally, both VPP and default FastClick achieve similar performance, as both of them use the Copying model that performs an extra copying. Moreover, these results show that FastClick can achieve similar throughput as BESS after disabling extra features and using the Overlaying model.

---

\*It is available at [tbarbette/xchange/examples/l2fwd-xchg](https://github.com/tbarbette/xchange/examples/l2fwd-xchg)



(a) X-Change improves the performance of two DPDK-based applications (*i.e.*, FastClick and l2fwd application). Moreover, PacketMill enables FastClick to process packets faster than a simple forwarding application (*i.e.*, l2fwd) with limited metadata.



(b) PacketMill performs better than other packet processing frameworks.

Figure 3.11: Comparison of packet processing frameworks forwarding fixed-size packets with a single core [45].

### 3.4 Frequently Asked Questions

**Why should I use PacketMill instead of PGO?** Specializing a binary for a specific workload via profiling + re-compiling, *aka* Profile-Guided Optimization (PGO), has recently gained a lot of interest, due to modular Programming Language (PL) tools (*e.g.*, LLVM toolchain) and convenient/accessible profiling tools (*e.g.*, perf). For example, Bolt [142] and Propeller [143] have shown that exploiting dynamic program control flow information, extracted via profiling, could improve large-

scale applications' performance. However, these techniques work best when there exists *a* defined workload, rather than different and varying workloads.

Software packet processing frameworks typically have to process various kinds of packets. For instance, a router should handle ARP requests while forwarding different versions/types of IP packets, which requires different execution paths, thereby reducing cache locality. To improve locality, Metron [16] exploited modern networking equipment to distribute traffic classes to different cores, *i.e.*, each core independently operates on a specific workload. Although Metron defines a per-core workload, it cannot substantially benefit from PGO\* since it relies on one codebase for all execution paths (or traffic classes). PacketMill could be extended to duplicate the necessary code per-traffic class and benefit from PGO, which remains as our future work.

**Does PacketMill affect the correctness?** Premature optimizations could be *the root of all evil* [144], as it may result in unexpected bugs. While our goal was to emphasize the impact of low-level optimizations for 100-Gbps packet processing, deploying optimized network functions should be accompanied by a verification stage to *formally* prove the correctness of the optimized NF and avoid bugs. Since PacketMill relies on optimizing the whole IR code, it is possible to integrate our system with a IR-based symbolic execution engine (*e.g.*, KLEE [116]), as done by VigNAT [145] and Vigor [146]. Furthermore, using a symbolic execution engine could facilitate further optimizations as demonstrated in NFReducer [115] and Castan [147].

**Why should I trust your measurements?** Mytkowicz *et al.*, [148] showed that measurement bias is a common phenomenon in evaluating computer systems. We have taken the following measures to *avoid* drawing incorrect conclusions.

- ① We use NPF to perform our experiments to ensure testbed consistency— as it clones & compiles repositories, sets up & configures the testbed, and collects measurements. Additionally, it randomizes the environment variables in each run to mitigate the measurement bias due to stack alignment.
- ② We randomize the location where the binary is loaded for each run of the experiment using Address Space Layout Randomization (ASLR).
- ③ We use different NF configurations and various traces to broaden our evaluation space. The best approach to mitigate measurement bias is to use & develop tools, such as STABILIZER [149], which is no longer maintained. Developing a tool for evaluating software packet processing frameworks' performance remains as our future work.

**Is PacketMill applicable to other frameworks?** Although we have designed PacketMill with an eye toward optimizing Click-based packet processing, our optimizations are not limited to Click, as the code inefficiencies (*e.g.*, unembedded

---

\*See PacketMill's public repository for more information.

graph and parameters) are also present in other packet processing frameworks such as BESS and VPP. Moreover, PacketMill's X-Change can be easily adapted to improve the performance of other frameworks since it modifies DPDK userspace drivers, making it a common optimization for DPDK-based frameworks. As our target was to achieve 100-Gbps per core, we modified the MLX5 driver used by NVIDIA/Mellanox NICs. However, X-Change is applicable to other drivers, as other (*e.g.*, Intel) drivers are implemented similarly and have the same inefficiencies. Moreover, our techniques & optimizations could be combined with other existing middlebox optimizations such as Metron [16].

**Would PacketMill still be relevant given current technology trends?** While many try to exploit recent accelerators to improve packet processing, we think our approach would still be relevant for two reasons: (*i*) current accelerators (*e.g.*, programmable switches & NICs) have many limitations, which force applications to perform all stateful processing in the commodity hardware, and (*ii*) our research shows that performing holistic optimizations makes it possible to achieve performance similar to accelerators while still benefiting from the flexibility of commodity hardware. Moreover, we believe PacketMill optimizations become even more critical with increasing link speeds (*i.e.*, 200 and 400 Gbps).

**What are future directions for PacketMill?** PacketMill's use of LTO and access to the whole program's IR facilitates application of additional techniques to improve & evaluate the performance of software packet processing frameworks, such as: *llvm-mca* [150] (*i.e.*, successor to the Intel® Architecture Code Analyzer (IACA) [151]) for performance estimation, IR-based *superoptimizers* [152] (*e.g.*, Souper [153, 154]), and/or customized code generation/instruction scheduling [155]. Additionally, it is possible to extend our code optimizations with new passes, *e.g.*, performing liveness analysis to overlay the metadata that is not alive at the same processing stage. Examining these techniques remains as future work.

## 3.5 Related Works

This section discusses other efforts relevant to our work.

**Profile-guided optimizations.** The work of K. Pettis *et al.*, in 1990 is the basis for the majority of profile-guided code reordering and optimization works. Many algorithms [156, 157, 158, 159, 160] and tools, such as AutoFDO [161], Ispike [162], PLTO [163], and Codestitcher [164] focus on the CFG and try to carefully layout the chain of frequently executed basic blocks to increase locality. Other efforts focus on function placement [165, 166, 167, 142] to merge the chain of commonly executed functions to reduce iTLB misses. Although we do not employ profiled-guided optimizations (*i.e.*, basic block reordering & function placement),

we believe these efforts would complement our work, see Section 3.4.

**Fast packet processing.** In addition to the works discussed throughout the chapter, the work on NFV performance acceleration can be classified into three categories: ① relies on hardware accelerators to improve processing speed by offloading (part of) packet processing into an FPGA, GPU, or modern NIC [168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179]; ② focuses on NFV execution models and tries to improve the performance of either the pipeline/parallelism model [180, 181, 16, 182, 183] or Run-to-Completion (RTC) model\* [112, 189]; and ③ improves the performance of NFV by reducing/eliminating redundant operations and/or merging similar packet processing elements into (one) consolidated optimized equivalent [190, 17, 191, 192, 193, 182]. The second category also includes efforts toward better scheduling & load balancing [184, 185, 186, 187, 188, 194] or more efficient I/O [27, 26]. Our work is orthogonal and complementary to these.

## 3.6 Summary

In this chapter, we highlighted the importance of *low-level* optimizations in order to utilize the full potential of commodity hardware. We introduced PacketMill that addressed some of the software limitations in packet processing by mitigating code inefficiencies & improving metadata management. Our evaluation results showed that PacketMill increases throughput (up to 36.4 Gbps, *i.e.*, 70%) & reduces latency (up to 101  $\mu$ s, *i.e.*, 28%) and enables nontrivial packet processing (*e.g.*, a router) at  $\approx 100$  Gbps, when new packets arrive  $> 10\times$  faster than main memory access times, while using only *one* processing core. The main takeaway of this chapter is that efficient packet processing at multi-hundred-gigabit-per-second rates calls for holistic system optimizations, *i.e.*, *milling* the whole solution/software stack to *squeeze* every bit of performance from the available hardware.

---

\*This category also includes efforts toward better scheduling & load balancing [184, 185, 186, 187, 188] or more efficient I/O [27, 26].

## Chapter 4

# Toward Secure Multi-Hundred-Gigabit-Per-Second Services

**T**HE introduction of multi-hundred-gigabit-per-second network interfaces (100/200/400 Gbps) has motivated many researchers to study networked systems & their underlying hardware to understand the challenges of achieving suitable performance at higher rates. These studies have explored various topics, such as Linux kernel network stack [48], PCIe [49], RDMA [50], DDIO & LLC [27, 26], and smart NIC [51, 52]. Most studies focused on improving performance and paid less attention to other aspects, such as security and privacy, which are critical real-world requirements. Consequently, these studies often *disable* many software & hardware features to maximize the performance of networking applications. This chapter examines one of these features, specifically the *IOMMU*, which facilitates I/O virtualization and provides I/O security, to quantify its performance overheads at multi-hundred-gigabit-per-second rates.

IOMMU makes it possible to use virtual addresses rather than physical addresses when performing I/O via DMA, thereby creating an abstraction layer between the I/O devices and the processor. IOMMU provides hardware, similar to the Memory Management Unit (MMU) and TLB, to translate I/O Virtual Addresses (IOVAs) to their respective Physical Addresses (PAs). This can be used to restrict DMAs from I/O devices to only limited regions of the processor’s memory address space (*i.e.*, the pages mapped to the IOMMU hardware), hence (*i*) providing a higher degree of protection and (*ii*) easing virtualization & backward compatibility, which comes *at*



*the cost of performance*. We aim to characterize the performance overheads of the IOTLB *and* then mitigate these overheads for high-speed networking applications running on the Linux kernel.

Throughout this dissertation, we mainly concentrated on improving the performance of packet processing deployed on top of the kernel-bypass frameworks (*e.g.*, DPDK). In contrast, this chapter primarily focuses on the performance of TCP-based applications deployed on top of the Linux kernel\*. This chapter starts by providing the necessary background on the IOMMU and Linux buffer management; then it provides a thorough characterization of the performance impact of IOMMU & IOTLB at high network speeds, and its dependency on multiple parameters (*e.g.*, offered load, MTU, and network features offloads). Finally, it carefully investigates the challenges of employing hugepages in Linux network device drivers, to find a sweet spot between memory stranding, page locality, and CPU overhead; and then demonstrates the effectiveness of employing hugepages to mitigate IOTLB overheads at 200 Gbps using our *prototypical* hugepage-aware memory allocator (called HPA) for device drivers.

## 4.1 Background

IOMMU provides hardware support for I/O virtualization, allowing DMA operations to use virtual addresses rather than physical addresses. IOMMU hardware is present in most recent processors (*e.g.*, Intel VT-d[195], AMD-Vi[196], and ARM SMMU[197]), and typically performs two main tasks: (i) DMA remapping and (ii) interrupt remapping.

**DMA remapping** maps virtual addresses in DMA operations to physical addresses in the processor's memory address space. Similar to the MMU, the IOMMU uses a multi-level page table to keep track of the IOVA-to-PA mappings at different page-size granularity (*e.g.*, 4-KiB, 2-MiB, and 1-GiB pages). The hardware also implements a cache (*aka* IOTLB) of page table entries to speed up translations.

**Interrupt remapping** enables I/O devices to trigger interrupts by performing DMAs to a dedicated memory range [198, 199]; thus, Virtual Machine (VMs) can receive interrupts from I/O devices. Note that the IOMMU cannot differentiate interrupts and DMAs without interrupt remapping.

**Use cases.** Besides providing isolation and protection against DMA attacks, the IOMMU can be used to (i) support 32-bit PCIe devices on x86-64 systems, (ii) hide physical memory fragmentation, and (iii) perform scatter-gather I/O operations.

---

\*Section 4.5 elaborates on the importance of having high-performance Linux-based solutions alongside kernel-bypass frameworks (*e.g.*, DPDK).

**Monitoring capabilities.** Intel VT-d and AMD-Vi provide performance counter registers to measure different IOMMU-related metrics, such as the number of hits & misses in IOTLB and in various levels of the I/O page table.

#### 4.1.1 IOMMU Performance Impact on DMA Operations

**Basic model.** I/O subsystems have an upper bound on the number ( $N$ ) of outstanding DMA transactions. This implicitly creates a throughput bottleneck  $B \propto N / (T_{\text{MEM}} + T_{\text{IOMMU}})$ , where  $T_{\text{MEM}}$  is the time to complete a transaction without IOMMU, and  $T_{\text{IOMMU}}$  is the time to perform related address translations. In ideal conditions,  $B$  exceeds the desired data rate and the capacity of other buses (*e.g.*, NIC & PCIe) in the data path; but exceedingly high translation times (*e.g.*, due to excessive IOTLB misses) can reduce  $B$ , making it the bottleneck.

**Dependencies.**  $T_{\text{IOMMU}}$  is negligible if a translation can be served by the IOTLB; otherwise, it depends linearly on the probability and number of IOTLB misses per translation. Misses in turn are heavily affected by (i) the size and management policy of the IOTLB (unfortunately, these are unmodifiable hardware features [200, 49, 195, 201, 196], see Appendix B.1) and (ii) by the memory request patterns (and implicitly the IOTLB request patterns), which we study and optimize in this chapter.

**IOTLB miss metrics.** The absolute number of IOTLB requests and misses can be determined using dedicated performance counters on the processor. Depending on the data direction and system configuration, each block of data may require one or more translations. As a consequence, neither the absolute value nor the percentage of IOTLB misses are especially meaningful to assess their effect, or compare system behavior with different configurations. Instead, we found that a metric of *IOTLB misses per MiB* (or per unit of data) is especially convenient, as this metric can be compared between different system configurations (*e.g.*, different MTU and buffer organizations).

**Buffer shuffling and positive feedback loops.** Common buffer management strategies try to reduce IOTLB misses by improving buffer address locality. However, in the presence of traffic drops and retransmissions, accesses to data buffers end up being shuffled, thereby causing more IOTLB misses, more drops, and more shuffling in a positive feedback loop. Consequently, the system may enter a poor performance regime from which it is difficult to escape, see Section 4.2.7 & Section 4.4.3.

## 4.1.2 Networking Data Path in the Linux Kernel

We briefly summarize the data path in the Linux kernel for both RX & TX paths [202, 48].

### 4.1.2.1 RX path

To receive packets from the wire, the network driver initially populates the RX queues of the NIC with pointers to *receive buffers* allocated from host memory. The size of receive buffers and their number per queue (*aka* RX descriptors) are configurable at run-time. Upon incoming packets, the NIC issues DMA transfers over the bus (typically PCIe) to move incoming data from the NIC's internal buffers to the receive buffers in host memory (or cache [27]). When one or more DMA transfers are complete, the NIC issues an interrupt, and the CPU core running an interrupt driver attaches newly filled receive buffers to a packet descriptor (`sk_buff`) and passes them to the kernel for further protocol processing. The RX queue is then replenished with new receive buffers. Eventually, applications will issue a `recvfrom()`, or an equivalent system call, to consume received buffers. In non-zero-copy cases, issuing a receive system call will copy the content of in-kernel receive buffers to userspace buffers, making it possible to immediately return the backing memory to the kernel. However, using a receive zero-copy mechanism could delay returning the memory to the kernel, as the application may hold buffers longer. To the best of our knowledge, there are only a few experimental techniques (with many limitations) to perform receive-side zero-copying within the Linux kernel. These techniques require the applications to return the buffers within a reasonable time; otherwise, other optimizations to reduce memory mapping costs would break. Therefore, most zero-copying applications rely on userspace network stacks and kernel-bypass frameworks, which target a different domain than the work described in this chapter. Section 4.5 elaborates on the importance of having high-performance Linux-based solutions alongside kernel-bypass frameworks.

**Receive buffer management.** NIC device drivers use a custom or general-purpose *RX buffer pool* (e.g., the Page Pool API\* [203]) to speed up the allocation and deallocation of receive buffers. The RX buffer pool generally implements a per-CPU-core cache of receive buffers, with optimizations to re-use and *recycle* the receive buffers once they are consumed by applications. Buffer recycling reduces calls to the kernel's page allocator, which may improve memory access locality and in turn, reduce the MMU and IOMMU translation costs.

---

\*See Appendix B.3 for implementation details.

Each RX buffer pool is used concurrently by multiple CPU cores (*e.g.*, the core serving interrupts requests new buffers, while cores running applications release buffers after completing `recvfrom()` system calls). Therefore, the RX buffer pool may trade recycling efficiency for reduced lock contention. Additionally, applications may be slow in consuming data, consequently depleting the cache and forcing new allocations. Moreover, buffers are often delivered to different applications, hence these buffers are released in a different order, which over time may cause *buffer shuffling* and reduce address locality.

**Receive buffer lifetime.** If a non-zero-copy application is *not* starved of CPU or poorly designed, a receive buffer lifetime will be reasonably short, making it possible to recycle it fast enough. However, packet drops and TCP re-transmissions cause the buffers to be held for one or more Round-Trip Times (RTTs), delaying the buffer recycling. Section 4.2.7 discusses the impact of packet drops on IOTLB misses. This chapter does not focus on kernel-based applications that hold kernel buffers for unreasonably long times, as they have much worse throughput bottlenecks than IOMMU & IOTLB.

#### 4.1.2.2 TX path

Transmissions are initiated by applications calling `sendmsg()` or equivalent system calls, which allocate in-kernel *transmit buffers* to copy user-supplied data, attach them to `sk_buffs`, and pass them to the network stack; some may use zero-copy mechanisms, see Section 4.5. Eventually, the addresses of the data buffers will be passed to the NIC's TX queue so that the NIC will be able to issue DMA transfers and perform the transmission. When the transmission is complete, the NIC issues an interrupt and the device driver can return the buffer memory to the kernel.

**Transmit buffer management.** Similar to the RX path, the TX path may use caches to allocate & release buffers. The constraints are slightly different, as in this case there may be multiple CPU cores allocating buffers, and only one core releasing them.

#### 4.1.2.3 Operation When IOMMU is Enabled

When IOMMU is enabled, the driver must create IOMMU mappings for receive buffers & descriptors, and pass IOVAs instead of physical addresses to the NIC. The granularity of these mappings is constrained by the granularity of physical pages backing these memory buffers. Currently, DMA mapping is mainly performed at the 4-KiB page granularity. For an IOMMU address space of 48 bits, one address

translation requires accessing four entries in the IOMMU page table\*. When a buffer is not used for DMA anymore, its IOMMU mapping is not needed anymore. Depending on the protection vs. performance trade-off policy, the operating system may decide to remove the mappings right away (*i.e.*, a very expensive operation, since it involves flushing the IOTLB entries); do the unmappings in batches at a later time; or keep the mappings alive in case the buffers are recycled. Note that deferring the unmappings slightly relaxes the security guarantees provided by IOMMU, but reduces its overheads [206].

## 4.2 IOMMU Performance Characterization

As reported in previous works (such as [207]), enabling & using IOMMU typically causes a drop in network bandwidth, due to factors such as (i) IOVA (de)allocation [202, 208, 209], (ii) IOTLB invalidation [206, 210], and (iii) IOTLB misses. Previous works have mainly focused on the first two factors, which typically show up as additional CPU load. This section studies the correlation between the IOTLB misses and the throughput drop, which is *independent* of CPU load. A common optimization is to make mapping entries more long-lived, thereby making the most out of the IOTLB while addressing security concerns. In particular, DAMN [202] presented a DMA-aware memory allocator that used permanently mapped buffers for packet buffers to re-use pages for DMA-ing. DAMN primarily focused on 4-KiB pages and measured IOMMU performance only for 9000-B jumbo frames. However, our work looks at a different aspect of the problem, *i.e.*, **data path impact of IOMMU and IOTLB misses**.

Following the description in Section 4.1.1, IOMMU translation costs do not affect throughput up to the point where serving IOTLB misses takes too long, thus creating a throughput bottleneck. The IOTLB antagonization threshold, *i.e.*, IOTLB wall, depends in part on hardware features we cannot modify: the number and management of IOTLB entries, the maximum number of outstanding translation requests (*aka* I/O page walkers), and the completion time of an IOMMU translation reading entries from the main memory. A second dependency, which is partly under our control, is on the request pattern. Since IOTLB is a cache of the IOMMU page tables, we can reduce misses by managing buffers in a way that improves address locality. As an example, placing sub-4-KiB buffers belonging to the same 4-KiB page on consecutive RX descriptors generally saves at least one IOTLB miss for descriptors after the first one. Reducing the total number of buffers, making full use of buffer space, and batching accesses to descriptors, all may contribute to reducing the number of IOTLB misses.

---

\*48 bits address space with 4-B entries and 2048-B page-table size requires 4 page tables (each having 512 entries) [204, 205].

This section characterizes IOTLB behavior in a variety of scenarios, to better understand the correlation between the number of IOTLB misses and the throughput drop. We mainly rely on iPerf, a network performance measurement tool, to run microbenchmarks and characterize the performance of IOTLB. We use iPerf due to its efficiency in saturating a 200-Gbps link, whereas real-world applications (*e.g.*, Memcached) typically cannot reach line rate due to other throughput bottlenecks. Section 4.2.7 shows that our iPerf analysis would still be applicable to less I/O intensive applications. Most of our experiments use Intel Xeon IceLake processors with NVIDIA/Mellanox NICs, but Section 4.2.5 and Section 4.2.6 examines other Intel & AMD EPYC processors and a 100-Gbps Intel E810 NIC, respectively.

**Testbed.** Our testbed uses two workstations connected via a 200-Gbps link (or 100-Gbps in a few experiments); the RTT is negligible. One workstation acts as a traffic source & sink, without IOMMU for maximum performance. The other one acts as the DuT and runs different microbenchmarks with & without the presence of IOMMU; we use the default setting in Linux for IOMMU (*e.g.*, `intel_iommu=on` that uses deferred IOTLB invalidation [206]). Table 4.1 shows the different processors used for the DuT. In most experiments DuT is equipped with Intel Xeon Gold 6346 @ 3.1 GHz\* (*i.e.*, IceLake) and a 200-Gbps NVIDIA/Mellanox PCIe-4.0 ConnectX-6<sup>†</sup>; see Appendix B.2 for more info. In all cases the DuT runs Ubuntu-20.04.3 (Linux kernel 5.15). Unless stated otherwise, we set the core frequency to the processor’s maximal frequency (*i.e.*, 3.6 GHz), and the uncore frequency to its maximum (*i.e.*, 2.4 GHz), for best memory/cache access latency [136, 137]. To read IOMMU-related performance counters, we use the Performance Counter Monitor (PCM) tool (*i.e.*, `pcm-iio.x`) on Intel, and the Linux `perf` tool on AMD processors. IOTLB hit & miss values and their rates are derived from performance counters sampled once per second. We run our experiments automatically via the NPF tool [138] to improve the reproducibility of our results, and we mainly report the median of five 60-s runs, **with min/max error bars** (though in many experiments the range is small and almost invisible).

Table 4.1: Processors used for DUT. The entry marked with a star (★) is our primary testbed.

Generation \ Properties	Model	Physical Cores	Freq. (GHz)	LLC (MB)
Intel Xeon Skylake	Gold 6140	18	2.3	24.75
Intel Xeon CascadeLake	Gold 6246R	16	3.4	35.75
Intel Xeon IceLake ★	Gold 6346	16	3.1	36
AMD EPYC (3rd gen. Milan)	74F3	24	3.2	256

\*Nominal frequency

<sup>†</sup>We use the main slot of a Socket Direct PCIe 4.0 adapter.

**200-Gbps considerations.** We used the following options to operate iPerf at 200 Gbps without becoming CPU-bound, so our measurements emphasize IOTLB effects: (i) enable hyper-threading to increase available CPU cycles (our processors only have 16-24 physical cores each); (ii) run all interrupt and application threads on Socket 0, the socket local to the NIC (the cost of cross-socket memory access exceeds the benefits of using the additional cores on Socket 1); (iii) configure the NIC with one TX/RX queue pair and interrupt for each logical core on Socket 0; (iv) enable TCP Segmentation Offload (TSO) to reduce transmit CPU load, enable Generic Receive Offload (GRO), and enable/disable Large Receive Offload (LRO) depending on the configuration, see Section 4.2.3; and (v) enable interrupt moderation mechanisms as appropriate for the platform\*.

**iPerf configurations.** We use iPerf-2.0.14 [211] in TCP mode<sup>†</sup>, which uses multiple threads to transfer unidirectional traffic, with a configurable rate, over one or more TCP sockets between one sender and one receiver. In our experiments, we tune the number of TCP connections and message blocks to maximize iPerf throughput when IOMMU is disabled; more specifically, we use 384 TCP connections and 128-KiB blocks for read/write system calls (of course, TCP segments these blocks according to window bounds), and report the transport level throughput.

**Parameters.** We explore the impact of the following parameters on experiments: (i) *iommu configuration* (off, on, mapping size); (ii) *offered rate*,  $R_{IN}$ , enforced by rate limiting the sender, which affects both CPU load and IOMMU translation rates; (iii) *MTU*, which affects the receive buffer size and IOMMU translation rate for a given offered rate, and (iv) *platform*, to study the behavior of different CPUs and NICs.

MTU experiments use multiples of 1500 between 1500 and 9000 (both commonly used values), plus MTU=3690 that is the largest value allowed with 4-KiB buffers for our NIC. We have verified that our testbed can get within 2-3% of line rate at those MTUs<sup>‡</sup>.

---

\*We observed that the AMD EPYC 74F3 processor had a hard time dealing with a large number of interrupts per second, which occurs at high packet rates and with a large number of queues. We mitigate the problem by setting the sysctl parameter `napi_defer_hard_irqs=1` that was introduced in the Linux kernel specifically to address that issue.

<sup>†</sup>iPerf uses standard Linux system calls, and in UDP mode it cannot saturate a 200-Gbps link due to high per packet costs caused by per-packet system calls and fewer batching opportunities. Appendix B.5 investigates the IOTLB overheads on both throughput & latency with UDP traffic using DPDK-based FastClick [45, 24].

