Architectural Enhancements for Data Transport in Datacenter Systems

by

Hossein Golestani

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2021

Doctoral Committee:

Professor Thomas F. Wenisch, Chair
Associate Professor Robert Dick
Assistant Professor Baris Kasikci
Professor Scott Mahlke

Hossein Golestani

hosseing@umich.edu

ORCID iD: 0000-0001-7293-6965

To my family, the pieces of my heart...

ACKNOWLEDGEMENTS

Well, I have come to the end of a journey full of invaluable experiences and wonderful memories. Over the last five years in my PhD, I learned so many things not just in the matter of computer science but also about many aspects of life. True growth is always accompanied by discomfort and even pain. This was no exception for me during my PhD; I could only survive through the support and love of many amazing people around me, both physically and virtually. Words may not be enough to acknowledge them, but I will do my best.

First, I would like to thank my PhD adviser, Prof. Tom Wenisch. My first encounter with him was in my first semester at Michigan, where he was the instructor of the Computer Architecture course. I was absolutely amazed by how passionately he gave the lectures. Later, I realized he has the same passion and devotion in all aspects of his work, particularly in the research with his students. I started to work with Tom at one of those "in the right place, at the right time" moments. I joined his research group a few months before he took a sabbatical. Before leaving, he suggested a research direction to me, which eventually resulted in the material of this dissertation. Even after he left, he was there all the time to help me not only with my research but also with my career decisions. Tom has been an incredible source of research insight, technical knowledge, and professional advice for me.

I would also like to thank a number of other professional researchers who assisted me during my PhD. Profs. Robert Dick, Baris Kasikci, and Scott Mahlke kindly served in my doctoral committee and provided valuable feedback to my proposal and to the writing of this dissertation. Additionally, I had the chance to work directly with Prof. Scott Mahlke along with Prof. Satish Narayanasamy at the beginning of my PhD. They helped me, an

inexperienced researcher at the time, to learn how to define and approach a research problem and find solutions for it through critical thinking. I also appreciate Drs. Gagan Gupta and Rathijit Sen, who provided great mentorship for me during my two internships at Microsoft. The trust they placed in me helped me build confidence in conducting research, which has been beneficial to me ever since I worked with them.

I must also thank my colleagues in Tom's research group, unofficially called the "Sanctuary Lab". Vaibhav Gogte, Kevin Loughlin, Amirhossein Mirhosseini, Harini Muthukrishnan, Akshitha Sriraman, Ofir Weisse, Brendan West, and Steve Zekany are brilliant people, and I have fond memories of working with them in a dynamic, lively lab. In particular, I thank Amirhossein very much, who helped me hit the ground running when I joined the group. He was easy to reach and willing to cooperate, and in fact, I co-authored the research related to Chapters II and III jointly with him. I also particularly thank Steve for he helped me a lot with accessing the group servers and also ordering and building new ones. Additionally, I was lucky to be a labmate of another group of wonderful people: Jonathan Bailey, Armand Behroozi, John Kloosterman, Salar Latifi, Shikai Li, Brandon Nguyen, Sunghyun Park, Jiecao Yu, Babak Zamirai, Pedram Zamirai, and Ze Zhang. I truly enjoyed working and being friends with them. Ze and I also did two internships at Microsoft at exactly the same times and offices. I learned so much from and was inspired by such great colleagues and friends, in both my previous and current labs.

The BBB building—that is, U of M's Computer Science and Engineering (CSE) Department—was the reason to build many valuable connections. I was fortunate to interact with fabulous people other than my labmates, including Shaizeen Aga, Tanvir Ahmed Khan, Subarno Banerjee, Daichi Fujiki, Shruti Padmanabha, Mehrzad Samadi, Arun Subramaniyan, Shahab Tajik, and Hanyun Tao. Tanvir and I also interned in the same group at Microsoft, working at desks right next to each other in a memorable summer. Furthermore, I must thank the great CSE staff, who work hard to enable us to study and work smoothly. In particular, I appreciate Ashley Andreae, Jamie Goldsmith, Karen Liska, and Stephen Reger,

who were always prompt and perfect in many occasions I sought their help.

The PhD life would be too tedious without close friends. I have been fortunate enough to have Agreen Ahmadi, Ramin Ansari, Javad Bagherzadeh, Salar Latifi, Babak Zamirai, and Pedram Zamirai as my true friends. My friendship with Salar goes back to our undergrad school, Sharif University of Technology. We did almost all the projects we had in the last three years of our undergrad together. We did the same in our grad courses after we both got admitted to U of M. Beyond that, we have been roommates in Ann Arbor for five years; what a marvelous blessing. I also knew Babak back from Sharif, but our true friendship began thanks to U of M. He was always there to help me in Ann Arbor in every step of life and study. I was honored to officiate the wedding of two of my best friends, Agreen and Pedram, in Ann Arbor. This experience was full of ineffable joy and thrill for me. Ramin is a cool, outgoing friend, who hosted me in a one-month interval when I was switching apartments. Javad is also an amazingly energetic friend, and we have awesome memories of playing football (known as soccer in the U.S.) to death! In addition, I have been very lucky to have friends like Samin Aayanifard, Mehran Amini, Omid Bahrami, Navid Barani, Farima Fatahi, Ashkan Kazemi, Alireza Khadem, and Milad Moosavifar, happy moments with whom will always remain in my memory.

Last but definitely not least, I must express my immense gratitude towards my family for their everlasting, infinite love and support. Above all, my parents, Mahmoud and Fooziyeh, are the reason I got this far. Apart from the huge love and care they always have had for me, they provided for my education with all they had and taught me the highest levels of integrity, dedication, and commitment. My lovely sisters—Nasim, Sahar, and Tolou—mean the world to me, and the bond we have cannot be described by words. Moreover, I am so happy that I have such wonderful brothers-in-law—Mehrdad, Abolfazl, and Mohammadhossein. Specifically, I was only able to survive several mental meltdowns during my PhD through the counseling and psychological support that Abolfazl provided for me. The brutal travel restrictions imposed on Iranian students in the U.S. and their families outside the U.S.

not only deprived me from meeting my family during my PhD, but I have also never met Mehrdad and Mohammadhossein in person. I have yet to meet and hug my cute nephew, Karan, as well. Nevertheless, Sahar and Abolfazl have been kindly sending me videos of Karan, so that I don't miss the everyday growth of this adorable, little boy. All in all, my family is the pieces of my heart and all the strength I have; I literally feel them under my skin.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Datacenter systems run myriad applications, which frequently communicate with each other and/or Input/Output (I/O) devices—including network adapters, storage devices, and accelerators. Due to the growing speed of I/O devices and the emergence of microservice-based programming models, the I/O software stacks have become a critical factor in end-to-end communication performance. As such, I/O software stacks have been evolving rapidly in recent years. Datacenters rely on fast, efficient "Software Data Planes", which orchestrate data transfer between applications and I/O devices. The goal of this dissertation is to enhance the performance, efficiency, and scalability of software data planes by diagnosing their existing issues and addressing them through hardware-software solutions.

In the first step, I characterize challenges of modern software data planes, which bypass the operating system kernel to avoid associated overheads. Since traditional interrupts and system calls cannot be delivered to user code without kernel assistance, kernel-bypass data planes use spinning cores on I/O queues to identify work/data arrival. Spin-polling obviously wastes CPU cycles on checking empty queues; however, I show that it entails even more drawbacks: (1) Full-tilt spinning cores perform more (useless) polling work when there is less work pending in the queues. (2) Spin-polling scales poorly with the number of polled queues due to processor cache capacity constraints, especially when traffic is unbalanced. (3) Spin-polling also scales poorly with the number of cores due to the overhead of polling and operation rate limits. (4) Whereas shared queues can mitigate load imbalance and head-of-line blocking, synchronization overheads of spinning on them limit their potential benefits.

Next, I propose a notification accelerator, dubbed *HyperPlane*, which replaces spin-

polling in software data planes. Design principles of HyperPlane are: (1) not iterating on empty I/O queues to find work/data in ready ones, (2) blocking/halting when all queues are empty rather than spinning fruitlessly, and (3) allowing multiple cores to efficiently monitor a shared set of queues. These principles lead to queue scalability, work proportionality, and enjoying theoretical merits of shared queues. HyperPlane is realized with a programming model front-end and a hardware microarchitecture back-end. Evaluation of HyperPlane shows its significant advantage in terms of throughput, average/tail latency, and energy efficiency over a state-of-the-art spin-polling–based software data plane, with very small power and area overheads.

Finally, I focus on the data transfer aspect in software data planes. Cache misses incurred by accessing I/O data are a major bottleneck in software data planes. Despite considerable efforts put into delivering I/O data directly to the last-level cache, some access latency is still exposed. Cores cannot prefetch such data to nearer caches in today's systems because of the complex access pattern of data buffers and the lack of an appropriate notification mechanism that can trigger the prefetch operations. As such, I propose *HyperData*, a data transfer accelerator based on targeted prefetching. HyperData prefetches exact (rather than predicted) data buffers (or a required subset to avoid cache pollution) to the L1 cache of the consumer core at the right time. Prefetching can be done for both core–peripheral and core–core communications. HyperData's prefetcher is programmable and supports various queue formats—namely, direct (regular), indirect (Virtio), and multi-consumer queues. I show that with a minor overhead, HyperData effectively hides data access latency in software data planes, thereby improving both application- and system-level performance and efficiency.

CHAPTER I

Introduction

Datacenters are composed of multi-tenant systems, each running a massive number of processes. Individual users rely on datacenter systems for services like cloud computing (through Virtual Machines (VMs), containers, etc.), cloud storage, Web search, emails, and cloud-assisted applications such as video/audio streaming, social media, and so on. Industrial and academic users leverage datacenter systems for high-performance computing [10, 12, 15], software-defined networking [4, 16], and network function virtualization [1, 3], to name but a few. Tenants of datacenter systems—i.e., host applications and client applications/VMs—frequently interact with each other as well as Input/Output (I/O) devices to connect to the outside world or other systems within the datacenter, or use peripherals such as storage devices or accelerators. Datacenters aim for high-throughput, low-latency data communication between CPUs and/or I/O devices, while being energy-efficient and keeping the systems highly utilized.

The Operating System (OS) performs I/O coordination and processing in conventional systems. Today's high-speed I/O devices—such as network cards, solid state drives, persistent memory, and PCIe-attached accelerators—have microsecond-scale access time, and due to their high throughput, they demand CPU attention every few microseconds. The OS and the underlying hardware are capable of effectively handling millisecond-scale and nanosecond-scale stalls, respectively. Nevertheless, they fall short of covering microsecond-

scale stalls, also known as the "Killer Microseconds" [61]. Furthermore, modern online data-intensive applications are shifting away from monolithic software architectures with millisecond-scale computation time, and leverage a distributed microservice-based software model that involves microsecond-scale computation time and much finer-grained inter-server communication [167, 169]. At such low latencies, high throughputs, and microsecond-scale service times, the overhead of I/O software stacks becomes absolutely critical.

Due to the shortcomings and overheads of existing operating systems in I/O processing, I/O software stacks are being re-architected [64, 87, 113, 115, 127, 151, 174, 176, 179]. State-of-the-art I/O software stacks, "Software Data Planes", bypass the OS kernel to avoid corresponding overheads such as context switches, system calls, interrupts, and cross–address-space copies. Kernel-bypass software brings the necessary OS functionalities to the user space, such as user-level thread/task scheduling, networking/storage transport processing, and device drivers [24, 43, 223]. In this dissertation, I show that software data planes have their own particular inefficiencies, especially when the I/O count and the number of tenants are scaled up. I propose hardware-software solutions to enhance the performance and efficiency of software data planes, as will be summarized in the following sections.

1.1 Characterization of Software Data Planes *

Modern software data planes bypass the OS software stack to avoid the attendant overheads of kernel-based I/O processing, and rely on cores spinning on user-level shared I/O queues as a fast notification mechanism. In fact, spinning cores are notified of the arrival of new work/data in I/O queues through cache coherence invalidation signals corresponding to cache lines holding queue heads. Whereas spin-polling can improve latency and throughput, it entails significant shortcomings, especially when scaling to large numbers of cores/queues. In the first set of contributions of this dissertation, I pinpoint and quantify challenges of spin-polling–based software data planes using Intel's Data Plane Development Kit (DPDK),

* Published in the 2019 ACM Symposium on Cloud Computing (*SoCC'19*) [104]

as a representative infrastructure, on a real system. I characterize four scalability issues of software data planes: (1) Spin-polling lacks work proportionality, meaning that even more (useless) work may be performed when there is less I/O traffic. (2) Throughput and latency are severely affected at high queue counts because a large fraction of time is wasted interrogating empty queues, especially when reading empty queue heads incurs cache misses, which is quite likely. (3) Operation rate limits (transactions per second) as well as a *Polling Tax* (the overhead of polling, which is considerable even when operating at saturation throughput) result in poor core scalability. (4) Multiple cores cannot efficiently spin on shared queues, which have theoretical merits in mitigating load imbalance and head-of-line blocking, because of coherence and synchronization costs. I identify root causes of these issues and discuss solution directions to improve hardware and software abstractions for better performance, efficiency, and scalability in software data planes. The design and implementation of a promising solution is the subject of the next part.

1.2 Acceleration of the Notification Mechanism [†]

In the spin-polling–based software data planes, cores often poll empty queues before finding work in non-empty ones. Interrogating empty queues hurts peak throughput, tail latency, and energy efficiency as it often entails fruitless cache misses. The second major contribution of this dissertation is *HyperPlane*, an efficient accelerator for the notification mechanism of software data planes. The key features of HyperPlane are (1) avoiding iteration over empty I/O queues, unlike software-only designs, resulting in queue scalability, (2) halting execution when I/O queues are idle, leading to work proportionality and energy efficiency, and (3) efficiently sharing queues across cores to enjoy strong theoretical properties of scale-up queuing. HyperPlane is realized through a hardware subsystem associated with a familiar programming model, centering on the `QWAIT` instruction. `QWAIT` either returns a ready queue to the calling core to be processed, or halts execution. HyperPlane's

[†] Published in the 2020 IEEE/ACM Symposium on Microarchitecture (*MICRO'20)* [161]

microarchitecture consists of a *monitoring set* and a *ready set*. The monitoring set, which is a lookup table structure with high or full associativity, watches cache coherence write transactions that indicate new data or work item arrivals in I/O queues. The ready set, which effectively functions as a task scheduler, tracks ready queues and distributes work to cores based on various service policies and priority levels. I model HyperPlane in a simulator, and show that it improves peak throughput by 4.1× and tail latency by 16.4× compared to a state-of-the-art software data plane. Furthermore, HyperPlane reduces the core power consumption down to only 16.2% at zero load or idle state. The monitoring and ready sets incur only less than 1% per-core power and area overheads.

1.3 Acceleration of Data Transfer [‡]

In addition to the notification mechanism, data transfer itself is another key aspect of data communication. I aim to enhance data transfer among the software data plane, I/O devices, and applications/VMs by designing the *HyperData* accelerator, the third major contribution of this dissertation. Data items in software data plane systems, such as network packets or storage blocks, are transferred through shared memory queues. Consumer cores typically access the data from DRAM or, thanks to technologies like Intel DDIO [27], from the (shared) last-level cache. Today, consumers cannot effectively prefetch such data to nearer caches due to the lack of a proper arrival notification mechanism and the complex access pattern of data buffers. HyperData is designed to perform *targeted* prefetching, wherein the *exact* data items (or a required subset) are prefetched to the L1 cache of the consumer core. Furthermore, HyperData is applicable to both core–device and core–core data communication, and it supports complex queue formats like Virtio [185] and multi-consumer queues. HyperData is realized with a *per-core programmable prefetcher*, which issues the prefetch requests, and a *system-level monitoring set*, which monitors queues for data arrival and triggers prefetch operations. I show that HyperData improves processing

[‡] Under review in the 2021 IEEE International Conference on Computer Design (*ICCD'21*)

latency by 1.20-2.42× in a simulation of a state-of-the-art software data plane, with only a few hundred bytes of per-core overhead.

1.4 Road Map

Throughout the remainder of this dissertation, I first elaborate on characterization of software data planes in Chapter II. I then discuss acceleration of the notification mechanism and data transfer in software data planes in Chapters III and IV, respectively. Finally, I summarize the dissertation in Chapter V and describe how the proposed solutions can be implemented in the near-future systems and how software data planes can be enhanced with further hardware support.

Additionally, I describe the earliest project that I completed during my PhD studies[§] in Appendix A. While the focus of this dissertation is on enhancement of datacenter systems, I initially concentrated on the performance and efficiency of Web applications at the client side, i.e., user phones and computers. I briefly describe the problem and findings here; the details are elaborated on in the appendix.

Web applications are widely used in many different daily activities—such as online shopping, navigation through maps, and social networking—in both desktop and mobile environments. Advances in technology, such as network connection, hardware platforms, and software design techniques, have empowered Web developers to design Web pages that are highly rich in content and engage users through an interactive experience. However, the performance of Web applications is not ideal today, and many users experience poor quality of service, including long page load times and irregular animations. One of the contributing factors to low performance is the very design of Web applications, particularly Web browsers. In the appendix, I argue that there are unnecessary computations in today's Web applications, which are completely or most likely wasted. I first describe the potential

[§] Published in the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (*ISPASS'19*) [103]

unnecessary computations at a high level, and then design a profiler based on dynamic backward program slicing that detects such computations. The profiler reveals that for four different websites, only 45% of dynamically executed instructions are useful in rendering the main page, on average. I then analyze and categorize the unnecessary computations. The analysis shows that processing JavaScript codes is the most notable category of unnecessary computations, specifically during page loading. Therefore, such computations are either completely wasted or could be deferred to a later time, i.e., when they are actually needed, thereby providing higher performance and better energy efficiency.

CHAPTER II

Software Data Planes: You Can't Always Spin to Win [*†]

2.1 Introduction

Software data planes, which use shared-memory queues and spinning cores to enable fast data transfer among application software, accelerators, and I/O subsystems, have become critical to the performance of datacenter systems. Originally conceived to enable fast network packet processing (e.g., firewalls, routing, denial-of-service protection, deep packet inspection), software data planes are now widely used to virtualize network and storage systems [87, 218], eliminate OS overheads to I/O latency and throughput [174, 179], administer shared I/O bandwidth [125, 211], construct virtual networks [122], enable Remote Direct Memory Access (RDMA) [48, 125, 171], implement network switches in software [35], facilitate high-performance computing applications [11] and microservices [200], and transfer data to hardware accelerators [184] for functionalities as diverse as erasure coding, encryption, and video transcoding [9, 44, 142].

Intel's Data Plane Development Kit (DPDK) [24] is a representative software infrastructure for building data planes to run on conventional Intel Xeon cores. Its central abstractions are (1) spinning cores—cores that execute a poll loop and never yield or invoke blocking OS functionality, and (2) user-level queue pairs—shared memory structures for the data

[*] Published in the 2019 ACM Symposium on Cloud Computing (*SoCC'19*) [104]
[†] Joint research with Amirhossein Mirhosseini

plane to communicate with client software and hardware devices. These abstractions are general enough, and they have been also used in many other software and hardware infrastructures, such as the Storage Performance Development Kit (SPDK) [43]. The key enabling mechanism for the spin-polling communication model of software data planes is that it relies on cores spinning on cacheable memory-mapped locations. The mechanism is similar to shared memory communication between two cores; it relies on cache coherence to propagate a write with low latency, and it generally does not produce much coherence traffic, as cores can spin locally in their cache, unless there is work, making it fast-reacting and low-overhead.

Software data planes improve latency and throughput of conventional systems through (1) bypassing the OS software stacks, and (2) enabling a fast signaling and notification mechanism by replacing hardware-managed interrupts and their associated overheads (e.g., switching address spaces, flushing hardware pipelines) with spin-polling. Data plane operating systems—such as IX [64], ZygOS [179], and Shenango [174]—leverage these mechanisms to implement low-latency and high-throughput OS-bypass I/O and networking stacks. Nevertheless, due to the rapid growth of the number of cores on a chip and the advent of Terabit Ethernet [8] and other high-bandwidth I/O devices [44], software data planes face considerable scalability challenges. In this chapter, we show that whereas software data planes provide an easy-to-use and efficient model for communication and signaling, they are far from ideal, especially when scaled to serve numerous clients/flows or require many cores to scale transport processing for high throughput.

Using Intel's DPDK as an example software infrastructure, we characterize four scalability challenges of software data planes, identify their root causes, and discuss solution directions and alternative approaches. We summarize our findings as follows:

Spin-polling performs more work when there is less. Since spinning cores run full-tilt even when they have no work, polling performs more (useless) work when there is no I/O traffic or work items in the queue. Therefore, spin-polling lacks energy proportionality [62]

and speeds up core/chip aging [172], especially at low system loads. Furthermore, useless spinning can drastically slow co-running applications on Simultaneously Multi-Threaded (SMT) cores and result in severe quality-of-service violations.

Spin-polling is not scalable to many queues. We show that increasing the number of queues on which a core spins increases processing latency and, depending on traffic balance, can harm peak throughput. This lack of scalability is caused by excessive pressure on processor caches. Moreover, we show that the performance overhead is greatest when most command queues are empty and traffic is concentrated in only a few queues; the overhead increases with the number of queues.

Spin-polling is not scalable to many cores. Spin-polling incurs a non-negligible instruction overhead for iterating over the body of the poll loop, which we call the *Polling Tax*. Even when operating at saturation throughput (100% load), we show that the polling tax is non-trivial. The polling tax increases the number of cores it takes to saturate network line rate, even for the simplest packet forwarding use case. Furthermore, various I/O devices and interconnects on the data-path (e.g., NIC, PCIe, DDIO) are constrained by operation rate limits (transactions per second), in addition to data rate limits. The Polling Tax and operation rate limits result in poor core scalability of software data planes.

Spin-polling is not well-suited for scale-up queuing. For many application classes, scale-up queuing organizations [160], wherein a single queue is shared among multiple cores, holds promise to improve latency and throughput through better queuing behavior by avoiding load imbalance and head-of-line blocking. However, most software data planes are currently implemented in a scale-out fashion, wherein there is a dedicated queue per core, to avoid the synchronization overheads of sharing a single queue across cores. We quantify this effect and show that software data planes can greatly leverage scale-up queuing organizations, specifically for high-disparity service distributions, but current spin-polling mechanisms are not well-suited for such queuing organizations.