<sup>‡</sup>A very fine-grained sweep of MTU values showed some significant throughput drop with MTUs below 1000, or MTUs above 1500 and below 3000; see Appendix B.2. For other values, the testbed behaves as expected so there is no need to run subsequent experiments with too fine granularity.

The maximum achievable data and packet rate depend on the MTU. On the wire, each packet occupies the link for  $\text{WIRE\_SIZE} = (\text{MTU} + 38)$  bytes (*i.e.*, 14-B Ethernet header, 4-B CRC, 20-B preamble & inter-packet gap) limiting the maximum packet rate to  $\text{link\_rate}/(8 \times \text{WIRE\_SIZE})$ . Each packet carries a payload of  $\text{MSS}^* = (\text{MTU} - 52)$  bytes (20-B IP header, 20-B TCP header, 12-B TCP options in IPv4), thus the maximum achievable transport-level throughput (“application line rate” for brevity) is a fraction  $\text{MSS}/\text{WIRE\_SIZE}$  of line rate (*e.g.*, 0.9418 for MTU=1500, 0.9900 for MTU=9000).

**Metrics of interest** are averaged across the entire duration of the experiment. They include: (i) *throughput (TP)*, which deviates from offered rate in presence of bottlenecks; (ii) *throughput drop percentage*,  $100 \times \frac{(\text{TP}_{\text{NO\_IOMMU}} - \text{TP}_{\text{IOMMU}})}{\text{TP}_{\text{NO\_IOMMU}}}$ , which details the relative loss introduced by IOMMU; and (iii) *IOTLB misses per MiB*, which indicates how often we incur IOTLB misses. This unusual third unit is chosen to make the metric comparable across different experiments, as will be discussed later.

## 4.2.1 Data Rate and IOTLB Wall

Our first set of experiments explores the receiver behavior on IceLake (with 32 logical cores and a 200-Gbps NIC), with MTU=1500 B and 1024 RX descriptors. Figure 4.1 shows how the throughput and CPU utilization on a TCP receiver vary with offered rate. Without IOMMU, we almost reach line rate before becoming CPU-bound. Enabling IOMMU *on the receiver* caps the rate causing a significant *throughput drop*, not due to CPU overload (as CPU utilization remains around 60% in this experiment), but to reasons explained in Section 4.1.1.

Figure 4.1 shows the average throughput and CPU utilization at different offered traffic rates. **Without IOMMU**, throughput matches the offered rate up to ~160 Gbps, when occasional drops start to occur, and CPU load becomes super-linear because of the extra cost to process retransmissions and out-of-order traffic. At CPU saturation, the system achieves ~185 Gbps, not far from the theoretical maximum of 188 Gbps. **With IOMMU ON**, linearity breaks and drops start to appear much earlier, ~130 Gbps, and throughput eventually tops ~150 Gbps with CPU utilization never going above 60%.

To explain the throughput drop, which is clearly not due to a CPU bottleneck, we measured and show in Figure 4.2, the throughput drop percentage and the IOTLB misses per MiB at different offered rates.

---

\*Maximum Segment Size (MSS)



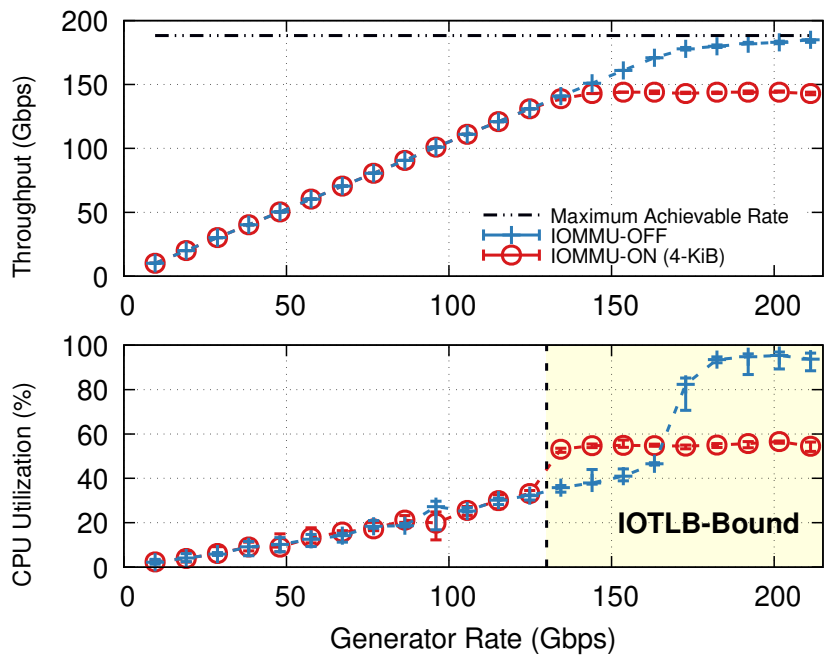


Figure 4.1: TCP throughput and receiver CPU utilization with and without IOMMU on the receiver (200-Gbps link, MTU=1500). Enabling IOMMU on a receiver puts a hard limit on the TCP throughput. Note that the receiver’s CPU utilization (lower graph) is well below saturation [53].

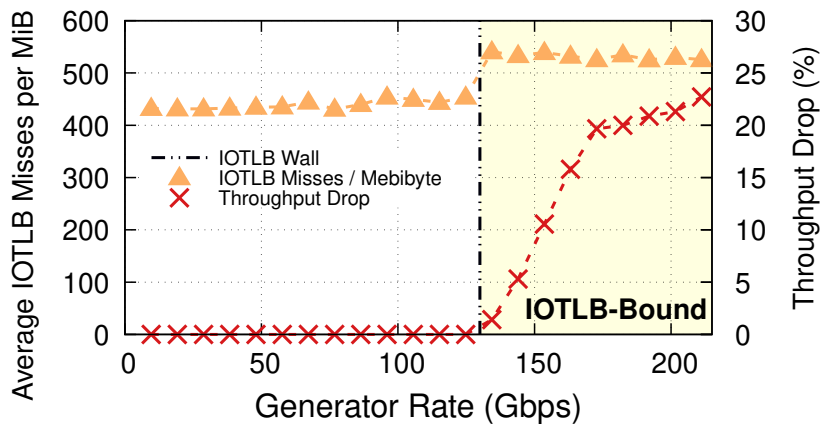


Figure 4.2: Throughput drop and IOTLB misses per MiB versus offered rate, MTU=1500. Misses have a sudden jump when the receiver starts dropping packets, because drops create buffer shuffling and in turn reduces address locality [53].

**IOTLB misses discussion.** In this experiment with MTU=1500 & MSS=1448, 1 MiB requires  $2^{20}/1448 \approx 725$  RX buffers, each one using half of a 4-KiB page. In normal conditions, it is very likely that contiguous RX descriptors use contiguous half pages; accessing the first half page will cause an IOTLB miss on the last level IOMMU entry, while the second half page will be served from the IOTLB. This alone causes  $\sim 420$  misses per MiB. The remaining 80-100 misses per MiB can be explained by misses on the next level IOMMU entries, and the interleaving of traffic on a large number of queues, which breaks the regular pattern and causes additional misses also on the next IOMMU level.

When packets start being dropped (around 130 Gbps), the misses per MiB jumps up suddenly. We theorize that the packet drops cause shuffling in received buffers, which in turn reduces locality and creates more misses per packet.

To prove our hypothesis, we ran another experiment with 4-KiB buffers (MTU=3690). In this case, we expect a last level IOTLB miss on every buffer, even without shuffling, but there are  $\sim 288$  buffers per MiB, fewer than before. Figure 4.3 proves our theory, and also shows that the throughput drop now occurs at a higher rate (*i.e.*,  $\sim 150$  Gbps) because of the lower number of misses per MiB. Note how the step in misses between the two regimes still exists, but it is smaller. The reason for this step is as follows: while shuffling within the same 4-KiB block causes no additional IOTLB miss, there is a significant number of 4-KiB blocks in each RX buffer pool (almost double the number of blocks for MTU=1500), and shuffling across these boundaries does cause extra IOTLB misses.

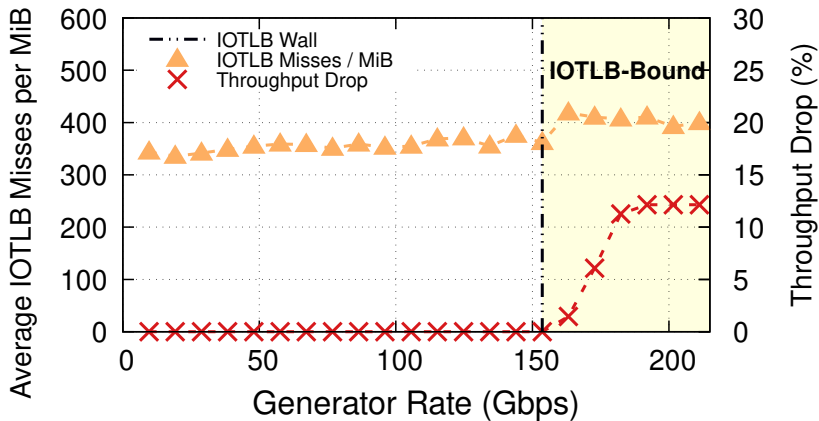


Figure 4.3: Throughput drop and IOTLB misses per MiB versus offered rate for MTU=3690. The modest reduction in misses compared to Figure 4.2 causes a significant increase in the rate where drops start to appear [53].

**Main takeaway.** Comparing Figure 4.2 and Figure 4.3 gives an important piece of information: a modest reduction in IOTLB misses per MiB (*e.g.*, from  $\sim 420$  to  $\sim 360$  as in our experiments) can significantly shift the rate at which we start experiencing drops. This knowledge will be useful when designing our mitigations.

### 4.2.2 Effect of MTU Size

We now explore the impact of different MTUs on throughput in two different regimes: (*i*) underloaded (100-Gbps offered load) and (*ii*) overloaded (200-Gbps offered load). Figure 4.4 shows the IOTLB miss per MiB for both regimes. The misses per MiB follow our expectations, with slightly higher values in the overloaded case, matching the steps we see around the IOTLB wall in Figures 4.2 and 4.3.

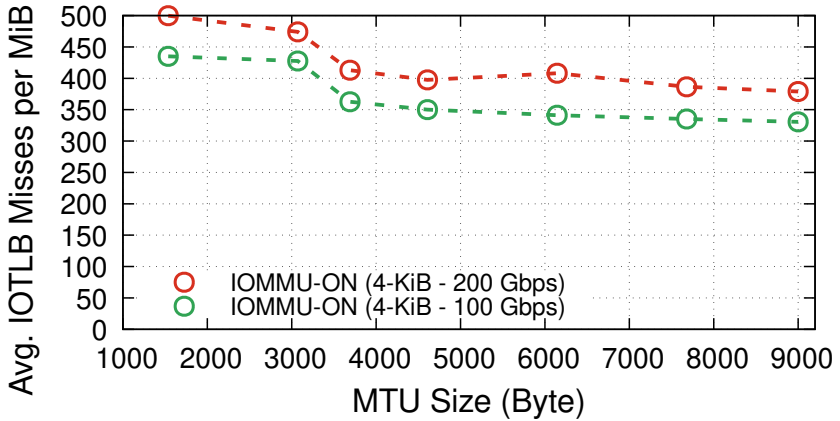


Figure 4.4: Receiver side average IOTLB misses per MiB for different MTU sizes, in underloaded (100-Gbps) and overloaded (200-Gbps) conditions [53].

When underloaded, with and without IOMMU, throughput matches the offered load, whereas overloading the system can cause additional drops. Figure 4.5 compares the throughput in the overloaded condition with and without IOMMU, where we see up to 20% drop introduced by IOMMU.

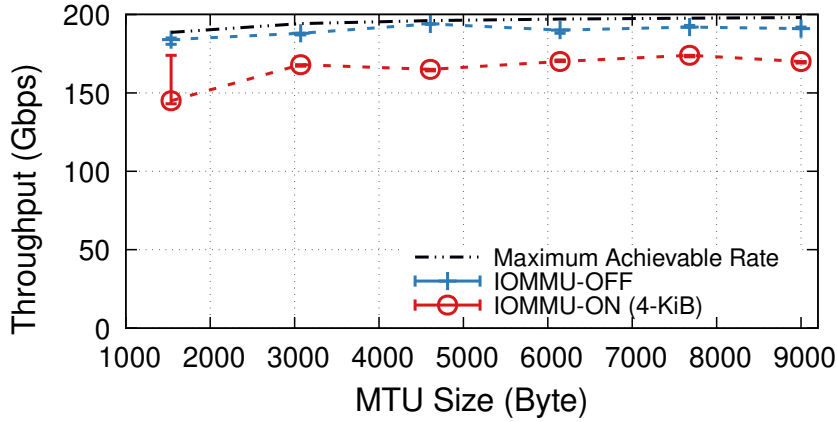
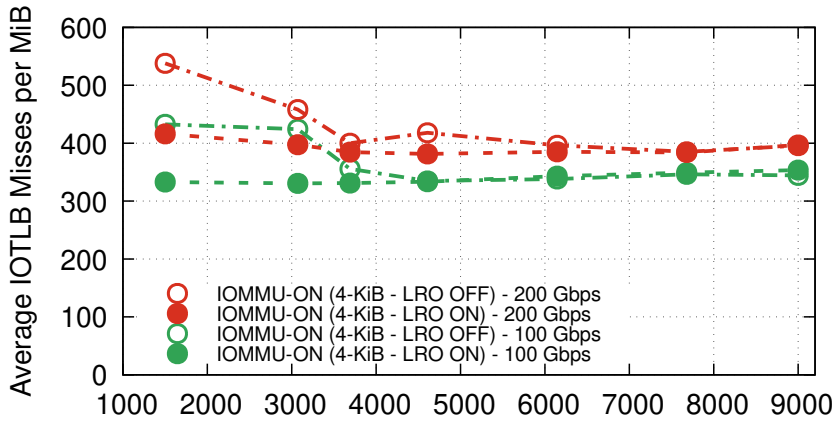


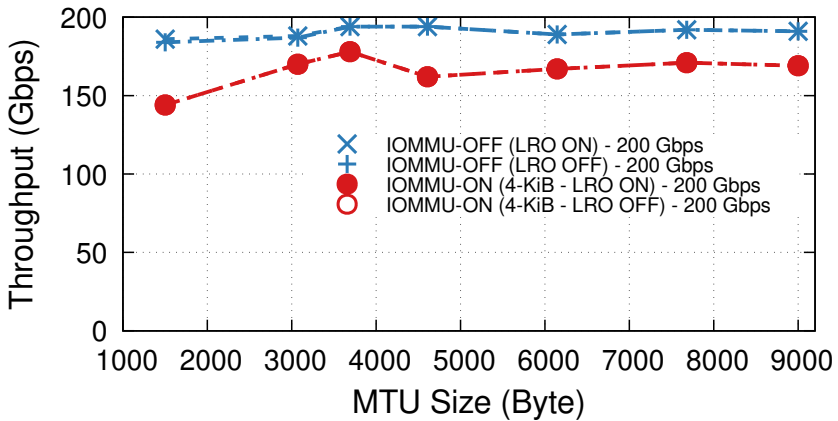
Figure 4.5: Throughput with/without IOMMU for different MTU sizes, in overloaded (200-Gbps) condition [53].

### 4.2.3 Large Receive Offload (LRO)

LRO enables the NIC to reassemble received *contiguous* packets into larger buffers, thus increasing address locality and reducing the number of IOTLB translations. However, the dynamics depend heavily on the buffer size. Figure 4.6a shows the effect of LRO on the average IOTLB misses per MiB in both underloaded and overloaded conditions. As expected, LRO effectively reduces IOTLB misses per MiB for sub-4-KiB MTUs where the NIC could assemble multiple buffers into a single 4-KiB page. Figure 4.6b shows that effect of LRO on throughput when operating at overloaded conditions with and without IOMMU; our processor is capable of achieving the same throughput without LRO.



(a) Average IOTLB misses per MiB.



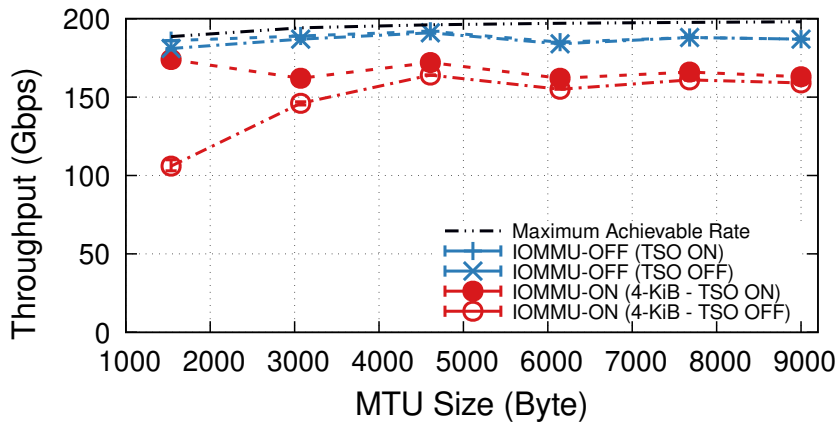
(b) Throughput in overloaded conditions.

Figure 4.6: Effect of LRO on IOTLB misses and throughput in underloaded and overloaded conditions. Note that LRO does not affect throughput in Figure 4.6b [53].

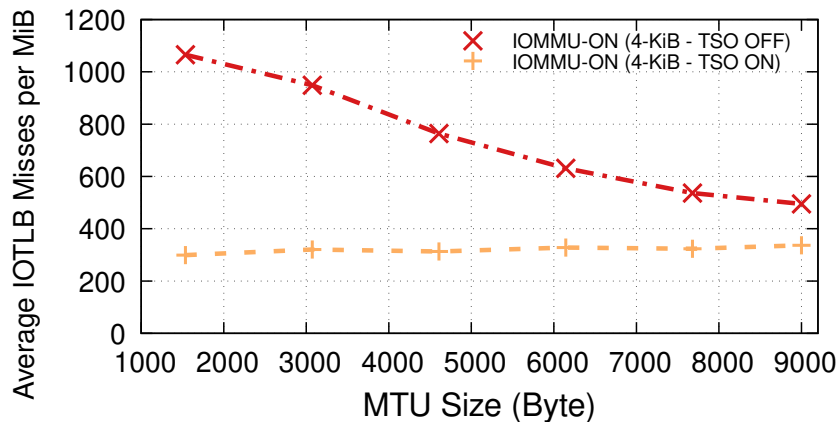
#### 4.2.4 TX & TCP Segmentation Offload (TSO)

This section explores the behavior of the DuT as a sender and studies the impact of IOMMU on the TX path. Figure 4.7a plots the maximum throughput achieved at different MTUs. Without IOMMU, the sender is close to the maximum theoretical line rate at all MTUs, with or without TSO. Enabling the IOMMU, without TSO, introduces a throughput drop even greater than on the receive side. Figure 4.7b

explains that the difference occurs due to a much higher number of IOTLB misses per MiB than what we experience on the receive side (see Figure 4.4). Without TSO, each TX packet is made of at least two memory buffers: one for the header, one for the payload, and sometimes more if the payload comes from two different `sendmsg()` calls (as opposed to the receiver where each RX packet uses a single buffer). Larger MTUs reduce the number of segments, resulting in fewer misses per MiB and higher throughput.



(a) Throughput at different MTUs with various combinations of IOMMU and TSO on the sender side.



(b) Average IOTLB misses per MiB with and without TSO.

Figure 4.7: Using TSO reduces the IOTLB overheads on the TX path due to fewer IOTLB translations [53].

With TSO, TX packets sent to the NIC are made of one header followed by up to 64-KiB of data, generally making full use of 4-KiB pages. The NIC takes care of segmentation, thus requesting approximately one translation request per 4-KiB of data (256 per MiB) irrespective of the MTU. This reflects perfectly in the TSO-ON curve in Figure 4.7b, with approximately constant  $\sim 300$  misses per MiB (the extra ones, once again, come from accesses to descriptors, interrupts, and next level IOMMU entries).

### 4.2.5 Different Processors

This section compares the impact of IOMMU on different processors. The key difference, for our study, is the PCIe bus version, which in turn affects the maximum PCIe bandwidth, NIC link speed, and also influences the vendor design choice of the maximum number of supported outstanding PCIe transactions. In fact, PCIe “posted data” credits, a hardware feature that affects the maximum number of outstanding PCIe write transactions, tend to be proportional to the bus speed (*i.e.*, PCIe-4 buses target approximately twice the target bus speed as PCIe-3 ones).

We separate the evaluation into two blocks: 100 Gbps and 200 Gbps, and present throughput without and with IOMMU, and IOTLB misses per MiB.

**100-Gbps link speed.** This set of experiments uses the NIC at 100 Gbps, which is the highest supported NIC link rate on PCIe-3.0. We compare Intel Skylake & CascadeLake (with PCIe-3.0) and our previously used IceLake (with PCIe-4.0).

Figure 4.8a shows the receiver throughput on the various platforms with and without IOMMU at different MTUs. Noticeably, one of our workstations is unable to achieve line rate at the smaller MTUs\*.

Enabling IOMMU further differentiates the behavior, with IceLake still being able to achieve line rate across the board, whereas SkyLake and CascadeLake experience throughput drop. More specifically, CascadeLake suffers from larger drops due to IOMMU across different sized MTUs.

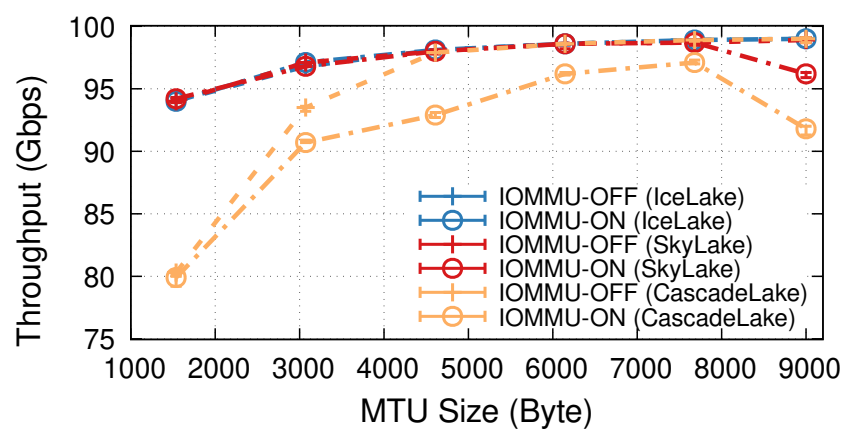
Figure 4.8b shows IOTLB drops per MiB with IOMMU. Despite similar numbers for IceLake and SkyLake, the latter performs significantly better, suggesting that it can handle a larger number of outstanding transactions without creating a bottleneck, per the model in Section 4.1.1. Our IceLake processor has a PCIe-4.0 bus, as opposed to the PCIe-3.0 used on Skylake.

Worth noting is the higher number of IOTLB misses for CascadeLake, which suggests a much higher number of misses in the next-level IOMMU table, in turn, an indication of a less effective IOTLB (smaller size, or different management) than the other processors.

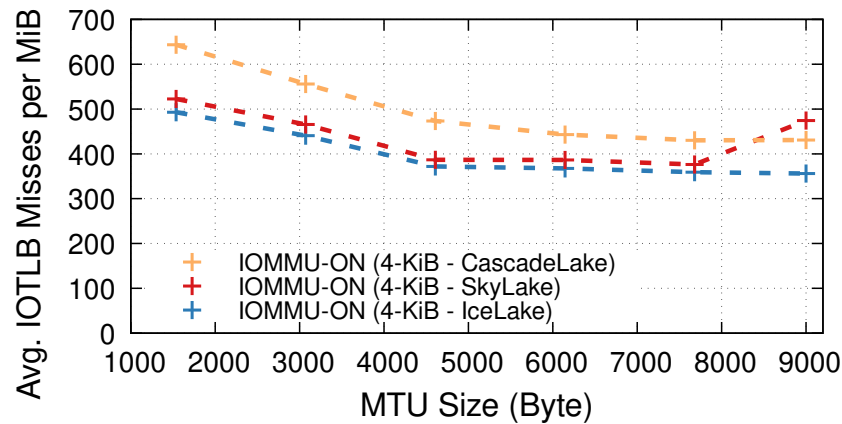
---

\*Some tuning might help, but that is beyond the purpose of this study.





(a) Throughput with and without IOMMU (zoomed Y axis).



(b) Average IOTLB misses per MiB on the receiver.

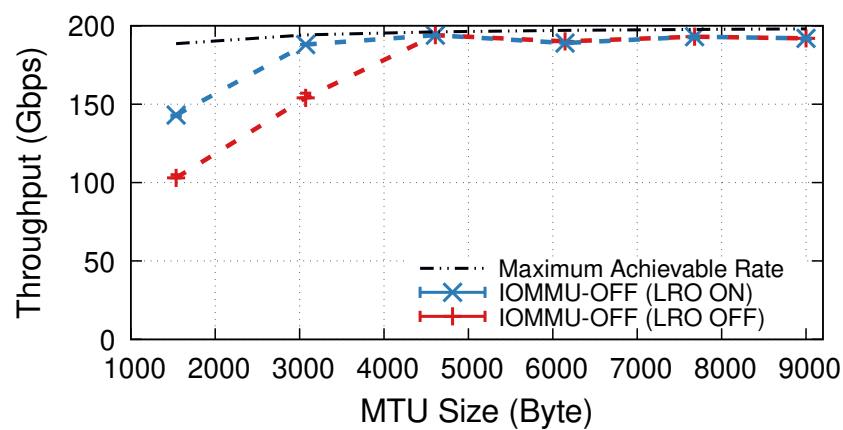
Figure 4.8: Throughput and IOTLB misses on several Xeon processors, operating at 100 Gbps [53].