This study aims to motivate better hardware and software abstractions to overcome

these challenges and enable greater data plane scalability to higher data rates and larger core counts. Future data planes should leverage advantages of spin-polling (e.g., OS kernel bypassing, low-latency I/O signaling) while avoiding the corresponding overhead and scalability issues. We envision a multi-address monitoring scheme, wherein multiple memory locations are monitored (through sufficient hardware support) for work arrival, as a promising solution direction. Moreover, data plane applications with unbalanced service distributions might benefit from work distribution schemes based on scale-up or hierarchical queuing. Such schemes could be enabled by wider SMT processors [159] with hardware task scheduling [158], or I/O-managed pull-based work distribution based on core availability [84].

We first provide a brief background on software data planes (Section 2.2) and describe our DPDK-based measurement methodology (Section 2.3). We then present each of our four main findings (Sections 2.4-2.7) and discuss potential solution directions to mitigate them (Section 2.8). Finally, we discuss related work (Section 2.9) and conclude (Section 2.10).

2.2 Background

We briefly describe the operation of software data planes and outline their applications in datacenter networking, storage systems, and I/O virtualization.

2.2.1 Software Data Plane Mechanisms

Conventional Systems. Figure 2.1 contrasts conventional kernel-based I/O stacks with modern software data planes. In kernel-based I/O stacks (Figure 2.1(a)), a user process signals the kernel that it wishes to perform I/O via a system call (e.g., through sockets or file system APIs). The protocol and transport processing needed to both read and write data is carried out by kernel threads, either by directly borrowing the CPU of the user process (e.g., during a system call), or by using interrupt mechanisms and kernel scheduling to place work on another core. Kernel I/O stacks are able to scale their CPU usage dynamically

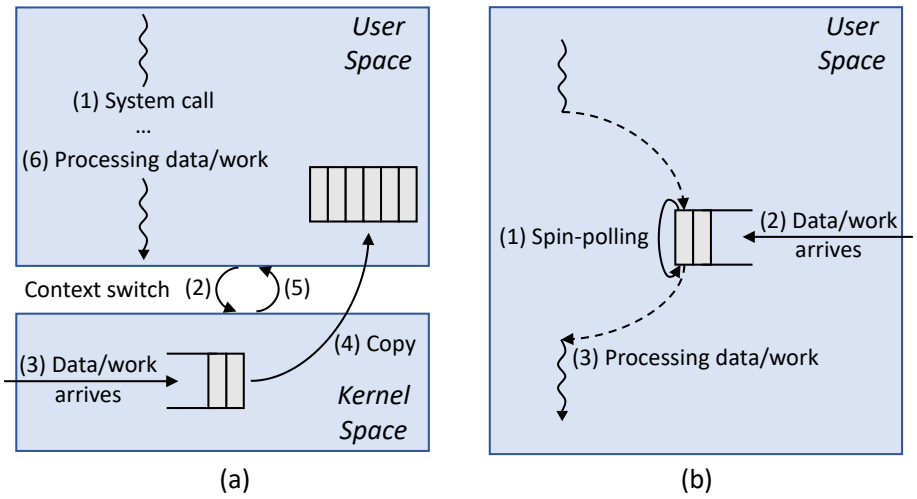


Figure 2.1: (a) Kernel-based I/O processing, (b) Spin-polling–based software data planes.

with I/O load, stealing CPU from user processes as needed to perform transport work. However, system call and interrupt-based signalling mechanisms are slow and transport processing is disruptive (especially for incoming data) to the user processes it interrupts. Although the kernel goes to great lengths to spread transport work and optimize for affinity to user processes, the aggregate CPU cost for gigabit- and terabit-scale networking is significant [113].

Key Data Plane Mechanisms. As illustrated in Figure 2.1(b), software data planes rely on *spin-polling cores* and *user-level queue pairs* for low-latency and high-throughput communication, enabling both faster signalling than conventional system call/interrupt-based mechanisms and greater CPU efficiency for high throughput. In such systems, user processes communicate with transport processing code and I/O devices via in-memory queue pairs to schedule I/O operations and get notified of their completion. The key enabling mechanism of this approach is that the two queues, which may be called the submission/completion or send/receive queues, are typically in cacheable shared memory. Therefore, the two communicating end points can quickly signal one another if each spins on the head of its respective inbound queue. These locations are cached in a *shared* state within local L1/L2 caches, and the end points may locally spin awaiting a change without triggering any coherence transaction or on-chip network traffic [195]. In the shared state, all sharers

maintain a read-only copy of a memory location. Before any processor or I/O device may write to the location (i.e., to append a new request or completion to a queue), it must first invalidate these copies. These invalidation messages propagate rapidly (nanoseconds within a chip; hundreds of nanoseconds across I/O interconnects) and serve as a low-latency notification mechanism. The subsequent read of an invalidated line will obtain its new value. The key to this mechanism is that memory traffic between the communicating end points only occurs when the originator of a request or completion writes to a queue—a receiver spin-polling on an idle queue produces no traffic.

Zero-Copy Data Transfer. Software data planes can further improve CPU efficiency relative to kernel transport by using *zero-copy* data transfer mechanisms. With zero-copy data transfer, two communicating end points access common buffer pools in a shared memory space. Unlike traditional OS-based approaches, where data are copied multiple times within main memory to traverse address spaces (frequently incurring context switches along with each copy), zero-copy mechanisms source outgoing data directly from user-space buffers where the data were first prepared, and land incoming data directly in the address space of the user process that will receive them. The transport software in the data plane arranges for data to flow directly between buffer pools in user processes and I/O devices, without any intermediate copies into memory owned by the data plane or kernel. Zero-copy mechanisms typically require some hardware support, and rely on user applications to adhere to more stringent lifetime and flow control guarantees for data and buffers in shared pools than synchronous kernel I/O interfaces (i.e., they are harder for programmers to use correctly).

Off-Chip Devices. I/O devices on the same chip as the CPU core or connected via a Non-Uniform Memory Access (NUMA) fabric (e.g., integrated NICs or accelerators) can directly share memory with the user process running on the CPU through virtual memory. For PCIe attached I/O devices, data typically must be transferred to a local buffer on the I/O device before they can be transmitted/stored. This transfer is typically accomplished

in one of two ways. In a Memory-Mapped I/O (MMIO), individual CPU store instructions trigger a PCIe transaction to update buffers on the I/O device. To reduce the number of PCIe transactions, CPUs implement a "write combining" optimization, which combines stores to generate cache-line–sized PCIe transactions. In a doorbell-based approach, instead of generating PCIe transactions directly via stores, after preparing a larger chunk of data in memory, the CPU issues a single PCIe write to a "doorbell" location, which triggers the target device to perform a Direct Memory Access (DMA) transfer to the device's memory. The doorbell-based approach may use PCIe bandwidth more efficiently, but can result in higher latency and more CPU work for the doorbell operations [118].

Bringing Data on Chip. When PCIe-attached I/O devices receive data on behalf of a user process, the updates are propagated from device buffers to CPU-side shared memory buffers via PCIe transactions. A hardware steering mechanism that can determine to which user process incoming data are destined is required for zero-copy receive; high end network cards and Solid-State Drives (SSDs) provide such flow steering mechanisms. Conventionally, PCIe transactions write data to main memory. However, if data plane software or the receiving user process will immediately access the data, the CPU will incur a costly Last Level Cache (LLC) miss to retrieve the cache line. To avoid this cache miss, prior work has investigated direct cache placement of I/O data [66, 112]. Accordingly, Intel has introduced Data Direct I/O (DDIO) technology [27], where a bus-mastering PCIe device can write directly into and read from the processor's LLC. With this mechanism, if a processor accesses incoming I/O data shortly after they are received, they will still be present in the LLC and costly main memory reads can be avoided. Similarly, data targeted to the I/O device do not need to be written back to memory; the device can read outbound data directly from the LLC. To prevent the I/O device from displacing too much CPU data from the LLC, it is restricted to write only to a limited number of LLC ways.

Steering to Multiple Queues. When multiple processes/CPUs communicate with the same I/O device, multiple queue pairs are typically provisioned, to avoid synchronization

overheads on each queue. Most modern I/O devices (e.g., multi-queue NVMe SSDs [207]) readily support numerous queue pairs. When a CPU submits work to a specific submission queue, the I/O device knows that it has to acknowledge through the corresponding completion queue. However, directing completions to the correct queue is more challenging when communication is not solicited by the CPU. For example, when a NIC receives packets, it must choose to which CPU's receive queue to append the packet. This steering is challenging because the NIC seeks to load-balance transport work across CPUs, but also seeks to ensure that packets sent in the same flow (between the same communicating end-user processes) are delivered in order. The commonly used solution for this problem is Receive-Side Scaling (RSS) [40]. In RSS, the NIC performs a hash of various fields in packet headers to identify the flow to which a packet belongs. The result of the hash function is used as an index to an indirection table, which specifies a set of queues/cores to which the packet should be directed. For TCP/IP traffic, the hash function is typically a 4-tuple hash over source/destination IP addresses and port numbers.

2.2.2 Software Data Plane Applications

We describe recent applications of software data planes.

Networking. With the advent of Software-Defined Networking (SDN), flexibility and programmability have been brought to network backends by having a logically centralized control plane and a data plane. In SDN, a decoupled control plane, as the network brain, programs the data plane on how to forward packets. This approach has transformed traditional switches to programmable ones (through languages like P4 [69]), and has also enabled implementation of network functions—such as routing, load balancing, address translation, and firewalls—on industry standard servers. As a result, many software data plane solutions in datacenter networks have been proposed and deployed [87, 122, 174, 179], which benefit from cores spinning on queues tightly coupled to Network Interface Controllers (NICs). Intel's DPDK [24] is a representative open-source infrastructure for building

spin-polling–based network data planes.

Storage. High-speed storage devices such as SSDs and new persistent memory technologies like Intel's Optane [29] have been explored for use in demanding applications in databases and big data analytics. Data planes for such storage systems demand fast mechanisms for data transfer between the CPU and the device, leading to the introduction of new protocols, such as NVM Express (NVMe) and NVMe over Fabrics (NVMe-oF). Consequently, the concept of spinning cores to process queues for such storage devices has been increasingly adopted. The Storage Performance Development Kit (SPDK) [43] is the canonical example of using CPU to spin-poll user-space queues for storage devices.

I/O Virtualization. Data centers are commonly virtualized, where multiple Virtual Machines (VMs) run on the same host. Through I/O virtualization, a single network or storage device can be used by multiple VMs. This sharing is enabled by virtual machine managers or hypervisors, which manage access of multiple VMs to the I/O device(s). Alternatively, VMs can directly access I/O devices through nascent Single Root I/O Virtualization (SR-IOV) technology, bypassing the hypervisor. These mechanisms make it feasible for the VM data plane to take advantage of the previously described spin-polling mechanisms of network and storage devices. A VM can therefore spin-poll an I/O device queue through the hypervisor (e.g., DPDK's Vhost library) or directly through SR-IOV.

2.3 Methodology

At the heart of the software data planes in software defined networking, whether in a virtualized environment or not, CPU cores read packets from receive (RX) queues, process them, and then send processed packets or newly generated ones to transmit (TX) queues. Each RX or TX queue pairs a CPU core with an I/O device or another core (e.g., a client process). In a generic scenario, a core must handle packets received from I/O devices or cores in a number of RX queues and eventually transmit a number of packets through TX queues back to them or to other I/O devices and cores (Figure 2.2(a)). As a result,

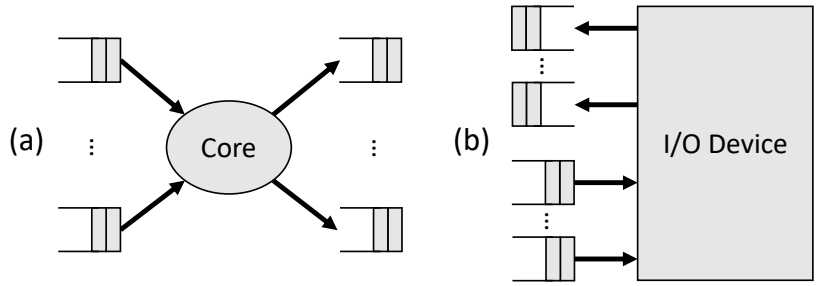


Figure 2.2: Data communication through RX/TX queues: (a) From a core to I/O devices or other cores, (b) From an I/O device to cores.

each I/O device (e.g., a NIC) communicates with CPU cores through these RX/TX queues (Figure 2.2(b)).

CPU cores and queues are points of scalability in software data planes. As in Figure 2.2(b), many queues (up to several hundred) may be introduced for a single NIC. Scaling up the number of queues communicating with a NIC is done either for application-specific purposes (e.g., differentiated QoS classes, each represented by a queue) or to scale transport processing to fully utilize the line rate offered by the NIC. In the latter case, CPU cores are also scaled up, each coupled with the NIC through one or more queues. We design experiments to investigate the performance and scalability of modern software data planes.

We use Intel's Data Plane Development Kit (DPDK) [24], an open-source project, as a representative infrastructure for building spin-polling-based software data planes. DPDK provides poll mode drivers for numerous modern NICs, which enable cores to spin on user-level queues to communicate with the NICs. DPDK is heavily optimized to offer high performance. For instance, DPDK pins its threads to specific cores to reduce context switches as much as possible. NUMA-aware memory allocation, use of 2MB and 1GB huge pages, and cache-alignment–aware data placement are among other optimizations done in DPDK.

Our experiments primarily use DPDK's Test Poll Mode Driver (TestPMD) and routing / Layer 3 Forwarding (L3Fwd) applications, and we modify them as needed to implement scalability experiments and to measure additional statistics, such as poll count and CPU

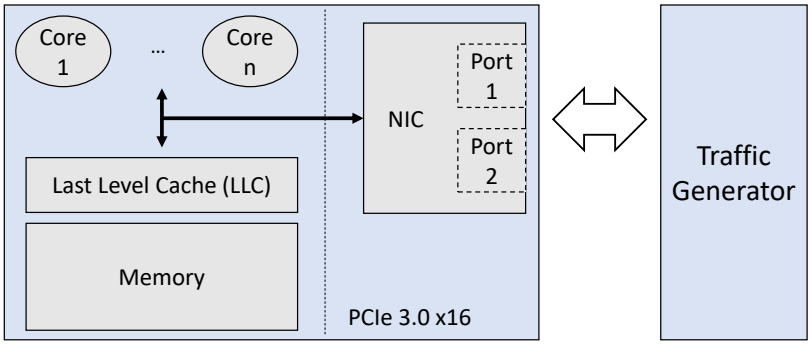


Figure 2.3: The machine under test receives packets directly from the packet generator.

Table 2.1: HW/SW specs of experimental machines.

Item	**Machine under test**	**Packet generator**
CPU	Single-socket, Xeon Platinum 8160 24 physical cores @ 2.10-3.70 GHz 32 KB L1 I/D-cache (per core) 1024 KB L2-cache (per core) 33 MB LLC	Dual-socket, Xeon Gold 6138 2×20 physical cores @ 2.00-3.70 GHz 32 KB L1 I/D-cache (per core) 1024 KB L2-cache (per core) 27.5 MB LLC (per socket)
Memory	96 GB (6×16 GB) DDR4 @ 2666 MHz	96 GB (12×8 GB) DDR4 @ 2666 MHz
NIC	Mellanox ConnectX-5 MCX556A-ECAT 100 GbE dual-port PCIe 3.0 x16	Mellanox ConnectX-5 MCX556A-ECAT 100 GbE dual-port PCIe 3.0 x16
OS	Ubuntu 18.04.2 (LTS)	Ubuntu 18.04.2 (LTS)
Software	DPDK 18.11.1 (LTS)	DPDK 18.11.1 (LTS) Pktgen 3.6.6

cycles spent in different parts of the application. For CPU cycle classification, we read the x86 Time Stamp Counter (TSC) register at appropriate places in the code. In the L3Fwd application, the Longest Prefix Match (LPM) algorithm is used, and the routing table is filled with 16K entries. We have also developed a similar application to investigate the use of shared queues. Details of each application will be presented in the following sections as we introduce each experimental setup. Unless otherwise specified, we use 64-byte IPv4 UDP packets (96-byte for experiments requiring timestamps for latency measurement), but we vary packet sizes in several experiments.

Our system under test is a Skylake server equipped with a dual-port 100 GbE Mellanox ConnectX-5 NIC. Detailed specifications of this machine appear in Table 2.1. As shown in Figure 2.3, the NIC in the system under test receives traffic from a separate packet generator

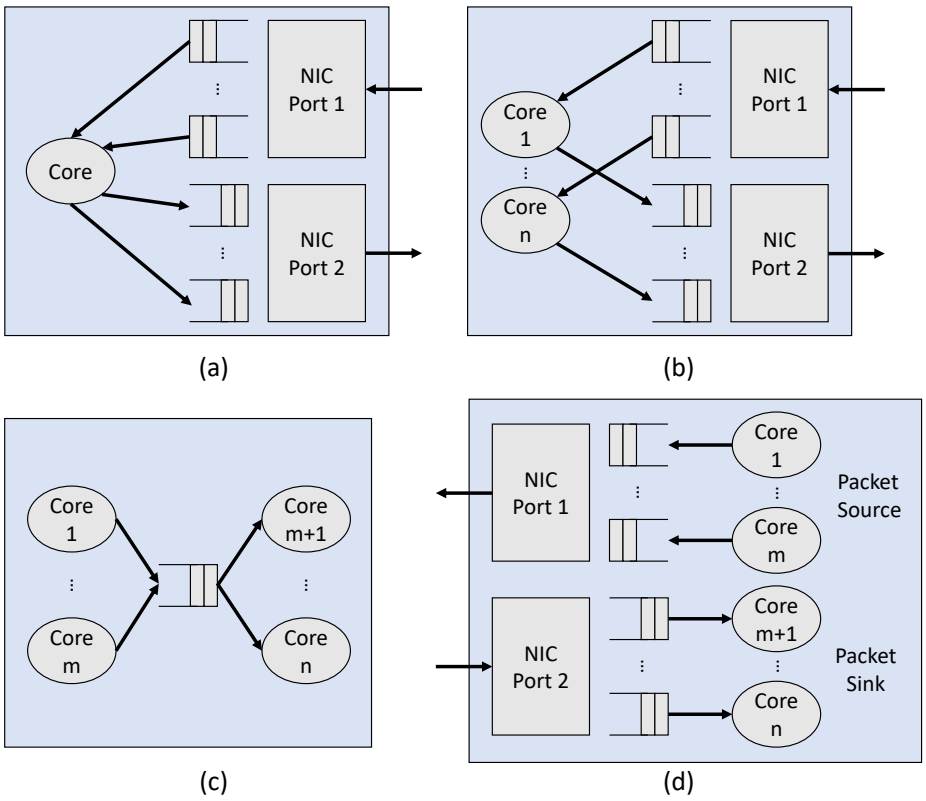


Figure 2.4: Configurations of cores and queues in experiments: (a) Scaling up the number of queues in the machine under test; (b) Scaling up the number of core-queue pairs in the machine under test; (c) A shared queue accessed by multiple cores in the machine under test; (d) The generic setup of cores and queues in the packet generator machine.

connected directly to the NIC ports. The setup for core/queue scalability experiments is shown in Figure 2.4(a) and (b). DPDK does not support sharing NIC queues among multiple cores; therefore, for queue sharing experiments, we instead investigate an application that distributes incoming packets among a set of cores (e.g., an intra-machine virtual network) as shown in Figure 2.4(c). Note that, unless otherwise specified, we use only a single hyperthread on each physical core (i.e., all threads run on different physical cores). We use the Linux Perf tool to gather microarchitectural statistics, particularly, cache hit and miss rates.

Our packet generator is another Skylake server, which also has a dual-port 100 GbE Mellanox ConnectX-5 NIC, as described in Table 2.1. Packets are generated using Pktgen [36], an open-source DPDK-powered application. Figure 2.4(d) illustrates the generic setup we

use to generate packets. Ports 1 and 2 of the NIC in the packet generator are connected directly to ports 1 and 2 of the NIC in the machine under test, respectively. For experiments where we wish to measure round-trip latency, the packet generator appends a timestamp to the packet when it is sent from port 1 and calculates the round-trip latency when an echoed/response packet is received at port 2 using a local high-precision time source. We also modify Pktgen to output a latency distribution.

2.4 Inefficiencies of Spin-Polling

Spin-polling is the key mechanism for fast signalling between software data planes and their work sources (clients or devices). Although spin-polling typically enables lower latency than system-call or interrupt-based mechanisms, it suffers from two general inefficiencies: *polling tax* and *work disproportionality*.

2.4.1 Polling Tax

Spin-polling incurs a non-negligible overhead from the useless work of iterating over the body of the poll loop. We refer to the cycles wasted on these useless instructions as the *Polling Tax* as they are inherent/inevitable when polling is used. Figure 2.5 reports the breakdown of the cycles of a single CPU core spent on polling overhead (Tax) vs. useful work (i.e., routing packets) in an LPM-based routing application using the configuration shown in Figure 2.4(a), under different offered loads, and varying the number of RX queues on the first NIC port from one to eight (one TX queue for each port).

We have instrumented the application code to classify CPU cycles as "useful" and "polling overhead". The application body includes an infinite outer main loop, and an inner loop that traverses the queue heads. If an RX queue is non-empty, CPU cycles spent on reading packets from the RX queue, routing, and sending them to the TX queue are classified as useful. The rest of CPU cycles are classified as polling overhead (including loop branch overheads and polling of empty queues).

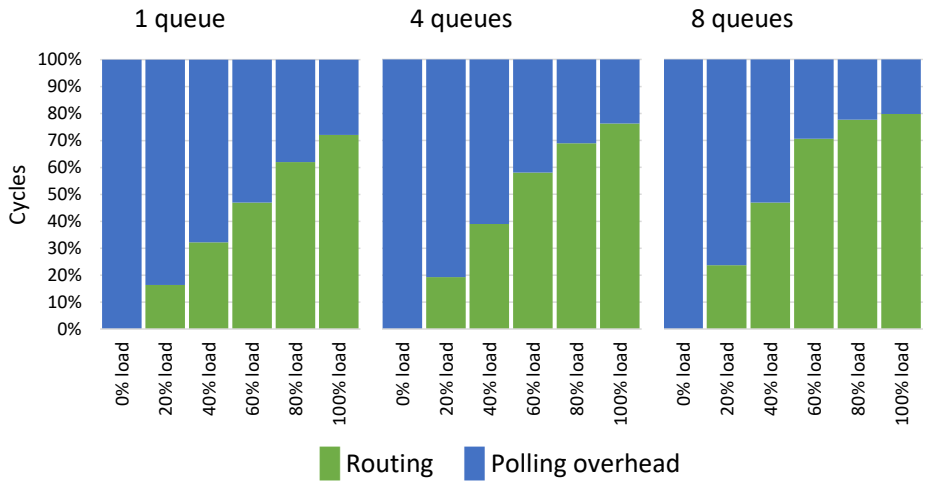


Figure 2.5: Useful cycles vs. cycles spent on polling in a core performing network routing.

As shown in Figure 2.5, even at 100% load (i.e., maximum routing throughput), a significant fraction of CPU cycles (~20-28%) are spent on spin-polling, rather than the useful work of routing packets. Note that in each iteration of the main loop, n queue heads are checked. Therefore, overhead is amortized over n queues. This amortization results in overhead decreasing as the number of queues increases at a specific offered load. The high percentage of polling tax is due to the fact that the useful work per iteration is relatively simple (i.e., lookups in forwarding tables), highlighting the impact of poll loop overhead; with more complicated tasks in the loop body, we expect a smaller polling tax.

2.4.2 Work Disproportionality

When polling, spinning cores run full-tilt even when they have no work. Modern cores can spin with remarkably high Instructions Per Cycle (IPC) while the core is not doing any useful work. To quantify this effect, we measure the IPC of a single core spin-polling 1, 4, and 8 queues for LPM routing at varying offered loads. Figure 2.6 reports the IPC of the poller core as throughput increases. As shown in the Figure, IPC decreases up to 29% as throughput increases. As throughput increases, more time is spent on useful work, which exhibits substantially lower IPC than spin-polling (e.g., due to cache misses). Our measurement confirms the observations of Alameldeen and Wood [56] that IPC is a harmful

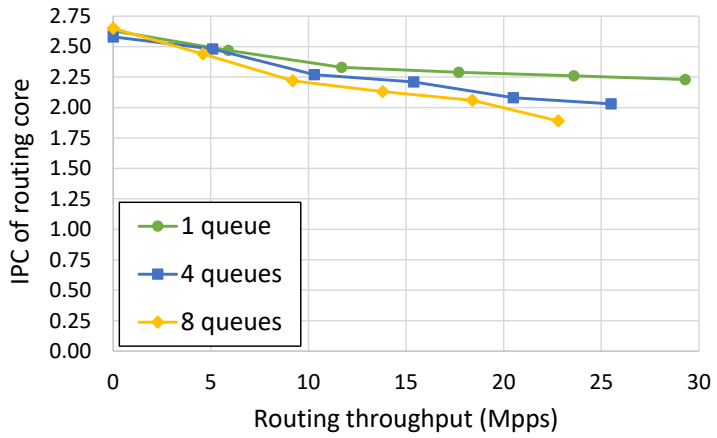


Figure 2.6: Instructions Per Cycle (IPC) of a spin-polling core performing network routing.

metric for performance evaluation of workloads that use busy spinning for communication, as it does not necessarily represent useful work. Furthermore, our result shows that spin-polling harms core efficiency (the lower the load, the more instructions are executed). The high IPC of useless spin-polling has three implications:

Energy disproportionality and inefficiency. Spin-polling results in poor energy proportionality [62] as the core energy consumption when spinning is higher than when performing useful work. Moreover, spinning often costs turbo-boost head-room for other (application) cores. We have observed that 8 and 16 cores spinning full-tilt on empty queues reduce the frequency of a core running a regular application from ~3.45 GHz to ~3.25 GHz and ~2.95 GHz in our machine under test, respectively.

Modern datacenter applications usually exhibit low IPCs (often < 1; at most 1.5) on Xeon-class cores [57, 58, 95, 141, 198]. However, as shown in Figure 2.6, software data planes can yield average IPCs of 2-2.6, depending on the load and the number of queues. CPU IPC translates directly to circuit-level switching activity, which drives the switching power of VLSI systems. As a result, we expect cores running data plane software to consume at least 30% more switching power. These effects have also been observed in the case of spin-locks for synchronization-heavy applications [90].

Faster aging. Spin-polling also has an adverse effect on processor aging, due to the high

IPC and core power consumption despite doing no useful work. Prior work has reported a 7-10 year lifetime for server-class cores in 32nm technology [210]. We expect shorter nominal core lifetimes in more recent technology, as the aging rate of silicon increases substantially with smaller transistors [172]. Among various physical effects that cause transistor aging, *Negative Bias Temperature Instability (NBTI)* [215] and *Hot Carrier Injection (HCI)* [205] are dominant. HCI, in particular, arises from the cumulative effect of switching activity [172], which correlates with IPC at the microarchitecture level. In comparison to typical server applications, software data planes have higher IPC and never enter sleep states. Given 30% or more higher total activity, spinning core lifetimes may drop below the 5-year threshold, requiring larger supply voltage guard bands (implying higher power) or reduced peak frequency as chips age. Both energy proportionality and server lifetimes are particularly critical in datacenter environments, as they have significant impacts on the Total Cost of Ownership (TCO) [62, 129, 155, 156, 186].

Co-runner interference. Spin-polling has an adverse effect on co-runner threads in SMT cores, since fruitless polling consumes a large fraction of execution resources that could otherwise accelerate the co-runner thread. In fact, the ICOUNT policy, which grants fetch bandwidth to competing SMT threads in proportion to their IPC and is widely employed to schedule execution resources among competing SMT threads [212], is counterproductive for a mostly idle poll loop. SMT efficiency is also critical in datacenter environments because (1) datacenter operators prefer to utilize SMT to improve overall utilization and reduce TCO [95, 120], and (2) modern datacenter services have tight latency targets that are easily violated due to heavy co-runner interference [143, 159].

We further investigate the co-runner interference effects of spin-polling. Figure 2.7(a) reports the IPC of a matrix multiplication application when (1) not collocated with any other thread, (2) collocated with the LPM routing workload at zero load (i.e., polling idle queues), (3) collocated with the LPM routing workload at 100% load (i.e., highest possible ratio of work to polling overhead), and (4) collocated with Geekbench 4 [46], which is a compute-

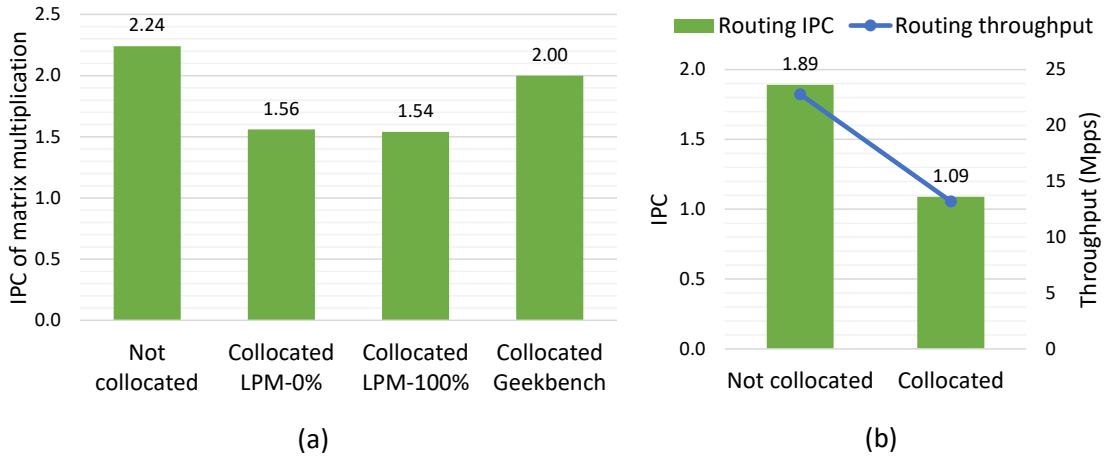


Figure 2.7: The adverse effect of a spin-polling application and a regular matrix multiplication application on each other when collocated on two SMT hyperthreads: (a) The IPC of the co-running matrix multiplication decreases. (b) Packet throughput of the spin-polling application drops.

and memory-intensive benchmark suite of real-world applications. As shown in the Figure, when co-running with LPM under either load condition, the matrix multiplication IPC is substantially lower than without a co-runner (by 30% and 31% with empty and full queues, respectively) and the case where it is collocated with Geekbench by 11%. The antagonistic impact of the LPM code is slightly higher under load as it competes for cache capacity as well as execution bandwidth. Nevertheless, even with no load (and hence, no useful work), the high IPC of the idle poller thread (see Figure 2.6) drastically slows the matrix-multiply co-runner.

Figure 2.7(b) demonstrates the co-runner antagonistic impact of the matrix multiply on LPM routing at 100% load. The LPM routing application also experiences 42% IPC reduction and 42% packet throughput reduction due to SMT collocation. Note that, while both co-runners suffer lower IPC, the overall IPC of the core is about 39% better, which confirms that SMT is desirable to improve utilization [95, 120, 159].

Note that many of the undesirable effects we characterized may be mitigated by `MWAIT`-like monitoring instructions that avoid spinning by waiting on a location to change values [6], or spin-loop detectors that are already present in some processors to reduce the IPC of a spinning core [137]. As an example, prior work has shown that using `MWAIT` inside spin-

locks can improve energy efficiency of synchronization-heavy applications by 1.5× [90]. However, both of these mechanisms are only suitable for polling a single location; `MWAIT` can only monitor a contiguous address range, and spin loop detectors usually fail to detect a loop that iterates over multiple queues, as demonstrated by the high IPC at zero load shown in Figure 2.6. The x86 Instruction Set Architecture (ISA) also offers a `PAUSE` instruction, which is intended to slow instruction throughput in tight spin loops. However, prior work has reported mixed effectiveness in using `PAUSE` in synchronization spin loops [90].

2.5 Lack of Queue Scalability

Network traffic typically consists of numerous flows, which are spread among multiple queues by data plane applications (e.g., because they originate from multiple clients). The application needs to take actions on each flow, such as metering, routing, filtering, encapsulation/decapsulation, and encryption/decryption. Flows may also be associated with priorities to provide differentiated quality of service. Since the data plane application is provisioned a limited number of CPU cores, each core may be responsible for processing multiple queues of traffic flows. Due to the limited size of L1/L2 and LLC caches, we expect the time to poll a set of queues to grow non-linearly with the number of queues, thereby reducing maximum throughput and increasing latency. In this section, we inspect caching effects from increasing the number queues on packet processing throughput and latency.

We illustrate the experimental setup used in this section in Figure 2.4(a). Ingress traffic flows are spread among RX queues through RSS. To focus on the effect of scaling up the number of queues, we use a single polling thread running TestPMD's forwarding mode, where ingress packets on the first NIC port are forwarded to the second port.

First, we consider the case where there is no traffic and all queues are idle. We vary the number of queues up to 512, which is the maximum allowed by the DPDK poll mode driver for this NIC. Figure 2.8 shows that the poll rate (queue heads polled per second; left axis) decreases as the number of queues are increased. Polling scales poorly as queue heads

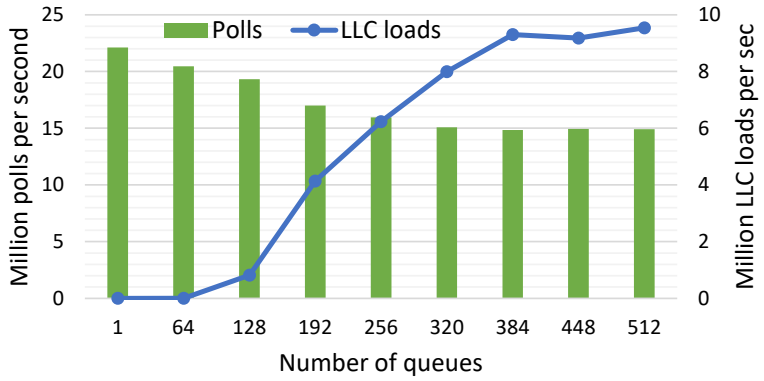


Figure 2.8: Idle (zero-traffic) polling rate (left axis) and LLC loads per second (right axis) vs. number of queues.

fall out of caches. In particular, from one queue to 512 queues, there is a 33% decrease in polling rate, even though the work is the same (i.e., spinning on empty queues). Figure 2.8 also reports the number of LLC loads per second (right axis). Above 64 queues, we observe a gradual increase in the number of LLC loads. This trend shows that queue heads no longer fit in the L1 and L2 caches and some must be read from the LLC each poll loop iteration. Above 384 queues, reading any queue head results in an L1/L2 cache miss. Note that the LLC is large enough to accommodate all queue heads, and the decrease in the polling rate can be solely attributed to the limited L1 and L2 capacity.

Next, we examine the effect of scaling the number of queues under load. Note that in addition to the undesirable effect of limited cache size with multiple queues, the time a core spends spin-polling empty queues adversely affects queues that contain packets. To isolate these effects, we analyze two cases. First, we consider the case where the traffic comprises a single flow that passes through only one queue; therefore, only one queue has useful work and the remainder are idle. Second, we consider a fair-share case where the traffic is spread over multiple flows so all queues are well-utilized.

Figure 2.9 shows the average latency under light load ($<$ 1 Mpps). The packet forwarding latency for both single-flow and multi-flow traffic increases with the number of queues. With 512 queues, the average latency is more than 6$\times$ worse than the single-queue case. Since load is low in this scenario, there is minimal queuing delay in both the single-flow

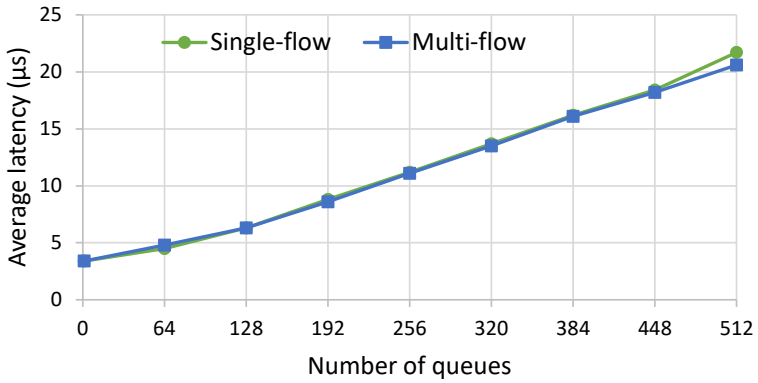


Figure 2.9: Round-trip latency of packet forwarding under light traffic (< 1 Mpps), with varying number of queues.

and multi-flow cases. Furthermore, the number of cache misses incurred between the arrival and forwarding of a packet is about the same under both the single and multi-flow cases, and both exhibit the same average latency.

In contrast, when we examine a high-throughput scenario, we see a substantial difference between single-flow and multi-flow performance. Figure 2.10 shows the maximum forwarding throughput when scaling queues for single-flow (a) and multi-flow (b) traffic alongside the rate of LLC load hits and misses per second (secondary axis). For single-flow traffic, we observe that the forwarding throughput decreases from one queue to 8 queues and then remains roughly constant up to 64 queues. It then gradually decreases from 64 queues to 512 queues. There are no LLC load misses. In this case, both the queue descriptors and packet data from the single active queue fit within the LLC (ingress packets are delivered directly to the cache via DDIO). For 1-64 queues, all queue heads fit in L1/L2 caches, and we see no decrease in the throughput since the core spins fast on the queues. For 64 or more queues, although the additional queues are idle, polling their heads often incurs L1/L2 cache misses as the queue heads are displaced by data accessed when servicing the active queue. As such, the number of L1/L2 cache misses (to the idle queue heads) per transmit burst grows. It takes long enough to traverse the set of empty queue heads that the NIC runs out of work from the active queue's transmit batch and PCIe bandwidth is underutilized. The dip

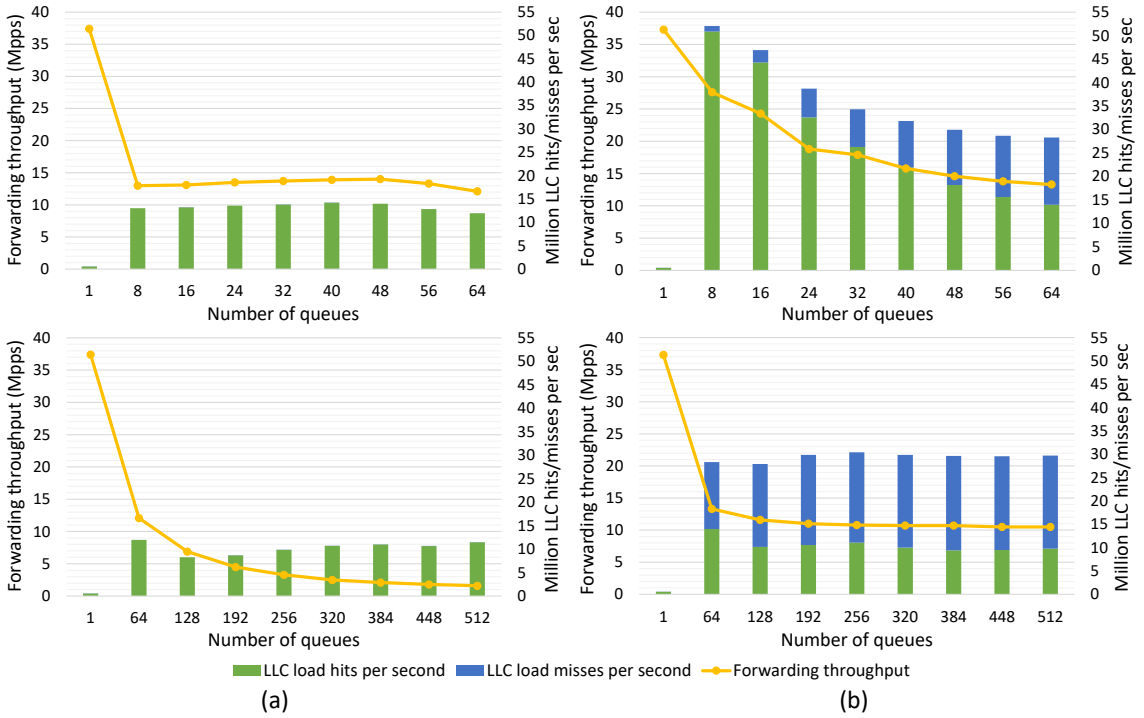


Figure 2.10: Maximum forwarding throughput and LLC load hits/misses per second as the number of queues increases: (a) Single-flow, (b) Multi-flow

in LLC load hits per second from 64 to 320 queues arises due to the interaction of slower poll loop iterations (causing LLC load hits per second to drop) and increasing contention for L2 capacity (causing queue heads be replaced to the LLC, increasing LLC accesses per loop iteration).

In the multi-flow case, throughput gradually decreases from one queue to 64 queues, and then it remains stable above 10 Mpps from 64-512 queues—substantially better than the paltry 1.6 Mpps achievable by a single flow with 512 queues. Once the core attempts to access packet data, it is more likely that the data have been displaced by other ingress packets as the number of queues increases. For 64 or more queues, the number of misses to queue heads and packet data per transmit burst is constant. With traffic spread over all queues, each queue head miss will yield a transmit burst. As a result, the NIC pipeline and PCIe bandwidth remain utilized throughout the poll loop. Note that throughput is much higher than in the single-flow case even though packet data must now traverse main memory

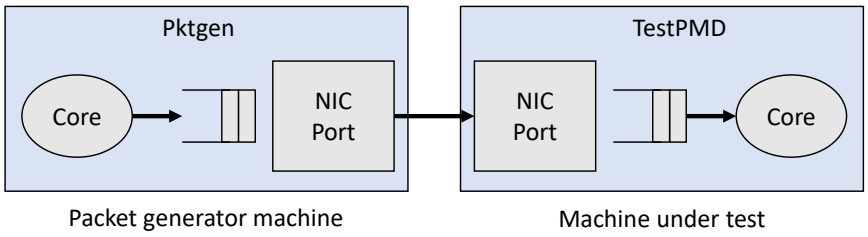


Figure 2.11: Setup for measuring single-core throughput.

(as evidenced by the large number of LLC misses per second; blue bars in Figure 2.10(b)). Main memory provides sufficient bandwidth at these packet rates.

In summary, increasing the number of queues increases average latency and decreases peak throughput. These effects arise because the number of L1/L2 misses to queue heads grows due to cache pressure, and are magnified in the single-flow case because the misses are not interleaved with transmit bursts and do not amortize, leading to underutilization of the NIC egress pipeline—while peak throughput converges to a constant in the multi-flow case, it continues to decrease with more queues in the single-flow case.

2.6 Lack of Core Scalability

In this section, we investigate how many cores it takes to saturate the line rate of our 100 Gbps network adapter under varying packet sizes. We first measure single-core throughput, where only one core is used in both the packet generator and the machine under test in the setup shown in Figure 2.11. Packets are generated in advance and are sent on the line as fast as possible. The receiver processes the NIC descriptors for every received packet, but does not examine the packets, which are simply discarded. Transmit (TX) and receive (RX) packet and data rates for varying packet sizes are illustrated in Figure 2.12. A single core is unable to saturate line rate even with the largest packets (i.e., Maximum Transmission Unit (MTU)). Note that for smaller packets, the RX core cannot keep up with the packet rate even though it only reaps the receive packet descriptors; the NIC ends up discarding some packets. The RX core is unable to keep up because of polling overhead. Note that the

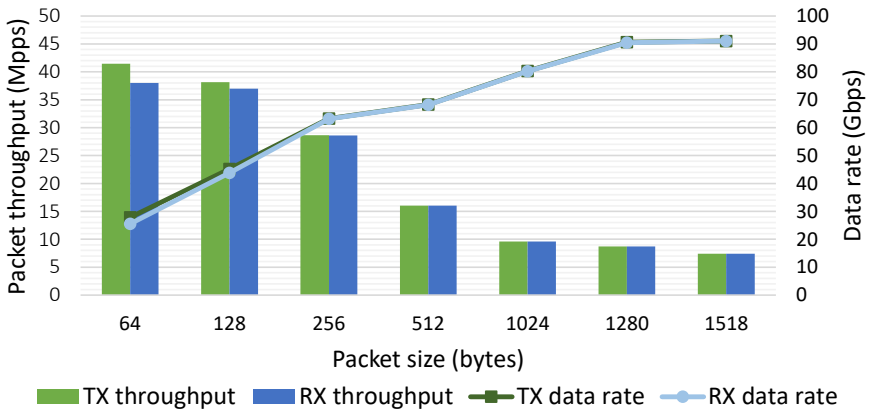


Figure 2.12: Maximum throughput of a single core.