**200-Gbps link speed.** Two of our systems, IceLake and AMD, support PCIe-4.0 and 200-Gbps NICs, so we compare the throughput drop introduced by IOMMU for them\*. As our AMD EPYC processor achieves a higher throughput *with* LRO even without IOMMU (see Figure 4.9a), we compare two processors with this feature enabled.

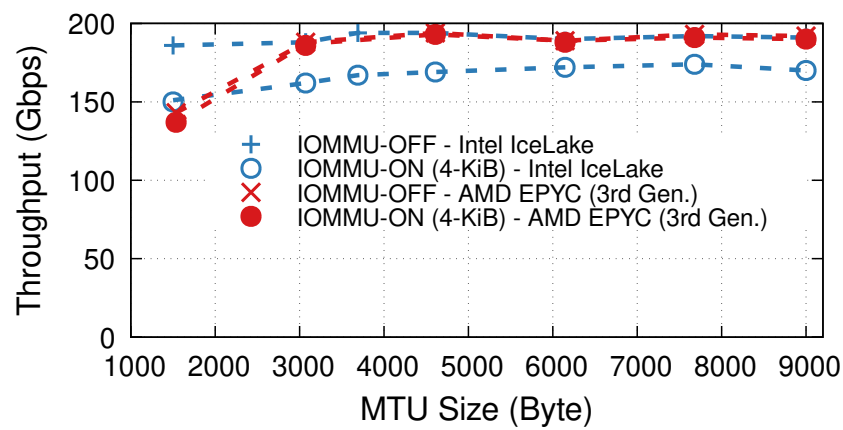
Figure 4.9b plots the maximum throughput of Intel IceLake and AMD EPYC processors with and without IOMMU for different MTUs. These results show that AMD EPYC’s IOMMU induces a very low overhead on throughput at almost all MTUs, but it is only capable of achieving line rate for MTUs larger than 3000 bytes.

---

\*We do not compare IOTLB misses per MiB since Intel and AMD have different IOMMU architectures.



(a) Throughput with and without LRO, AMD at 200 Gbps.



(b) Throughput with and without IOMMU, IceLake and AMD at 200 Gbps.

Figure 4.9: Throughput of IceLake and AMD workstations with and without IOMMU at 200 Gbps [53].

### 4.2.6 Different NICs

NIC device drivers may use different implementations for RX buffer pools to (de)allocate receive buffers, see Section 4.1.2.1. This section evaluates the impact of the driver implementation on IOTLB misses. More specifically, we compare the `ice` driver used by a 100-Gbps Intel E810 NIC and the `mlx5` driver used by a 100-Gbps NVIDIA/Mellanox ConnectX-6 NIC. The `mlx5` driver uses the Page Pool API [203] for receive buffer management, whereas the `ice` driver has its own custom implementation. Figure 4.10 shows that `mlx5` achieves slightly fewer IOTLB misses per MiB than `ice` driver; both NICs reach the maximum achievable rate for all MTUs. Furthermore, we observed that both drivers often perform continual page allocations at high rates.

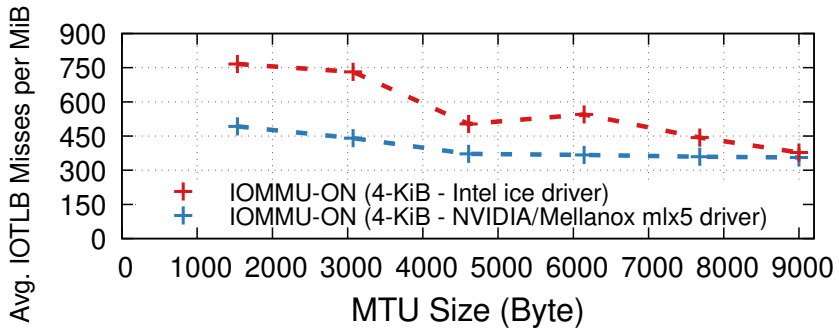


Figure 4.10: IOTLB misses of Intel E810 and NVIDIA/Mellanox ConnectX-6 NICs at 100 Gbps [53].

### 4.2.7 Other Workloads

Our study has mainly focused on iPerf microbenchmarks that represent applications capable of running at line rate with data-center-like RTTs and limited packet drops. This section concludes our evaluation by measuring IOTLB misses in two other scenarios considering (i) workloads with higher rates of packet drops and (ii) a less I/O intensive application (*i.e.*, Memcached).

**Packet drops.** Packet drops could cause receive buffers to be held longer in the Linux kernel, negatively affecting the buffer recycling and shuffling (see Section 4.1.2.1). To measure the impact of longer receive buffer lifetimes (caused explicitly by packet drops), we developed a kernel patch that artificially drops packets in the kernel network stack to induce TCP re-transmissions. Figure 4.11 shows that the higher the drop percentage, the higher the IOTLB misses per MiB.

In other words, longer buffer lifetimes put greater pressure on the IOTLB. For instance, dropping  $\sim 3\%$  of the packets for every TCP flow could increase IOTLB misses per MiB by up to  $\sim 2.5\times$ .

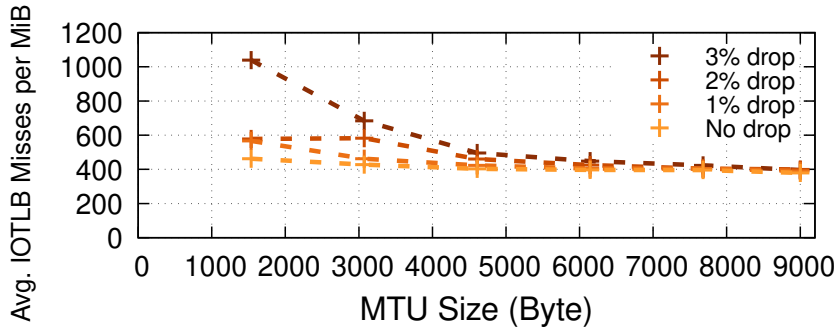


Figure 4.11: Packet drops put more pressure on IOTLB due to buffer shuffling and poor recycling [53].

**Memcached.** To show the applicability of our evaluations to less I/O intensive applications, we measure the number of IOTLB misses for a Memcached workload. We use a 32-core Memcached-SR [212] & mutilate [213] and 64-B keys & 128-KiB values with 50/50 SET/GET workloads [202] to achieve a comparable rate (90 Gbps when TPS= $\sim 170k$ ) to our iPerf measurements. Figure 4.12 shows that Memcached causes 450-500 IOTLB misses per MiB (same range as iPerf), despite being less I/O intensive and having other throughput bottlenecks. We interpret the existence of these iPerf-comparable results as a validation of our choice to use iPerf as the main driving application (in addition to its inherently deterministic behavior).

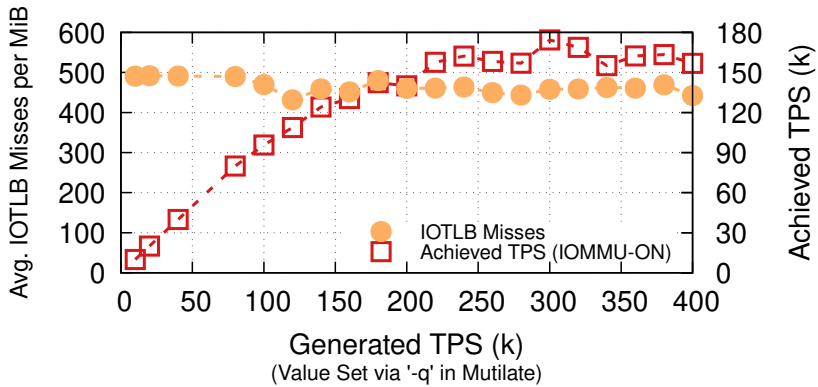


Figure 4.12: Memcached (a less I/O insensitive application) has a similar IOTLB pressure to iPerf at 90 Gbps (TPS= $\sim 170$ k) [53].

### 4.2.8 Summary of Characterization

This section showed that IOMMU is a major performance bottleneck when the offered network rate approaches 200 Gbps. Using LRO & TSO can reduce the number of IOTLB translations and partly mitigate the IOTLB overheads, especially for small MTU sizes, but still cannot eliminate the overheads at full line rate. Therefore, it is essential to come up with more effective solutions.

## 4.3 A Solution to IOTLB Wall

Current processors support IOMMU mappings with 2-MiB and 1-GiB entries. Having established that reducing IOTLB misses, even by a modest amount, has great benefits on throughput (Figure 4.3), the use of larger mappings is an obvious way to increase the chance that multiple buffers share the same IOMMU entry.

To prove the effectiveness of this approach, we ran a crude experiment backing buffers with 2-MiB memory “hugepages” and mapping them with 2-MiB IOMMU entries, without much attention to CPU costs and memory usage. Figure 4.13 proves that with such large mappings, we can achieve almost the same throughput as without IOMMU.

Our proposal thus focuses on how to build a *hugepage-aware buffer allocator* to be used for NIC buffers. In the next section, we discuss challenges that arise in designing this subsystem and explore available options to address them.

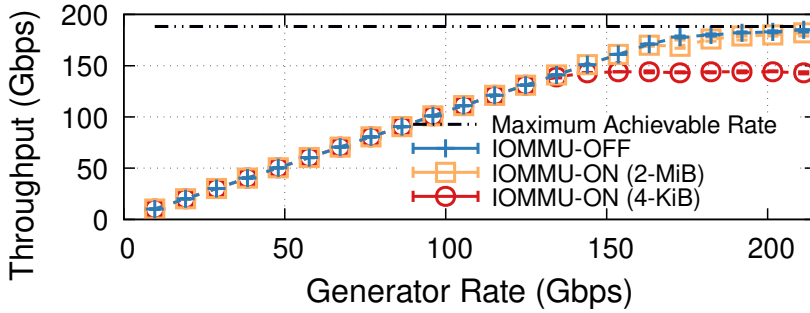


Figure 4.13: Throughput comparison with IOMMU-off, IOMMU-4-KiB and IOMMU-2-MiB for MTU=1500. The dashed line shows the maximum achievable rate for MTU=1500. The use of 2-MiB mappings recovers the throughput drop introduced by the IOMMU [53].

### 4.3.1 Challenges and Design Options

**Allocation availability and CPU cost.** The kernel allocates memory in 4-KiB units and uses virtual memory to map non-physically-contiguous pages into larger, virtually contiguous segments to satisfy any request from clients.

The kernel runs background compaction tasks to merge free pages into larger, physically contiguous blocks, to satisfy *e.g.*, requests from peripherals that do not support scatter/gather I/O, or for performance-sensitive systems that need to reduce MMU and IOMMU translation costs. Part of this effort includes remapping pages in use to some other locations, to ease compaction. A 2-MiB page, however, requires a significant amount of coalescing (*i.e.*, 512 contiguous 4-KiB-sub-pages), and in a long-running system with significant memory fragmentation, it may be difficult to find a spare one without asking for explicit compaction.

**Memory stranding.** One common option to obtain hugepages is to reserve them at boot time for a specific use, but this makes the reserved memory unavailable even when the intended client has no need for it. Furthermore, a boot-time reservation requires knowledge of the worst-case memory requirements.

For a network device, that is complicated: on the RX side, there is a minimum amount of buffers needed to populate receive queues, but these buffers are passed to applications' socket buffer and become unavailable for the NIC until the application has consumed the data. Similarly, on the TX side, buffers (allocated after the `sendmsg()` call and used for TCP) cannot generally be released until congestion control allows transmission *and* an acknowledgment has been received from the remote endpoint.

As a minimum, a sender should allow for an amount of buffering equal to the bandwidth-delay product. Frontend servers for residential customers are likely to deal with  $O(10\text{--}100\text{ ms})$  RTT, which at 200 Gbps is 2–3 GiB of memory. A similar minimum amount may be needed on the receive side, to buffer out-of-order data. Occasional CPU overloads resulting in delayed reads, or systems with a very large number of outstanding TCP connections, may cause the total socket buffer requirements to exceed our bandwidth-delay product size.

Depending on the use case, reserving several GiB of memory for network buffers may not be completely out of the question. However, an interesting feature of the problem is that we do not need to completely eliminate IOTLB misses, but simply *reduce their impact just enough* to prevent them from becoming the primary throughput bottleneck.

**Memory fragmentation.** Once we solve the problem of obtaining hugepages from the kernel, our memory allocator should be able to handle memory fragmentation by itself: transmit and receive buffers only need to handle MTU-sized blocks of data, each with a possibly different lifetime. The buffer managers, mentioned in Section 4.1.2.1, typically receive 4-KiB pages as input, and split them internally into a small number (*e.g.*, 2) of smaller chunks to be used as packet buffers. In our case, the input is 2-MiB pages, split in 512..1024 chunks. Over time some of these pages may become unavailable (*e.g.*, because they are stuck in socket buffers that are never read from). We call this phenomenon “buffer leakage”, which is something that requires the buffer manager to slowly allocate new buffers.

**Locality.** Using 2-MiB IOMMU mappings does not guarantee locality if we randomly pick buffers from different hugepages. Our allocator should thus track and in a way support returning contiguous buffers to the NIC when it replenishes the receive queues.

Both fragmentation and locality can be addressed with solutions that allow fast access to the state of contiguous buffers (such as, a bitmap indicating which chunks from a 2-MiB page are available, combined with existing kernel reference counts on the underlying 4-KiB pages).

### 4.3.2 Proposed Solution

Given the above constraints, we designed and implemented our initial prototype by interposing a hugepage allocator, HPA, between the NIC memory allocator (currently PagePool API\*, which for us is an opaque block) and the kernel memory allocator. Future versions will completely replace the Page Pool API and integrate functionality into the HPA for better locality and performance. The system works as follows:

---

\*Appendix B.4 elaborates on our current implementation.



- When the HPA needs memory, it calls the kernel memory allocator asking for one 2-MiB page, maps it into the IOMMU, and splits it into 4-KiB fragments kept in its internal cache. It also keeps track of the mapping to undo the mapping when the NIC releases its memory. In future versions, to avoid blocking in the hot path, the HPA will try to issue kernel requests in the background. If a 2-MiB page is not immediately available, it will fall back to regular 4-KiB pages.
- When the NIC device driver needs new buffers, it passes the request to our HPA, which replenishes the cache if needed, and responds with 4-KiB pages from its cache. The NIC then uses the IOVA for the page that has been supplied by the HPA, instead of creating (and removing when done) a new IOVA.
- In steady state, the NIC should be able to recycle pages through its own memory allocator, *e.g.*, Page Pool API, and not issue new requests to the HPA.
- Occasionally pages may not be recycled and are instead returned back to the kernel (“leaked”, from the point of view of the NIC; but fully recovered, in terms of memory usage) triggering more requests to the HPA.

The HPA does not waste memory but does keep pages mapped in the IOVA until its pages have been fully released.

#### 4.3.2.1 Discussion and Future Works

Our current solution is constrained by the desire to avoid massive changes to the existing device drivers, and is amenable to several optimizations, especially regarding the handling of corner cases. Nevertheless, it allows us to study in more detail the behavior of the system when using large IOTLB mappings. Open issues and future works include the followings (some have already been mentioned above):

**Background page allocation, fallback to 4-KiB pages.** We have not yet implemented this feature, which would be useful in systems with low memory availability.

**Locality improvements.** By completely removing the existing page buffer and absorbing its functionality in our allocator, we can do a better job in feeding the NIC with buffers that are closer in the IOMMU address space, following the indications given previously.

**Transmit side support.** Our allocator currently handles only the receive side. Transmit buffers are allocated during a `sendmsg()`, agnostic to the destination of

the message. It should be possible to infer, from the file descriptor, whether such data are destined to a NIC using IOMMU, and allocate the TX buffer from a pool of memory backed by and mapped as a hugepage. Our current implementation is deployable as-is and it is not dependent on the transmit-side solution.

**Recovery from thrashing.** Despite our best efforts, it may happen that the system enters a thrashed state where available buffers cause excessive IOTLB misses. The system should be able to detect this regime (*e.g.*, by looking at NIC packet drop counters on the receive side) and replenish the HPA with a fresh set of hugepages, so that the current, poorly placed pages can be quickly phased out in favor of a new set.

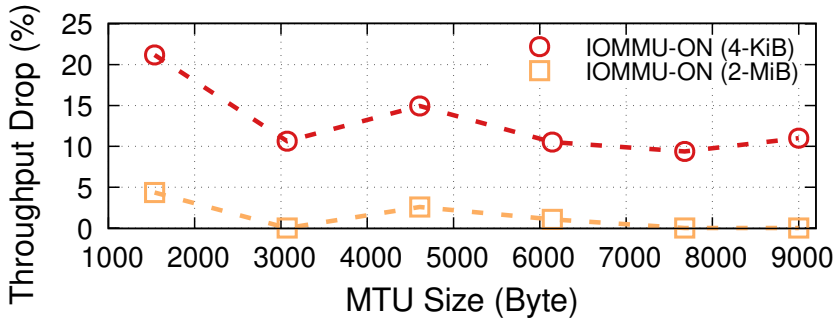
## 4.4 Evaluation

This section evaluates the effectiveness of using large IOTLB entries for 200-Gbps networking using our proposed memory allocator. Section 4.5 answers some additional questions about our approach.

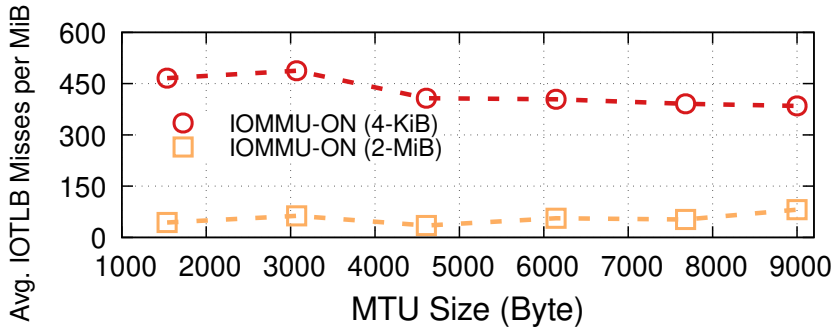
### 4.4.1 IOTLB Misses and Throughput Drop

We already showed in Figure 4.13 that the 2-MiB mappings almost completely recover the throughput drop introduced by IOMMU. This is shown in more detail in Figure 4.14a, where we see that 2-MiB mappings only causes a modest 5% drop for MTU=1500, and enables the system to get very close to the no-IOMMU case for larger MTUs.

Figure 4.14b compares the average number of IOTLB misses per MiB for the original IOMMU mappings (with 4-KiB IOTLB entries) and our implementation (with 2-MiB IOTLB entries). These results show that employing hugepages helps overcoming the IOTLB wall at 200 Gbps, as the number of IOTLB per MiB misses is reduced (by up to  $\sim 10\times$ , giving us a large safety margin for operation). The gap (350-400 misses per MiB) is in good accordance with our estimates in Section 4.2.1.



(a) Throughput drop.



(b) Average IOTLB misses per MiB.

Figure 4.14: Larger IOMMU mappings on the receiver side significantly reduce throughput drop and IOTLB misses [53].

#### 4.4.2 Run-Time Allocation Overhead

To quantify the overhead of run-time allocation, we benchmarked the execution time of the page-allocation function in the Page Pool API (*i.e.*, `page_pool_alloc_pages_slow`) via `kstats` [214] over a period of 2 hours with an offered load of 200 Gbps. The default buffer allocator requests blocks of  $64 \times 4$ -KiB pages from the kernel, whereas our mechanism has a granularity of 512 (contiguous) 4-KiB pages. Individual allocations are thus very likely to take much longer, but their cost is amortized over  $8 \times$  more pages. The correct model of operation is to request buffers in the background, so they do not block critical sections, and then look at the amortized, per-page cost, which is what we plot in Figure 4.15. The cost distribution for the two scenarios is similar, which is encouraging. We surmise that this may depend on the relatively low memory pressure in our testbed, making it possible to retrieve 2-MiB pages with relatively

little extra CPU overhead.

We should keep in mind that in the presence of high memory pressure, we can always fall back to 4-KiB pages, on the grounds that such a scenario will likely introduce other bottlenecks (CPU or memory contention) that prevent reaching line rate anyways.

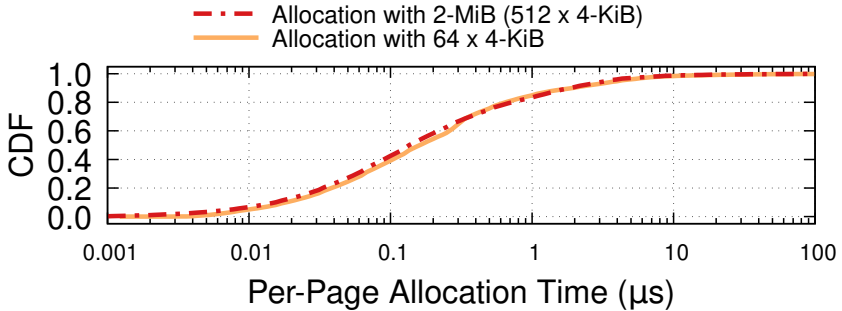


Figure 4.15: Per 4-KiB page allocation costs are similar with 4-KiB and 2-MiB pages in our experiments [53].

#### 4.4.3 Buffer Shuffling

To conclude, we want to revisit the dynamics of buffer shuffling with hugepage mappings. Our previous experiments with 4-KiB mappings were extremely prone to this problem, which showed up quickly at the first signs of congestion (see Section 4.2.1). Due to the nature of this problem, we did not expect it to disappear completely even with hugepage mappings. To understand the potential overheads associated with buffer shuffling, where the long-term change in the order of buffers causes consecutive pages to have *different* IOTLB mappings, we ran two types of experiments: ① (“No Pool”) where the buffer allocator does not recycle buffers internally, and instead always gets fresh buffers, and ② where the existing allocator recycles buffers very aggressively rather than making new allocations, reusing the same block of 256 or  $512 \times 2$ -MiB pages per queue.

Figure 4.16 shows the IOTLB misses per MiB over time. All experiments have a modest amount of packet drops. As expected from our previous experiments all three cases start with a low number of misses. As expected, the “No Pool” case has a low miss rate for the duration of the experiment. This occurs since there is no source of buffer shuffling, despite the packet drops. The experiment with 512 pages starts to show an increasing number of misses after about 2 minutes, and the curve grows over time as more and more packet drops cause more severe buffer

shuffling (the growth is bounded because eventually there is a hard limit given by the maximum link data rate). In the experiment with 256 pages, the shuffling appears later but grows more rapidly because the smaller pool gives more opportunities for reshuffling.

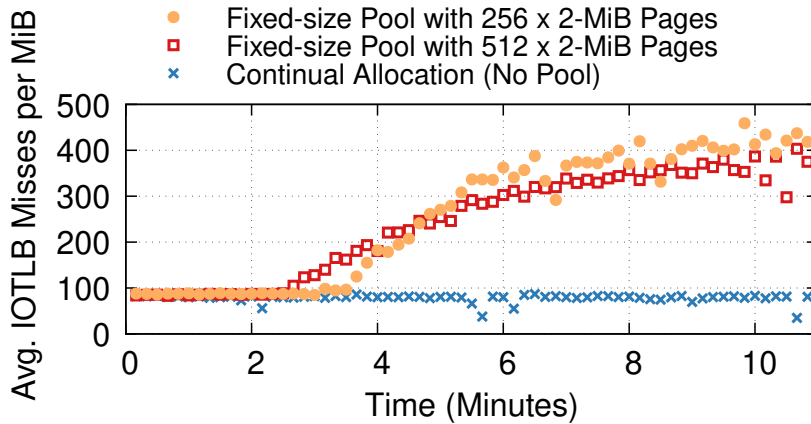


Figure 4.16: Increased IOTLB misses due to the buffer shuffling could cause IOTLB to again become a bottleneck for an iPerf server receiving 1500-B frames with different pool sizes per queue. Data points are shown every 10 seconds [53].

## 4.5 Frequently Asked Questions

**What makes your approach more adoptable?** Previous attempts [215, 209, 208, 210, 202] focused on improving IOMMU performance, but most of them have not been adopted, *speculatively* due to introducing hardware modifications [215, 209], imposing other overheads [210], and requiring massive kernel changes [210] (including a larger “struct page” that is a central and fragile kernel data structure). We believe our work is much less intrusive, as it is confined to a buffer allocator without introducing dependencies. Our Page Pool extension can be deployed on a per-driver or per-module basis as needed *without* global kernel changes, making experimentation and gradual adoption a lot simpler and less risky.

**Are hugepages the panacea?** We showed that using *statically* mapped hugepages can mitigate the performance overheads of IOMMU for high-speed networking. However, hugepages are not a panacea, and faster link speeds would require larger & larger hugepages to achieve optimal performance with the same number of IOTLB entries. Therefore, future-generation hardware is expected to introduce

changes to the IOMMU architecture, *e.g.*, adapting the IOMMU page table as proposed by [209] and/or modifying the IOTLB to support more concurrent translations per second.

### **Does using hugepages relax security guarantees provided by IOMMU?**

Switching to *statically* mapped hugepages may cause security concerns if we do not use a specific memory pool, as the deallocated pages might be used by the kernel or an application while still being exposed to an I/O device [216]. However, (de)allocating hugepage-backed buffers to/from a specific (I/O) memory pool addresses these concerns. Note that each I/O device is limited to its own domain mappings; therefore, using larger IOTLB entries provides the *same* I/O device with a larger access domain *and* a longer access time, neither of which should raise any critical concerns as long as the huge IOTLB mappings are only used for I/O.