RX throughput shortfall is less than the polling tax for a single core at 100% load reflected in Figure 2.5. As such, in the absence of this overhead, we expect the RX core would keep pace (or exceed) TX performance. In contrast, the TX core does not perform any wasted work. As packets get larger, packet rate becomes smaller, and the bottleneck shifts from the CPU core to PCIe bandwidth. Thus, the RX core is able to keep up with the TX core for 256-byte and larger packets.

Next, we analyze how throughput scales when we increase the number of cores. On the packet generator machine, we provision sufficient cores to ensure we saturate the 100 Gbps line rate. We configure the machine under test as shown in Figure 2.4(b). To avoid any synchronization on queues, we provision a dedicated queue on each NIC port for each core. We use TestPMD's forwarding mode, in which packets received from the first port are simply forwarded to the second port without examining the packet. The goal is to forward all the (saturated) incoming traffic by increasing the number of forwarding cores. Note that incoming packets are spread across RX queues evenly using RSS.

Figure 2.13 shows packet throughput as the number of cores increases for four different packet sizes. The RX and TX rates in this Figure are from the machine under test. Forwarding throughput (i.e., the TX rate) does not scale linearly even though none of the cores contend on shared memory. Even for the simplest forwarding task, we observe that line rate can often be saturated with a large number of cores. More complex packet processing would

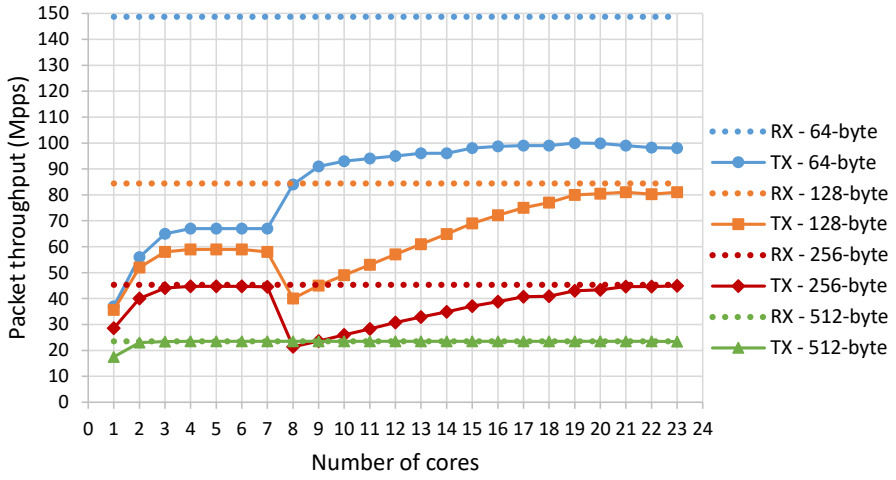


Figure 2.13: Packet throughput as the number of core-queue pairs increases. The RX line is saturated in all the cases.

demand even more cores to saturate line rate. The non-linearity and asymptoticity of core scaling can be attributed partially to the polling tax, and is more visible with smaller packets. These effects also can arise due to various operation rate limits, for example, for NIC, PCIe MMIO, and DDIO/LLC transactions. Furthermore, we observe surprising behavior for 3-8 cores/queues. From three to seven cores, we see a plateau in the throughput for 64-, 128-, and 256-byte packets. At eight cores, we see a sudden throughput discontinuity. Scaling behavior beyond eight cores matches our expectations, and ultimately reaches line rate for all packet sizes above 128 bytes. We note that, for 64-byte packets, transmit performance cannot reach line rate. According to the vendor specifications, 100 million packets per second is the limit of the NIC hardware capability for RX, which is insufficient to reach line rate for small packets.

The discontinuity from seven to eight cores coincides with a sudden decrease in memory bandwidth and a sudden increase in LLC bandwidth. This shift in memory behavior suggests that the DDIO mechanism is sourcing packets from the last-level cache (as it should) above eight cores/queues, but is sourcing packets from main memory with fewer cores. From seven to eight cores, forwarding throughput decreases for 128- and 256-byte packets but increases for 64-byte packets. As mentioned in Section 2.2, the DDIO mechanism uses a

limited number of LLC ways. Apparently, the limited size of the LLC subset used by DDIO is insufficient for larger packets and results in a sudden performance drop upon activation of DDIO. Furthermore, we have determined that this behavior is tied to the number of queues rather than the number of cores interacting with the NIC. As such, we believe the anomalous performance plateau from 3-8 cores is most likely due to the specific implementation of the DDIO mechanism in the NIC. As a result, we discount performance over this range when drawing conclusions.

2.7 Scale-up Queuing is Impractical

We next consider the opportunities that might arise when sharing queues across multiple cores, rather than dedicating queues for each core. Sharing queues is often challenging in the context of networking applications, as common implementations of higher-level networking protocols (like TCP/IP) expect ordered delivery of packets in a single flow; shared queues make it difficult to guarantee such ordering. Nevertheless, shared queues provide strong theoretical properties that merit further exploration.

Figure 2.14 illustrates two different queuing models for concurrent work: scale-out and scale-up. In the scale-out model, each core obtains work from a separate queue and a dispatcher steers work into queues to balance the arrival rate across queues. The scale-out model captures the typical RSS mechanism used to distribute work among cores in networking applications. The 4-tuple hash ensures ordering within individual flows, but distributes work irrespective of the load on each core. In the scale-up model, instead a single queue is shared among all cores, wherein each fetches work from the central queue. This model allows work to spread evenly over cores irrespective of the distribution of traffic across flows. However, this model requires synchronization of the central request queue and does not naturally ensure ordered flow delivery.

Neglecting ordering and synchronization costs, the scale-up organization always outperforms the scale-out organization, in terms of average response time, for two reasons [160]:

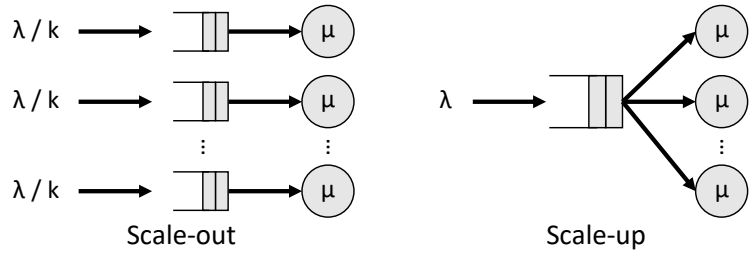


Figure 2.14: Scale-out vs. scale-up queuing organizations with k cores. (λ and μ represent arrival and service rates.)

First, in the scale-up organization, a core will not remain idle if there is work waiting in the central queue (i.e., it is work conserving). However, in scale-out systems, a core may remain idle if its own queue is empty even while work is outstanding elsewhere. Second, when a packet takes longer to process than average in a scale-out organization, all the packets behind it suffer from Head-of-Line (HoL) blocking delays. In contrast, in scale-up architectures, packets may be serviced by any core; stalling at one server has less impact on the system-wide instantaneous service rate.

Software data plane infrastructures are usually optimized for scale-out organizations because (1) as already noted, many networking applications do not tolerate out-of-order delivery within flows, and (2) the RSS mechanism of modern NICs already distributes the packets almost uniformly into different queues when an application serves many similar flows. In this section, we seek to quantify the trade-off between the performance advantages and synchronization costs of implementing a scale-up queuing organization in software data planes by sharing a queue across multiple cores. We do not further consider packet ordering in this study; we consider applications where out-of-order delivery is allowed, such as layer three routing, unordered datagrams, higher-level protocols for adaptive routing, or non-networking applications.

We design an experiment as shown in Figure 2.15. In both the scale-out (Figure 2.15(a)) and scale-up (Figure 2.15(b)) configurations, the machine under test receives packets on its first NIC port spread over multiple queues by RSS. Next, n "dispatcher" cores read the packets and write them to multiple queues in the scale-out configuration or to a single shared

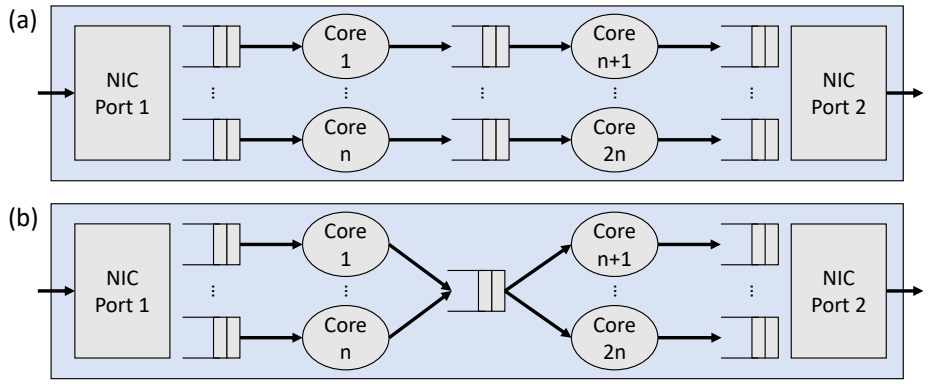


Figure 2.15: Experimental setup of (a) scale-out, and (b) scale-up, configurations.

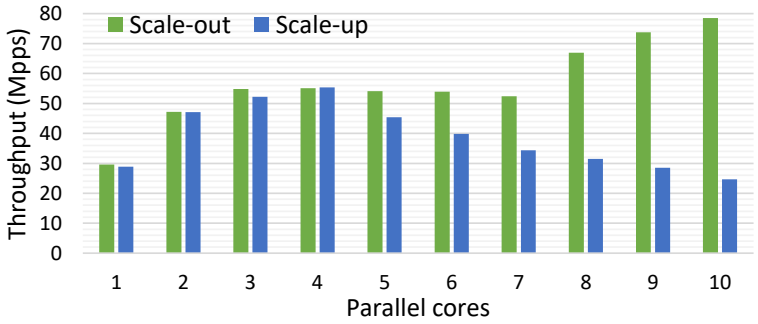


Figure 2.16: Maximum throughput achieved by scale-out and scale-up configurations.

queue in the scale-up configuration. Finally, another set of n cores read the packets from the multiple scale-out queues or the single shared queue and forward them to corresponding transmit queues of the second NIC port. Note that we designed this relatively complex setup to compare the scale-out and scale-up methodologies because DPDK does not support sharing a NIC queue among multiple cores.

Figure 2.16 reports the maximum throughput achieved by different numbers of cores forwarding packets, organized in both scale-up (sharing a single queue) and scale-out (each core has a distinct queue) fashions. As shown in the Figure, the throughput achieved by the scale-out organization scales with the number of cores; the lack of scalability between 3 to 7 cores is due to the NIC issue discussed in the previous section. However, with the scale-up organization, throughput only scales up to 4 cores and then falls off due to synchronization overhead. This overhead is caused by serialized updates to the shared queue by multiple cores, although the shared queue is implemented in a lock-free manner in DPDK.

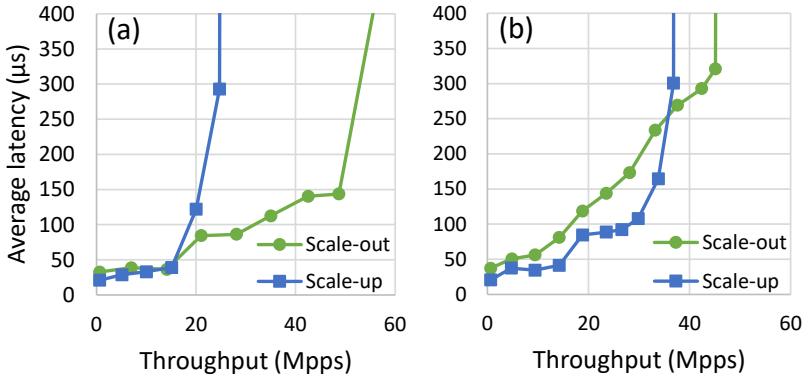


Figure 2.17: Average round-trip latency of scale-out vs. scale-up configurations with 10 cores: (a) No hiccups, (b) 1 µs processing hiccup with 1% probability.

Figure 2.17(a) reports the average packet latency when forwarding with 10 cores using both scale-up and scale out organizations, under various load levels. As shown in the Figure, while the scale-up organization results in slightly lower unloaded latencies, it saturates much earlier and results in considerably higher latency at loads above 15 Mpps. At higher load, synchronization overheads outweigh the latency advantages of the scale-up organizations. However, as the figure shows, the knee of the latency graph for the scale-up organization is much steeper than the one for the scale-out organization (sudden saturation vs. gradual latency increase before saturation). This is an inherent feature of scale-up queuing systems; with sufficient concurrency, scale-up systems eliminate queuing delays at loads lower than the saturation throughput, forming a perfect "hockey stick" curve.

Nonetheless, scale-up systems provide the greatest advantage for service distributions that entail high variability, which leads to a high probability of HoL blocking. One of the main reasons for such service distributions are *system hiccups* [160], wherein processing is stalled because it is interrupted by system tasks (e.g., garbage collection [214], memory compaction [130], power state transitions [135]), or the interference of collocated workloads, within SMT hyperthreads [159], on-chip caches [121, 143], and memory [225]. Such interference can arise even on cores dedicated to data plane processing. Whereas such hiccups are more apparent under large-scale deployments, to model them in our test environment, we add a hiccup condition, wherein forwarding a packet might be delayed by 1 µs with

1% probability, following prior work [160]. The resulting average packet latencies for 10 cores under various offered loads are depicted in Figure 2.17(b). As shown in the Figure, the scale-up organization now achieves considerably lower unloaded latency compared to the scale-out organization and saturates at higher load (relative to the case with no hiccups; Figure 2.17(a)). This advantage arises because the scale-up organization smooths service variability and mitigates HoL blocking. However, due to synchronization costs, the scale-up organization is still unable to match the peak throughput of the scale-out organization. The resulting trade-off might be exploited by designing an adaptive system that switches queuing disciplines in response to load based on the latency break-even point observed in Figure 2.17(b).

Scale-out queuing may benefit from more sophisticated load balancing schemes than the RSS mechanism. For example, a core may be applied as a load balancer, reading packets from the NIC(s) and pushing them to per-core queues based on the load of each flow. Nevertheless, scale-out queuing with better load balancing is still not equivalent to scale-up queuing as the latter both balances the load across cores and also eliminates HoL blocking. Load balancing is most beneficial when load is below peak and some cores become idle. However, at near-saturation load, especially if the task size distribution is heavy-tailed (as with the hiccups we consider in Figure 2.17(b)) and the metric of interest is tail latency (rather than throughput), load balancing has little effect as it does not eliminate HoL blocking. An adaptive load balancer might do better by balancing load according to queue depth, but again, little effect is expected on latency at high load.

In summary, we observed that the scale-up organization can result in significant performance gains over scale-out, in principle, especially in the presence of system hiccups or if the distribution of packet processing time entails high variability. However, due to synchronization costs, these advantages cannot be fully exploited. Therefore, any effort to reduce the synchronization costs in software/hardware [190] or to alleviate the serialization effect of lock-free synchronization [101] may potentially unlock the performance

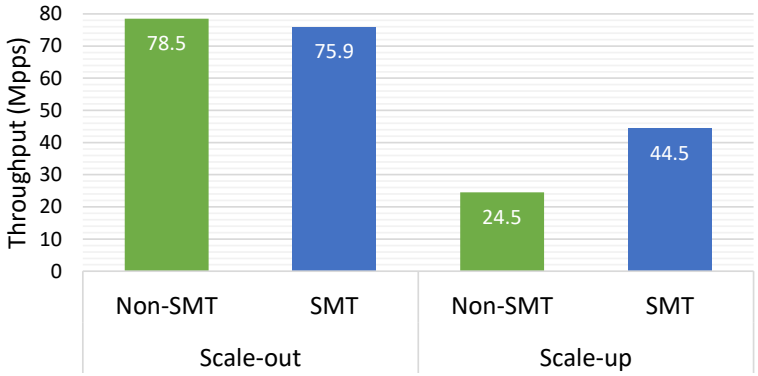


Figure 2.18: Throughput of scale-out vs. scale-up configurations with 10 logical cores in case of using 2-threaded SMT cores or separate physical cores.

advantages of the scale-up organization. To highlight this effect, Figure 2.18 compares the maximum throughput achieved by 10 forwarding cores under both scale-up and scale-out organizations when (1) each logical core is mapped to a separate physical core, and (2) when each two logical cores are mapped to the two hyperthreads of the same physical core. As shown in the figure, while the scale-out organization experiences a slight slow-down with hyperthreading, due to resource contention, the scale-up organization experiences significant performance improvement (45%) when hyperthreading is enabled. This advantage arises because both hyperthreads of each physical core share the same L1 and L2 caches, and hence, are less sensitive to synchronization overheads. We expect further improvement in scale-up performance may be possible in cores with a larger number of hyperthreads [158, 159].

2.8 Discussion: Solution Directions

In Sections 2.4-2.7, we discussed high-level deficiencies of software data planes, which make it difficult to close the gap between a traditional OS/interrupt-based system and an ideal notification mechanism as throughput demands and core counts scale. In this section, we discuss potential solution directions.

Spin-polling induces a non-trivial polling tax, even when operating at saturation throughput. Spinning also results in poor energy proportionality and is disruptive to SMT co-runners.

Future data planes require an alternative mechanism that retains the latency advantage of spin-polling while avoiding these pitfalls. `MWAIT`-like mechanisms that monitor memory locations for changes could bridge this gap, but current implementations monitor only a single address [6]. Such mechanisms mimic the behavior of an interrupt-based system, without the common inefficiencies (e.g., kernel crossing and switching address spaces) of conventional implementation.

The key enabling mechanism for address-monitoring is cache coherence [195]; coherence invalidation messages act as a signal that a location's value is changing. An address monitoring solution might compare incoming coherence invalidation messages with a set of monitored addresses. To scale to many addresses, this address set might be maintained in large cache-like associative structures at each core or be tracked in the on-chip cache coherence directory, which already tracks the sharers of each cache block. Similar solutions have been proposed to enable many watchpoints for debugging [105, 213], conflict detection in transactional memories [163], and memory consistency violation detection [75].

We have observed that a scale-up queuing organization (shared queues) can provide substantial performance advantages, especially in mitigating tail latency. However, synchronization overheads presently prevent shared queues from being scalable and practical. RPCValet [84] proposes a potential solution for on-chip integrated NICs. Instead of the integrated NIC "pushing" packets into each core's dedicated queues, which may result in load imbalance and HoL blocking, each core "pulls" a packet from the NIC as soon as it is done processing the previous packet. The single shared packet queue is managed in hardware by the on-chip NIC and distributes packets into the cores' local queues. RPCValet's solution might be generalized to non-integrated I/O through a specialized on-chip dispatcher unit at the LLC to implement such a pull-based queuing solution for any shared-queue application. Mechanisms like DDIO might facilitate transfer of off-chip I/O data to this hardware dispatcher. Properly addressing concurrency and ordering constraints among arriving work remains an open challenge in this model.

In summary, we believe hardware mechanisms for address monitoring and work distribution may boost the performance of software data planes by replacing spin-polling and enabling shared queues across multiple cores without their attendant synchronization costs.

2.9 Related Work

Kernel bypass and spinning cores. Frequent OS intervention is a performance antagonist. Many prior works use kernel bypass to mitigate the system call and/or interrupt overheads. MICA [139], Sandstorm [150], mTCP [113], and eRPC [117] provide user-level networking stacks. Arrakis [176] uses SR-IOV for direct access to virtualized I/O. User-level storage frameworks have also been proposed [126, 218]. Additionally, many works rely on spinning cores as a low-latency notification mechanism. IX [64], ZygOS [179], Shenango [174], Shinjuku [115], and Andromeda [87] are examples of spin-polling network data planes, and ReFlex [127] and PASTE [111] provide polling-based network and storage data planes for remote Flash devices and persistent main memory, respectively.

Energy inefficiency and the poor scalability of spin-polling have been explored in the context of lock-based synchronization [90, 144]. Although spinning is the key enabler of low-latency software data planes, its shortcomings in such a context have not been systematically characterized and compiled into a single work before. The intuition that fruitless spinning should be avoided has motivated introduction of user-level threading and scheduling for spin-based software data planes [87, 174, 179]. However, in this work, by quantifying and characterizing inefficiencies of spinning at the architecture level in the context of software data planes, we set the ground for exploring generic notification mechanisms that are fast-reacting, efficient, and scalable. Such generic mechanisms can avoid complicated user-level schedulers and be deployed in the data planes of networks as well as storage systems.

Data movement. Efficient data movement is one of the key factors in the performance of software data planes. Prior work has proposed solutions to shorten CPU-I/O data

path [27, 67, 138], take the CPU out of the data path [81, 97, 201], optimize data movement within user space [87, 117], and accelerate memory copies [114].

Shortening the path through which data travels in software data planes can improve processing latency and throughput. For example, using Intel's Data Direct I/O technology [27], the CPU can immediately read incoming I/O data from PCIe directly in the LLC, avoiding long-latency memory reads. Similarly, a PCIe device can read outgoing data directly from the LLC. Integrated NICs have also been proposed to enable close coupling of CPUs and network adapters [67, 138], which eliminates the need for long-latency data movement across PCIe.

Programmable NICs have enabled rich packet processing offload capabilities, which can completely remove the CPU from the data path (i.e., complete offload rather than partial offload). Microsoft Azure has deployed an FPGA-based smart NIC [97], which has an embedded programmable switch to forward packets to VMs through SR-IOV without the intervention of hypervisor cores. Similar offloading features have been proposed for storage systems [81, 201].

Although bypassing the kernel removes many unnecessary data movements, user-level data movement must also be efficient in software data planes. Google's Andromeda [87] deploys carefully crafted fast-paths for data movement to and from VMs, done entirely at user level. Kalia and co-authors [117] seek to minimize data movement in a user-level RPC platform. Note that software data planes can generally benefit from memory copy accelerators (e.g., [114]) to accelerate data movement.