**How about zero-copy mechanisms?** There are a few old (*e.g.*, `sendfile` and `splice`) and recent (*i.e.*, `MSG_ZEROCOPY` [217, 218] and `MAIO` [219]) techniques to pass userspace buffers directly to the kernel, avoiding the data copying overheads; however, most of these techniques are generally only effective for large messages [220] (*e.g.*, 10 KiB [221]). Our approach is not designed with zero-copy mechanisms in mind. As for the RX path, our approach may work for the recent zero-copy receive mechanism introduced by Eric Dumazet [222, 223] & VMware's I/O memory allocator, called `MAIO` [219, 224]. However, it could raise additional security concerns since DMA attacks may affect the user/application workflow. As for the TX path, we rely on the current Linux kernel implementation to map buffers to IOMMU. Drivers usually assume that the allocated buffers (*i.e.*, in the RX path) and/or received skbuffs from the kernel (*i.e.*, in the TX path) are backed by `PAGE_SIZE` (*i.e.*, 4-KiB) pages. Therefore, they create a 4-KiB IOMMU entry even if a buffer is allocated from a hugepage (*e.g.*, a 2-MiB/1-GiB hugepage) in userspace and passed to the driver via zero-copying. It is possible to extend the available machinery with additional options/flags to notify the driver about the granularity of IOTLB mappings, which remains as our future work.

**Why should I care about IOMMU overheads in the Linux kernel instead of using hugepages in kernel-bypass frameworks?** While bypassing the kernel provides better performance by avoiding some inherent kernel overheads (*e.g.*, data copying), it excludes applications from benefiting from the existing infrastructure in the Linux kernel, thus requiring programmers to re-develop & maintain the needed infrastructure in userspace. In particular, many networking applications require different network stacks [225], which has resulted in some userspace network stacks such as `mTCP` [226], `f-stack` [227], and `TLDK` [228]. However, these userspace stacks are rarely maintained for a long period and they may (&

do) contain bugs. Consequently, many applications are still being deployed that use the Linux kernel\* (despite its known overheads); therefore, it is important to be able to mitigate IOTLB overheads in the Linux kernel.

## 4.6 Related Works

In addition to the works discussed throughout this chapter, other IOMMU-related works can be grouped into three categories: ① investigating the IOMMU security vulnerabilities & the possibility of performing different DMA attacks [216, 229]; ② optimizing the performance of IOMMU for hypervisors and virtualized setups [230, 231, 232, 233, 234]; and ③ presenting hardware proposals for future-generation IOMMU technologies [209, 230, 235]. Additionally, Lesokhin *et al.*, [236] enable Infiniband and Ethernet NICs to support I/O page faults. Moreover, Agarwal *et al.*, [237] measure the impact of IOMMU on host interconnect congestion. Our contributions are complementary to these works.

## 4.7 Summary

IOMMU is one of the available technologies that was introduced to ease I/O virtualization and to provide security & isolation. However, high-performance systems tend to disable IOMMU to mitigate its performance overheads, thus voiding the security & isolation guarantees. This chapter empirically studied IOMMU in recent hardware and demonstrates that IOTLB overheads become even more costly when moving toward 200-Gbps networking. However, we showed that it is possible to alleviate these overheads in software by employing hugepage IOTLB mappings, but there are many challenges in permanently eliminating these overheads. The main takeaway of this chapter is that supporting 200/400-Gbps networking demands fundamental changes in the Linux kernel's I/O management.

---

\*It is worth mentioning that future Internet services may rely on QUIC and UDP (*e.g.*, HTTP/3), which makes it easier to rely on userspace implementations, see Section 6.1.

# Chapter 5

## Related Works

**T**HROUGHOUT this dissertation, we highlighted the most relevant works to each contribution. This chapter briefly summarizes the related work to this dissertation in a more consolidated way in order to have a better understanding of the ongoing research targeting similar problems. More specifically, we discuss the recent efforts geared toward improving packet processing and facilitating the deployment of multi-hundred-gigabit-per-second network services. From a high-level perspective, most of the relevant works (i) propose/employ software optimizations to mitigate inefficiencies and/or (ii) utilize/propose programmable hardware to benefit from (introduce) more customized packet processing capabilities at high rates. Figure 5.1 illustrates a more detailed overview of the relevant works, where software & hardware optimizations are divided into subcategories. Sections 5.1 (software) and 5.2 (hardware) go through each category and list the most relevant & influential related works.

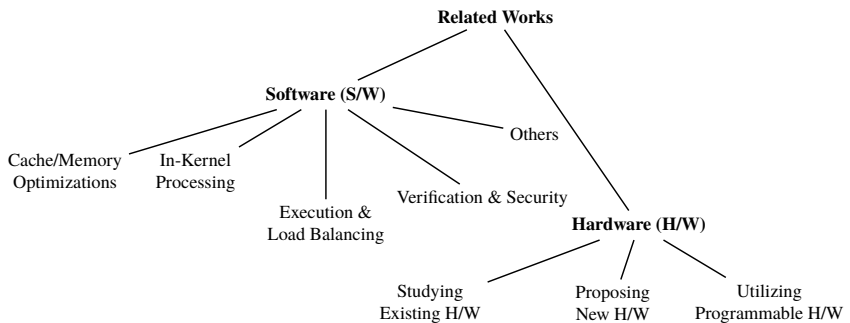


Figure 5.1: An overview of recent efforts geared toward high-speed networking.



## 5.1 Software-Related Works

Section 1.2 emphasized the importance of utilizing the underlying resources efficiently due to the saturation of per-core performance. To do so, it is important to mitigate software inefficiencies and make the most out of the available cache memories to be able to process packets at multi-hundred-gigabit-per-second rates [238, 26]. This section enumerates some of the key recent trends in optimizing software for high-speed networking applications.

**Cache/memory optimizations.** While many have previously shown the benefits of cache partitioning & cache-aware memory management for different non-networking applications [239, 240, 241], a recent series of publications have focused on I/O-aware cache partitioning for networking applications. In particular, SLOMO [242], ResQ [97], and IAT/IOCA [243, 244] have used CAT to perform I/O-aware cache partitioning to improve performance predictions *and* minimize I/O contentions caused by DDIO. Moreover, Cavast [245] has employed page coloring to optimize the cache utilization of large key-value stores.

**In-kernel processing.** Section 4.5 already highlighted the importance of having high-performance in-kernel solutions despite the existence of kernel-bypass frameworks for packet processing. In this regard, Cai *et al.*, [48] studied the Linux kernel and presented measurements & insights to improve its performance at multi-hundred-gigabit-per-second rates. Additionally, Cai *et al.*, [246] proposed a disaggregated network stack architecture to enable efficient & independent resource allocation & scheduling for terabit-per-second packet processing pipelines. Moreover, some works have relied on Extended Berkeley Packet Filter (eBPF) to (i) offload the application logic (*e.g.*, keeping a key-value store cache in the kernel [212]), (ii) process packets [247, 248], or (iii) balancing application requests [249] in the kernel. Section 6.1 discusses this trend in more detail.

**Execution & load balancing.** To find the appropriate trade-off between the memory footprint & the inter-core communication overheads, some works have thoroughly studied different packet processing (*aka* NFV) execution models [16, 250] (*i.e.*, RTC vs. pipeline models). Achieving high performance via a RTC model that minimizes the inter-core communication requires careful balancing of the load among different cores. To do so and to improve load balancing, RSS++ [186] and Metron [16] introduced a dynamic RSS table and employed flow steering techniques, respectively. Moreover, some works (*e.g.*, SNF [17] and OpenBox [92]) have intensively optimized the NF processing graph to improve the performance of packet processing frameworks.

**Verification & security.** The transition toward NFV has improved time-to-market & flexibility, but the frequent software changes/updates make the NFs more

error-prone, increasing the chances of shipping software with uncaught bugs. Some works (*e.g.*, KLINT [251], Vigor [146], and VigNAT [145]) have focused on formal verification of NFs to address this issue. In addition, some works (*e.g.*, PIX [252] and BOLT [253]) have proposed performance contracts/interfaces to predict *and* improve the performance of NFs. Furthermore, CASTAN [147] has employed symbolic execution techniques to synthesize adversarial workloads and reason about NF performance. In addition to verification, a few works have focused on network security at high rates. For instance, Retina [254] proposes a framework to analyze 100-Gbps traffic on commodity hardware.

**Others.** Moving toward multi-hundred-gigabit-per-second rates requires optimizing different aspects of software such as locking/synchronization *and* data structures. In particular, many stateful NFs heavily rely on different types of hash tables to keep track of connection states, which requires a good understanding of data structures to implement an efficient solution. For instance, Girondi *et al.*, [255] thoroughly studied different implementations of connection tracking in modern high-speed NFs. Moreover, some works (*e.g.*, MiddleClick [256] and ClickNF [257]) have tried to provide an efficient userspace infrastructure to process stateful & TCP-based NFs and modify the content of flows.

## 5.2 Hardware-Related Works

While performing software optimization is necessary for achieving high performance, there is a fundamental limit to the maximum achievable performance of hardware. Therefore, an active area of research is constantly trying to study, propose, and utilize new hardware to facilitate the processing of multi-hundred-gigabit-per-second network traffic. This section highlights some of the key trends in this context.

**Studying existing hardware.** A series of publications have focused on understanding the performance of the existing technologies in accordance with higher network traffic in order to improve future generations of technologies *and* utilize them more efficiently. First, Neugebauer *et al.*, [49] studied the *de facto* I/O interconnect\* of commodity hardware, *i.e.*, PCIe, presents a theoretical model *and* benchmark to understand its performance implications. Second, Katsikas *et al.*, [51] studied the benefits & limitations of offloading capabilities offered by smart (commodity) NICs. Last but not least, Wang *et al.*, [259] developed an analytical

---

\*It is worth mentioning that NVIDIA uses NVLink to accelerate all-to-all Graphics Processing Unit (GPU) communication and to perform AllReduce operation at NVSwitches [258]. Additionally, Compute Express Link (CXL) is a recently introduced open standard for high-speed CPU-to-device and CPU-to-memory connections, which may replace PCIe.

framework to predict the effectiveness of DCA\*.

**Proposing new hardware.** A large series of work (mainly from the computer architecture community) focus on proposing architectures for future-generation hardware to process packets more efficiently. Enumerating all publications is out of the scope of this dissertation, but we briefly highlight some of the interesting & more relevant proposals for multi-hundred-gigabit-per-second packet processing. HALO [260] proposes changes to (i) the NUCA architecture of Intel CPUs and (ii) x86-64 instruction set to perform near-cache flow classification. IDIO [261] extends DDIO to L2 cache (*aka* Mid-Level Cache (MLC)) and proposes three mechanisms to improve tail latency of networking applications. NanoPU [262] proposes a new NIC-CPU co-design to accelerate the *fast path* between the applications & the network by directly placing the incoming packets into CPU registers (as opposed to DCA and DDIO that inject the incoming packets in the CPU cache). ORCA [263] presents a network & architecture co-design solution to offload data center applications using RDMA and cache-coherence accelerators. HTA [264] proposes a few instructions to accelerate hash tables lookups and updates. RPCValet [265] introduces a software-hardware co-design to perform dynamic load balancing in multicore CPUs with an integrated NIC.

**Utilizing programmable hardware.** Multi-pipeline programmable switches and smart/FPGA NICs have recently gained a lot of traction. Many researchers focus on exploiting the processing capabilities offered by this hardware to perform costly operations in the network. However, this hardware comes with limitations, *e.g.*, it supports limited arithmetic operations and/or has limited memory, which typically requires meticulous scheduling to run applications on variant devices in heterogeneous setups with CPU-based servers, multi-pipeline programmable switches, and smart NICs. For instance, Galium [59] designs & implements a compiler to split the input NF code into two parts running on x86 servers and programmable switches. Additionally, Flightplan proposes a toolchain to disaggregate a single P4 program into subprograms and execute them across different dataplanes (*e.g.*, programmable switches, FPGAs, and x86 servers). Many works have used this hardware to (i) load balance requests (*e.g.*, Cheetah [188]), (ii) accelerate key-value stores (*e.g.*, NetCache [266] and KVSwitch [267, 268]), (iii) utilize the available bandwidth more efficiently by storing the unnecessary parts of packets in the network (*e.g.*, PayloadPark [269], Ribosome [270], NFSlicer [271], and nicmem [272]), and (iv) design FPGA accelerators (*e.g.*, FlowBlaze [273], KV-Direct [274], and Pegasus [179]). Moreover, some works have used smart NICs to accelerate the protocol stack [275, 276] and perform non-uniform DMA on multi-socket CPUs (*e.g.*, IOctopus [277]).

---

\*We have also done a similar study in this doctoral research project; see C2 in Section 1.5.2

## Chapter 6

# Conclusion and Future Works

**T**HIS dissertation focused on optimizing the performance of packet processing by identifying opportunities in *both* software and hardware to realize low-latency Internet services.

Chapter 2 focused on improving the cache utilization for virtual NFs. To do so, we scrutinized the cache architecture in modern Intel processors. We identified the non-uniform architecture of LLC and showed that it could be exploited to reduce average access latency to LLC by 20%. Finally, we used this knowledge and introduced CacheDirector, a network I/O solution, which can reduce the tail latency of time-critical NFV service chains.

Chapter 3 demonstrated the impact of performing whole-stack optimization on the performance of high-speed software packet processing frameworks. We introduced PacketMill that significantly increased performance when processing packets at  $\approx 100$  Gbps.

Chapter 4 characterized the performance of IOMMU & IOTLB at 200 Gbps. We showed the necessity of employing hugepages to provide performant I/O security solutions in the Linux kernel; and discussed the challenges of implementing a hugepage-aware memory manager for the device drivers.

The works done during this dissertation successfully satisfied our pre-defined objectives (see Section 1.3 on page 5). However, there still remains work, which could be pursued to further facilitate realizing low-latency Internet services on multi-hundred-gigabit-per-second commodity hardware by providing more optimized packet processing solutions. The next section proposes some possible directions to continue this project.

## 6.1 Future Works

In addition to the future works already described throughout this dissertation, a continuation of this project should be able to answer the following questions in order to realize low-latency Internet services on multi-hundred-gigabit-per-second commodity hardware.

- Q.1** *What are the performance implications of existing features in current hardware & software (e.g., mitigations against microarchitectural attacks and power & frequency management) when shifting toward multi-hundred-gigabit-per-second rates? How should we use/reform them to fit the needs of future networking trends?*
- Q.2** *What are the shortcomings of modern networking equipment, such as network interfaces and switches? How should we design the next-generation networking equipment? How should we overcome the challenges of employing current offloading capabilities when considering the trade-offs between cost, flexibility, and performance?*
- Q.3** *How does the current Linux-based networking infrastructure scale with increasing link speeds? What are the required changes to adapt/reform the Linux network stack to enable high-performance, low-latency, and low-overhead processing?*
- Q.4** *Can eBPF act as a performant packet-processing alternative solution compared to its kernel-bypass counterparts? What are eBPF's current challenges and possible opportunities to address them?*
- Q.5** *What are the potential replacements for the TCP protocol in the future? How can they perform at multi-hundred-gigabit-per-second rates? How should we adapt the existing packet processing pipelines to provide an efficient and performant platform for all Internet services relying on different protocols?*
- Q.6** *What are the possible ways to adapt general-purpose hardware to make them more compatible with the current packet processing requirements?*
- Q.7** *What are the anticipated trends for future-generation data center architecture? How do they affect existing networked systems?*

- Q.8** *Can we improve the training of large machine-learning models with the recent advances in networking equipment? What are the potential solutions to perform machine learning and inferencing at multi-hundred-gigabit-per-second rates? Additionally, can we improve packet processing by employing machine-learning techniques? Is it possible to accurately predict network traffic patterns?*
- Q.9** *Is P4 the suitable domain-specific language for programming any kind of data plane? What are its shortcomings? How can we improve it for future-generation packet processing?*

The following list summarizes some works that could potentially answer some of these questions.

**Building multi-hundred-gigabit-per-second Linux-based infrastructure.** As shown by a recent study [48], the current Linux networking infrastructure is not performant enough to run 100-Gbps networking applications. Consequently, many high-performance applications have to rely on kernel-bypass frameworks (e.g., DPDK) to be able to run at the speed of the underlying hardware. While bypassing the kernel provides better performance by avoiding some inherent kernel overheads (e.g., data copying), it excludes applications from benefiting from the Linux kernel's existing infrastructure in (see Section 4.5). Consequently, the preferred way to deploy applications is to use the Linux kernel to benefit from the continuous maintenance and long-term efforts put into the kernel by the community. Future work can take a step toward building a more multi-hundred-gigabit-friendly network stack in the Linux kernel. In particular, one could study the current I/O memory (buffer) management and propose techniques/modifications to facilitate integrating overhead-free zero-copying networking into the kernel *without* imposing new criteria for applications while still benefiting from the available network stack (as opposed to XDP or DPDK).

**Building performant packet processing solutions via eBPF.** This dissertation has primarily focused on the performance of packet processing done via a kernel-bypass framework (i.e., DPDK); however, relying on kernel-bypass solutions has its downsides (see Section 4.5). eBPF has recently become a trendy method to realize specific userspace-developed packet-processing solutions for large networking & data center companies, which enables developers to *safely* insert customized applications into the Linux kernel but with some limitations (e.g., limited program size and limited support for loops). eBPF provides variant hooking points to insert programs into the Linux kernel infrastructure. The two common networking hooking points are XDP and `tc` (traffic control). XDP enables programs to

access raw packets at the earliest stage of packet reception, whereas `tc` provides access to packets after the metadata extraction by the Linux network stack. Meta, Google, Isovalent, Microsoft, and Netflix have recently launched the eBPF Foundation as part of the Linux Foundation [278], which could promote eBPF-based solutions and facilitate its further development. Some works have tried to (i) improve the performance of eBPF-based packet processing solutions via H/W offloading [279, 280] and low-level optimizations [248] and (ii) facilitate the development of complex NFs despite eBPF's limited programmability [247]. However, eBPF development is still in progress and may require solving additional challenges in order to provide packet processing solutions capable of processing packets at multi-hundred-gigabit-per-second rates.

**Examining potential replacements for TCP.** TCP has been the most dominant Internet protocol in the last few decades, as most Internet services require a reliable connection. For instance, both HTTP/1.x & HTTP/2 *primarily* rely on TCP\* to exchange information over the web. However, TCP was not designed for modern data centers and imposes many challenges on multi-hundred-gigabit-per-second networking applications. For example, TCP is one of the major contributors to “data center tax”, as it requires a considerable amount of CPU resources to operate at multi-hundred-gigabit-per-second rates [281, 282]. Future-generation Internet services may rely on other transport protocols (*e.g.*, Homa [283, 284] and QUIC [285, 286]) to transfer data. For instance, HTTP/3 [287] has already started using QUIC over UDP (instead of TCP), potentially making TCP less dominant. Moreover, RDMA could become another alternative for transferring large files, where out-of-order delivery is unnecessary. In contrast to TCP, RDMA requires much less CPU resources since most of the work can be offloaded to storage devices and NICs. To facilitate the integration of these alternatives, it is crucial to (i) empirically examine the available solutions for these alternatives, (ii) measure their performance at multi-hundred-gigabit, and (iii) adapt and/or build more performant solutions for processing alternative protocols. For instance, Yang *et al.*, [288] analyzes the CPU usage of different QUIC implementations and proposes a hardware/software co-design of QUIC to offload the CPU-hungry parts of the protocol (*i.e.*, the crypto module running Authenticated Encryption with Associated Data (AEAD) algorithms *and* packet reordering) to smart NICs<sup>†</sup>. It is worth mentioning that QUIC & UDP traffic can potentially rely on high-performance kernel-bypass frameworks and use userspace protocol stacks; however, kernel-bypass frameworks (*e.g.*, DPDK) typically overutilize CPU resources due to their continuous polling mechanisms [290]. Therefore,

---

\*HTTP can also use SCTP as a transport layer protocol.

<sup>†</sup> According to Google [289], QUIC requires  $3.5\times$  more CPU cycles than TCP+TLS.

building a performant Linux-based infrastructure would still be of importance.

**Future-generation general-purpose hardware.** Deploying low-latency multi-hundred-gigabit-per-second applications on commodity hardware calls for new changes in hardware since the per-core performance is not increasing at its previous rate\*. Previous research has shown the benefits of introducing new features & accelerators within general-purpose processors. For instance, HALO [260] proposes a near-cache flow classification accelerator and extends the x86-64 instruction set to improve the performance of packet processing. Additionally, CacheDirector [26] (see Chapter 2) suggests making the slice selection hash function programmable to reduce access time to LLC, thereby improving tail latency of 100-Gbps applications. There are still many other changes that could be made to the available hardware. For instance, many networking applications perform boilerplate tasks that require many instructions and processing power. To improve instruction cache locality & applications' performance, we can introduce modifications to processors and modern NICs to accelerate the execution of these tasks. For instance, PacketMill [45] (see Chapter 3) showed the benefits of optimizing metadata management and making network drivers application-aware. One can push this idea further and make the NIC application-aware. By doing so, not only can we improve cache locality, but also save PCIe bandwidth and avoid transmitting unnecessary data between the NIC and the processor. For example, Pismenny *et al.*, [272] proposes to make the small on-NIC memory available to accelerate some networking applications (*e.g.*, NFs and key-value stores). One can (*i*) identify other acceleration possibilities, (*ii*) propose changes to the hardware, and (*iii*) evaluate their potential value in terms of cost & performance via programmable NICs and cycle-accurate simulators such as gem5 [291], which requires understanding the challenges of processing packets at high rates. Moreover, processor manufacturers constantly introduce new features in new generations of processors to improve the performance of applications. For instance, Intel has recently introduced the 4<sup>th</sup> generation of Xeon scalable family processors (*aka* Sapphire Rapids) that are shipped with various new features (*e.g.*, userspace interrupts), instructions (*e.g.*, CLDEMOT), and on-chip accelerators (*e.g.*, Advanced Matrix Extension (AMX), Dynamic Load Balancer (DLB), Data Streaming Accelerator (DSA), In-Memory Analytics Accelerators (IAA), and QuickAssist Technology (QAT)). It is important to evaluate the effectiveness of these newly introduced features and utilize them to improve the performance of different applications deployed on top of commodity hardware.

---

\*These changes are not limited to networking applications. Server-grade processors are now being shipped with new instructions to accelerate machine learning.



**Future-generation data centers.** Data center traffic has tremendously increased in the last few years, which has resulted in new proposals for future generations of data centers. First, a few works [292, 293, 294, 295] advocate for optical switching to address the low capacity *and* high power consumption of electrical switches. Second, FLEET [296] makes a case for offering shared computational & storage resources for high-end servers via customized optical NICs. FLEET suggests connecting resources/components via PCIe to mitigate the overheads of (de)composing Ethernet packets. Third, higher variance in resource requirements for different cloud-hosted services requires better solutions to maximize the data center resource utilization. Some works [39, 40] have proposed different models for resource disaggregation and building customized resource pools, which require meticulous scheduling for achieving a trade-off between cost, performance, and utilization [68, 297, 298, 299]. Fourth, the recent high demand for artificial intelligence applications has driven some data center companies to build customized hardware (*e.g.*, Google Tensor Processing Unit (TPU) pods [300, 301]) and interconnects (*e.g.*, HPE Slingshot interconnect, NVIDIA’s Ethernet Cloud Fabric, Ayar Labs’ optical I/O interconnect [302], and SiP-ML [303]) to accelerate High Performance Computing (HPC) workloads, such as training & inference of large deep learning models. Additionally, some works [262, 304, 305] have focused on NIC-CPU co-design to increase the performance of networking applications. Future works could (*i*) contribute to each of these categories, (*ii*) study their impact on the existing networked systems, and (*iii*) facilitate the potentially gradual integration of these proposals into future-generation data centers.

**Building high-performance ML-related networked systems.** Machine Learning (ML) has recently become very trendy given the advancements in computing and the availability of data, resulting in many emerging applications for ML. Therefore, it is crucial to (*i*) build high-performance networked systems for ML to perform network-accelerated distributed training [306, 307], (*ii*) utilize network accelerators to enable low-latency ML inferencing at multi-hundred-gigabit-per-second rates [308, 309, 310, 311, 312], and (*iii*) employ ML techniques to improve packet processing at high rates. Some works have already tried to use ML to accelerate flow/traffic classifications [313, 314, 315] and network intrusion detection [316, 317, 318]. However, many other unexplored opportunities exist to employ ML techniques for packet processing. For instance, previous works [61] have shown that *packet order* can significantly affect the performance of high-speed NFs. If we can build sufficiently accurate ML models to predict the upcoming network traffic and potentially its characteristics (*e.g.*, flow size and packet inter-arrival time), one can use this information to further optimize packet processing software. For example, sufficiently accurate models for network traffic could

enable stateful NFs to *(i)* use simplified data structures (*e.g.*, FIFO queues) rather than hash tables [255] and *(ii)* produce workload-optimized binaries (*e.g.*, with different basic block orderings [160]), and *(iii)* prefetch data in advance. Note that some of this work involves improving the network performance of specialized processors rather than simply being focused on CPU-based commodity hardware. For example, with network traffic going directly from a NIC to GPUs or other specialized processors, it is important to minimize the communication overhead between different components.

**Improving P4 language and its infrastructure.** P4 is a Domain-Specific Language (DSL) for programming and controlling packet processors, which was introduced in 2014 [319]. P4 is specifically designed for packet processing, which requires three main steps to deserialize (*aka* parse), transform/process, and serialize (*aka* deparse) the packets. Since its introduction, P4 has been primarily used for configuring multi-pipeline programmable switches. However, P4 has recently started to become popular for programming other data plane elements, such as smart NICs (*e.g.*, Netronome [320]), Data Processing Units (DPUs) (*e.g.*, NVIDIA Bluefield via DOCA SDK [321]), eBPF, and software switches (*e.g.*, DPDK [322]). Introducing new targets for P4 facilitates scheduling & offloading of NFs on setups with heterogeneous hardware, but requires developing an infrastructure to do so [323]. Future works should focus on addressing the challenges of producing efficient, performant, & verifiable code for various hardware *and* scheduling & offloading different parts of the code to the hardware [59, 324, 325] while guaranteeing the program's correctness and verifying program semantics [326, 327]. Moreover, having a DSL for packet processing could inspire the networking community to come up with other applications than just programming data plane devices. For instance, Google has recently used P4 to build a model for its fixed-function ASIC-based switches and formally verify it [328].



# References

- [1] A. Roozbeh, J. Soares, G. Q. Maguire Jr., F. Wuhib, C. Padala, M. Mahloo, D. Turull, V. Yadhav, and D. Kostić, “Software-Defined “Hardware” Infrastructures: A Survey on Enabling Technologies and Open Research Directions,” *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2454–2485, thirdquarter 2018. doi: 10.1109/COMST.2018.2834731
- [2] K. Bogdanov, M. Peón-Quirós, G. Q. Maguire Jr., and D. Kostić, “The Nearest Replica Can Be Farther Than You Think,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015. doi: 10.1145/2806777.2806939. ISBN 978-1-4503-3651-2 pp. 16–29. [Online]. Available: <http://doi.acm.org/10.1145/2806777.2806939>
- [3] A. Patrizio, “Data centers are set to grow and become more complex, survey finds,” *Network World*, 2018, Accessed 2019-02-21. [Online]. Available: <https://www.networkworld.com/article/3324598/data-center/data-centers-are-set-to-grow-and-become-more-complex-survey-finds.html>
- [4] B. Kleyman, “State of the Data Center Industry, 2018 – Where We are and What to Expect,” *DataCenter Knowledge*, 2018, Accessed 2019-02-21. [Online]. Available: <https://www.datacenterknowledge.com/afcom/state-data-center-industry-2018-where-we-are-and-what-expect>
- [5] M. Allen, “Why the US Needs 4,000 New Data Centers by 2020,” *Datacenters.com*, 2018, Accessed 2019-02-21. [Online]. Available: <https://www.datacenters.com/news/why-the-us-needs-4-000-new-data-centers-by-2020>
- [6] Cisco, “Cisco Global Cloud Index: Forecast and Methodology, 2016-2021,” White paper, Tech. Rep., 2018, Accessed 2019-02-21. [Online]. Available: <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html>

- [7] A. Chakravarty, K. Schmidtke, S. Giridharan, J. Huang, and V. Zeng, “100G CWDM4 SMF optical interconnects for facebook data centers,” in *2016 Conference on Lasers and Electro-Optics (CLEO)*, June 2016, pp. 1–2.
- [8] E. Watkins, “How data centers can keep up with massive Internet user growth,” TechTarget, 2018, Accessed 2019-02-21. [Online]. Available: <https://searchdatacenter.techtarget.com/feature/QA-How-data-centers-can-keep-up-with-massive-Internet-user-growth>
- [9] N. Liu, “Will Google Debut 800G Data Center Switches in 2022?” SDxCentral, 2022, Accessed 2022-07-27. [Online]. Available: <https://www.sdxcentral.com/articles/news/will-google-debut-800g-data-center-switches-in-2022/2022/01/>
- [10] E. Zeman, “What is 5G? A Guide to the Transformative Wireless Tech That’s Being Hyped to Change Everything,” Fortune, 2018, Accessed 2019-02-21. [Online]. Available: <http://fortune.com/2018/10/08/what-is-5g/>
- [11] 3GPP, “5G: Study on Scenarios and Requirements for Next Generation Access Technologies,” 2017, (3GPP TR 38.913 version 14.2.0 Release 14) Accessed 2019-02-24. [Online]. Available: [https://www.etsi.org/deliver/etsi\\_tr/138900\\_138999/138913/14.02.00\\_60/tr\\_138913v140200p.pdf](https://www.etsi.org/deliver/etsi_tr/138900_138999/138913/14.02.00_60/tr_138913v140200p.pdf)
- [12] M. Z. Chowdhury, M. Shahjalal, S. Ahmed, and Y. M. Jang, “6G Wireless Communication Systems: Applications, Requirements, Technologies, Challenges, and Research Directions,” *IEEE Open Journal of the Communications Society*, vol. 1, pp. 957–975, 2020. doi: 10.1109/OJCOMS.2020.3010270
- [13] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, “A comprehensive survey of Network Function Virtualization,” *Computer Networks*, vol. 133, pp. 212 – 262, 2018. doi: <https://doi.org/10.1016/j.comnet.2018.01.021>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1389128618300306>
- [14] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, “Making Middleboxes Someone else’s Problem: Network Processing As a Cloud Service,” in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’12. New York, NY, USA: ACM, 2012. doi: 10.1145/2342356.2342359. ISBN 978-1-4503-1419-0 pp. 13–24. [Online]. Available: <http://doi.acm.org/10.1145/2342356.2342359>

- [15] A. Nandugudi, M. Gallo, D. Perino, and F. Pianese, “Network function virtualization: through the looking-glass,” *Annals of Telecommunications*, vol. 71, no. 11, pp. 573–581, Dec 2016. doi: 10.1007/s12243-016-0540-9. [Online]. Available: <https://doi.org/10.1007/s12243-016-0540-9>
- [16] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire Jr., “Metron: NFV Service Chains at the True Speed of the Underlying Hardware,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018. ISBN 978-1-931971-43-0 pp. 171–186. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/katsikas>
- [17] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr., and D. Kostić, “SNF: Synthesizing high performance NFV service chains,” *PeerJ Computer Science*, vol. 2, p. e98, Nov. 2016. doi: 10.7717/peerj-cs.98. [Online]. Available: <http://dx.doi.org/10.7717/peerj-cs.98>
- [18] 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)*, June 2011. ISSN 1063-6897 pp. 365–376.
- [19] K. Rupp, “42 Years of Microprocessor Trend Data,” <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>, February 2018, (Online; accessed 2019-02-01).
- [20] Ethernet Alliance, “The 2018 Ethernet Roadmap,” <https://ethernetalliance.org/the-2018-ethernet-roadmap/>, 2018, (Online; accessed 2019-02-01).
- [21] R. Zhao, W. Fischer, E. Aker, and P. Rigby, “Broadband Access Technologies,” *A White Paper by the Deployment & Operations Committee*, 2013, Accessed 2019-04-05. [Online]. Available: [https://www.ftthcouncil.eu/documents/Publications/DandO\\_White\\_Paper\\_2\\_2013\\_Final.pdf](https://www.ftthcouncil.eu/documents/Publications/DandO_White_Paper_2_2013_Final.pdf)
- [22] G. E. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965, Accessed 2019-02-21. [Online]. Available: <https://www.cs.utexas.edu/users/fussell/courses/cs352h/papers/moore.pdf>
- [23] G. P. Katsikas, G. Q. Maguire Jr., and D. Kostić, “Profiling and accelerating commodity NFV service chains with SCC,” *Journal of Systems and Software*, vol. 127, pp. 12 – 27, 2017. doi: <https://doi.org/10.1016/j.jss.2017.01.005>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121217300055>

- [24] T. Barbette, C. Soldani, and L. Mathy, “Fast Userspace Packet Processing,” in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '15. Washington, DC, USA: IEEE Computer Society, 2015. ISBN 978-1-4673-6632-8 pp. 5–16.
- [25] J. W. Creswell and J. D. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017. ISBN 9781506386713
- [26] A. Farshin, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, “Make the Most out of Last Level Cache in Intel Processors,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: ACM, 2019. doi: 10.1145/3302424.3303977. ISBN 978-1-4503-6281-8 pp. 8:1–8:17. [Online]. Available: <http://doi.acm.org/10.1145/3302424.3303977>
- [27] —, “Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020. ISBN 978-1-939133-14-4 pp. 673–689. [Online]. Available: <https://www.usenix.org/conference/atc20/presentation/farshin>
- [28] A. Roozbeh, A. Farshin, D. Kostić, and G. Q. Maguire Jr., “Memory Allocation in a Hierarchical Memory System,” 2018, WO Patent WO2019245445A1, Filed 2018-06-21. [Online]. Available: <https://patents.google.com/patent/WO2019245445A1>
- [29] Amir Roozbeh, Alireza Farshin, Dejan Kostić, and Gerald Q. Maguire Jr., “Methods and nodes for handling memory,” 2018, WO Patent WO2020122779A1, Filed 2018-12-13. [Online]. Available: <https://patents.google.com/patent/WO2020122779A1>
- [30] A. Roozbeh, A. Farshin, D. Kostić, and G. Q. Maguire Jr., “Methods and devices for controlling memory handling,” 2019, WO Patent WO2020167234A1, Filed 2019-02-14. [Online]. Available: <https://patents.google.com/patent/WO2020167234A1>
- [31] Amir Roozbeh, Alireza Farshin, Dejan Kostić and Gerald Q. Maguire Jr., “Entities, system and methods performed therein for handling memory operations of an application in a computer environment,” 2019, WO Patent WO2021066687A1, Filed 2019-10-02. [Online]. Available: <https://patents.google.com/patent/WO2021066687A1>

- [32] A. Roozbeh, A. Farshin, C. Padala, D. Kostić, and G. Q. Maguire Jr., “Efficient Loading of Code Portion to a Cache,” 2020, WO Patent WO2021235988A1, Filed 2019-05-15. [Online]. Available: <https://patents.google.com/patent/WO2021235988A1>
- [33] A. Roozbeh, A. Farshin, D. Kostić, and G. Q. Maguire Jr., “Method and System for Efficient Input/Output Transfer in Network Devices,” 2020, WO Patent WO2022108497A1, Filed 2020-11-20. [Online]. Available: <https://patents.google.com/patent/WO2022108497A1>
- [34] Amir Roozbeh and Alireza Farshin and Dejan Kostić and Gerald Q. Maguire Jr., “Method and System for Efficient Input/Output Transfer in. Network Devices (2),” 2020, wO Patent WO2022108498A1, Filed 2020-11-20. [Online]. Available: <https://patents.google.com/patent/WO2022108498A1>
- [35] A. Roozbeh, A. Farshin, T. Barbette, D. Kostić, and G. Q. Maguire Jr., “Method and System for Efficient Metadata and Data Delivery Between a Network Interface and Application,” 2021, PCT Patent PCT/IB2021/052976, Filed 2021-04-09. [Online]. Available: <https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2022214854>
- [36] A. Roozbeh, A. Farshin, C. Padala, D. Kostić, and G. Q. Maguire Jr., “System, Method, and Apparatus for Fine-grained Control of I/O Data Placement in Memory Subsystem,” 2021, PCT Patent PCT/SE2021/050803, Filed 2021-08.
- [37] A. Roozbeh, A. Farshin, and C. Padala, “System and Method for Cache pooling and Efficient Usage and I/O Transfer in disaggregated and Multi-Processor Architectures via Processor Interconnect,” 2021, PCT Patent PCT/SE2021/051016, Filed 2021-10.
- [38] A. Farshin, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, “Optimizing Intel Data Direct I/O Technology for Multi-hundred-gigabit Networks,” in *Proceedings of Fifteenth EuroSys Conference 2020*, ser. EuroSys ’20, 2020. [Online]. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-272720>
- [39] A. Roozbeh, “Toward Next-generation Data Centers,” Licentiate thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Kista, Sweden, May 2019, TRITA-EECS-AVL-2019:39 ISBN 978-91-7873-169-5.
- [40] —, “Realizing Next-Generation Data Centers via Software-Defined “Hardware” Infrastructures and Resource Disaggregation: Exploiting your



- cache,” Doctoral dissertation, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Kista, Sweden, December 2021, TRITA-EECS-AVL-2021:77 ISBN 978-91-8040-065-7.
- [41] Ericsson, “Supercharging the Evolved Packet Gateway,” Ericsson, Tech. Rep., June 2017, <https://www.ericsson.com/assets/local/digital-services/doc/Supercharging-the-Evolved-Packet-Gateway.pdf>, accessed 2020-07-24. [Online]. Available: <https://www.ericsson.com/assets/local/digital-services/doc/Supercharging-the-Evolved-Packet-Gateway.pdf>
  - [42] M. Konstantynowicz, P. Lu, and S. M. Shah, “Benchmarking and Analysis of Software Data Planes,” Cisco, Intel Corporation, FD.io, Tech. Rep., Dec 2017, [https://fd.io/wp-content/uploads/sites/34/2018/01/performance\\_analysis\\_sw\\_data\\_planes\\_dec21\\_2017.pdf](https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf), accessed 2019-07-24. [Online]. Available: [https://fd.io/wp-content/uploads/sites/34/2018/01/performance\\_analysis\\_sw\\_data\\_planes\\_dec21\\_2017.pdf](https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf)
  - [43] G. Tkachuk, M. Konstantynowicz, and S. M. Shah, “Benchmarking Software Data Planes - Intel Xeon Skylake vs. Broadwell,” Cisco, Intel Corporation, FD.io, Tech. Rep., March 2019, [https://www.lfnetworking.org/wp-content/uploads/sites/55/2019/03/benchmarking\\_sw\\_data\\_planes\\_skk\\_bdx\\_mar07\\_2019.pdf](https://www.lfnetworking.org/wp-content/uploads/sites/55/2019/03/benchmarking_sw_data_planes_skk_bdx_mar07_2019.pdf), accessed 2020-07-24. [Online]. Available: [https://www.lfnetworking.org/wp-content/uploads/sites/55/2019/03/benchmarking\\_sw\\_data\\_planes\\_skk\\_bdx\\_mar07\\_2019.pdf](https://www.lfnetworking.org/wp-content/uploads/sites/55/2019/03/benchmarking_sw_data_planes_skk_bdx_mar07_2019.pdf)
  - [44] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, “Dataplane Specialization for High-Performance OpenFlow Software Switching,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2934872.2934887. ISBN 9781450341936 p. 539–552. [Online]. Available: <https://doi.org/10.1145/2934872.2934887>
  - [45] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, “PacketMill: Toward per-Core 100-Gbps Networking,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3445814.3446724. ISBN 9781450383172 p. 1–17. [Online]. Available: <https://doi.org/10.1145/3445814.3446724>
  - [46] A. Farshin and T. Barbette, “PacketMill: Toward per-core 100-Gbps Networking - Artifact for ASPLOS'21,” Jan. 2021, Note that this is

- just an archive for ASPLOS'21 artifact evaluation; you can access the latest version at <https://github.com/aliireza/packetmill>. [Online]. Available: <https://doi.org/10.5281/zenodo.4435970>
- [47] A. Roozbeh, A. Farshin, T. Barbette, H. Ghasemirahni, D. Kostić, and G. Q. Maguire Jr., “Reordering and reframing packets,” 2020, WO Patent WO2021240215A1, Filed 2020-05-26. [Online]. Available: <https://patents.google.com/patent/WO2021240215A1>
  - [48] Q. Cai, S. Chaudhary, M. Vuppalaapati, J. Hwang, and R. Agarwal, “Understanding Host Network Stack Overheads,” in *Proceedings of the 2021 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '21. New York, NY, USA: ACM, 2021. doi: 10.1145/3452296.3472888. [Online]. Available: <http://doi.acm.org/10.1145/3452296.3472888>
  - [49] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, “Understanding PCIe Performance for End Host Networking,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018. doi: 10.1145/3230543.3230560. ISBN 978-1-4503-5567-4 pp. 327–341. [Online]. Available: <http://doi.acm.org/10.1145/3230543.3230560>
  - [50] A. Kalia, M. Kaminsky, and D. G. Andersen, “Design Guidelines for High Performance RDMA Systems,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016. ISBN 978-1-931971-30-0 pp. 437–450. [Online]. Available: <https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia>
  - [51] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, and G. Q. Maguire Jr., “What You Need to Know About (Smart) Network Interface Cards,” in *Passive and Active Measurement*, O. Hohlfeld, A. Lutu, and D. Levin, Eds. Cham: Springer International Publishing, 2021. doi: 10.1007/978-3-030-72582-2\_19. ISBN 978-3-030-72582-2 pp. 319–336. [Online]. Available: [https://doi.org/10.1007/978-3-030-72582-2\\_19](https://doi.org/10.1007/978-3-030-72582-2_19)
  - [52] 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*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3341302.3342079. ISBN 9781450359566 p. 318–333. [Online]. Available: <https://doi.org/10.1145/3341302.3342079>

- [53] A. Farshin, L. Rizzo, K. Elmeleegy, and D. Kostić, “Overcoming the IOTLB Wall for Multi-100-Gbps Linux-based Networking,” *PeerJ Computer Science*, 2022.
- [54] A. Farshin, “aliireza/iommu-bench: PeerJ-CS,” Nov. 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.7278161>
- [55] —, “aliireza/linux: PeerJ-CS-hugepage,” Nov. 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.7278172>
- [56] —, “aliireza/linux: PeerJ-CS-Drop,” Nov. 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.7278174>
- [57] “Intel Ethernet Flow Director and Memcached Performance,” <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf>, 2014, (Online; accessed: 2019-02-01).
- [58] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, “MICA: A Holistic Approach to Fast In-memory Key-value Storage,” in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’14. Berkeley, CA, USA: USENIX Association, 2014. ISBN 978-1-931971-09-6 pp. 429–444. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2616448.2616488>
- [59] K. Zhang, D. Zhuo, and A. Krishnamurthy, “Gallium: Automated Software Middlebox Offloading to 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*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3387514.3405869. ISBN 9781450379557 p. 283–295. [Online]. Available: <https://doi.org/10.1145/3387514.3405869>
- [60] Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen, “Automated SmartNIC Offloading Insights for Network Functions,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3477132.3483583. ISBN 9781450387095 p. 772–787. [Online]. Available: <https://doi.org/10.1145/3477132.3483583>
- [61] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh, M. Girondi, M. Chiesa, G. Q. Maguire Jr., and D. Kostić, “Packet Order

- Matters! Improving Application Performance by Deliberately Delaying Packets,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. ISBN 978-1-939133-27-4 pp. 807–827. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni>
- [62] A. Roozbeh, A. Farshin, M. Chiesa, and F. L. Verdi, “System and Method for Accurate Traffic Monitoring on Multi-Pipeline Switches,” 2021, PCT Patent PCT/EP2021/084572, Filed 2021-12.
- [63] A. Roozbeh, A. Farshin, and D. Kostić, “System and Method for Organizing Physical Queues into Virtual Queues,” 2022, PCT Patent PCT/EP2022/051103, Filed 2022-01.
- [64] A. Roozbeh, A. Farshin, M. Chiesa, T. Barbette, and D. Kostić, “Apparatus, System, and Methods for Sliced Accelerated Packet Processing at Terabit-per-second Networking,” 2022, US Provisional Patent, Filed 2022-05.
- [65] A. Roozbeh, C. Padala, A. Farshin, D. Kostić, and G. Q. Maguire Jr., “Processing Unit, Packet Handling Unit, Arrangement and Methods for Handling Packets,” 2022, PCT Patent PCT/SE2022/050710, Filed 2022-07.
- [66] A. Roozbeh, A. Farshin, M. Chiesa, D. Kostić, and H. Ghasemirahni, “Hint Entity, Receiver Node, System and Methods Performed Therein for Handling Data in a Computer Environment,” 2022, PCT Patent PCT/SE2022/051036, Filed 2022-11.
- [67] A. Roozbeh, A. Farshin, and M. Chiesa, “Entity and Method Performed Therein for Handling Packets in a Computer Environment,” 2022, US Provisional Patent, Filed 2022-11.
- [68] A. Farshin, A. Roozbeh, C. Schulte, G. Q. Maguire Jr., and D. Kostić, “Scheduling - A Secret Sauce For Resource Disaggregation,” KTH, Network Systems Laboratory and Software and Computer systems and Ericsson Research, Tech. Rep., 2021. [Online]. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-302728>
- [69] A. Farshin, “Realizing low-latency Internet Services via low-level Optimization of NFV Service Chains,” Licentiate thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Kista, Sweden, Jun. 2019, TRITA-EECS-AVL-2019:41 ISBN 978-91-7873-175-6. [Online]. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-249664>

- [70] O. Giordano, “Design and Implementation of an Architecture-aware In-memory Key-Value Store,” Master’s thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2021. [Online]. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-291213>
- [71] The United Nations World Commission on Environment and Development (UNWCED), “Report of the World Commission on Environment and Development: Our common future,” The United Nations World Commission on Environment and Development (UNWCED), Tech. Rep., 1987, transmitted to the General Assembly as an Annex to document A/42/427 - Development and International Co-operation: Environment. [Online]. Available: <http://www.un-documents.net/wced-ocf.htm>
- [72] C. Kim, D. Burger, and S. W. Keckler, “An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches,” *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 211–222, Oct. 2002. doi: 10.1145/635506.605420. [Online]. Available: <http://doi.acm.org/10.1145/635506.605420>
- [73] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA ’09. New York, NY, USA: ACM, 2009. doi: 10.1145/1555754.1555779. ISBN 978-1-60558-526-0 pp. 184–195. [Online]. Available: <http://doi.acm.org/10.1145/1555754.1555779>
- [74] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, “Mapping the Intel Last-Level Cache,” Cryptology ePrint Archive, Report 2015/905, 2015, <https://eprint.iacr.org/2015/905>.
- [75] R. Hund, C. Willems, and T. Holz, “Practical Timing Side Channel Attacks against Kernel Space ASLR,” in *2013 IEEE Symposium on Security and Privacy*, May 2013. doi: 10.1109/SP.2013.23. ISSN 1081-6011 pp. 191–205.
- [76] M. Seaborn, “L3 cache mapping on Sandy Bridge CPUs,” 2015, Accessed 2018-05-09. [Online]. Available: <http://lackingrhoticity.blogspot.se/2015/04/13-cache-mapping-on-sandy-bridge-cpus.html>
- [77] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-Level Cache Side-Channel Attacks are Practical,” in *2015 IEEE Symposium on Security and Privacy*, May 2015. doi: 10.1109/SP.2015.43. ISSN 1081-6011 pp. 605–622.