2.10 Conclusion

We have presented a characterization of spin-polling software data planes, which bypass OS I/O stacks and rely on cores spinning on user-level queues. Although these mechanisms are known to be easy-to-use, low-latency approaches for communication and signaling, we demonstrated that they lead to deficiencies, especially when cores or queues are scaled to

serve numerous clients/flows or provide high transport throughput. We designed several experiments to reveal these deficiencies using Intel's Data Plane Development Kit. First, we quantified polling's lack of work and energy proportionality and its adverse effect on co-running applications because spinning cores run full-tilt at highest IPC. Second, we demonstrated the poor scalability of spin-polling with the number of queues, particularly in the case of unbalanced traffic, due to constrained processor cache capacity. Third, we quantified that the overhead of polling, which is considerable even when operating at saturation throughput, and operation rate limits result in poor core scalability. Finally, we considered the use of shared queues as a mitigation of head-of-line blocking, but found that synchronization overheads presently limit the potential benefit. In conclusion, we motivated better hardware/software notification mechanisms that enable fast-reacting, efficient, and scalable software data planes.

CHAPTER III

HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes [*†]

3.1 Introduction

Computer system designers are on the hunt to address "Killer Microseconds" [61, 78, 159]. The latency to access modern I/O devices—such as emerging storage-class and disaggregated memories [51, 52, 140, 171], 100+ gigabit networking devices [68], and high-throughput accelerators [74, 178]—is as low as single-digit microseconds. At such low latencies and high throughputs, the I/O software stack becomes a critical factor in end-to-end communication performance. Moreover, modern cloud applications are shifting away from ms-scale single-binary monoliths to loosely-coupled µs-scale microservices, to achieve better scalability, reliability, and programmablity [80, 100, 199]. With µs-scale service times, the I/O software stack's latency becomes comparable to computation time and must be aggressively optimized [197, 200].

In conventional systems, sharing an I/O device among multiple applications is orchestrated through the kernel—depicted in Figure 3.1(a). When a user process signals the kernel that it wishes to perform I/O via a system call, the protocol and transport processing software stack is carried out by kernel threads, either by directly borrowing the user process's CPU,

[*] Published in the 2020 IEEE/ACM Symposium on Microarchitecture (*MICRO'20)* [161]
[†] Joint research with Amirhossein Mirhosseini

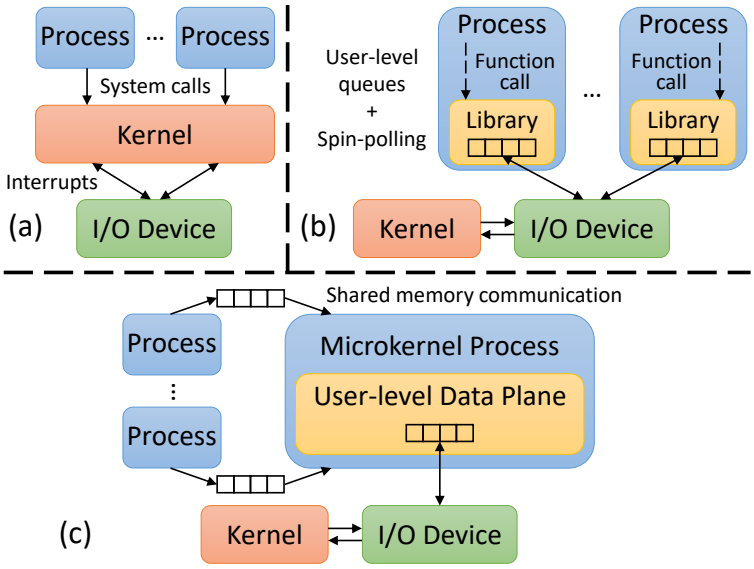


Figure 3.1: I/O communication approaches: (a) conventional kernel-based, (b) user-level library OS, (c) microkernel-based "software data planes".

or by using interrupt mechanisms and kernel scheduling to place work on another core. Nonetheless, the mechanisms involved in such kernel-based approaches—including synchronization, scheduling, inter-processor interrupts, switching address spaces, and copying data across address spaces—impose significant performance overheads when dealing with I/O devices that exhibit µs-scale latencies and gigabits- to terabits-per-second throughputs [113].

In contrast, as illustrated in Figure 3.1(b), modern I/O devices provide virtual user-level queue pairs for user processes to communicate directly with them, bypassing the kernel software stack. As such, since traditional interrupts cannot be delivered to user code without kernel assistance, user processes often perform spin-polling on the queues to be notified of new data/task arrivals. In these schemes, I/O software stacks are often implemented as a library operating system loaded as part of the user process, and invoked through function calls [64, 109, 113, 176]. Zero-copy data transfer mechanisms further improve CPU efficiency by placing I/O buffer pools in the user process address space, enabling incoming data items to directly land where the receiving user process can access them. The transport software in zero-copy I/O stacks arranges for data to flow directly between buffer pools in user processes and I/O devices, without any intermediate copies.

Despite all the performance benefits of kernel-bypass I/O stacks, the key shortcoming of these systems is the lack of centralized coordination that conventional kernel-based approaches enable. By providing a central view of all resources and the tenant processes/VMs that communicate with each resource, the kernel is able to deliver more efficient scheduling, fairness, and resource accounting. As a result, a popular alternative approach used by systems like Google's Snap [151] is deploying the I/O stack within a user-level software data plane microkernel module—illustrated in Figure 3.1(c)—communicating with the application processes through user-level shared memory queues. This way, the user-level software data plane manages all the I/O queues and exploits its centralized view to provide better resource management and scheduling while retaining most of the performance benefits of kernel-bypass I/O stacks. Furthermore, by decoupling and isolating the I/O stack from both the application and the kernel, this approach provides better locality and also enhances development and release velocity for the I/O software stack. Academic projects like Shenango [174] have advocated similar approaches.

Nonetheless, even microkernel-based software data planes typically rely on spinning cores and user-level queues in their transport software. Although spin-polling is an easy-to-use, fast-reacting approach for communication and signaling, it involves a number of inevitable drawbacks, especially when dealing with hundreds (or more) queues. First, the traffic that passes through these queues is often unbalanced, and consequently, a large subset of queues are empty at any given point in time. Traffic is unbalanced because (1) some I/O devices are inherently more frequently accessed than others, and (2) tenant applications/VMs typically experience bursty activity patterns at different times. As such, a large fraction of time in software data planes is wasted interrogating empty queues, especially when reading empty queue heads incurs cache misses, which is quite likely. This useless work substantially hurts the tail latency and peak throughput of software data planes and limits their queue scalability [104]. Furthermore, software data planes may even exhibit "work disproportionality"; that is, they perform more fruitless spinning work in terms of

Instructions Per Cycle (IPC) at lower transport load, leading to energy inefficiency. Finally, software data planes can benefit substantially from sharing queues across multiple cores to achieve better queuing properties. However, the coherence and synchronization costs of spinning on shared queues make such sharing impractical [84].

In this chapter, we propose *HyperPlane*, a hardware notification accelerator that facilitates fast (user-level) software data planes, which unlike software-only spinning alternatives, exhibits queue scalability and work proportionality and enables efficient queue sharing. HyperPlane comprises a programming model front-end and a hardware microarchitecture back-end that together enable efficient operation. The core of its programming model is the `QWAIT` instruction, which has similar semantics to the *select-case* construct in the Go programming language [47]. `QWAIT` waits on a set of doorbell locations associated with queues and blocks execution until a work item arrives to a queue. Once one or more queues are ready, `QWAIT` returns the next Queue ID (*QID*) that must be serviced according to the specified service policy—round-robin, weighted round robin, or strict priority. `QWAIT` is inspired by the x86 `MWAIT` and ARM `WFE` instructions, which halt execution until the contents of a single memory address or address range change.

The two key components of HyperPlane's microarchitecture back-end are a *monitoring set* and a *ready set*. The monitoring set comprises the locations of doorbells associated with each queue. These locations are monitored in hardware for cache coherence write transactions that indicate new work item arrivals. Once a write transaction for a doorbell is matched to a QID in the monitoring set, the QID is moved to the ready set. The ready set, which is HyperPlane's key departure from prior monitoring schemes, tracks ready QIDs and determines the next queue to service according to a service policy. `QWAIT` returns the QID of the next ready queue from the ready set. The ready set effectively functions as a task scheduler at non-trivial loads, sorting the order of ready queues to be serviced.

To the best of our knowledge, HyperPlane is the first hardware accelerator proposal that enhances the performance and energy efficiency of software data planes. HyperPlane

achieves queue scalability as `QWAIT` always returns the next QID to be processed without the need for checking many empty queues. It avoids work disproportionality of fruitless spinning because it halts execution when there is no work item in any queue, avoiding wasted energy/execution resources or harming the execution of another hyperthread on the same core. Finally, since the monitoring and the ready set units are shared across all cores within the chip, HyperPlane enables efficient cross-core queue sharing and enjoys the strong properties of scale-up queuing models, providing higher performance and better support for queue priorities. Our results show that HyperPlane improves peak throughput by 4.1× and tail latency by 16.4×, on average, in comparison to a state-of-the-art spin-polling–based software data plane, across a varying number of I/O queues (up to 1000). Moreover, HyperPlane achieves up to 6.2× lower idle (zero-load) power consumption. With 1024-entry monitoring and ready sets in a 16-core CMP, HyperPlane incurs $< 1\%$ per-core power and area overheads.

We first provide background on software data planes and motivate our work by illustrating challenges of software data planes (Section 3.2). Next, we describe the design and detailed microarchitecture of HyperPlane (Sections 3.3 and 3.4). We then evaluate HyperPlane (Section 3.5). Finally, we discuss related work (Section 3.6) and conclude (Section 3.7).

3.2 Background and Motivation

3.2.1 Software Data Planes

Software data planes manage data communication of tenants (i.e., host applications or client applications/VMs) with I/O devices, such as Network Interface Controllers (NICs) [151], Solid State Drives (SSDs) [133], persistent memory devices [29, 39], and accelerators [44]. Software data planes have two key functionalities: First, they manage I/O queues and direct traffic to the corresponding tenant and vice versa. Each tenant uses one or

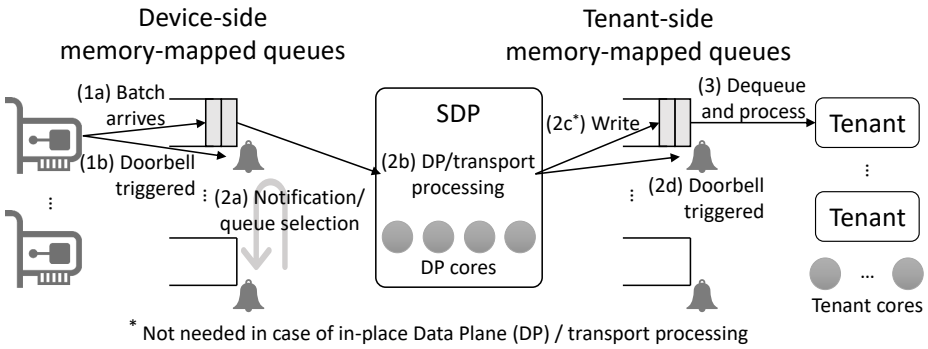


Figure 3.2: Software Data Plane (SDP) operations.

more queue pairs to communicate with particular I/O devices. Thus, software data planes must efficiently handle a large number of queues (i.e., ~1k) corresponding to many tenants and several I/O devices.

Second, software data planes provide low-level I/O operation services to tenants. Software defined networks and virtual network functions demand fast packet processing, which is enabled by software data planes through services like address translation, firewall, software switching, and deep packet inspection [87]. Software data planes also provide services to virtualize storage systems [218], administer shared I/O bandwidth [125, 211], enable Remote Direct Memory Access (RDMA) [48, 125], facilitate high-performance computing [11], perform erasure coding [28], and encrypt/decrypt data [24, 28].

Figure 3.2 illustrates the receive-side interactions in a software data plane (the transmit-side diagram looks similar): (1a) a batch of one or more packets/work items arrives to one of the device-side memory-mapped queues; (1b) the device triggers the corresponding doorbell; (2a) the software data plane module is informed of the arrival (through the doorbell); (2b) depending on the format/semantics of the work items in the queue, the software data plane either performs transport processing in-place or (2c) writes/copies the transformed packets/work items to the corresponding tenant-side queue; (2d) the software data plane triggers the tenant-side doorbell; and finally, (3) the tenant is informed of the packet/work item arrivals to process. Each tenant has a single or a few queues per (virtual) core. Therefore, it can easily monitor the queue via spin-polling or different variants of the `MWAIT` instruction.

However, the software data plane has to monitor all queues simultaneously and service them based on the predefined system policy, hence it cannot use `MWAIT` variants.

3.2.2 Software Data Plane Challenges and Goals

Even though software data planes rely on spin-polling to deliver high throughput and low latency notification, they suffer from several inherent inefficiencies. First, software data planes lack queue scalability [104]. Spinning cores iterate over all of the input queues at full tilt even when there is no work item in any of them. Increasing the number of queues puts excessive pressure on processor caches, which can hurt peak throughput and tail latency. This effect is exacerbated when traffic lacks balance, i.e., when a subset of queues contain no work items most of the time. Empty queues cost a spinning core time as it searches for the next ready work item in a non-empty queue. This cost is particularly high when interrogating empty queues may incur cache misses, slowing the polling loop. Since the time required to process a work item is usually short (i.e, a few microseconds), missing on multiple empty queue heads might take even longer than processing a ready queue.

Second, software data planes are not necessarily work-proportional. Modern cores can spin with high IPC. Therefore, spin-polling may require a core to perform more work when there are, in fact, fewer work items in the queues. Work disproportionality translates to energy disproportionality [62, 104]. It also has an adverse effect on workloads co-running on Simultaneously Multi-Threaded (SMT) cores. Useless spinning consumes execution resources and L1 cache bandwidth that could otherwise be effectively used by a co-runner hyperthread. Whereas spin-locks also exhibit the same drawback, the collateral damage of a spin-lock is lower because they spin only on a single memory location. Modern cores can easily detect such spin-loops and slow the spinning process [137]. Moreover, as shown by prior work, variants of the `MWAIT` instruction may be used to put the core in halt/sleep state until a write is performed to the lock location to prevent useless spinning and save energy [90].

Finally, scale-up queuing is impractical in software data planes [104, 174]. The scale-up queuing organization, wherein multiple cores fetch work items from a shared set of queues, has strong theoretical advantages compared to scale-out queuing, wherein each core is associated with a different set of queues [160]. First, scale-out designs may suffer from load imbalance as the traffic is usually unbalanced and only a subset of queues have work items—these queues are often non-uniformly distributed among cores [84]. In contrast, scale-up organizations provide an inherent load balancing property as all queues are visible to all cores in a work conserving setting. Second, scale-out organizations are prone to Head-of-Line (HoL) blocking [158, 162]—if the work item at the head of a queue takes longer than average to process, all work items behind it experience long queuing delays, yielding a high tail latency. Scale-up designs, however, are not susceptible to HoL blocking as all queues are visible to all cores—if an item takes long to process, the items queued behind it are drained by other cores. Finally, scale-up organizations provide better support for queue priorities. With scale-out organizations, each core can only prioritize over its own subset of queues. Despite these theoretical merits, software data planes that leverage scale-up queuing suffer from excessive synchronization and coherence performance overheads in practice, as the cores must frequently synchronize to dequeue items and the corresponding cache lines ping-pong among the cores' L1 caches.

Our goal is to design a hardware accelerator subsystem that can efficiently address these shortcomings and provide a scalable, low-latency, and work-proportional notification mechanism to enable high-performance software data planes. We will quantify these inefficiencies and how our proposed design, HyperPlane, can address them in Section 3.5. In the next subsection, we present a case study of DPDK's queue scalability.

3.2.3 Case Study: DPDK Queue Scalability

The Data Plane Development Kit (DPDK) [24] is a representative software infrastructure for building spin-polling–based user-level data planes. DPDK provides highly optimized

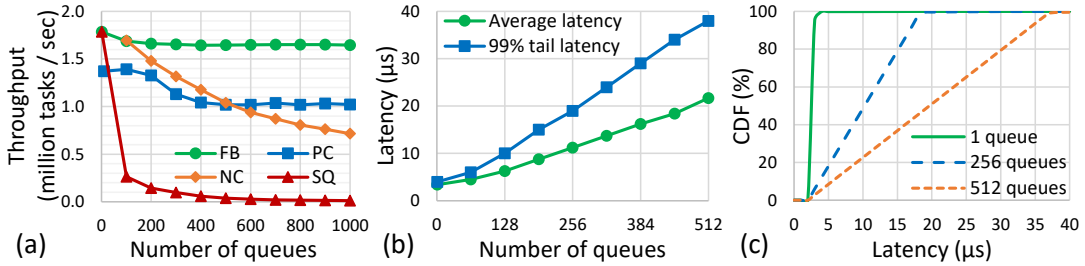


Figure 3.3: DPDK: (a) Throughput of packet encapsulation in DPDK, (b) Round-trip latency of packet forwarding under light traffic (~0.01 Mpps), (c) Distribution of round-trip latency.

poll mode drivers for numerous modern I/O devices—such as NICs and crypto devices—which enable cores to spin on user-level queues to communicate with the devices. Using DPDK, we illustrate the inherent queue scalability challenge of software data planes on a real server with a 24-core Xeon Skylake processor and a 100 GbE Mellanox ConnectX-5 NIC.

We first consider the effect of increasing the number of queues on the maximum achievable throughput of a core performing network packet encapsulation tasks with various traffic shapes: *Fully Balanced* (**FB**), where traffic passes through all the queues; *Proportionally Concentrated* (**PC**), where traffic passes through 20% of the queues all the time and through the rest with a probability of 5%; *Non-proportionally Concentrated* (**NC**), where traffic passes through 100 queues all the time and through the rest with a probability of 5%; *Single Queue* (**SQ**), where traffic passes through only one queue. Figure 3.3(a) shows task execution throughput at different numbers of queues for the mentioned traffic shapes. We observe a drastic drop in throughput with *SQ* traffic. This drop is caused by useless spinning on empty queues, which is exacerbated by cache misses incurred for fetching queue heads. The throughput drop with the *NC* traffic is milder since the ratio of non-empty to empty queues grows at a smaller rate by increasing the number of queues, compared to *SQ*. With *FB* and *PC*, the ratio of non-empty queues to empty queues is constant (i.e., zero and four, respectively). Therefore, the throughput stabilizes as the number of cache misses per task becomes constant. In summary, the throughput of software data planes is adversely affected when traffic is concentrated in a small number of queues, and the rest are usually empty,

which is the common case.

Next, we show how latency is affected by increasing the number of queues. Figure 3.3(b) reports the round-trip latency of a core forwarding packets received from the machine's NIC at different numbers of queues. Latency is measured at a packet generator, which sends/receives packets to/from the machine under test. To avoid queuing delays, we offer minimal load in this test (~0.01 Million packets per second (Mpps)). Therefore, the reported latency is composed of service time (packet forwarding by the core) plus round-trip time. As shown in the Figure, both average and 99^{th} percentile tail latencies grow almost linearly with the queue count because of more cache misses due to reading empty queue heads. Furthermore, tail latency grows with a higher slope—in the tail case, the data plane has to poll over far more empty queues before finding work in a ready queue, compared to the average case. This finding is further illustrated in Figure 3.3(c), which shows the Cumulative Distribution Function (CDF) of latency at three different queue counts. With more queues, the latency distribution spans a wider range, resulting in a larger difference between average and tail latencies.

3.3 HyperPlane Design

In this section, we explore the design of the HyperPlane notification system. HyperPlane seeks to enable efficient software data planes that, unlike spinning-based variants, (1) do not need to iterate over empty queues to find work in ready ones, (2) block/halt when all queues are empty rather than spinning fruitlessly, and (3) allow multiple cores to efficiently monitor a shared set of queues to provide higher performance with strong support for queue priorities and different service policies.

HyperPlane seeks to facilitate the notification/queue selection operation of software data planes (step (2a) in Figure 3.2) in both directions (transmit and receive). It comprises a programming model and a hardware notification subsystem. At a high level, the programming model centers around the `QWAIT` instruction, which waits on a set of queue head

doorbell locations and returns the QID for the next ready queue. The hardware subsystem relies on two key components, the *monitoring set* and the *ready set*. The monitoring set tracks the doorbell locations associated with each queue and observes cache coherence write transactions to these locations, which indicate that a work item has been enqueued. The ready set tracks, orders, and prioritizes queues that are ready to be processed.

3.3.1 Programming Model

The key component of the HyperPlane programming model is the `QWAIT` instruction. `QWAIT` is inspired by x86 `MWAIT` and ARM `WFE` instructions, which monitor a single memory address or a contiguous address range. `MWAIT` halts the execution of a hardware thread and waits until the contents of a specified address range change. Whereas `MWAIT` is a privileged instruction that cannot be used in user applications, Intel has recently introduced a user-mode variant of this instruction, called `UMWAIT`, which can also run in unprivileged code [13]. Nonetheless, the `MWAIT` variants can at best only partially address the work disproportionality of spin-based data planes by blocking execution when all queues are empty and waiting for a work item to arrive in some queue. However, they cannot indicate in which queue the work item is located, requiring the code to iterate across many (likely empty) queues, hurting latency and throughput.

In contrast, `QWAIT` monitors a set of queue doorbell locations and returns the QID of the next ready queue—similar semantics to the *select-case* construct in the Go programming language [47]. Each queue is associated with a doorbell in memory, which is usually a word composed of multiple fields that specify various properties of the I/O queue. We assume a doorbell implementation wherein a field represents an atomic counter, indicating the number of elements in the queue, with similar semantics to a semaphore [190]—producers atomically increment the counter after enqueuing each element and consumers decrement the counter before dequeuing each element. A write from the producer to the doorbell location indicates that an item has been enqueued. These writes typically either trigger interrupts

Algorithm 1: HyperPlane Programming Model

```
1  QWAIT_init(doorbell_addr_range, service_policy)                    // Control Plane
2
3  for all QIDs do
4      do
5          doorbell = allocate_address(doorbell_addr_range)
6      while (QWAIT-ADD(QID, doorbell) == FAIL)
7      doorbell_map[QID] = doorbell
8  end
9
10 while true do                                                      // Data Plane
11     QID = QWAIT()
12     doorbell = doorbell_map[QID]
13     if QWAIT-VERIFY(doorbell) == False then
14         continue
15     end
16
17     work_item = dequeue(QID)
18     QWAIT-RECONSIDER(QID, doorbell)
19     process(work_item)
20 end
21 ──────────────────────────────────────────────────────────────────────
22 QWAIT-VERIFY(doorbell):                                  // Atomic Instruction
23     if is_empty(doorbell) then
24         arm_in_monitoring_set(doorbell)
25     end
26
27 QWAIT-RECONSIDER(QID, doorbell):                          // Atomic Instruction
28     if is_empty(doorbell) then
29         arm_in_monitoring_set(doorbell)
30     else
31         activate_in_ready_set(QID)
32     end
```

(e.g., PCIe MSI-X mechanism) or are polled by the software data planes. By watching all doorbell locations, HyperPlane is able to determine the next ready queue without iterating across them and without the overheads of an interrupt. Algorithm 1 presents the high-level programming model of the HyperPlane architecture, centered around the `QWAIT` instruction. Each HyperPlane thread runs the code presented in Algorithm 1 and is pinned to a physical core to prevent it from being context-switched.

Control plane primitives. These primitives are required to setup and configure the HyperPlane hardware and modify the list of queue doorbells. They are privileged instructions as they need access to physical or kernel memory. Therefore, they are only used in the kernel driver code. `QWAIT_init` is used to initiate the HyperPlane hardware and specify the

address range from which doorbells can be allocated, as well as the service policy—round-robin, weighted round-robin, or strict priority. We will discuss service policies and their implementations in more detail in Section 3.4.2. `QWAIT-ADD` associates a doorbell address with a QID, adds it to the HyperPlane's monitoring set, and *arms* the address to be watched for work arrivals. It is used when a new tenant connects to the data plane. Conversely, when a tenant process terminates, either the tenant itself or the kernel driver must disconnect it from the data plane by removing its QIDs and releasing their space from the monitoring set via `QWAIT-REMOVE`.

Data plane primitives. In the body of the data plane thread, the `QWAIT` instruction is executed in a loop. Similar to `MWAIT`, the `QWAIT` instruction halts a hardware thread's execution if all queues are empty and waits for a work item to arrive in some queue. By halting, `QWAIT` prevents useless spinning and the consequent work disproportionality. A core may also enter a power-optimized mode to save more energy if all hardware threads are halted. When work arrives, `QWAIT` returns the QID of the ready queue. If multiple queues already have ready work items when `QWAIT` is executed, it returns the QID of the queue that should be serviced first according to the selected service policy, specified via `QWAIT_init`. The returned QID can then be used to service the corresponding queue. Hence, using the `QWAIT` instruction, HyperPlane does not waste time interrogating empty queues to find work, and immediately moves on to the next ready queue to be serviced.

The two blue highlighted parts of the code in Algorithm 1 are required for the correctness of the hardware implementation and do not impact the high-level semantics of the code. Prior to servicing the returned QID, a `QWAIT-VERIFY` instruction is called to check whether the returned QID is in fact ready. `QWAIT-VERIFY` atomically performs two functions: (1) it indicates whether the queue is empty (i.e., by checking the value of the doorbell's atomic counter), and if it is, (2) re-arms it in the monitoring set to detect the arrival of subsequent work items. This instruction is needed to detect potential spurious wake-ups or QID returns—i.e., a returned QID might not necessarily correspond to a ready queue with

available work items (e.g., due to false sharing). After the work item has been dequeued, the QWAIT-RECONSIDER instruction is called, which either re-arms a QID in the monitoring set or re-activates it in the ready set (we will discuss these structures later) based on whether additional work items are queued, and is atomic with respect to new work item arrivals. QWAIT-VERIFY and QWAIT-RECONSIDER are both atomic instructions with memory barrier semantics, to prevent the execution from advancing before their operation is complete. We will explain these instructions in more detail in the next subsection.

Whereas QWAIT provides work proportionality by halting execution and avoiding fruitless spinning when all queues are empty, it might be desirable to execute a latency-insensitive task on the core when it is waiting for work items to arrive. This can be achieved in two different ways: (1) QWAIT can provide a non-blocking variant, which returns a reserved QID immediately even if there is no ready QID in the ready set. This way, the code performing a background task might poll the entire ready set with a single QWAIT instruction to see if any work item has arrived. (2) A background task may run on the second hyperthread of the core, which can efficiently use the core resources while the QWAIT thread is halted. To ensure the background task does not hurt data plane performance, the core may prioritize its SMT threads—using mechanisms proposed by prior work [88]—and only execute instructions from the low-priority background thread when the high-priority foreground thread is halted. QWAIT may be used as the signaling mechanism to detect when the data plane thread is halted, waiting for work to arrive.

Finally, we also envision two additional primitives, QWAIT-ENABLE and QWAIT-DISABLE, which may be used by the service procedure of a queue to temporarily inhibit a queue being serviced despite having ready work items. If a queue is disabled via QWAIT-DISABLE, its QID will not be returned until its service procedure is re-enabled via QWAIT-ENABLE (e.g., by timer). An example use case of these primitives is to limit the processing rate of a queue for a period for, e.g., congestion control in networking applications [151].

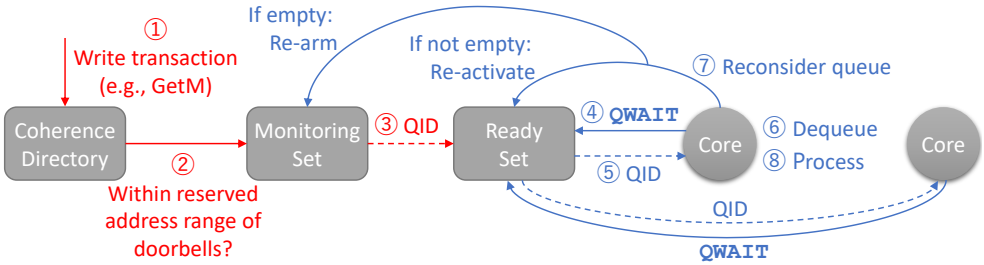


Figure 3.4: High-level hardware block diagram of HyperPlane.

3.3.2 Hardware Components

Figure 3.4 depicts the HyperPlane hardware block diagram. HyperPlane's operation is orchestrated by two hardware components: the *monitoring set* and *ready set*. At a high level, the monitoring set snoops the write transactions to a reserved address range dedicated to doorbells (steps 1 and 2). If a write transaction matches a QID in the monitoring set, it disarms the entry and activates the QID in the ready set (step 3). At this point, when a data plane core executes the `QWAIT` instruction (or has been blocked on it) (step 4), it will be able to return the corresponding QID, according to the service policy. When a QID is returned (step 5), the data plane core dequeues/locates a single or a batch of work items (step 6), and signals HyperPlane to reconsider the queue by either re-arming it in the monitoring set, if the queue is empty, or re-activating it in the ready set, if it is non-empty (step 7). Finally, the data plane core performs transport processing for the work item (step 8) and signals the tenant, prior to re-executing the `QWAIT` instruction (step 4). In the rest of this section, we explain the detailed functionality and interactions of the monitoring and the ready sets. We will later explore the detailed microarchitectural implementation of these components.

The monitoring set observes the doorbell memory addresses and detects work arrival by snooping the cache coherence write transactions to a specific pinned address range, reserved by the kernel driver for I/O queue doorbells. Any coherence transaction that grants exclusive ownership of a cache line to the requester will cause the monitoring set to indicate a wake-up/arrival on the corresponding queue (e.g., GetM transactions in the generic coherence protocols described in [195]). The monitoring set is independent of the

coherence organization and is able to snoop messages either at a bus or directory. At a high level, the internal structure of the monitoring set is similar to a large associative memory that maps cache line tags to QIDs.

A monitoring set entry is composed of the following fields: tag, QID, monitoring bit, valid bit. The monitoring bit indicates that a cache line is "armed", being watched for write transactions. The monitoring set snoops all incoming write transactions, and if their tag matches an entry, it disarms the entry (i.e., sets the monitoring bit to 0, to indicate the line is no longer being watched), and activates the associated QID in the ready set. The `QWAIT-ADD` instruction is used to add a new entry to the monitoring set, e.g., when a new tenant connects to the data plane. The entries may later be removed via the `QWAIT-REMOVE` instruction. `QWAIT-VERIFY` and `QWAIT-RECONSIDER` instructions are used to re-arm an entry in the monitoring set. When an entry is re-armed, a coherence read transaction (i.e., GetS) is issued to ensure the line has no owner and the writes cannot be performed locally.

Although the `QWAIT-VERIFY` instruction filters out spurious writes, it is desirable that only doorbell writes performed by a producer (not the data plane thread itself) signal a QID in the monitoring set. Due to the memory barrier semantics of the `QWAIT-RECONSIDER` instruction, it is not issued before the dequeue operation (line 17 in Algorithm 1) is completed. Therefore, potential write transactions issued by the dequeue operation (i.e., decrementing the doorbell counter) do not trigger any QID in the monitoring set, since the corresponding entry is not armed during the dequeue operation. Note that once an item arrives to an armed queue, its entry in the monitoring set is disarmed, and further arrivals have no effect in the monitoring set until the queue is armed again (via `QWAIT-RECONSIDER`). When a QID is returned by the `QWAIT` instruction, the dequeue operation can retrieve a batch of items provided it correspondingly decrements the doorbell counter. Furthermore, note that Algorithm 1 seeks to deliver maximum intra-queue concurrency in multicore data planes, to eliminate potential HoL blocking scenarios and improve tail latency (see Section 3.2). However, in various flow-based stateful networking applications—such as

TCP/IP processing—packets or work items have to be processed in order [60], and intra-queue concurrency is not allowed. In such cases, lines 18 and 19 should be swapped to ensure a queue may only be serviced again when its previous work item has been processed.

The ready set is responsible for returning the QID of the next ready queue upon `QWAIT`, according to the selected service policy. Conceptually, it is composed of a list of the QIDs with available work items and an iterator that searches over the list and finds the next ready QID according to the service policy. For example, in the case of round-robin policy, the iterator searches over an unsorted list of QIDs to find the first one after the last serviced QID in a circular order. The ready set and its iterator may in principle be implemented either in hardware or in software. In case of a software implementation, the iterator code would be embedded into the `QWAIT` function (`QWAIT` would no longer be a single atomic instruction). However, in this case, in addition to the complications of providing atomicity, with fully- or semi-balanced traffics, the iterator code may need to iterate over potentially ~1k QIDs in the list, adding a significant runtime overhead to the data plane performance, which may even be longer than the time required for processing an individual work item after locating it.

The `QWAIT-VERIFY` instruction in Algorithm 1 ensures that the returned QID indeed corresponds to a queue with ready work items. In case the queue is empty, its QID is atomically re-armed in the monitoring set. This instruction filters spurious wake-ups/activations—due to exclusive reads, false sharing, or doorbell writes that do not correspond to work item arrivals—while ensuring there is no window of opportunity for actual work arrivals to be missed. The `QWAIT-RECONSIDER` instruction in Algorithm 1 arranges for a QID to be considered again for service in a future iteration if it has ready work items. It atomically checks whether the queue is empty or already has work items available (e.g., more work items have arrived while the QID was waiting in the ready set). If the queue is empty, its entry is re-armed in the monitoring set. If it already has work items, the QID is directly activated in the ready set, so the iterator will select it again for service according to the service policy. The entire `QWAIT-RECONSIDER` operation must be implemented atomically to

prevent various possible data races, including a scenario wherein the queue tests empty but a work item arrives before the QID is re-armed in the monitoring set, leading to a missed write transaction and consequent missed wake-up/activation.

When running the data plane software on multiple cores to distribute the load, two options might be considered, as described in Section 3.2: scale-up and scale-out. The scale-up organization is theoretically preferred as it achieves better throughput and latency, since it is not susceptible to load imbalance and HoL blocking. It also provides better support for queue priorities and weights, as it makes all queues visible to all cores. However, despite the theoretical advantages of scale-up queuing, scale-out organizations are often implemented in practice due to the synchronization overheads of scale-up organizations in spinning data planes. In contrast, HyperPlane enables efficient scale-up queuing organizations as the monitoring and ready sets are shared across the chip, accessible by all of the cores running the data plane software. As we will show in Section 3.5, HyperPlane enjoys optimal latency and throughput characteristics, as it relies on scale-up queuing.

In case of NUMA systems, we envision a multi-socket HyperPlane deployment to employ separate data plane cores on each socket, with NUMA affinity between tenants and the data plane cores (e.g., doorbells mapped to the local memory channel address space) to avoid costly inter-socket communication. In this case, the deployment would exhibit scale-out queuing properties across sockets, which may result in suboptimal performance if there is heavy load imbalance across sockets. To mitigate this issue, a work-stealing approach may be used, wherein the data plane cores fetch ready QIDs from remote ready sets if the local ready set is empty. We defer exploration of such mechanisms to future work.

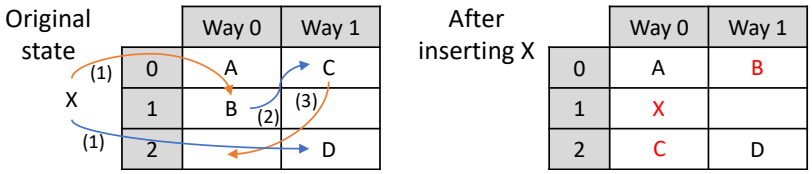


Figure 3.5: An example 2-way Cuckoo hash table insertion.

3.4 Detailed Microarchitecture

3.4.1 Monitoring Set

Conceptually, the monitoring set provides a fully associative key-value lookup functionality, wherein the cache line tags for the doorbell locations of all QIDs are watched simultaneously. However, large fully associative hardware structures are costly in terms of area and especially energy, due to performing many comparisons in parallel. Alternatively, serializing the comparisons significantly increases lookup latency [165]. Set-associative structures can suffer from high conflict rates, unless associativity is high (e.g., 64 or higher), which makes them almost as costly as fully associative structures. We propose to use a ZCache-like structure [187] to overcome these challenges.

The key idea behind ZCache operation is Cuckoo hashing [175], which decouples ways and associativity, enabling a highly associative structure with low lookup latency and energy. Figure 3.5 depicts a simple Cuckoo hash table with two ways. Each key is hashed using two hash functions, H0 and H1. When key X is looked up, it is hashed into row 1 in way 0 and row 2 in way 1, but both of those rows are occupied by other keys (i.e., a miss). To insert key X, way 0 is picked, and key B that is currently occupying the location where X must be inserted is rehashed using H1 and moved to way 1. By moving B to row 0 of way 1, C will be rehashed by H0 and moved to row 2 of way 0. The "table walk" process terminates at this point as C is placed in an empty location. If no empty location is found, despite the table containing empty entries, a *conflict* is said to have occurred.

As illustrated in the example, to look up a key, Cuckoo hashing need only check as many locations as the number of ways. However, by walking the table and performing a chain

of replacements, the scheme provides a high effective associativity, and thus low conflict rate. Even though lookup latency and energy of this scheme are low, insertions can take substantially more time and energy than a conventional set-associative cache. However, insertions are only performed by `QWAIT-ADD` instructions, which are only executed when a new tenant is connected to the data plane (i.e., second or minute time scales). When an entry is disarmed and then re-armed in the monitoring set, it is not evicted and then re-inserted, but instead only its monitoring bit is set/reset. As a result, the costly hash table walks for insertion are only performed once for each `QWAIT-ADD` instruction. Re-arming a QID in the monitoring set via `QWAIT-VERIFY` or `QWAIT-RECONSIDER` instructions only involves a single tag lookup, similar to snooping the incoming transactions.

When a new tenant wishes to connect to the data plane, it executes the `QWAIT-ADD` instruction for every *<QID, doorbell address>* pair. Whereas Cuckoo hash tables exhibit much lower conflict rates than typical set-associative structures, conflicts are still possible. If `QWAIT-ADD` fails to insert a QID into the monitoring set due to a conflict, it returns an error code, and invokes driver code to reallocate a different doorbell address to the QID. Nonetheless, to minimize the conflict rate and ensure doorbell address reallocations are rare, the monitoring set (Cuckoo hash table) may be over-provisioned with respect to the maximum number of supported doorbells. Prior work has shown that over-provisioning the size of a Cuckoo hash table by 5%-10% reduces the conflict rate down to 0.1% [187], which is negligible. Note that after a new QID is added to the monitoring set, it stays there conflict-free, and is only removed by an explicit `QWAIT-REMOVE`.

Because the monitoring set snoops all coherence transactions to the doorbell memory address range, it is not subject to the conflict replacement behavior of a directory-based coherence scheme. In most directory-based schemes, when an entry is evicted from the directory, it sends invalidations to all the sharers of the line. However, the monitoring set is not an explicit sharer, but rather snoops all relevant coherence transactions (i.e., conceptually implemented as part of the directory). Therefore, it retains all the monitored doorbell tags,

even in case of evictions in the directory. When an external write is about to be performed on the doorbell, the directory has be informed via a write transaction (i.e., GetM), which also informs the monitoring set. Since doorbells are allocated from a restricted address range (managed by the HyperPlane kernel driver), the monitoring set only need snoop addresses in this range and the snooping bandwidth is tractable. In the case of distributed directories, the monitoring set must also be banked, attached to individual directory banks. In such cases, the driver must spread doorbell addresses across banks.

3.4.2 Ready Set

When the monitoring set matches a coherence transaction to a monitored doorbell, it disarms the entry and activates the QID in the ready set. The main responsibility of the ready set is to determine the next QID to be returned by `QWAIT`, according to the service policy. Whereas the monitoring set must be implemented in hardware (since coherence transactions are not visible to software), the ready set may in principle be implemented in software or hardware. In a software implementation, an iterator traverses a list of ready QIDs to find the next QID to be processed based on the service policy. However, in fully- or semi-balanced traffic scenarios where most queues are non-empty, the code must iterate over a large number of QIDs, imposing a substantial runtime overhead.

Instead, we propose a hardware implementation for the ready set, presented in Figure 3.6. Our hardware implementation takes as input a bit vector representing "ready bits" that correspond to different QIDs. That is, when a QID is returned by the monitoring set, the corresponding ready bit is set. As shown in the Figure, there is also a "mask bits" vector, which filters ready bits that should not be returned. These mask bits are manually set/reset via `QWAIT-ENABLE` and `QWAIT-DISABLE` to temporarily disable queues. The ready set hardware produces the "select bits" vector as its output, which is encoded in a "one-hot" fashion—that is, at most one of the bits can be set, indicating the selected QID to be returned by `QWAIT`. When the `QWAIT` instruction is executed, it is the ready set's responsibility to

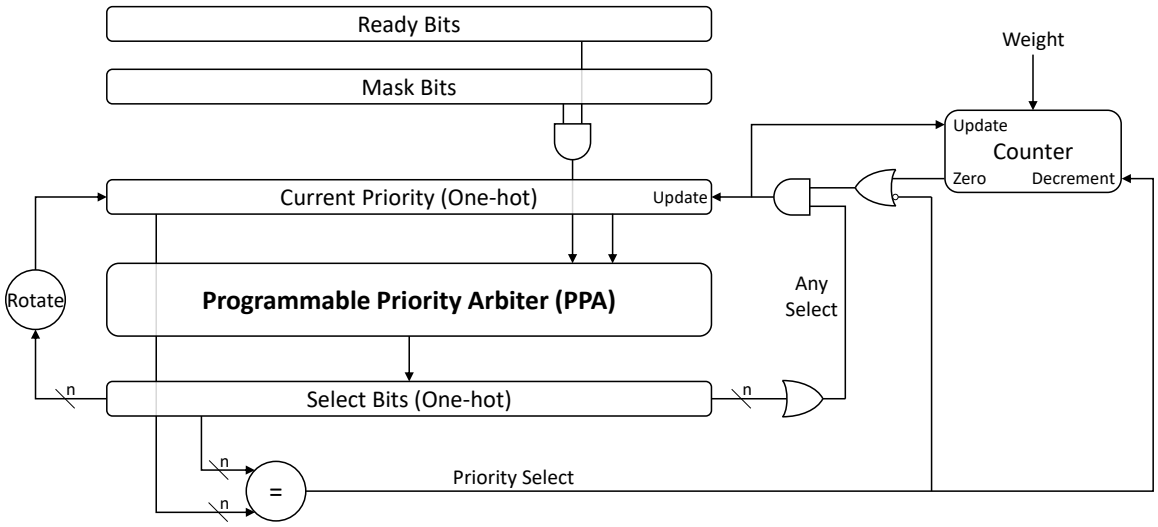


Figure 3.6: High-level block diagram of the ready set hardware.

compute the "select bits", based on the the ready bits and the service policy.

The core of our ready set hardware implementation is a Programmable Priority Arbiter (PPA)—a widely used building block in on-chip networks and switching devices to grant access to one of the many requesters of a shared resource [85]. Besides "ready bits", the PPA module also takes a "current priority" one-hot bit vector as an input. The only bit position set to one in the current-priority bit vector indicates the QID with the highest priority. If that QID is ready, it is selected. Otherwise, its priority is propagated to the next bit position, wrapping around, until a ready QID is found.

To explain the operation of PPA, Figure 3.7 presents the ripple-priority bit-slice implementation of the PPA module, which is one of its simplest implementations. Its operation is similar to a ripple-carry adder. As shown in the Figure, at each bit position, the hardware checks (1) whether the ready bit is set to one and (2) whether priority is given to that bit position (via the one-hot *Priority* input or from a previous bit position via P_{in}). If both conditions are met, the corresponding select bit is set. Otherwise, if *Priority* or P_{in} is asserted, but the ready bit is not set, priority is propagated to the next bit position via P_{out}. Ripple-priority implementation of PPA results in linear delay and hardware complexity. Furthermore, as shown in Figure 3.7, it requires a "wrap-around" connection that results in a combinational

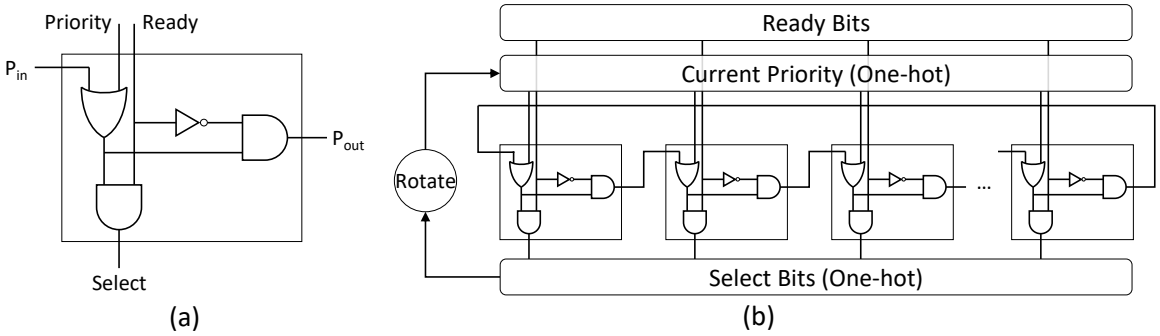


Figure 3.7: (a) A bit-slice Programmable Priority Arbiter (PPA) cell, and (b) a multi-bit ripple-priority PPA design.

loop, making it difficult for EDA tools to synthesize and analyze the hardware.