- [78] G. I. Apecechea, T. Eisenbarth, and B. Sunar, “Systematic Reverse Engineering of Cache Slice Selection in Intel Processors,” *IACR Cryptology ePrint Archive*, vol. 2015, p. 690, 2015, Accessed 2019-03-19. [Online]. Available: <https://eprint.iacr.org/2015/690>
- [79] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters,” in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015. ISBN 978-3-319-26361-8 pp. 48–65. [Online]. Available: [http://dx.doi.org/10.1007/978-3-319-26362-5\\_3](http://dx.doi.org/10.1007/978-3-319-26362-5_3)
- [80] Data Plane Development Kit (DPDK), 2022, Accessed 2022-07-14. [Online]. Available: <https://www.dpdk.org/>
- [81] eXpress Data Path (XDP), <https://www.iovisor.org/technology/xdp>, 2016.
- [82] S. Gallenmüller, D. Scholz, F. Wohlfart, Q. Scheitle, P. Emmerich, and G. Carle, “High-performance packet processing and measurements,” in *2018 10th International Conference on Communication Systems Networks (COMSNETS)*, Jan 2018. doi: 10.1109/COMSNETS.2018.8328173 pp. 1–8.
- [83] T. Hudek, “Introduction to Receive Side Scaling,” 04 2017, Accessed 2018-06-09. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>
- [84] Mellanox Technologies, “Mellanox NIC’s Performance Report with DPDK 17.05,” Mellanox Technologies, Tech. Rep. MLNX-15-52365, 2017, Accessed 2018-06-09. [Online]. Available: [http://fast.dpdk.org/doc/perf/DPDK\\_17\\_05\\_Mellanox\\_NIC\\_performance\\_report.pdf](http://fast.dpdk.org/doc/perf/DPDK_17_05_Mellanox_NIC_performance_report.pdf)
- [85] R. Sudarikov and P. Lu, “Hardware-Level Performance Analysis of Platform I/O,” p. 7, June 2018, Accessed 2019-01-10. [Online]. Available: <https://dpdkprcsummit2018.sched.com/event/EsPa/hardware-level-performance-analysis-of-platform-io>
- [86] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, “Bobtail: Avoiding Long Tails in the Cloud,” in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013. ISBN 978-1-931971-00-3 pp. 329–341. [Online]. Available: [https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu\\_yunjing](https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_yunjing)

- [87] L. Rizzo, “netmap: A Novel Framework for Fast Packet I/O,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012. ISBN 978-931971-93-5 pp. 101–112. [Online]. Available: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo>
- [88] OpenOnload, <http://www.openonload.org>, 2018.
- [89] S. Han, K. Jang, K. Park, and S. Moon, “PacketShader: A GPU-accelerated Software Router,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, Aug. 2010. doi: 10.1145/1851275.1851207. [Online]. Available: <http://doi.acm.org/10.1145/1851275.1851207>
- [90] J. H. Salim, R. Olsson, and A. Kuznetsov, “Beyond Softnet,” in *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ser. ALS ’01. Berkeley, CA, USA: USENIX Association, 2001. [Online]. Available: [http://www.usenix.org/publications/library/proceedings/als01/full\\_papers/jamal/jamal.pdf](http://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf)
- [91] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, “MegaPipe: A New Programming Interface for Scalable Network I/O ,” in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012. ISBN 978-1-931971-96-6 pp. 135–148. [Online]. Available: <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han>
- [92] A. Bremner-Barr, Y. Harchol, and D. Hay, “OpenBox: Enabling Innovation in Middlebox Applications,” in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMiddlebox ’15. New York, NY, USA: ACM, 2015. doi: 10.1145/2785989.2785992. ISBN 978-1-4503-3540-9 pp. 67–72. [Online]. Available: <http://doi.acm.org/10.1145/2785989.2785992>
- [93] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, “RouteBricks: Exploiting Parallelism to Scale Software Routers,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP ’09. New York, NY, USA: ACM, 2009. doi: 10.1145/1629575.1629578. ISBN 978-1-60558-752-3 pp. 15–28. [Online]. Available: <http://doi.acm.org/10.1145/1629575.1629578>
- [94] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and Implementation of a Consolidated Middlebox Architecture,” in

- Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 24–24. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2228298.2228331>
- [95] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, “NFP: Enabling Network Function Parallelism in NFV,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017. doi: 10.1145/3098822.3098826. ISBN 978-1-4503-4653-5 pp. 43–56. [Online]. Available: <http://doi.acm.org/10.1145/3098822.3098826>
- [96] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, “OpenNetVM: A Platform for High Performance Network Service Chains,” in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMiddlebox '16. New York, NY, USA: ACM, 2016. doi: 2940147.2940155. ISBN 978-1-4503-4424-1 pp. 26–31. [Online]. Available: <http://doi.acm.org/2940147.2940155>
- [97] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, “ResQ: Enabling SLOs in Network Function Virtualization,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018. ISBN 978-1-931971-43-0 pp. 283–297. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/tootoonchian>
- [98] P. Veitch, E. Curley, and T. Kantecki, “Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation,” in *2017 IEEE Conference on Network Softwarization (NetSoft)*, July 2017. doi: 10.1109/NETSOFT.2017.8004214 pp. 1–5.
- [99] P. Veitch, “Cache Allocation Technology: A Telco’s NFV Noisy Neighbor Experiments,” Aug 2017, Accessed 2019-01-10. [Online]. Available: <https://software.intel.com/en-us/articles/cache-allocation-technology-telco-nfv-noisy-neighbor-experiments>
- [100] N. McDonnell and G. Eads, “Queue Management and Load Balancing on Intel Architecture,” 2020, <https://tinyurl.com/yxv9cgpj>, accessed 2020-08-08.
- [101] 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 Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. doi: 10.1145/2656877.2656890. [Online]. Available: <https://doi.org/10.1145/2656877.2656890>
- [102] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, “The Case For In-Network Computing On Demand,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys ’19. New York, NY, USA: ACM, 2019. doi: 10.1145/3302424.3303979. ISBN 978-1-4503-6281-8 pp. 21:1–21:16. [Online]. Available: <http://doi.acm.org/10.1145/3302424.3303979>
- [103] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, “Fast Packet Processing: A Survey,” *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3645–3676, 2018. doi: 10.1109/COMST.2018.2851072. [Online]. Available: <https://doi.org/10.1109/COMST.2018.2851072>
- [104] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, “Survey of Performance Acceleration Techniques for Network Function Virtualization,” *Proceedings of the IEEE*, vol. 107, no. 4, pp. 746–764, 2019. doi: 10.1109/JPROC.2019.2896848. [Online]. Available: <https://doi.org/10.1109/JPROC.2019.2896848>
- [105] ntop, “PF\_RING ZC (Zero Copy),” 2020, [https://www.ntop.org/products/packet-capture/pf\\_ring/pf\\_ring-zc-zero-copy/](https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/), accessed 2020-08-02.
- [106] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, “The Design and Implementation of Open vSwitch,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015. ISBN 978-1-931971-218 pp. 117–130. [Online]. Available: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>
- [107] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, and D. Rossi, “Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned,” *IEEE Communications Magazine*, vol. 55, no. 3, pp. 163–169, 2017. doi: 10.1109/MCOM.2017.1600756CM. [Online]. Available: <https://doi.org/10.1109/MCOM.2017.1600756CM>
- [108] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, “FlowMon-DPDK: Parsimonious Per-Flow Software Monitoring at Line Rate,” in *2018 Network Traffic Measurement and Analysis Conference*

- (TMA), 2018. doi: 10.23919/TMA.2018.8506565 pp. 1–8. [Online]. Available: <https://doi.org/10.23919/TMA.2018.8506565>
- [109] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, “The Click Modular Router,” in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’99. New York, NY, USA: Association for Computing Machinery, 1999. doi: 10.1145/319151.319166. ISBN 1581131402 p. 217–231. [Online]. Available: <https://doi.org/10.1145/319151.319166>
- [110] FD.io, “Vector Packet Processing - One Terabit Software Router on Intel Xeon Scalable Processor Family Server,” Cisco, Intel Corporation, FD.io, Tech. Rep., July 2017, <https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf>, accessed 2020-07-24. [Online]. Available: <https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf>
- [111] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho, “SnabbSwitch user space virtual switch benchmark and performance optimization for NFV,” in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015. doi: 10.1109/NFV-SDN.2015.7387411 pp. 86–92. [Online]. Available: <https://doi.org/10.1109/NFV-SDN.2015.7387411>
- [112] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, “SoftNIC: A Software NIC to Augment Hardware,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>
- [113] —, “Berkeley Extensible Software Switch (BESS),” 2015, <http://span.cs.berkeley.edu/bess.html>, accessed 2020-07-22.
- [114] E. Kohler, R. Morris, and B. Chen, “Programming Language Optimizations for Modular Router Configurations,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: Association for Computing Machinery, 2002. doi: 10.1145/605397.605424. ISBN 1581135742 p. 251–263. [Online]. Available: <https://doi.org/10.1145/605397.605424>
- [115] B. Deng, W. Wu, and L. Song, “Redundant Logic Elimination in Network Functions,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR

- '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3373360.3380832. ISBN 9781450371018 p. 34–40. [Online]. Available: <https://doi.org/10.1145/3373360.3380832>
- [116] C. Cadar, D. Dunbar, and D. 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*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.
- [117] J. M. Westall, “Management of *sk\_buffs*,” <https://people.cs.clemson.edu/~westall/853/notes/skbuff.pdf>, 2011.
- [118] S. Reddy, “What is SKB in Linux kernel? What are SKB operations? Memory Representation of SKB? How to send packet out using skb operations?” <http://amsekharkernel.blogspot.com/2014/08/what-is-skb-in-linux-kernel-what-are.html>, 2014.
- [119] DPDK, “Mbuf Library,” [https://doc.dpdk.org/guides/prog\\_guide/mbuf\\_lib.html](https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html), 2020.
- [120] FastClick, “Packet Class,” <https://github.com/tbarbette/fastclick/blob/master/include/click/packet.hh>, 2019.
- [121] BESS, “Packet,” <https://github.com/NetSys/bess/blob/master/core/packet.h>, 2019.
- [122] —, “sn\_buff Layout,” [https://github.com/NetSys/bess/blob/master/core/snbuf\\_layout.h](https://github.com/NetSys/bess/blob/master/core/snbuf_layout.h), 2017.
- [123] S. Pirelli and G. Candea, “A Simpler and Faster NIC Driver Model for Network Functions,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. ISBN 978-1-939133-19-9 pp. 225–241. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/pirelli>
- [124] M. Godbolt, “Optimizations in C++ Compilers,” *Commun. ACM*, vol. 63, no. 2, p. 41–49, Jan. 2020. doi: 10.1145/3369754. [Online]. Available: <https://doi.org/10.1145/3369754>
- [125] LLVM, “LLVM Link Time Optimization: Design and Implementation,” 2020, <https://llvm.org/docs/LinkTimeOptimization.html>, accessed 2020-06-15.

- [126] GCC, “Link Time Optimization,” 2009, <https://gcc.gnu.org/wiki/LinkTimeOptimization>, accessed 2020-06-15.
- [127] T. Glek and J. Hubička, “Optimizing real world applications with GCC Link Time Optimization,” 2010, <http://sciencewise.info/media/pdf/1010.2196v2.pdf>, accessed 2020-06-15.
- [128] LLVM, “ThinLTO,” 2020, <https://clang.llvm.org/docs/ThinLTO.html>, accessed 2020-06-15.
- [129] The Rust Language Reference, “Struct Types,” 2008, <https://github.com/rust-lang/reference/blob/master/src/types/struct.md>, accessed 2020-06-15.
- [130] Stack Overflow, “Struct Reordering by compiler,” 2016, <https://stackoverflow.com/questions/38244689/struct-reordering-by-compiler>, accessed 2020-06-15.
- [131] —, “Why can’t C compilers rearrange struct members to eliminate alignment padding?” 2012, <https://tinyurl.com/yxncnqk8>, accessed 2020-08-07.
- [132] —, “Why doesn’t GCC optimize structs?” 2008, <https://stackoverflow.com/questions/118068/why-doesnt-gcc-optimize-structs>, accessed 2020-06-15.
- [133] LLVM, “Four bitcode generated with plugin-opt=save-temps,” 2018, <http://lists.llvm.org/pipermail/llvm-dev/2018-May/123341.html>, accessed 2020-06-15.
- [134] R. Keller and S. Fan, *PINstruct – Efficient Memory Access to Data Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 127–128. ISBN 978-3-642-35893-7. [Online]. Available: [https://doi.org/10.1007/978-3-642-35893-7\\_14](https://doi.org/10.1007/978-3-642-35893-7_14)
- [135] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: Association for Computing Machinery, 2005. doi: 10.1145/1065010.1065034. ISBN 1595930566 p. 190–200. [Online]. Available: <https://doi.org/10.1145/1065010.1065034>

- [136] V. Sundriyal, M. Sosonkina, B. M. Westheimer, and M. Gordon, "Comparisons of Core and Uncore Frequency Scaling Modes in Quantum Chemistry Application GAMESS," in *Proceedings of the High Performance Computing Symposium*, ser. HPC '18. San Diego, CA, USA: Society for Computer Simulation International, 2018. ISBN 9781510860162
- [137] C. Gough, I. Steiner, and W. A. Saunders, *Energy Efficient Servers: Blueprints for Data Center Optimization*, 1st ed. USA: Apress, 2015. ISBN 1430266376
- [138] T. Barbette, "Network Performance Framework (NPF)," 2020, <https://github.com/tbarbette/npf>, accessed 2020-07-24.
- [139] —, "Architecture for programmable network infrastructure," Ph.D. dissertation, University of Liege, 2018, <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1249035>, accessed 2020-12-23.
- [140] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-Speed Software Data Plane via Vectorized Packet Processing," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 97–103, 2018. doi: 10.1109/MCOM.2018.1800069
- [141] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the Performance of State-of-the-Art Software Switches for NFV," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3359989.3365415. ISBN 9781450369985 p. 68–81. [Online]. Available: <https://doi.org/10.1145/3359989.3365415>
- [142] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A Practical Binary Optimizer for Data Centers and Beyond," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019. ISBN 9781728114361 p. 2–14.
- [143] Google, "Github - Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker," 2020, <https://github.com/google/llvm-propeller>, accessed 2020-06-15.
- [144] D. E. Knuth, "Structured Programming with Go to Statements," *ACM Comput. Surv.*, vol. 6, no. 4, p. 261–301, Dec. 1974. doi: 10.1145/356635.356640. [Online]. Available: <https://doi.org/10.1145/356635.356640>

- [145] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, “A Formally Verified NAT,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3098822.3098833. ISBN 9781450346535 p. 141–154. [Online]. Available: <https://doi.org/10.1145/3098822.3098833>
- [146] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, “Verifying Software Network Functions with No Verification Expertise,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3341301.3359647. ISBN 9781450368735 p. 275–290. [Online]. Available: <https://doi.org/10.1145/3341301.3359647>
- [147] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, “Automated Synthesis of Adversarial Workloads for Network Functions,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3230543.3230573. ISBN 9781450355674 p. 372–385. [Online]. Available: <https://doi.org/10.1145/3230543.3230573>
- [148] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Producing Wrong Data without Doing Anything Obviously Wrong!” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009. doi: 10.1145/1508244.1508275. ISBN 9781605584065 p. 265–276. [Online]. Available: <https://doi.org/10.1145/1508244.1508275>
- [149] C. Curtsinger and E. D. Berger, “STABILIZER: Statistically Sound Performance Evaluation,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013. doi: 10.1145/2451116.2451141. ISBN 9781450318709 p. 219–228. [Online]. Available: <https://doi.org/10.1145/2451116.2451141>
- [150] A. D. Biagio and M. Davis, “llvm-mca - LLVM Machine Code Analyzer,” 2020, <https://llvm.org/docs/CommandGuide/llvm-mca.html>, accessed 2020-06-15.

- [151] I. Hirsh and G. Stupp, “Intel Architecture Code Analyzer,” 2019, <https://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer.html>, accessed 2020-06-15.
- [152] H. Massalin, “Superoptimizer: A Look at the Smallest Program,” in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Washington, DC, USA: IEEE Computer Society Press, 1987. doi: 10.1145/36206.36194. ISBN 0818608056 p. 122–126. [Online]. Available: <https://doi.org/10.1145/36206.36194>
- [153] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, “Souper: A Synthesizing Superoptimizer,” *CoRR*, vol. abs/1711.04422, 2017. [Online]. Available: <http://arxiv.org/abs/1711.04422>
- [154] Google, “Github - Souper: A superoptimizer for LLVM IR,” 2020, <https://github.com/google/souper>, accessed 2020-06-15.
- [155] R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, “Combinatorial Register Allocation and Instruction Scheduling,” *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, Jul. 2019. doi: 10.1145/3332373. [Online]. Available: <https://doi.org/10.1145/3332373>
- [156] F. T. Boesch and J. F. Gimpel, “Covering Points of a Digraph with Point-Disjoint Paths and Its Application to Code Optimization,” *J. ACM*, vol. 24, no. 2, p. 192–198, Apr. 1977. doi: 10.1145/322003.322005. [Online]. Available: <https://doi.org/10.1145/322003.322005>
- [157] C. Young, D. S. Johnson, M. D. Smith, and D. R. Karger, “Near-Optimal Intraprocedural Branch Alignment,” in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI ’97. New York, NY, USA: Association for Computing Machinery, 1997. doi: 10.1145/258915.258932. ISBN 0897919076 p. 183–193. [Online]. Available: <https://doi.org/10.1145/258915.258932>
- [158] B. Calder and D. Grunwald, “Reducing Branch Costs via Branch Alignment,” *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, p. 242–251, Nov. 1994. doi: 10.1145/381792.195553. [Online]. Available: <https://doi.org/10.1145/381792.195553>
- [159] A. Ramírez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, “Software Trace Cache,” in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New

- York, NY, USA: Association for Computing Machinery, 1999. doi: 10.1145/2591635.2667175. ISBN 9781450328401 p. 261–268. [Online]. Available: <https://doi.org/10.1145/2591635.2667175>
- [160] A. Newell and S. Pupyrev, “Improved Basic Block Reordering,” *IEEE Transactions on Computers*, p. 1–1, 2020. doi: 10.1109/tc.2020.2982888. [Online]. Available: <http://dx.doi.org/10.1109/TC.2020.2982888>
- [161] D. Chen, D. X. Li, and T. Moseley, “AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications,” in *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, New York, NY, USA, 2016, pp. 12–23.
- [162] C.-K. Luk, R. Muth, Harish Patil, R. Cohn, and G. Lowney, “Ispike: a post-link optimizer for the Intel/spl reg/ Itanium/spl reg/ architecture,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004. doi: 10.1109/CGO.2004.1281660 pp. 15–26. [Online]. Available: <https://doi.org/10.1109/CGO.2004.1281660>
- [163] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, “PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture,” in *In Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.
- [164] R. Lavaee, J. Criswell, and C. Ding, “Codestitcher: Inter-Procedural Basic Block Layout Optimization,” in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3302516.3307358. ISBN 9781450362771 p. 65–75. [Online]. Available: <https://doi.org/10.1145/3302516.3307358>
- [165] G. Ottoni and B. Maher, “Optimizing function placement for large-scale data-center applications,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017. doi: 10.1109/CGO.2017.7863743 pp. 233–244. [Online]. Available: <https://doi.org/10.1109/CGO.2017.7863743>
- [166] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, “Code layout optimizations for transaction processing workloads,” in *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001. doi: 10.1145/379240.379260 pp. 155–164. [Online]. Available: <https://doi.org/10.1145/379240.379260>
- [167] P. Li, H. Luo, C. Ding, Z. Hu, and H. Ye, “Code Layout Optimization for Defensiveness and Politeness in Shared Cache,” in *2014 43rd International*



- Conference on Parallel Processing*, 2014. doi: 10.1109/ICPP.2014.24 pp. 151–161. [Online]. Available: <https://doi.org/10.1109/ICPP.2014.24>
- [168] G. S. Niemiec, L. M. S. Batista, A. E. Schaeffer-Filho, and G. L. Nazar, “A Survey on FPGA Support for the Feasible Execution of Virtualized Network Functions,” *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 504–525, 2020. doi: 10.1109/COMST.2019.2943690. [Online]. Available: <https://doi.org/10.1109/COMST.2019.2943690>
- [169] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, “ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2934872.2934897. ISBN 9781450341936 p. 1–14. [Online]. Available: <https://doi.org/10.1145/2934872.2934897>
- [170] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, “NetFPGA SUME: Toward 100 Gbps as Research Commodity,” *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014. doi: 10.1109/MM.2014.61. [Online]. Available: <https://doi.org/10.1109/MM.2014.61>
- [171] X. Li, X. Wang, F. Liu, and H. Xu, “DHL: Enabling Flexible Software Network Functions with FPGA Acceleration,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018. doi: 10.1109/ICDCS.2018.00011 pp. 1–11. [Online]. Available: <https://doi.org/10.1109/ICDCS.2018.00011>
- [172] W. Sun and R. Ricci, “Fast and flexible: Parallel packet processing with GPUs and click,” in *Architectures for Networking and Communications Systems*, 2013. doi: 10.1109/ANCS.2013.6665173 pp. 25–35. [Online]. Available: <https://doi.org/10.1109/ANCS.2013.6665173>
- [173] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, “GASPP: A GPU-Accelerated Stateful Packet Processing Framework,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014. ISBN 978-1-931971-10-2 pp. 321–332. [Online]. Available: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis>
- [174] X. Yi, J. Duan, and C. Wu, “GPUNFV: A GPU-Accelerated NFV System,” in *Proceedings of the First Asia-Pacific Workshop on Networking*, ser. APNet’17. New York, NY, USA: Association for Computing Machinery,

2017. doi: 10.1145/3106989.3106990. ISBN 9781450352444 p. 85–91. [Online]. Available: <https://doi.org/10.1145/3106989.3106990>
- [175] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, “G-NET: Effective GPU Sharing in NFV Systems,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018. ISBN 978-1-939133-01-4 pp. 187–200. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/zhang-kai>
- [176] 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)*. Renton, WA: USENIX Association, Jul. 2019. ISBN 978-1-939133-03-8 pp. 345–362. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/eran>
- [177] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, “High Performance Packet Processing with FlexNIC,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2872362.2872367. ISBN 9781450340915 p. 67–81. [Online]. Available: <https://doi.org/10.1145/2872362.2872367>
- [178] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, “Azure Accelerated Networking: SmartNICs in the Public Cloud,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018. ISBN 978-1-939133-01-4 pp. 51–66. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/firestone>
- [179] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, “Achieving 100Gbps Intrusion Prevention on a Single Server,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. ISBN 978-1-939133-19-9 pp. 1083–1100. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng>

- [180] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, “NFP: Enabling Network Function Parallelism in NFV,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3098822.3098826. ISBN 9781450346535 p. 43–56. [Online]. Available: <https://doi.org/10.1145/3098822.3098826>
- [181] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, “ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3050220.3050236. ISBN 9781450349475 p. 143–149. [Online]. Available: <https://doi.org/10.1145/3050220.3050236>
- [182] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, “Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3230543.3230563. ISBN 9781450355674 p. 504–517. [Online]. Available: <https://doi.org/10.1145/3230543.3230563>
- [183] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, “ClickOS and the Art of Network Function Virtualization,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014. ISBN 978-1-931971-09-6 pp. 459–473. [Online]. Available: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins>
- [184] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumathurai, and X. Fu, “NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains,” *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 639–652, 2020. doi: 10.1109/TNET.2020.2969971. [Online]. Available: <https://doi.org/10.1109/TNET.2020.2969971>
- [185] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, “ResQ: Enabling SLOs in Network Function Virtualization,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018. ISBN 978-1-939133-01-4 pp. 283–297. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/tootoonchian>

- [186] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr., and D. Kostić, “RSS++: Load and State-Aware Receive Side Scaling,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3359989.3365412. ISBN 9781450369985 p. 318–333. [Online]. Available: <https://doi.org/10.1145/3359989.3365412>
- [187] K. Kaffes, 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)*. Boston, MA: USENIX Association, Feb. 2019. ISBN 978-1-931971-49-2 pp. 345–360. [Online]. Available: <https://www.usenix.org/conference/nsdi19/presentation/kaffes>
- [188] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr., P. Papadimitratos, and M. Chiesa, “A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. ISBN 978-1-939133-13-7 pp. 667–683. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/barbette>
- [189] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, “NetBricks: Taking the V out of NFV,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016. ISBN 978-1-931971-33-1 pp. 203–216. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda>
- [190] A. Bremler-Barr, Y. Harchol, and D. Hay, “OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2934872.2934875. ISBN 9781450341936 p. 511–524. [Online]. Available: <https://doi.org/10.1145/2934872.2934875>
- [191] B. Anwer, T. Benson, N. Feamster, and D. Levin, “Programming Slick Network Functions,” in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015. doi: 10.1145/2774993.2774998. ISBN 9781450334518. [Online]. Available: <https://doi.org/10.1145/2774993.2774998>

- [192] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and Implementation of a Consolidated Middlebox Architecture,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012. ISBN 978-931971-92-8 pp. 323–336. [Online]. Available: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar>
- [193] Y. Jiang, Y. Cui, W. Wu, Z. Xu, J. Gu, K. K. Ramakrishnan, Y. He, and X. Qian, “SpeedyBox: Low-Latency NFV Service Chains with Cross-NF Runtime Consolidation,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019. doi: 10.1109/ICDCS.2019.00016 pp. 68–79. [Online]. Available: <https://doi.org/10.1109/ICDCS.2019.00016>
- [194] T. Barbette, M. Chiesa, G. Q. Maguire Jr., and D. Kostić, *Stateless CPU-Aware Datacenter Load-Balancing*. New York, NY, USA: Association for Computing Machinery, 2020, p. 548–549. ISBN 9781450379489. [Online]. Available: <https://doi.org/10.1145/3386367.3431672>
- [195] Intel, “Intel Virtualization Technology for Directed I/O,” April 2021, <https://tinyurl.com/2jexuusd>, accessed 2021-08-10.
- [196] AMD, “AMD I/O Virtualization Technology (IOMMU) Specification,” April 2021, <https://tinyurl.com/27w2tw2p>, accessed 2021-08-10.
- [197] ARM, “System MMU Support,” August 2021, <https://developer.arm.com/architectures/system-architectures/system-components/>, accessed 2021-08-10.
- [198] U. Hafeez and V. Patov, “I/O Virtualization with Hardware Support,” May 2017, [https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp17/cse506/slides/hw\\_io\\_virtualization.pdf](https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp17/cse506/slides/hw_io_virtualization.pdf), accessed 2021-08-10.
- [199] E. Bugnion, J. Nieh, and D. Tsafir, “Hardware and Software Support for Virtualization,” *Synthesis Lectures on Computer Architecture*, vol. 12, no. 1, pp. 1–206, 2017. doi: 10.2200/S00754ED1V01Y201701CAC038. [Online]. Available: <https://doi.org/10.2200/S00754ED1V01Y201701CAC038>
- [200] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle, “User Space Network Drivers,” in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*, Sep. 2019.
- [201] A. Kegel, P. Blinzer, A. Basu, and M. Chan, “Virtualizing I/O Through the I/O Memory Management Unit (IOMMU),” April

- 2016, [http://pages.cs.wisc.edu/~basu/isca\\_iommu\\_tutorial/IOMMU\\_TUTORIAL\\_ASPLoS\\_2016.pdf](http://pages.cs.wisc.edu/~basu/isca_iommu_tutorial/IOMMU_TUTORIAL_ASPLoS_2016.pdf), accessed 2021-12-19.
- [202] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafirir, “DAMN: Overhead-Free IOMMU Protection for Networking,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3173162.3173175. ISBN 9781450349116 p. 301–315. [Online]. Available: <https://doi.org/10.1145/3173162.3173175>
- [203] The kernel development community, “Page Pool API,” 2021, linux kernel documentation: [https://www.kernel.org/doc/html/latest/networking/page\\_pool.html](https://www.kernel.org/doc/html/latest/networking/page_pool.html), accessed 2021-12-09.
- [204] M. Malka, “Rethinking the I/O Memory Management Unit (IOMMU),” Master’s thesis, Technion, Computer Science Department, 2015. [Online]. Available: <https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2015/MS/MS-2015-10.pdf>
- [205] Stack Overflow, “Multi-level page tables - hierarchical paging,” 2011, <https://stackoverflow.com/questions/5558886/multi-level-page-tables-hierarchical-paging>, accessed 2022-07-18.
- [206] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafirir, “Utilizing the IOMMU Scalably,” in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. USA: USENIX Association, 2015. ISBN 9781931971225 p. 549–562.
- [207] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn, “The price of safety: Evaluating IOMMU performance,” in *The Ottawa Linux Symposium*, 2007, pp. 9–20, <https://www.kernel.org/doc/ols/2007/ols2007v1-pages-9-20.pdf>, accessed 2021-08-06.
- [208] M. Malka, N. Amit, and D. Tsafirir, “Efficient Intra-Operating System Protection against Harmful DMAs,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. USA: USENIX Association, 2015. ISBN 9781931971201 p. 29–44.
- [209] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafirir, “rIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15.

- New York, NY, USA: Association for Computing Machinery, 2015. doi: 10.1145/2694344.2694355. ISBN 9781450328357 p. 355–368. [Online]. Available: <https://doi.org/10.1145/2694344.2694355>
- [210] A. Markuze, A. Morrison, and D. Tsafir, “True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2872362.2872379. ISBN 9781450340915 p. 249–262. [Online]. Available: <https://doi.org/10.1145/2872362.2872379>
- [211] iPerf, “iPerf - The ultimate speed test tool for TCP, UDP and SCTP,” 2021, <https://iperf.fr/iperf-doc.php>, accessed 2021-08-11.
- [212] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, “BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021. ISBN 978-1-939133-21-2 pp. 487–501. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/ghigoff>
- [213] 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*, ser. EuroSys ’14. New York, NY, USA: Association for Computing Machinery, 2014. doi: 10.1145/2592798.2592821. ISBN 9781450327046. [Online]. Available: <https://doi.org/10.1145/2592798.2592821>
- [214] L. Rizzo, “kstats - collecting statistics in the Linux kernel,” 2020, slides discussing kstats, a library to collect statistics in the linux kernel, posted on the upstream linux mailing lists <https://lwn.net/Articles/813303/>.
- [215] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, “IOMMU: Strategies for Mitigating the IOTLB Bottleneck,” in *Computer Architecture*, A. L. Varbanescu, A. Molnos, and R. van Nieuwpoort, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN 978-3-642-24322-6 pp. 256–274.
- [216] M. Alex, S. Vargaftik, G. Kupfer, B. Pismeny, N. Amit, A. Morrison, and D. Tsafir, “Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys ’21. New York, NY, USA: Association for Computing Machinery, 2021. doi:

- 10.1145/3447786.3456249. ISBN 9781450383349 p. 395–409. [Online]. Available: <https://doi.org/10.1145/3447786.3456249>
- [217] Jonathan Corbet, “Zero-copy networking,” 2017, <https://lwn.net/Articles/726917/>, accessed 2021-08-11.
- [218] W. de Bruijn and E. Dumazet, “sendmsg copy avoidance with MSG\_ZEROCOPY,” 2017, <https://legacy.netdevconf.info/2.1/papers/debruijn-msgzerocopy-talk.pdf>, accessed 2021-08-11.
- [219] A. Markuze, I. Golikov, and C. Dar, “Rethinking Zero-Copy Networking with MAIO,” VMware, Tech. Rep., August 2021, [https://netdevconf.info/0x15/papers/1/maio\\_netdev0x15.pdf](https://netdevconf.info/0x15/papers/1/maio_netdev0x15.pdf), accessed 2021-08-19. [Online]. Available: [https://netdevconf.info/0x15/papers/1/maio\\_netdev0x15.pdf](https://netdevconf.info/0x15/papers/1/maio_netdev0x15.pdf)
- [220] A. Kesavan, R. Ricci, and R. Stutsman, “To Copy or Not to Copy: Making In-Memory Databases Fast on Modern NICs,” in *Data Management on New Hardware*, S. Blanas, R. Bordawekar, T. Lahiri, J. Levandoski, and A. Pavlo, Eds. Cham: Springer International Publishing, 2017. ISBN 978-3-319-56111-0 pp. 79–94.
- [221] Linux Networking Documentation, “MSG\_ZEROCOPY,” 2021, [https://www.kernel.org/doc/html/latest/networking/msg\\_zerocopy.html](https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html), accessed 2021-08-17.
- [222] Jonathan Corbet, “Zero-copy TCP receive,” 2018, <https://lwn.net/Articles/752188/>, accessed 2021-08-11.
- [223] Eric Dumazet, “Path to TCP 4K MTU and RX zerocopy,” 2020, <https://legacy.netdevconf.info/0x14/pub/slides/62/Implementing%20TCP%20RX%20zero%20copy.pdf>, accessed 2021-08-11.
- [224] A. Markuze, “Rethinking Zero Copy Networking with MAIO,” 2021, <https://netdevconf.info/0x15/session.html?Rethinking-Zero-Copy-Networking-with-MAIO>, accessed 2021-08-19.
- [225] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, “The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3477132.3483569. ISBN 9781450387095 p. 195–211. [Online]. Available: <https://doi.org/10.1145/3477132.3483569>



- [226] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USA: USENIX Association, 2014. ISBN 9781931971096 p. 489–502.
- [227] Tencent Cloud, "F-Stack," 2021, <http://www.f-stack.org/>, accessed 2021-08-19.
- [228] FD.io, "Transport Layer Development Kit (TLDK)," 2021, <https://github.com/FDio/tldk>, accessed 2021-08-19.
- [229] Y. Gui, C. Bhure, M. Hughes, and F. Saqib, "A Delay-Based Machine Learning Model for DMA Attack Mitigation," *Cryptography*, vol. 5, no. 3, 2021. doi: 10.3390/cryptography5030018. [Online]. Available: <https://www.mdpi.com/2410-387X/5/3/18>
- [230] A. Lavrov and D. Wentzlaff, "HyperTRIO: Hyper-Tenant Translation of I/O Addresses," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020. doi: 10.1109/ISCA45697.2020.00048. ISBN 9781728146614 p. 487–500. [Online]. Available: <https://doi.org/10.1109/ISCA45697.2020.00048>
- [231] K. Tian, Y. Zhang, L. Kang, Y. Zhao, and Y. Dong, "coIOMMU: A virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020. ISBN 978-1-939133-14-4 pp. 479–492. [Online]. Available: <https://www.usenix.org/conference/atc20/presentation/tian>
- [232] N. Amit, M. Ben-Yehuda, I. Research, D. Tsafir, and A. Schuster, "vIOMMU: Efficient IOMMU Emulation," in *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. Portland, OR: USENIX Association, Jun. 2011. [Online]. Available: <https://www.usenix.org/conference/usenixatc11/viommu-efficient-iommu-emulation>
- [233] P. Willmann, S. Rixner, and A. L. Cox, "Protection Strategies for Direct Access to Virtualized I/O Devices," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. USA: USENIX Association, 2008, p. 15–28.
- [234] J. T. Lim and J. Nieh, "Optimizing Nested Virtualization Performance Using Direct Virtual Hardware," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and*

- Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3373376.3378467. ISBN 9781450371025 p. 557–574. [Online]. Available: <https://doi.org/10.1145/3373376.3378467>
- [235] Y. Hao, Z. Fang, G. Reinman, and J. Cong, “Supporting Address Translation for Accelerator-Centric Architectures,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017. doi: 10.1109/HPCA.2017.19 pp. 37–48.
- [236] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafir, “Page Fault Support for Network Controllers,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3037697.3037710. ISBN 9781450344654 p. 449–466. [Online]. Available: <https://doi.org/10.1145/3037697.3037710>
- [237] S. Agarwal, R. Agarwal, B. Montazeri, M. Moshref, K. Elmeleegy, L. Rizzo, G. Kumar, S. Ratnasamy, D. Culler, and A. Vahdat, “Understanding Host Interconnect Congestion,” in *Proceedings of the Twenty-First ACM Workshop on Hot Topics in Networks*, ser. HotNets '22. New York, NY, USA: Association for Computing Machinery, 2022.
- [238] S. Thomas, G. M. Voelker, and G. Porter, “CacheCloud: Towards speed-of-light datacenter communication,” in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, Jul. 2018. [Online]. Available: <https://www.usenix.org/conference/hotcloud18/presentation/thomas>
- [239] S. Mittal, “A Survey of Techniques for Cache Partitioning in Multicore Processors,” *ACM Comput. Surv.*, vol. 50, no. 2, may 2017. doi: 10.1145/3062394. [Online]. Available: <https://doi.org/10.1145/3062394>
- [240] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, “KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018. doi: 10.1109/HPCA.2018.00019 pp. 104–117.
- [241] J. Park, S. Park, and W. Baek, “CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware

- Workload Consolidation on Commodity Servers,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3302424.3303963. ISBN 9781450362818. [Online]. Available: <https://doi.org/10.1145/3302424.3303963>
- [242] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, “Contention-Aware Performance Prediction For Virtualized Network Functions,” 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*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3387514.3405868. ISBN 9781450379557 p. 270–282. [Online]. Available: <https://doi.org/10.1145/3387514.3405868>
- [243] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, “Don’t Forget the I/O When Allocating Your LLC,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021. doi: 10.1109/ISCA52012.2021.00018 pp. 112–125.
- [244] Y. Yuan, M. Alian, Y. Wang, I. Kurakin, R. Wang, C. Tai, and N. S. Kim, “IOCA: High-Speed I/O-Aware LLC Management for Network-Centric Multi-Tenant Platform,” 2020. [Online]. Available: <https://arxiv.org/abs/2007.04552>
- [245] K. Wang, J. Liu, and F. Chen, “Put an Elephant into a Fridge: Optimizing Cache Efficiency for in-Memory Key-Value Stores,” *Proc. VLDB Endow.*, vol. 13, no. 9, p. 1540–1554, may 2020. doi: 10.14778/3397230.3397247. [Online]. Available: <https://doi.org/10.14778/3397230.3397247>
- [246] Q. Cai, M. Vuppapapati, J. Hwang, C. Kozyrakis, and R. Agarwal, “Towards  $\mu$ s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3544216.3544230. ISBN 9781450394208 p. 767–779. [Online]. Available: <https://doi.org/10.1145/3544216.3544230>
- [247] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, “Creating Complex Network Services with eBPF: Experience and Lessons Learned,” in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018. doi: 10.1109/HPSR.2018.8850758 pp. 1–8.

- [248] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi, “Domain Specific Run Time Optimization for Software Data Planes,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3503222.3507769. ISBN 9781450392051 p. 1148–1164. [Online]. Available: <https://doi.org/10.1145/3503222.3507769>
- [249] P. Enberg, A. Rao, and S. Tarkoma, “Partition-Aware Packet Steering Using XDP and EBPF for Improving Application-Level Parallelism,” in *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ser. ENCP ’19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3359993.3366766. ISBN 9781450370004 p. 27–33. [Online]. Available: <https://doi.org/10.1145/3359993.3366766>
- [250] P. Zheng, A. Narayanan, and Z.-L. Zhang, “A Closer Look at NFV Execution Models,” in *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, ser. APNet ’19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3343180.3343188. ISBN 9781450376358 p. 85–91. [Online]. Available: <https://doi.org/10.1145/3343180.3343188>
- [251] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea, “Automated Verification of Network Function Binaries,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. ISBN 978-1-939133-27-4 pp. 585–600. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/pirelli>
- [252] R. Iyer, K. Argyraki, and G. Candea, “Performance Interfaces for Network Functions,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. ISBN 978-1-939133-27-4 pp. 567–584. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/iyer>
- [253] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, “Performance Contracts for Software Network Functions,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019. ISBN 978-1-931971-49-2 pp. 517–530. [Online]. Available: <https://www.usenix.org/conference/nsdi19/presentation/iyer>

- [254] G. Wan, F. Gong, T. Barbette, and Z. Durumeric, “Retina: Analyzing 100GbE Traffic on Commodity Hardware,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM ’22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3544216.3544227. ISBN 9781450394208 p. 530–544. [Online]. Available: <https://doi.org/10.1145/3544216.3544227>
- [255] M. Girondi, M. Chiesa, and T. Barbette, “High-speed Connection Tracking in Modern Servers,” in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, 2021. doi: 10.1109/HPSR52026.2021.9481841 pp. 1–8.
- [256] T. Barbette, C. Soldani, and L. Mathy, “Combined Stateful Classification and Session Splicing for High-Speed NFV Service Chaining,” *IEEE/ACM Transactions on Networking*, vol. 29, no. 6, pp. 2560–2573, 2021. doi: 10.1109/TNET.2021.3099240
- [257] M. Gallo and R. Laufer, “ClickNF: a Modular Stack for Custom Network Functions,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018. ISBN 978-1-939133-01-4 pp. 745–757. [Online]. Available: <https://www.usenix.org/conference/atc18/presentation/gallo>
- [258] P. Kennedy, “NVIDIA NVLink4 NVSwitch at Hot Chips 34,” ServeTheHome, 2022, Accessed 2022-08-23. [Online]. Available: <https://www.servethehome.com/nvidia-nvlink4-nvswitch-at-hot-chips-34/>
- [259] M. Wang, M. Xu, and J. Wu, “Understanding I/O Direct Cache Access Performance for End Host Networking,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, feb 2022. doi: 10.1145/3508042. [Online]. Available: <https://doi.org/10.1145/3508042>
- [260] Y. Yuan, Y. Wang, R. Wang, and J. Huang, “HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA: ACM, 2019. doi: 10.1145/3307650.3322272. ISBN 978-1-4503-6669-4 pp. 601–614. [Online]. Available: <http://doi.acm.org/10.1145/3307650.3322272>
- [261] M. Alian, S. Agarwal, J. Shin, N. Patel, Y. Yuan, D. Kim, R. Wang, and N. S. Kim, “IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022. doi: 10.1109/MICRO56248.2022.00042 pp. 480–493.

- [262] S. Ibanez, A. Mallery, 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)*. USENIX Association, Jul. 2021. ISBN 978-1-939133-22-9 pp. 239–256. [Online]. Available: <https://www.usenix.org/conference/osdi21/presentation/ibanez>
- [263] Y. Yuan, J. Huang, Y. Sun, T. Wang, J. Nelson, D. R. K. Ports, Y. Wang, R. Wang, C. Tai, and N. S. Kim, “ORCA: A Network and Architecture Co-design for Offloading us-scale Datacenter Applications,” 2022. [Online]. Available: <https://arxiv.org/abs/2203.08906>
- [264] G. Zhang and D. Sanchez, “Leveraging Caches to Accelerate Hash Tables and Memoization,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3352460.3358272. ISBN 9781450369381 p. 440–452. [Online]. Available: <https://doi.org/10.1145/3352460.3358272>
- [265] 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*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3297858.3304070. ISBN 9781450362405 p. 35–48. [Online]. Available: <https://doi.org/10.1145/3297858.3304070>
- [266] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, “NetCache: Balancing Key-Value Stores with Fast In-Network Caching,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3132747.3132764. ISBN 9781450350853 p. 121–136. [Online]. Available: <https://doi.org/10.1145/3132747.3132764>
- [267] Y. Shi, J. Fei, M. Wenz, and C. Zhang, “KVSwitch: An In-network Load Balancer for Key-Value Stores,” in *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019. doi: 10.1109/ISCC47284.2019.8969745 pp. 1–7.
- [268] Y. Shi, J. Fei, M. Wen, and C. Zhang, “Balancing Distributed Key-Value Stores with Efficient In-Network Redirecting,” *Electronics*, vol. 8, no. 9, 2019. doi: 10.3390/electronics8091008. [Online]. Available: <https://www.mdpi.com/2079-9292/8/9/1008>

- [269] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. Seltzer, “Parking Packet Payload with P4,” in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3386367.3431295. ISBN 9781450379489 p. 274–281. [Online]. Available: <https://doi.org/10.1145/3386367.3431295>
- [270] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, “A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/scazzariello>
- [271] A. Sarma, H. Seyedroudbari, H. Gupta, U. Ramachandran, and A. Daglis, “NFSlicer: Data Movement Optimization for Shallow Network Functions,” 2022. [Online]. Available: <https://arxiv.org/abs/2203.02585>
- [272] B. Pismenny, L. Liss, A. Morrison, and D. Tsafirir, *The Benefits of General-Purpose on-NIC Memory*. New York, NY, USA: Association for Computing Machinery, 2022, p. 1130–1147. ISBN 9781450392051. [Online]. Available: <https://doi.org/10.1145/3503222.3507711>
- [273] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano, “FlowBlaze: Stateful Packet Processing in Hardware,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019. ISBN 978-1-931971-49-2 pp. 531–548. [Online]. Available: <https://www.usenix.org/conference/nsdi19/presentation/pontarelli>
- [274] 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*, ser. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3132747.3132756. ISBN 9781450350853 p. 137–152. [Online]. Available: <https://doi.org/10.1145/3132747.3132756>
- [275] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, “AccelTCP: Accelerating Network Applications with Stateful TCP Offloading ,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

- Santa Clara, CA: USENIX Association, Feb. 2020. ISBN 978-1-939133-13-7 pp. 77–92. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/moon>
- [276] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, “Autonomous NIC Offloads,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3445814.3446732. ISBN 9781450383172 p. 18–35. [Online]. Available: <https://doi.org/10.1145/3445814.3446732>
- [277] I. Smolyar, A. Markuze, B. Pismenny, H. Eran, G. Zellweger, A. Bolen, L. Liss, A. Morrison, and D. Tsafrir, “IOctopus: Outsmarting Nonuniform DMA,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3373376.3378509. ISBN 9781450371025 p. 101–115. [Online]. Available: <https://doi.org/10.1145/3373376.3378509>
- [278] T. L. Foundation, “Meta, Google, Isovalent, Microsoft and Netflix Launch eBPF Foundation as Part of the Linux Foundation,” The Linux Foundation, 2021, Accessed 2022-08-01. [Online]. Available: <https://www.linuxfoundation.org/press-release/facebook-google-isovalent-microsoft-and-netflix-launch-ebpf-foundation-as-part-of-the-linux-foundation/>
- [279] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, “hXDP: Efficient software packet processing on FPGA NICs,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. ISBN 978-1-939133-19-9 pp. 973–990. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/brunella>
- [280] M. Bonola, G. Belocchi, A. Tulumello, M. S. Brunella, G. Siracusano, G. Bianchi, and R. Bifulco, “Faster Software Packet Processing on FPGA NICs with eBPF Program Warping,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022. ISBN 978-1-939133-29-58 pp. 987–1004. [Online]. Available: <https://www.usenix.org/conference/atc22/presentation/bonola>



- [281] J. Ousterhout, “We Need a Replacement for TCP in the Datacenter,” 2022, <https://web.stanford.edu/~ouster/cgi-bin/papers/replaceTcp.pdf>, accessed 2022-07-28.
- [282] —, “It’s Time to Replace TCP in the Datacenter,” 2022. [Online]. Available: <https://arxiv.org/abs/2210.00714>
- [283] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, “Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3230543.3230564. ISBN 9781450355674 p. 221–235. [Online]. Available: <https://doi.org/10.1145/3230543.3230564>
- [284] J. Ousterhout, “A Linux Kernel Implementation of the Homa Transport Protocol,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021. ISBN 978-1-939133-23-6 pp. 99–115. [Online]. Available: <https://www.usenix.org/conference/atc21/presentation/ousterhout>
- [285] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3098822.3098842. ISBN 9781450346535 p. 183–196. [Online]. Available: <https://doi.org/10.1145/3098822.3098842>
- [286] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette, “A High-Speed QUIC Implementation,” in *Proceedings of the 3rd International CoNEXT Student Workshop*, ser. CoNEXT-SW ’22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3565477.3569154. ISBN 9781450399371 p. 20–22. [Online]. Available: <https://doi.org/10.1145/3565477.3569154>
- [287] M. Bishop, “HTTP/3,” RFC 9114, Jun. 2022. [Online]. Available: <https://www.rfc-editor.org/info/rfc9114>
- [288] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, “Making QUIC Quicker With NIC Offload,” in *Proceedings of the Workshop on*

- the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3405796.3405827. ISBN 9781450380478 p. 21–27. [Online]. Available: <https://doi.org/10.1145/3405796.3405827>
- [289] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3098822.3098842. ISBN 9781450346535 p. 183–196. [Online]. Available: <https://doi.org/10.1145/3098822.3098842>
- [290] M. Faltelli, G. Belocchi, F. Quaglia, S. Pontarelli, and G. Bianchi, *Metronome: Adaptive and Precise Intermittent Packet Retrieval in DPDK*. New York, NY, USA: Association for Computing Machinery, 2020, p. 406–420. ISBN 9781450379489. [Online]. Available: <https://doi.org/10.1145/3386367.3432730>
- [291] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. doi: 10.1145/2024716.2024718. [Online]. Available: <https://doi.acm.org/10.1145/2024716.2024718>
- [292] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, B. Thomsen, K. Shi, and H. Williams, “Sirius: A Flat Datacenter Network with Nanosecond Optical Switching,” in *SIGCOMM*. ACM, August 2020. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/sirius-a-flat-datacenter-network-with-nanosecond-optical-switching/>
- [293] D. Gibson, H. Hariharan, E. Lance, M. McLaren, B. Montazeri, A. Singh, S. Wang, H. M. G. Wassel, Z. Wu, S. Yoo, R. Balasubramanian, P. Chandra, M. Cutforth, P. Cuy, D. Decotigny, R. Gautam, A. Iriza, M. M. K. Martin, R. Roy, Z. Shen, M. Tan, Y. Tang, M. Wong-Chan, J. Zbiciak, and A. Vahdat, “Aquila: A unified, low-latency fabric for datacenter networks,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association,

- Apr. 2022. ISBN 978-1-939133-27-4 pp. 1249–1266. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/gibson>
- [294] L. Poutievski, O. Mashayekhi, J. Ong, A. Singh, M. Tariq, R. Wang, J. Zhang, V. Beauregard, P. Conner, S. Gribble, R. Kapoor, S. Kratzer, N. Li, H. Liu, K. Nagaraj, J. Ornstein, S. Sawhney, R. Urata, L. Vicisano, K. Yasumura, S. Zhang, J. Zhou, and A. Vahdat, “Jupiter Evolving: Transforming Google’s Datacenter Network via Optical Circuit Switches and Software-Defined Networking,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM ’22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3544216.3544265. ISBN 9781450394208 p. 66–85. [Online]. Available: <https://doi.org/10.1145/3544216.3544265>
- [295] R. Urata, H. Liu, K. Yasumura, E. Mao, J. Berger, X. Zhou, C. Lam, R. Bannon, D. Hutchinson, D. Nelson, L. Poutievski, A. Singh, J. Ong, and A. Vahdat, “Mission Apollo: Landing Optical Circuit Switching at Datacenter Scale,” 2022. [Online]. Available: <https://arxiv.org/abs/2208.10041>
- [296] F. Douglass, S. Robertson, E. v. d. Berg, J. Micallef, M. Pucci, A. Aiken, K. Bergman, M. Hattink, and M. Seok, “FLEET—Fast Lanes for Expedited Execution at 10 Terabits: Program Overview,” *IEEE Internet Computing*, vol. 25, no. 3, pp. 79–87, 2021. doi: 10.1109/MIC.2021.3075326
- [297] C. Branner-Augmon, N. Galstyan, S. Kumar, E. Amaro, A. Ousterhout, A. Panda, S. Ratnasamy, and S. Shenker, “3PO: Programmed Far-Memory Prefetching for Oblivious Applications,” 2022. [Online]. Available: <https://arxiv.org/abs/2207.07688>
- [298] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, “Can Far Memory Improve Job Throughput?” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3342195.3387522. ISBN 9781450368827. [Online]. Available: <https://doi.org/10.1145/3342195.3387522>
- [299] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat, “Carbink: Fault-Tolerant Far Memory,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association,

- Jul. 2022. ISBN 978-1-939133-28-1 pp. 55–71. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/zhou-yang>
- [300] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmamghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3079856.3080246. ISBN 9781450348928 p. 1–12. [Online]. Available: <https://doi.org/10.1145/3079856.3080246>
- [301] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. P. Jouppi, and D. Patterson, “Google’s Training Chips Revealed: TPUV2 and TPUV3,” in *2020 IEEE Hot Chips 32 Symposium (HCS)*, 2020. doi: 10.1109/HCS49909.2020.9220735 pp. 1–70.
- [302] A. Labs, “Ayar Labs’ optical I/O unleashes the potential of next-generation AI compute architectures,” Ayar Labs, 2022, Accessed 2022-07-27. [Online]. Available: <https://ayarlabs.com/ayar-labs-to-accelerate-development-and-application-of-optical-interconnects-in-artificial-intelligence-machine-learning-architectures-with-nvidia/>
- [303] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi, “SiP-ML: High-Bandwidth Optical Network Interconnects for Machine Learning Training,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM ’21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3452296.3472900. ISBN 9781450383837 p. 657–675. [Online]. Available: <https://doi.org/10.1145/3452296.3472900>
- [304] S. Arslan, S. Ibanez, A. Mallery, C. Kim, and N. McKeown, “NanoTransport: A Low-Latency, Programmable Transport Layer for

- NICs,” in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, ser. SOSR '21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3482898.3483365. ISBN 9781450390842 p. 13–26. [Online]. Available: <https://doi.org/10.1145/3482898.3483365>
- [305] S. Ibanez, M. Shahbaz, and N. McKeown, “The Case for a Network Fast Path to the CPU,” in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3365609.3365851. ISBN 9781450370202 p. 52–59. [Online]. Available: <https://doi.org/10.1145/3365609.3365851>
- [306] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, “Scaling Distributed Machine Learning with In-Network Aggregation,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021. ISBN 978-1-939133-21-2 pp. 785–808. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/sapio>
- [307] R. Viswanathan, A. Balasubramanian, and A. Akella, “Network-Accelerated Distributed Machine Learning for Multi-Tenant Settings,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3419111.3421296. ISBN 9781450381376 p. 447–461. [Online]. Available: <https://doi.org/10.1145/3419111.3421296>
- [308] J. Langlet, “Towards Machine Learning Inference in the Data Plane,” 2019.
- [309] J. Langlet, A. Kassler, and D. Bhamare, “Towards Neural Network Inference on Programmable Switches,” in *2nd P4 Workshop in Europe (EUROP4)*, Cambridge UK, September 23, 2019 (peer-reviewed but no proceedings track, Poster)., 2019.
- [310] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, “Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks,” 2022. [Online]. Available: <https://arxiv.org/abs/2206.05592>
- [311] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, “Taurus: A Data Plane Architecture for per-Packet ML,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022.