In contrast, modern PPA implementations use thermometer coding [106] to eliminate the wrap-around connection and Parallel Prefix Network (PPN)-based designs to reduce the delay complexity of priority propagation to logarithmic [94]. PPNs are enhanced variants of look-ahead designs—such as carry look-ahead adders—and are used in almost all state-of-the-art high-speed adders [63]. PPNs provide a better hardware complexity vs. latency trade-off, compared to naïve carry look-ahead designs, making them scalable to thousands of bits [183]. In our implementation, we employ a Brent-Kung PPN [71], which is optimized for hardware complexity to be scalable to high bit counts—our RTL analysis shows the latency and hardware costs of the ready set to be small. We will provide a detailed analysis in the next subsection.

Our proposed ready set hardware design can efficiently implement the three most common service policies. With a *round-robin* policy, the selected QID in each round must exhibit the lowest priority in the next round. Thus, as shown in Figure 3.6, if the "Any Select" signal is set to one, indicating there was a QID selected at this round, the current-priority bit vector will be the rotated version of the select bit vector to give the highest priority to the bit position next to the one corresponding to the currently selected QID. The *weighted round-robin* policy is a generalization of round-robin, which allows each queue to be serviced for multiple consecutive rounds once it is selected. By giving different weights

to different queues, weighted round-robin accommodates the differentiated arrival rates and QoS requirements of various tenants. In this case, when the current-priority bit vector is reloaded, the corresponding weight for the current-priority QID is loaded into a counter. Every time the queue is serviced, the counter is decremented. When the counter reaches zero or the current queue runs out of work items, the priority is passed to a different QID by reloading the current-priority and the weight register. Finally, by fixing the value of the current-priority bit vector to "10...0", the hardware implements a strict priority policy, wherein lower-numbered QIDs are always prioritized over higher-numbered ones. However, this policy is usually not used in real applications as it would result in the starvation of low-priority queues; instead, a weighted round-robin policy is often used, which differentiates queue priorities through weights and avoids starvation.

3.4.3 Hardware Costs

We have considered a 1024-entry banked monitoring and unified ready set, shared across 16 cores. We modeled the hardware costs of the ready set via an RTL implementation in 32nm technology, and derived the area, power, and timing estimates for the core and the monitoring set via CACTI [165] and McPAT [136] models. Note that, during normal data plane operation, the monitoring set is similar to the tag array of a 2-way associative cache in terms of latency and energy since arming/disarming QIDs only involves 2-way lookups—the table walk process is performed only once for each `QWAIT-ADD` instruction. Synthesis of our RTL design reports the area of a 1024-entry ready set to be 0.13 mm^2. We estimate the area of the monitoring set to be 0.21 mm^2, while our baseline core occupies 8.4 mm^2 of area. Hence, the overall area overhead of the HyperPlane hardware components is within 0.26% of the total core area, for a 16-core chip. Similarly, we estimate the power costs of HyperPlane to be within 0.4% the total core power (within 6.2% of a single core; 2.1% for the ready set and 4.1% for the monitoring set). Note that our analysis considers 16 cores but does not include the uncore area/power. Thus, full-chip overheads are even smaller.

Table 3.1: Microarchitecture details.

Core	8-wide issue OoO, 192/32-entry ROB/LSQ
L1 I/D	Private, 32 KB, 64B lines, 4-way SA
LLC	1 MB per core, 64B lines, 16-way SA
CMP	16 cores, directory-based MESI coherence
HyperPlane	1024-entry monitoring and ready set

From a timing perspective, our RTL model reports the latency of the ready set to be 12.25 ns. Since processing each work item takes a few microseconds, the ready set can easily serve `QWAIT` requests from O(100) cores—the number of cores needed for software data planes is usually small (1-4 [151, 174]). We have considered the lookup latency of the monitoring set to be within 5 CPU cycles but this latency is not on the critical path of the `QWAIT` instruction, unless it is halted, waiting for arrivals. To simplify the complexities of non-uniform access latencies of different cores to the single unified ready set, we have conservatively considered the `QWAIT` instruction latency to be 50 cycles in our experiments, which is higher than sum of all the latencies involved.

3.5 Evaluation

3.5.1 Methodology

We use the gem5 simulator [65] and augment it to model the hardware components of HyperPlane. We model a 16-core x86-64 CMP system of which our data plane software runs on 1-4 cores [151, 174]. We model the core power consumption using McPAT [136]. Experimental microarchitecture details are described in Table 3.1. We use an in-house software data plane system based upon DPDK [24] that is able to be run in the simulator. Our software data plane infrastructure closely tracks the performance characteristics of DPDK. Producer and consumer cores communicate through lock-free task queues. Emulated I/O sources running on "producer" cores generate traffic with different shapes and loads, which is passed through the data plane. Traffic shapes are the same as those used in Section 3.2: *Fully Balanced* (**FB**), *Proportionally Concentrated* (**PC**), *Non-proportionally*

Concentrated (**NC**), and *Single Queue* (**SQ**). We only report results for the round-robin service policy, as we found the service policy to have minimal impact on the performance trends. We evaluate our software data plane framework using the following tasks:

- **Packet encapsulation**: Network tunneling protocols leverage packet encapsulation to enable data movement of a network (e.g., a private network) over another network (e.g., a public network). We use the GRE (Generic Routing Encapsulation) protocol [92] to encapsulate IPv4 packets within IPv6 packets.
- **Crypto forwarding**: Network traffic is often encrypted for secure communication. In this task, network packets are encrypted through AES-CBC-256 (Advanced Encryption Standard in Cipher Block Chaining mode) [99].
- **Packet steering**: Cloud providers practice various work distribution mechanisms to avoid datacenter network traffic congestion and scale network performance [40, 60]. We employ a packet steerer that redirects the traffic by obtaining a session affinity from a hash table.
- **Erasure coding**: Erasure codes are commonly used in storage applications to detect and/or correct errors in stored data [134]. We use Reed-Solomon erasure coding to encode data blocks/fragments using a Cauchy matrix.
- **RAID protection**: RAID (Redundant Array of Independent Disks) is another mechanism for storage fault tolerance. In this task, RAID with P+Q redundancy is used to calculate parity bytes of input data blocks [77].
- **Request dispatching**: Online data-intensive applications dispatch microservices between servers at different tiers [200]. Our dispatcher task identifies request types and prepares the remote procedure calls to be dispatched.

3.5.2 Queue Scalability

Peak throughput. We first characterize the peak achievable throughput for different numbers of queues to evaluate the queue scalability of HyperPlane and alternative spinning data planes. The peak throughput of the spinning data plane at different numbers of queues

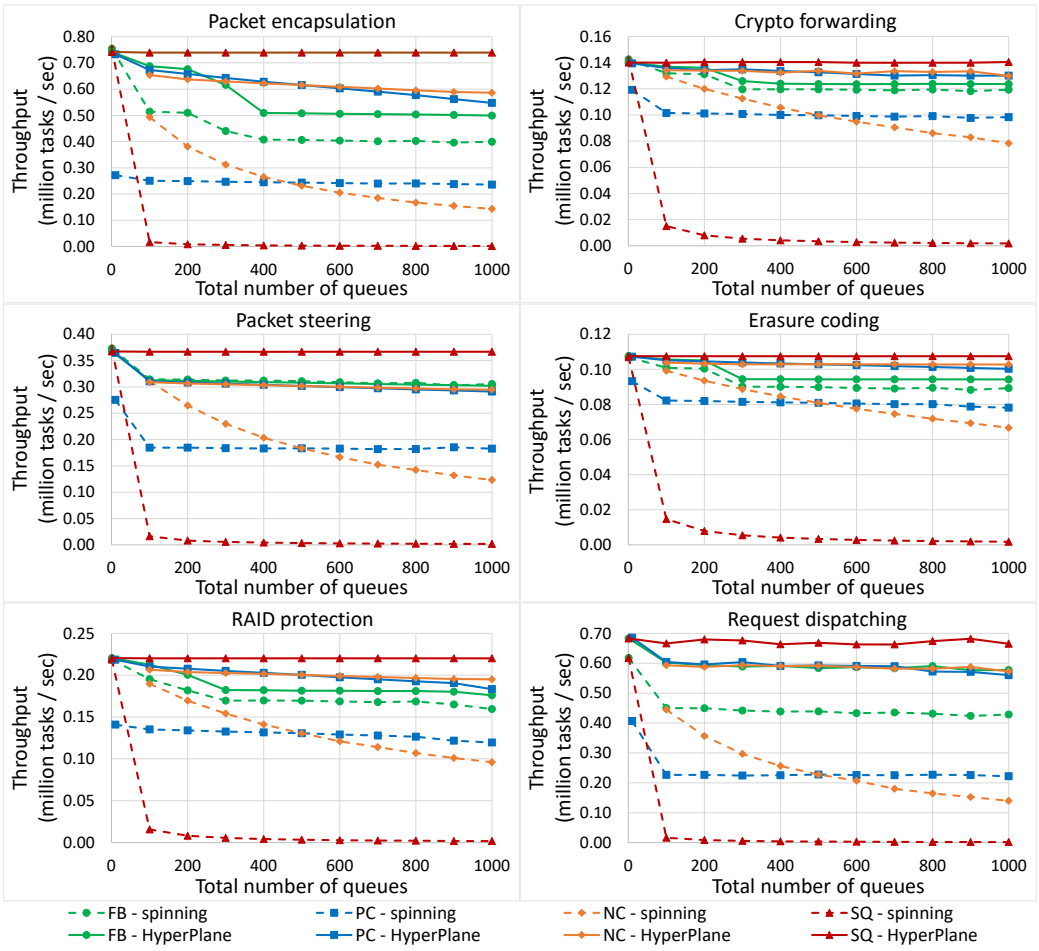


Figure 3.8: Peak throughput of a spinning data plane and HyperPlane.

on a single core is reported in Figure 3.8 for the various workloads. Consistent with Section 3.2, the throughput drop is the most drastic with the *SQ* traffic and is milder with the *NC* traffic since the core needs to spin-poll a larger number of empty queues before finding work in ready ones. With more queues, executed tasks use more data buffers in total, and as a result, we also observe a throughput decrease with the *PC* and *FB* traffics when the total size of task data and queue metadata exceeds the LLC size. However, since a task is executed for every n queue head polls ($n \approx 5$ for *PC* and $n = 1$ for *FB*)—each incurring a queue head cache miss at larger queue counts—throughput converges to a constant value with these two traffic shapes.

Figure 3.8 also reports the peak throughput achieved by HyperPlane. HyperPlane avoids the useless work of interrogating empty queues and the corresponding cache misses. Thus,

it recovers the lost throughput of the spinning data plane caused by the empty queues. In the *SQ* and *NC* traffics, where the number of active queues is constant (1 and 100, respectively), HyperPlane maintains its peak throughput when the total number of queues is increased. In the case of packet encapsulation, however, we observe a slight decrease in throughput for the *NC* traffic, due to an increase in the total data size of tasks and queues when the queue count increases. HyperPlane also exhibits a slightly larger throughput decrease for the *PC* and *FB* traffics, again due to the larger total data size of tasks and queues with more queues. Under the *FB* traffic, HyperPlane achieves better peak throughput in comparison to the spinning data plane in the case of packet encapsulation and request dispatching. Whereas the offered load fully saturates the processing capacity of the data plane core, empty queues are still occasionally observed, as our arrivals follow a Poisson process (memoryless inter-arrival times), which exhibits transient load variability. Thus, HyperPlane improves the peak throughput in the *FB* traffic particularly for shorter workloads, where the processing time of work items is more comparable to missing on empty queue heads in the spinning data plane. Overall with different traffic shapes and queue counts, HyperPlane improves the peak throughput by 4.1×, on average, compared to the spinning data plane.

Zero-load latency. Figure 3.9 reports the zero-load latency across workloads as the queue count is increased. Traffic is set to be very light ($< 1\%$ load) to avoid queuing delays. With the spinning data plane (Figure 3.9(a)), both average and tail latencies grow linearly as the number of queues is increased, because the core has to check more empty queues (and possibly incur cache misses) before finding work in the ready queue. Consistent with Section 3.2, the difference between tail and average latency grows with the queue count as the latency variation is higher with more queues—tail latency represents a worst case, wherein the iterator code has to traverse almost all queues before it reaches the ready one. Using HyperPlane, in contrast, the core avoids additional latency of checking empty queues. As a result, HyperPlane is perfectly queue-scalable, and neither average nor tail latency is affected with more queues, as depicted in Figure 3.9(b) (tail latency is not shown

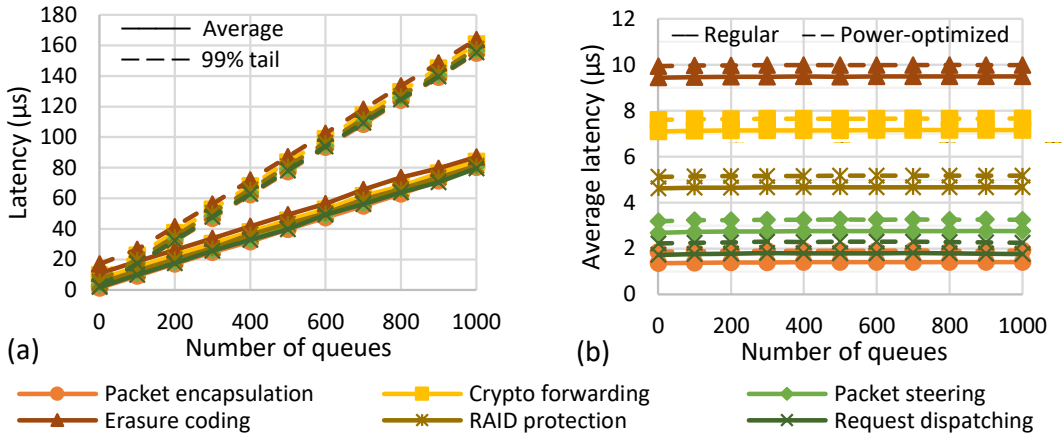


Figure 3.9: Latency under light traffic ($<$ 1% load): (a) Average and tail latency of a spinning data plane, (b) Average latency of HyperPlane in regular and power-optimized modes.

for HyperPlane as it does not differ significantly from the average latency at zero load). Whereas tail latency can be more than 100 µs for large queue counts in the spinning data plane, HyperPlane keeps both average and tail latencies below 10 µs even at 1000 queues. HyperPlane improves average/tail latency by 9.1$\times$ / 16.4$\times$, on average, at different queue counts. Note that with one queue, the core in the spinning data plane quickly finds a task in the queue upon its arrival. Nonetheless, due to the latency of the monitoring and ready sets (Section 3.4.3), HyperPlane underperforms the spinning data plane by at most 3% for a single queue. However, as the latency of the spinning data plane grows with queue count, HyperPlane outperforms the spinning data plane with as few as two queues.

HyperPlane may enter a power-optimized mode when it is idle and all queues are empty. Power saving in the idle state introduces an additional wake-up latency, which we will discuss in Section 3.5.4. The spinning data plane may outperform HyperPlane because of such a wake-up latency for small numbers of queues. Figure 3.9(b) reports the average latency of HyperPlane at zero load with a wake-up latency of ~0.5 µs (transitioning from C1 to C0 state). Our experiments show that because of this additional latency, the spinning data plane reacts faster to task arrival in comparison to HyperPlane for up to 6 queues on average (nine queues in the worst case). With more than six queues, even the power-optimized HyperPlane outperforms the spinning data plane.

3.5.3 Multicore Performance

In this section, we compare the performance of HyperPlane and spinning data planes under multicore scenarios. To provide a comprehensive analysis under the entire load spectrum, we report only results for the packet encapsulation workload. Other workloads follow the same performance trends. Figure 3.10 reports the 99th percentile tail latency under (a) *FB* and (b) *PC* traffics with four cores and 400 total queues. We report latency under the following configurations: scale-out, where each core is statically assigned 100 queues to serve; scale-up-2, where each 2-core cluster is assigned 200 queues; and scale-up-4, where all four cores share all 400 queues. Note that in HyperPlane, there is a single ready set shared among all serving cores. To fairly compare HyperPlane with the spinning data plane, we assume the ready set is partitioned in the scale-out and scale-up-2 configurations and only returns QIDs that belong to a core's subset of queues when the core executes `QWAIT`. In practice, any core can serve any ready queue in HyperPlane, as in the scale-up-4 configuration.

We make two key observations in Figure 3.10(a) for *FB* traffic: First, whereas a scale-out HyperPlane system does not considerably increase the saturation throughput compared to the spinning alternative, it significantly reduces the tail latency under pre-saturation loads (e.g., by 3.2$\times$ under 50% load). At lower loads, the expected number of empty queues the spinning data plane interrogates is higher, but this number reduces to zero at 100% load, and hence the performance of both designs converge. Second, whereas scale-up designs improve HyperPlane latency, especially at high loads, due to their queuing model advantages (see Sections 3.2), spinning alternatives experience significant performance drops due to (1) synchronization and cache coherence (ping-ponging of queue heads) costs, and (2) the expected number of empty queues traversed increases with the total number of queues—that is, each core in the scale-up-4 design iterates over 400 queues (compared to 100 in scale-out) and is likely to interrogate 4$\times$ more queues every time it looks for work.

For the *PC* traffic, Figure 3.10(b) compares the tail latency of the scale-out and scale-up-2

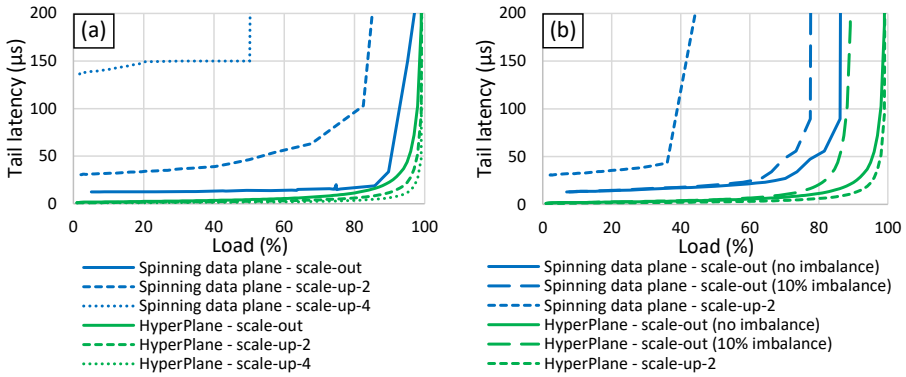


Figure 3.10: Multicore 99% tail latency: (a) Fully balanced traffic, (b) Proportionally concentrated traffic.

organizations as well as a variant of scale-out with 10% static load imbalance. As explained in Section 3.2, the scale-out organization is susceptible to load imbalance. Whereas for *FB* traffic, load imbalance might occur only dynamically, depending on the instantaneous availability of work items in queues, non–fully-balanced traffics, such as *PC*, are also subject to static load imbalance, wherein active queues are not assigned to cores in a balanced manner. Even though the runtime system may detect such load imbalance scenarios and reassign queues to cores, it can only react to traffic and workload changes at coarse-grain time scales—in practice, load imbalance of at least 10% is inevitable.

We make two observations in Figure 3.10(b): First, unlike *FB* traffic, HyperPlane increases the saturation throughput by 23% compared to the spinning data plane, in addition to improving tail latency under pre-saturation loads by at least 73% for *PC* traffic. Moreover, load imbalance is inevitable in real scenarios. The scale-up HyperPlane is not subject to load imbalance and improves the saturation throughput by 11% and 37% compared to scale-out HyperPlane and scale-out spinning data plane both with 10% load imbalance, respectively. However, the scale-up spinning data plane exhibits 54% lower saturation throughput, even compared to its scale-out alternative with 10% load imbalance, due to synchronization overheads.

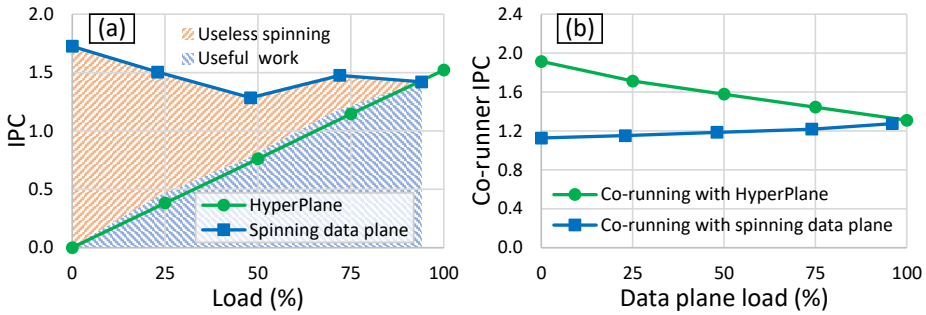


Figure 3.11: (a) IPC breakdown of a software data plane, (b) IPC of an application co-running with the software data plane.

3.5.4 Work Proportionality

HyperPlane is designed to avoid the useless spinning of software data planes and only execute when there is work in the system—that is, it halts execution when there is no work item in any queue. We quantify work proportionality of HyperPlane with respect to the data plane load. Figure 3.11(a) reports the Instructions Per Cycle (IPC) of a core running a packet encapsulation data plane. In HyperPlane, IPC—which is a measure of core activity—grows linearly with load. In contrast, when using a spinning data plane, the IPC is disproportionate to the amount of load and decreases as the load increases. The IPC of the spinning core is the highest at 0% load, meaning that the core spins full-tilt, desperately looking for work. Figure 3.11(a) divides the IPC based on performing useful work or useless spinning for the spinning data plane. At zero load, all the committed instructions are useless, and useful instructions gradually grow by increasing the load. Whereas the IPC of the spinning data plane generally decreases at higher loads, we observe an anomaly at loads above 50%. This anomaly arises because queue heads start to fall out of the L1 cache at higher loads, slowing the IPC of spinning.