- New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3503222.3507726. ISBN 9781450392051 p. 1099–1114. [Online]. Available: <https://doi.org/10.1145/3503222.3507726>
- [312] K. A. Simpson and D. P. Pezaros, “Revisiting the Classics: Online RL in the Programmable Dataplane,” in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022. doi: 10.1109/NOMS54207.2022.9789930 pp. 1–10.
- [313] T. Auld, A. W. Moore, and S. F. Gull, “Bayesian Neural Networks for Internet Traffic Classification,” *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 223–239, 2007. doi: 10.1109/TNN.2006.883010
- [314] M. Lotfollahi, R. S. H. Zade, M. J. Siavoshani, and M. Saberian, “Deep Packet: A Novel Approach For Encrypted Traffic Classification Using Deep Learning,” *CoRR*, vol. abs/1709.02656, 2017. [Online]. Available: <http://arxiv.org/abs/1709.02656>
- [315] A. Rashelbach, O. Rottenstreich, and M. Silberstein, “Scaling Open vSwitch with a Computational Cache,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. ISBN 978-1-939133-27-4 pp. 1359–1374. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/rashelbach>
- [316] E. Hodo, X. Bellekens, A. Hamilton, P.-L. Dubouilh, E. Iorkyase, C. Tachtatzis, and R. Atkinson, “Threat analysis of IoT networks using artificial neural network intrusion detection system,” in *2016 International Symposium on Networks, Computers and Communications (ISNCC)*, 2016. doi: 10.1109/ISNCC.2016.7746067 pp. 1–6.
- [317] A. Shenfield, D. Day, and A. Ayesh, “Intelligent intrusion detection systems using artificial neural networks,” *ICT Express*, vol. 4, no. 2, pp. 95–99, 2018. doi: <https://doi.org/10.1016/j.icte.2018.04.003> SI on Artificial Intelligence and Machine Learning. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2405959518300493>
- [318] C. Yin, Y. Zhu, J. Fei, and X. He, “A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks,” *IEEE Access*, vol. 5, pp. 21 954–21 961, 2017. doi: 10.1109/ACCESS.2017.2762418
- [319] 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 Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014. doi: 10.1145/2656877.2656890. [Online]. Available: <https://doi.org/10.1145/2656877.2656890>
- [320] N. S. Inc., “P4 Data Plane Programming for Server-Based Networking Applications,” Netronome, 2017, Accessed 2022-08-23. [Online]. Available: [https://www.netronome.com/media/redactor\\_files/WP\\_P4\\_Data\\_Plane\\_Programming.pdf](https://www.netronome.com/media/redactor_files/WP_P4_Data_Plane_Programming.pdf)
- [321] NVIDIA, “NVIDIA DOCA Software Framework,” NVIDIA, 2020, Accessed 2022-09-01. [Online]. Available: <https://developer.nvidia.com/networking/doca>
- [322] S. Laki, “T4P4S: When P4 meets DPDK,” September 2017, Accessed 2022-09-01. [Online]. Available: [https://www.dpdk.org/wp-content/uploads/sites/35/2017/09/DPDK-Userspace2017-Day2-12-SANDOR\\_LAKI-T4P4S.pdf](https://www.dpdk.org/wp-content/uploads/sites/35/2017/09/DPDK-Userspace2017-Day2-12-SANDOR_LAKI-T4P4S.pdf)
- [323] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, “A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research,” *CoRR*, vol. abs/2101.10632, 2021. [Online]. Available: <https://arxiv.org/abs/2101.10632>
- [324] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, “Flightplan: Dataplane Disaggregation and Placement for P4 Programs,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021. ISBN 978-1-939133-21-2 pp. 571–592. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/sultana>
- [325] A. Mohammadkhan, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, and L. N. Bhuyan, “P4NFV: P4 Enabled NFV Systems with SmartNICs,” in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2019. doi: 10.1109/NFV-SDN47374.2019.9040000 pp. 1–7.
- [326] R. Doenges, M. T. Arashloo, S. Bautista, A. Chang, N. Ni, S. Parkinson, R. Peterson, A. Solko-Breslin, A. Xu, and N. Foster, “Petr4: Formal Foundations for P4 Data Planes,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. doi: 10.1145/3434322. [Online]. Available: <https://doi.org/10.1145/3434322>

- [327] E. H. Campbell, W. T. Hallahan, P. Srikumar, C. Cascone, J. Liu, V. Ramamurthy, H. Hojjat, R. Piskac, R. Soulé, and N. Foster, “Avenir: Managing Data Plane Diversity with Control Plane Synthesis,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021. ISBN 978-1-939133-21-2 pp. 133–153. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/campbell>
- [328] K. D. Albab, J. DiLorenzo, S. Heule, A. Kheradmand, S. Smolka, K. Weitz, M. Timarzi, J. Gao, and M. Yu, “SwitchV: Automated SDN Switch Validation with P4 Models,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM ’22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3544216.3544220. ISBN 9781450394208 p. 365–379. [Online]. Available: <https://doi.org/10.1145/3544216.3544220>
- [329] PCI-SIG, “Address Translation Services Revision 1.1 ,” January 2009, [https://composter.com.ua/documents/ats\\_r1.1\\_26Jan09.pdf](https://composter.com.ua/documents/ats_r1.1_26Jan09.pdf), accessed 2021-08-10.
- [330] M. Krause, M. Hummel, and D. Wooten, “Address Translation Services,” June 2006, [https://composter.com.ua/documents/Address\\_Translation\\_Services.pdf](https://composter.com.ua/documents/Address_Translation_Services.pdf), accessed 2021-08-10.
- [331] L. Yao and J. Hu, “Revise 4K Pages Performance Impact For DPDK Applications,” 2018, [https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/LeiJiayu\\_Revise-4K-Pages-Performance-Impact-For-DPDK-Applications.pdf](https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/LeiJiayu_Revise-4K-Pages-Performance-Impact-For-DPDK-Applications.pdf), accessed 2021-08-11.





# Appendix A

## Network Function Configurations

This appendix explains the details of NF configurations used in Section 3.3.

### A.1 Simple Forwarder

The simple forwarder receives packets from DPDK, rewrites the Ethernet MAC addresses via `EtherRewrite` element, and transmits them.

### A.2 Router

The router configuration is the standard Click router, compliant with IP routing standards [109]. The processing graph produced by this configuration exhibits multiple, more complex paths. The whole IP header is loaded in memory, as the routing element (with only *one* rule per port) does a lookup for each destination IP address. We do not consider the impact of more rules, as they could be potentially offloaded to the NIC [16].

### A.3 NAT, IDS, and VLAN

All these configurations are optional supplements for the two previous configurations (*i.e.*, the simple forwarder and the router). The NAT rewrites source IP addresses of outgoing packets. The NAT configuration is stateful and it uses the DPDK Cuckoo hash table, resulting in more lookups and higher memory

usage. The IDS checks the correctness of TCP, UDP, and ICMP headers, except for the checksum that can be verified in hardware. The VLAN supplement eventually encapsulates the packet in a VLAN header.

## A.4 WorkPackage

This configuration uses `WorkPackage` element that is a purely synthetic element built for microbenchmarking [139]. It generates  $N$  random accesses to a static memory of  $S$  MiB. Additionally, it generates  $W$  pseudo-random numbers to artificially emulate more CPU-bound workloads. The `WorkPackage` element enables us to study the impact of our optimizations on more memory- & compute-intensive configurations. The `WorkPackage` is also supplementary to the base configurations. In our tests, we use it along with the forwarding configuration.

# Appendix B

## IOMMU – Supplementary Information

This appendix provides supplementary information for Chapter 4.

### B.1 IOTLB Implementation on Different Architectures

IOTLB features vary across different processors, with little if any public documentation. The available performance counters and public data sheets suggest that Intel processors use a single IOTLB for all levels of mapping (*e.g.*, first-level, second-level, nested, and pass-through mappings) [195], whereas AMD processors use two distinct IOTLBs for caching Page Directory Entry (PDE) and Page Table Entry (PTE) [196, 201]. Additionally, Emmerich *et al.*, [200] and Neugebauer *et al.*, [49] have speculated (based on their experiments) that the number of IOTLB entries in Intel processors is 64. Furthermore, some PCIe devices support Address Translation Service (ATS) [329, 330] that enables the devices to cache the address translation in a local cache to minimize latency and provide a scalable distributed caching solution for IOMMU. Based on our discussions with NVIDIA/Mellanox support, since Intel IceLake processors only caching of 1-GiB IOTLB entries is supported via ATS (*i.e.*, this caching is not helpful for 4-KiB entries).

## B.2 Testbed Illustration and Additional Results

Figure B.1 illustrates our testbed. We use the main slot of a Socket Direct ConnectX-6 for 200-Gbps experiments and a normal ConnectX-6 NIC with custom Dell firmware for 100-Gbps experiments.

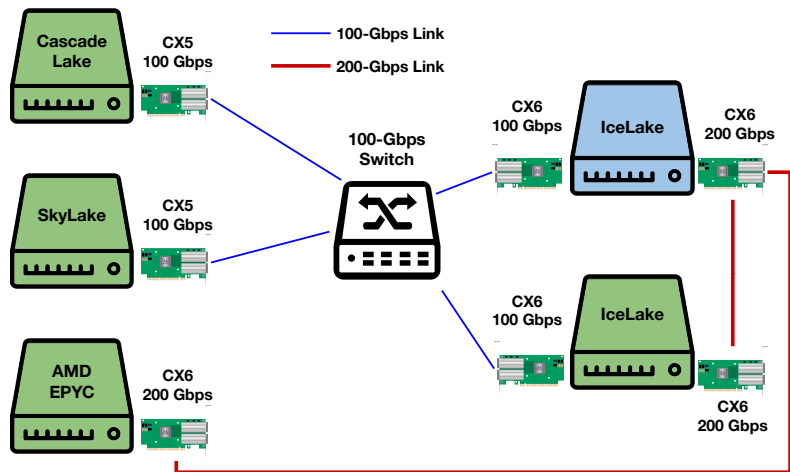


Figure B.1: Our testbed. Green servers represent DuTs. CX5 and CX6 stand for ConnectX-5 and ConnectX-6, respectively.

Figure B.2 shows the throughput for a fine-grained sweep of MTU values when the NIC uses 1024 RX descriptors per queue (*i.e.*,  $32 \times 1024$ ). Looking at the throughput values, we notice jumps at page splitting points. The `m1x5` driver splits pages based on the H/W MTU size that is 22 B larger than the value specified by the software (*e.g.*, via `ifconfig`). The extra 22 B ensures that the H/W buffer has enough space for the 14-B Ethernet header, 4-B VLAN header, and 4-B Ethernet Frame Check Sequence (FCS). The `m1x5` driver uses 512-B buffers for small H/W MTUs, but switches to 1024-B, 2048-B, and 4096-B buffers when the H/W MTU size exceeds 128, 640, and 1664 bytes. The driver uses a scatter-gather technique (*e.g.*, it uses multiple 256-B buffers) for “Jumbo” frames, *i.e.*, H/W MTU sizes  $> 3712$  B.

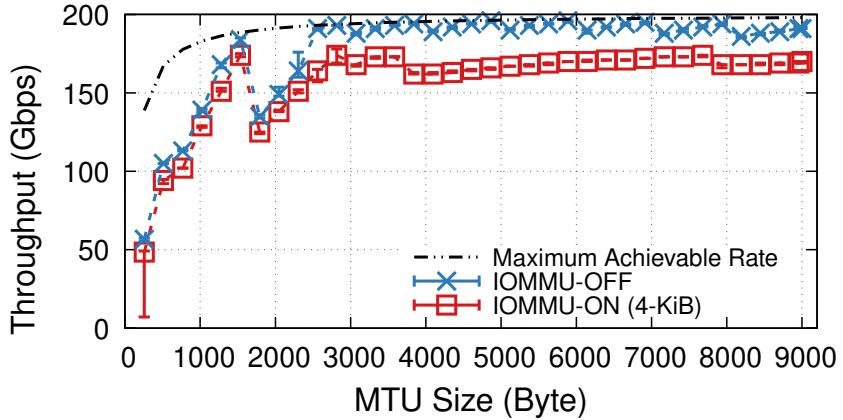


Figure B.2: IOMMU imposes performance overheads at all MTU sizes, but with a larger absolute throughput drop for MTU sizes larger than  $\sim 3000$  B.

### B.3 Page (De)Allocation via Page Pool API

Page Pool provides an API to device drivers to (de)allocate pages in an efficient, and preferably lockless way (see Figure B.3 for details in Linux kernel v5.15); the driver then splits each page into single or multiple buffers based on a given MTU size, see Section 4.2.2. A page pool is composed of two main data structures: (i) a fixed-size lockless array/LIFO (*aka* cache) and (ii) a variable-sized ring implemented via a `ptr_ring` data structure that is essentially a limited-size FIFO with spinlocks and that facilitates synchronization by using separate locks for consumers & producers. By default, the cache can contain up to 128 pages, and the size of the ring is determined based on the number of RX descriptors and MTU size. The purpose of the page pool is to (i) efficiently allocate pages from the cache without locking and (ii) use the ring to recycle returned pages. To avoid synchronization problems, each page pool should be connected to only *one* RX queue, thus it is protected by NAPI scheduling. The Page Pool API is mainly used to allocate memory at the granularity of a page; however, recent Linux kernels support page fragments that make it possible to allocate an arbitrary-sized memory chunk from an order- $n$  page, *i.e.*,  $2^n$  contiguous 4-KiB pages.

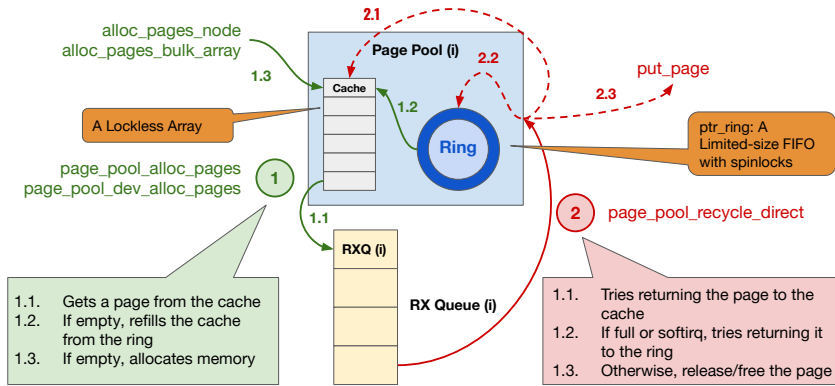


Figure B.3: Page Pool overview.

### Details of the steps in Figure B.3:

① **Allocating a page.** When a driver asks for a page: (1.1) the page pool initially checks the lockless cache; (1.2) if empty, it tries to refill the cache from the pages recycled in the ring; (1.3) if not possible due to software interrupts (softirq) or unavailability, it allocates page(s) and refills the cache. Recent Linux kernels perform bulk allocation (*e.g.*, 64 pages at a time) to amortize the allocation cost.

② **Returning a page.** When a driver returns a page: (2.1) the page pool attempts to recycle the page into the cache; (2.2) if not possible, it tries returning it to the ring; (2.3) if unsuccessful, it releases/frees the page. Since the Page Pool API is optimized for XDP and AF\_XDP socket (XSK), *i.e.*, an eBPF data path where each page is only used by one buffer, Steps 2.1 & 2.2 can only be performed if (i) a page has only a *single* reference (*i.e.*, `page_ref_count(page) == 1`) and (ii) a page is *not* allocated from the emergency pfmalloc reserves. The former causes a page pool to continually allocate new pages for drivers operating in a non-XDP mode that splits each page into multiple fragments and uses page references for bookkeeping/recycling. Unfortunately, continuous page allocation disables the driver from re-using a fixed set of pages, causing low page locality. We refer to this as the *leaky page pool* problem.

## B.4 Current Implementation of HPA

Given the previously enumerated design choices, it is clear that a hugepage-aware memory allocator for a network device driver should find a trade-off between allocation overhead & memory stranding while managing pages efficiently &

ensuring IOVA locality. However, the Linux kernel v5.15 imposes two main limitations on achieving a perfect design: ① Unfortunately, some drivers may perform continuous allocation due to the inability to recycle multi-referenced pages into their page pools. Therefore, pre-allocating a large pool of *statically* mapped memory for the network device driver will be insufficient, as the pool will eventually be depleted. ② Introducing a new memory allocator requires augmenting the page data structure, which is undesirable due to the aforementioned reasons. While it is possible to add a new data structure (*e.g.*, a hash table) to check whether a page belongs to the allocator’s pool or not; this will be more expensive than augmenting the page data structure.

**Implementation.** To partially address these two limitations, we decided to extend the current implementation of the Page Pool API [203], as it already contains a reference in the page data structure. In addition, extending an existing API makes it easier to integrate our solution into the Linux kernel, as it requires a smaller amount of changes. To provide a trade-off between run-time allocation overhead vs. memory stranding, we add a backup ring (`ptr_ring`) data structure to the Page Pool API to store a set of hugepage-backed 4-KiB pages; we chose the `ptr_ring` data structure due to its synchronization efficiencies.

We (i) allocate 2-MiB hugepages (via the buddy page allocator\*), (ii) map them to IOMMU, (iii) keep the mappings in a doubly linked-list to be able to unmap the pages at the page pool’s destruction, and (iv) store the sub-4-KiB pages in the backup ring. While it is possible to store the pre-allocated memory in the already existing cache or ring data structures, we consciously decided not to do so to avoid affecting the existing machinery. In our approach, the backup ring is only used to refill the cache when the page pool needs to allocate memory (see step 1.3 in Figure B.3). If the driver is *not* leaky, the backup ring would never be empty (*i.e.*, the pre-allocated memory is sufficient). However, drivers with a *leaky page pool* will eventually deplete the backup ring. We noticed that our NIC could deplete a 1-GiB per-queue memory (*i.e.*,  $512 \times 512$  pages) after  $\sim 100$  seconds when operating at 200 Gbps. Unfortunately, constantly refilling the backup ring with pages could become costly, as it requires using spinlocks. Therefore, when the backup ring is known to be depleted, we switch to an alternative mode where we allocate a new 2-MiB hugepage and *directly* inject its  $512 \times 4$ -KiB pages into the cache data structure. To do so, we increase the maximum size of the cache data structure to 1024 to potentially reduce the frequency of run-time allocations. Note that the page pool returns the *unrecyclable* pages (*i.e.*, individual 4-KiB pages within 2-MiB hugepage) back to the kernel and only saves their IOVA mappings. Therefore, run-time allocations do not over-allocate memory, but they may fragment memory over a long time and make the application more CPU-bound.

---

\*It is currently limited to allocating order-11 (8-MiB) pages.

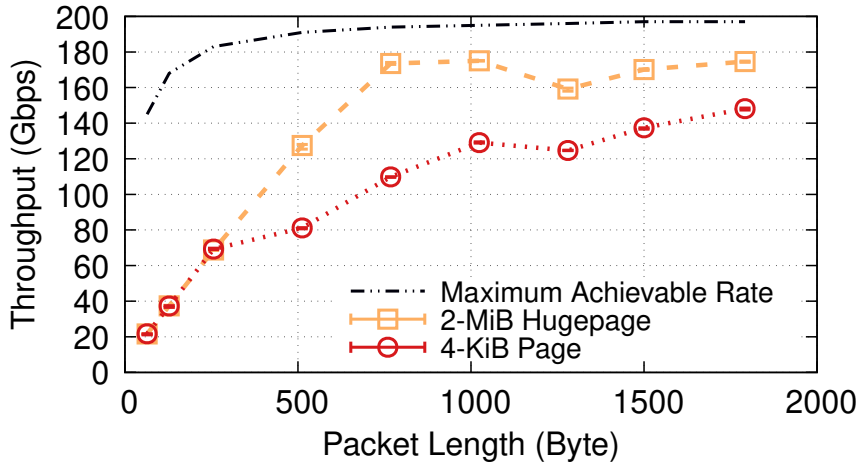


However, as long as the application falls into the IOTLB-bound region, our solution would improve its performance.

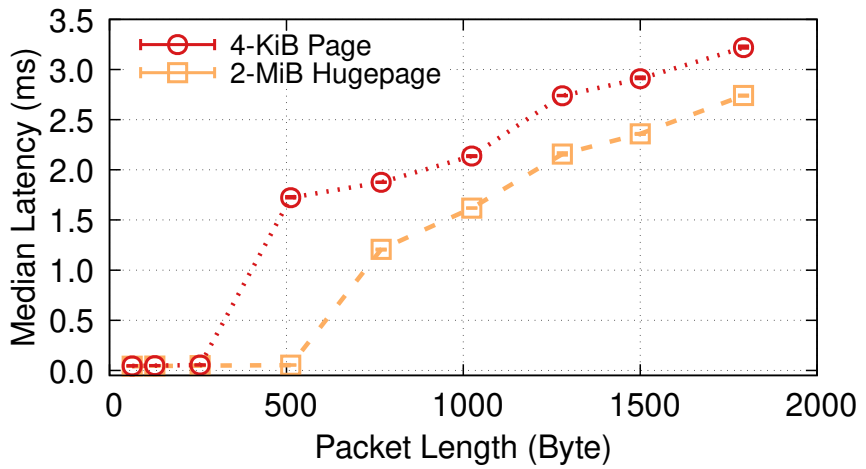
## B.5 IOTLB Overheads in DPDK

For the sake of completeness, this section examines a kernel-bypass framework, DPDK, to examine IOMMU in a high-performance & low-overhead setting. Additionally, using DPDK provides us with greater flexibility in memory management, enabling us to natively compare the impact of different (huge)page sizes (*i.e.*, 4-KiB & 2-MiB) on the performance of IOTLB. Furthermore, since DPDK allocates buffers contiguously (especially when we reserve hugepages at boot time), changing the (huge)page size will primarily show the impact of IOTLB misses. We use a DPDK-based packet-processing framework, FastClick [24], to generate & forward fix-sized packets at line rate. By doing so, we can benefit from all software optimizations shipped within FastClick and achieve the maximum achievable rate with DPDK [45].

Figure B.4 shows the throughput and median latency of a server running an L2 forwarder application that receives fixed-size packets with 16 cores and 1024 RX descriptors per core, mirrors the MAC address of received packets, and sends them back to the packet generator. These results show that kernel-bypass frameworks also experience similar performance degradations when they do not use hugepage IOTLB mappings. Note that the exact numbers may not match the iPerf results, as DPDK is less CPU-bound (*i.e.*, it achieves higher throughput with hugepages) and is less efficient without hugepages (*i.e.*, it achieves lower throughput in 4-KiB mode [331]).



(a) Throughput.



(b) Median latency.

Figure B.4: DPDK also experiences performance degradation with IOMMU when it uses 4-KiB pages.

# For DIVA

```
{
  "Author1": {
    "Last name": "Farshin",
    "First name": "Alireza",
    "Local User Id": "u1sop5xa",
    "E-mail": "farshin@kth.se",
    "ORCID": "0000-0001-5083-4052",
    "organisation": {"L1": "School of Electrical Engineering and Computer Science",
                     "L2": "Computer Science"}
  },
  "Degree": {"Educational program": "Information and Communication Technology"},
  "Title": {
    "Main title": "Realizing Low-Latency Packet Processing on Multi-Hundred-Gigabit-Per-Second Commodity Hardware",
    "Subtitle": "Exploit Caching to Improve Performance",
    "Language": "eng"},
  "Alternative title": {
    "Main title": "Realisering av Pakethantering med Låg Fördörjning på Tillgänglig Hårdvara med Stöd för Flera Hundra Gigabit Per
Sekund",
    "Subtitle": "Utnyttjande av Cacheteknik för att Förbättra Prestanda",
    "Language": "swe"
  },
  "Other information": {
    "Year": "2023",
    "Number of pages": "xxi,178"},
  "Supervisor1": {
    "Last name": "Kostić",
    "First name": "Dejan",
    "Local User Id": "u12eursm",
    "E-mail": "dmk@kth.se",
    "organisation": {"L1": "EECS",
                     "L2": "Computer Science"}
  },
  "Supervisor2": {
    "Last name": "Maguire Jr.",
    "First name": "Gerald Q.",
    "Local User Id": "u1d13i2c",
    "E-mail": "maguire@kth.se",
    "organisation": {"L1": "EECS",
                     "L2": "Computer Science"}
  },
  "Opponent": {
    "Last name": "Silberstein",
    "First name": "Mark",
    "E-mail": "mark@ee.technion.ac.il",
    "Other organisation": "Technion"
  },
  "Presentation": {
    "Date": "2023-03-06 17:00",
    "Language": "eng",
    "Room": "via Zoom and Sal C (Sven-Olof Öhrvik) at Electrum, Kungliga Tekniska Högskolan",
    "Address": "Kistagången 16",
    "City": "Kista"
  },
  "National Subject Categories": "20203, 20206",
}
```