The high IPC of useless spinning can harm system efficiency and restrict the performance of other applications. In particular, it has an implication on the applications co-located with the data plane through SMT. Scheduling execution resources among competing hyperthreads is typically performed based on thread activity or IPC (e.g., the ICOUNT policy [212]),

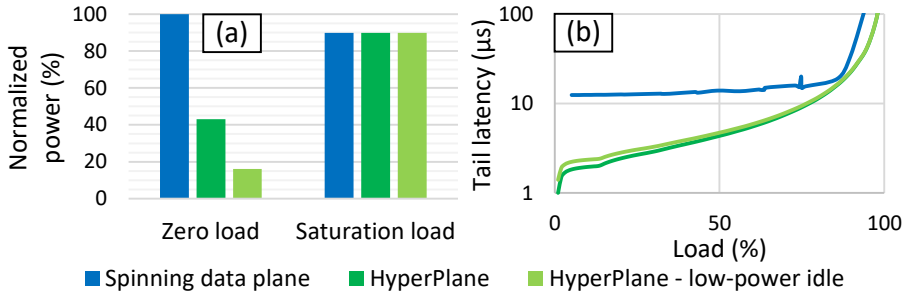


Figure 3.12: (a) Power consumption of a spinning data plane and HyperPlane with/without power optimization, (b) The effect of wake-up latency of power-optimized HyperPlane.

which is counterproductive for idle poll loops. We quantify interference of the spinning data plane as well as HyperPlane with an SMT co-runner, which is a regular application performing matrix multiplication, on a core with two hardware threads. Figure 3.11(b) reports the IPC of the co-runner at different loads of the software data plane. Interestingly, when the spinning data plane is used, the co-runner IPC increases with the data plane load—spinning is a more severe antagonist than performing actual work. With HyperPlane, however, the co-runner IPC decreases when data plane load increases. This again implies work proportionality of HyperPlane. HyperPlane does not interfere with a co-runner when there is no work.

Work disproportionality in spinning data planes also results in energy disproportionality of the core. We use McPAT [136] to model the core power consumption running the software data plane. Figure 3.12(a) reports the normalized power consumption of the core at zero and saturation loads. Perhaps surprisingly, the spinning data plane consumes more power at zero load compared to saturation. This is consistent with the previous observation of the disproportional IPC at zero load due to full-tilt useless spinning (see Figure 3.11). HyperPlane, however, exhibits higher energy proportionality. Whereas HyperPlane already consumes much less power at zero load by halting the execution, it can also enjoy a power-optimized mode, wherein the core enters a deeper "C state" to save power. We only consider transitioning from C0 to C1 state, as long latencies of deeper C states may hurt data plane performance. As shown in Figure 3.12(a), by using HyperPlane in the power-optimized

mode (i.e., core transitions to C1 when halted), core power consumption reduces down to only 16.2% at zero load.

Using HyperPlane in the power-optimized mode may cost additional wake-up latency. We consider the wake-up latency of the power-optimized HyperPlane to be ~0.5 µs to be consistent with performance characterizations of `MWAIT` [90] and C1-to-C0 transitions [189]. We report the tail latency of the experimental scenario of Figure 3.10(a) for the power-optimized HyperPlane in Figure 3.12(b). Results are reported in log-scale, so the differences can be visible. As the Figure shows, at zero load, power-optimized HyperPlane yields 38% higher tail latency, compared to regular HyperPlane. However, its achieved tail latency is still 8.9× lower than the one achieved by the spinning data plane. As the load increases, HyperPlane enters the power-optimized mode less often, and hence the gap shrinks rapidly—only 8% higher latency at 50% load.

3.5.5 Ready Set Implementation

As discussed in Section 3.3.2, the ready set may be implemented either in hardware or software. We now consider performance implications of a software-based ready set. In the software implementation, `QWAIT` either waits or iterates over the list of ready QIDs in a piece of code and returns one of them based on the system policy. We measure the peak throughput of a single core in HyperPlane monitoring 1000 queues. Figure 3.13 reports the normalized throughput of the software-based implementation over the hardware-based implementation for different workloads with the *PC* and *FB* traffic shapes. For both shapes, throughput of the software-based implementation is considerably lower than its hardware-based counterpart. Throughput drop is more severe with the *FB* traffic (down to 50%) as the iterator code of the ready set must choose a QID from a larger set of ready QIDs.

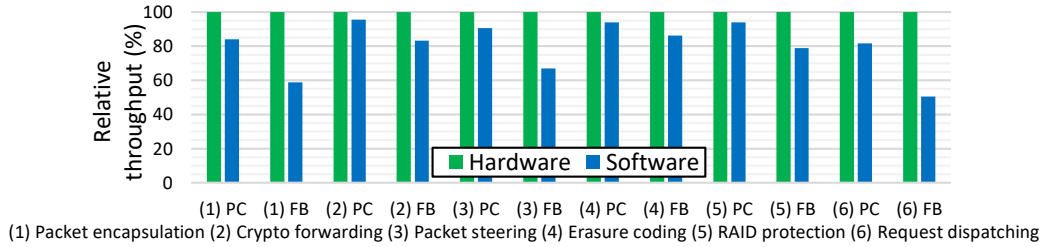


Figure 3.13: Throughput of a software-based vs. hardware-based ready set with two different traffic shapes.

3.6 Related Work

Memory monitoring. There are various hardware-assisted memory monitoring proposals for reliability and security applications [86, 168, 203, 213, 224], none of which is readily usable for software data planes. We consider one of the most general-purpose designs for a more detailed comparison: ECMon [168] is able to monitor various cache events (e.g., invalidation) for different ranges of addresses, specified in multiple entries of a *per-core* event descriptor table. Each entry corresponds to a handler function. However, ECMon does not provide any mechanism to keep certain cache lines (i.e., queue doorbells) in the caches. Even if cache lines are assumed to be locked in the cache, the event descriptor table is a small associative structure, which cannot efficiently support ~1k events for different doorbells. Furthermore, almost all of these proposals only provide a scheme for monitoring memory locations but no efficient mechanism to provide priority among ready events. In other words, the prior mechanisms at best replace only HyperPlane's monitoring set functionality. If multiple events are ready, handlers are called in the order the events are received (i.e., FIFO), or a bit-vector representing the ready events is passed to software. Similarly, HypePlane differentiates from list/queue-based locking schemes (such as MCS [157], CLH [83, 147], and QOLB [116]) in that they avoid spinning on a single lock location by forming a FIFO queue of the requesting processors, whereas HyperPlane operates on multiple I/O queues, servicing them based on a wide range of defined policies, rather than the FIFO order of work item arrivals in the queues. In software data planes, work items arrive at a high rate

and the system must perform task scheduling for non-trivial loads, prioritizing the service order among queues.

I/O software stacks. Several works enhance interrupts by reducing corresponding overheads [110, 194], combining them with spin-polling as a hybrid notification mechanism [89], or bringing them to user level [70, 82, 164, 170, 202]. HyperPlane, on the other hand, avoids the overheads of interrupts and spin-polling altogether. Kernel-bypass software stacks enable user processes to directly communicate with I/O. In such systems, application and transport software are integrated via a library OS. IX [64], Arrakis [176], ZygOS [179], and Andromeda [87] are specialized networking data planes with different features—such as task stealing [179], task preemption [115], virtualization [87, 176]—while ReFlex [127] and PASTE [111] target storage devices. Demikernel [223] specifies I/O abstractions that a library OS should provide in general. Other systems—such as Snap [151] and Shenango [174]—deploy centralized microkernel-like software, which orchestrates data communication of applications and I/O. HyperPlane, as a notification accelerator, can benefit transport software implementations, especially in case of microkernel-based software data planes like Snap [151] and Shenango [174].

Data plane optimizations. Prior works have proposed solutions to improve performance and efficiency of software data planes. DDIO [27], CacheDirector [93], and FlexNIC [123] optimize data transfer between I/O and CPU. Halo [222] proposes a near-cache accelerator for network packet flow classification. Compute-capable I/O devices, such as smart NICs/SSDs [38, 97, 142], and accelerators [102, 108, 132, 133, 178] are used to offload data plane operations from CPU. Particularly, hardware-managed transport protocols by RDMA NICs or SmartNIC-based network flow processing can ease tasks of data plane cores [84, 97, 119, 171]. Memory copy accelerators can also be used in software data planes for faster data movement [114, 149]. While HyperPlane, as a flexible centralized data plane, is compatible with commodity devices and protocols, it can leverage these proposals to further improve data plane performance.

3.7 Conclusion

In this chapter, we presented and evaluated *HyperPlane*, a hardware notification acceleration subsystem and programming model, which allows software data planes to efficiently monitor many I/O queues for work arrival. HyperPlane brings queue scalability, by avoiding spin-polling empty I/O queues unlike software-only designs, and work proportionality, by halting execution when I/O queues are idle. Furthermore, HyperPlane facilitates efficient sharing of queues across cores, enabling the strong properties of scale-up queuing. HyperPlane's programming model centers on the `QWAIT` instruction, which either returns a ready queue or halts execution. HyperPlane's microarchitecture comprises a *monitoring set*, which watches I/O queues for work arrival, and a *ready set*, the key component in HyperPlane's design that realizes various service policies, prioritization of ready queues, and work distribution among cores. We showed that HyperPlane improves peak throughput, tail latency, and idle power by 4.1×, 16.4×, and 6.2×, respectively, as compared to a modern spinning software data plane, while the monitoring and ready sets incur only $< 1\%$ per-core power and area overheads.

CHAPTER IV

HyperData: A Data Transfer Accelerator for Software Data Planes Based on Targeted Prefetching *

4.1 Introduction

Substantial efforts over recent decades have enabled us to utilize datacenters as warehouse-scale computers and benefit from "XaaS"—infrastructure, platform, software, function, etc. as a Service. The quality of service offered to individual users—such as cloud computing/storage and online applications—and industrial users—such as Software-Defined Networking (SDN) and big data analysis—depends heavily on how datacenter systems are architected and may necessitate rearchitecting them over time [61, 120, 186]. Tenants of datacenter systems—i.e., host applications and client applications/VMs—frequently interact with I/O devices like network adapters, storage devices, and accelerators. A crucial design principle of datacenter systems is high-throughput, low-latency, power-efficient data transfer between CPUs and I/O devices, while keeping the systems highly utilized.

Due to the growing speed of I/O devices, multi- or even hyper-tenancy [131], and the emergence of microservice-based programming models [167, 169], I/O software stacks have become a critical factor in end-to-end performance. Datacenters rely on fast, efficient "Software Data Planes" (SDPs, Figure 4.1), which orchestrate data transfer between ten-

* Under review in the 2021 IEEE International Conference on Computer Design (*ICCD'21*)

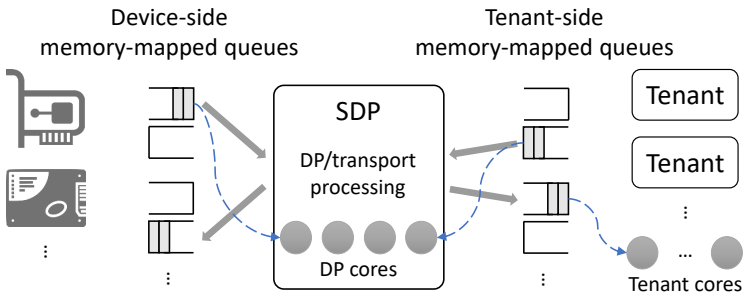


Figure 4.1: Software Data Plane (SDP) architecture. We aim to prefetch data buffers related to the items in the device- or tenant-side queues to the target data plane or tenant cores (shown by dashed arrows).

ants and I/O devices. Given the gigabit- to terabit-scale throughput and µs-scale access time of modern I/O devices as well as µs-scale processing time in the SDP and/or tenants, data transfer within a system—between the SDP and devices or tenants—should be performed smoothly. Unfortunately, data cache misses are a major bottleneck in existing SDP systems [104, 204, 208]. Recognizing the long latency of accessing data from DRAM, technologies like Intel's Data Direct I/O (DDIO) [27] and Arm's Cache Stashing [20] allow peripherals (e.g., a PCIe-attached network card) to directly deliver data to the Last-Level Cache (LLC). However, these mechanisms are unable to deliver data to private caches (e.g., L1), leaving some access latency exposed. Moreover, prior work has pinpointed additional challenges with these technologies like system-unaware data placement and eviction of unread data from the LLC due to restricted access to LLC ways [93, 221].

In this chapter, we propose HyperData to accelerate data transfer in SDPs through *targeted* prefetching. The SDP communicates with I/O devices and tenants using shared memory queues in the host address space (Figure 4.1), wherein the memory locations of the data buffers associated with enqueued items are described. We note that the consumer cores (either running the SDP or tenant software) cannot prefetch such data on their own because: (1) They identify data arrival either through interrupt mechanisms (e.g., PCIe MSI-X) or by explicitly checking the queue(s); the former approach is unable to trigger prefetch operations, and the latter is untimely to so. (2) The access pattern of data buffers in SDPs vary widely due to the highly dynamic allocation/deallocation of buffers, and thus, is

hard to predict (Section 4.2). HyperData takes advantage of the information available in the queues, i.e., data item descriptors, along with how the queues couple the SDP with devices and tenants to prefetch *exact* (rather than predicted) data to the right core. HyperData has the following distinguishing features:

- Data items are prefetched to the closest proximity of target cores, i.e., L1 caches.
- Only the required subset of data items (e.g., network packet headers) are prefetched to avoid cache pollution.
- Prefetching is done not only for SDP–device (i.e., core–peripheral) communication, but also for SDP–tenant (i.e., core–core) communication, wherein complex queues such as Virtio queues [185] are supported.
- In the case of scale-up queuing, where a queue is shared among multiple consumer cores, prefetching is done to the appropriate sharer core.

HyperData's hardware is composed of a *system-level monitoring set* and *per-core programmable prefetchers*. The monitoring set is a lookup table filled with addresses of doorbells associated with the queues and their mapping to target cores. It tracks cache coherence write transaction to the queue doorbell addresses, which indicate data item arrivals in the queues, and triggers the appropriate core's prefetcher. The prefetcher, which is programmed to understand the layout of the queue and descriptor data structures, discovers the address of related data buffers and makes the required prefetch requests to fetch data from LLC/DRAM. Our simulation results show that HyperData improves the processing latency of data items by up to 2.42× in a state-of-the-art SDP system, with only a 0.40-KB per-core overhead with a 1024-entry monitoring set.

4.2 Background and Motivation

Software data planes. The conventional approach for I/O communication and processing is using the OS kernel. The kernel manages access of user applications to (shared) I/O devices—such as Network Interface Controllers (NICs), Solid State Drives (SSDs),

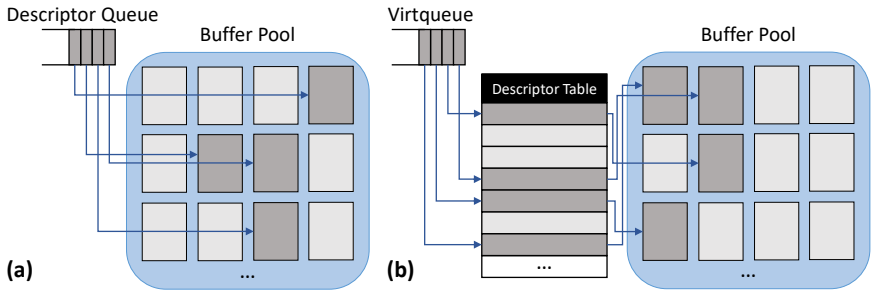


Figure 4.2: Allocation of buffers from the pool to items in the queue: (a) A regular descriptor queue, (b) A Virtio queue (Virtqueue) with a corresponding descriptor table.

persistent memory devices, and accelerators (GPUs, crypto modules, etc.)—and performs transport (e.g., TCP/IP) processing. However, the overheads associated with OS mechanisms like context switches, system calls, interrupts, and cross–address-space copies significantly limit the performance with today's µs-scale access time of I/O devices and their massive throughput. Consequently, state-of-the-art I/O software stacks, "software data planes", bypass the OS kernel to avoid the attendant overheads [24, 43, 151, 174]. SDPs typically rely on spin-polling cores, as a notification mechanism, and (user-level) queue pairs, as a means of data communication with client software and hardware devices through the shared memory (Figure 4.1).

Queue structures. Shared memory queues (or rings) in SDPs are used for exchanging data items—such as network packets, storage blocks, and RPC requests/responses—between processes and/or I/O devices. What pass through the queues are, in fact, data item *descriptors*, which contain the information (e.g., address and size) of the corresponding data buffers. Figure 4.2 shows the structure of a regular queue and a Virtqueue, which is commonly used in the driver of virtual devices in VMs and hypervisors [185]. Data buffers are typically allocated from a pre-`malloc`'ed pool of buffers for better performance, as depicted in the Figure.

Whereas queues are often organized in a scale-out manner, i.e., data items of each queue are consumed by exactly one core, the scale-up (a.k.a. shared queue) organization, wherein data items of a queue are consumed by multiple cores, demonstrates strong theoretical

properties that can benefit SDPs. Scale-up queuing avoids load imbalance as the traffic in the queue can be observed and serviced by any of the sharer cores, while in scale-out queuing, outstanding traffic may exist in a queue while some core is free. Additionally, the scale-up organization is less prone to head-of-line blocking, where processing an item takes longer than usual, as compared to the scale-out organization. While synchronization and coherence overheads of sharing a queue among multiple cores have discouraged its use in practice, such properties have motivated designs with efficient implementation of scale-up queuing [84, 161, 162]. We design HyperData in a way that supports both organizations.

Address correlation of data items. Making prefetch requests is predicated on knowing what memory references are likely to be made (or will definitely be made) in the near future. Conventional prefetchers leverage various memory access patterns that programs exhibit—namely, *strided*, *temporal*, and *spatial* [59, 91]—to predict future references. The strided address correlation, i.e., memory accesses with a constant distance, appears when accessing the contents (consecutive cache lines) of the buffer of a single data item in an SDP queue. Nevertheless, address correlation of data items of one or more queues, or the lack thereof, demands deeper analysis.

We investigate the existence of the strided access pattern and temporal address correlation (i.e., when a sequence of memory addresses, not necessarily with a constant stride, are referenced together) across the data items of an SDP queue. As such, we perform a real-world experiment using DPDK [24], a representative software infrastructure for building networking SDPs. The SDP runs in a Xeon Skylake server and steers packets received from a 100 GbE Mellanox NIC to a tenant (i.e., a QEMU VM). The buffer addresses of the packets exchanged between the SDP and the tenant is illustrated in Figure 4.3. As Figure 4.3(a) demonstrates, no particular correlation seems to exist in the shown 1024-packet sample sequence. Figure 4.3(b) shows that the buffer address strides (in a 10M-packet sequence) are distributed over a wide range. In fact, the stride distribution depends on how buffers are allocated/deallocated from/to the buffer pool, which is subject to the implementation of the

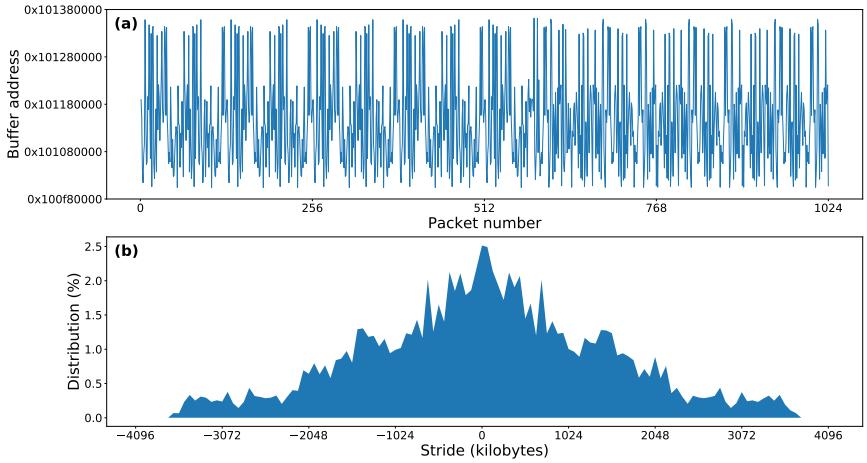


Figure 4.3: (a) Buffer addresses of a sequence of packets, (b) Distribution of strides of buffer addresses.

SDP and related I/O drivers and state of the system (e.g., different packet lifetimes due to prioritization or filtering). Such wildly distributed strides render conventional prefetchers and those designed for irregular workloads [55, 206] ineffective.

Spatial address correlation, i.e., accessing similar locations in different regions (e.g., pages) of memory, is also unlikely to appear in SDPs. Spatial variation in SDPs is related to data item buffers of different queues. SDP queues correspond to different tenants and I/O devices. Therefore, they may have independently variable traffic shapes [161], e.g., different bursts at different times. Thus, it is unexpected that the correlation of buffer addresses of a queue (if any) repeats in another queue.

4.3 HyperData Design

We design HyperData to accelerate data transfer in SDPs by prefetching the contents of data items (e.g., network packets, storage blocks) to the private cache(s) of SDP or tenant cores at the right time. As such, HyperData's key component is a prefetcher, which poses the question of *"when to prefetch what?"* Data items are communicated through shared memory queues between the SDP and I/O devices or tenants. Intuitively, the most effective time for prefetching is when data items are at or near the head of the queues, i.e., when they are about

to be dequeued. As for what exactly to prefetch, we take advantage of the fact that what actually passes through the queues are data item *descriptors*, which describe where data item contents/buffers are located. In the following sections, we first provide an overview of HyperData's components and then describe their operation details and implementation.

4.3.1 Design Overview

HyperData is composed of a centralized *monitoring set* and programmable per-core *prefetchers*. At a high level, the monitoring set watches for signals that indicate data items have been enqueued in the monitored queues and triggers the private prefetcher of appropriate cores to perform prefetching. The prefetcher does not predict what to prefetch (i.e., the address of data item buffers) but rather resolves the address of data item buffers based on the corresponding data item descriptors available in the queues.

Figure 4.4 shows the operations of HyperData. A doorbell is associated with each queue, by which the enqueuer (i.e., an I/O device or SDP/tenant core) indicates availability of data item(s) in the corresponding queue. Writing to queue doorbells (step 1 in Figure 4.4), which are shared memory locations, is realized by granting exclusive ownership of the related cache lines to the enqueuers through the cache coherence protocol (e.g., GetM in the generic coherence protocols described in [195]). The monitoring set snoops such coherence signals that correspond to doorbells (step 2), which are chosen from a specific address range reserved by the SDP kernel driver. The monitoring set is agnostic to the coherence organization and may snoop coherence signals either at a directory (as shown in the figure) or at a bus. The internal structure of the monitoring set, as will be elaborated on in Section 4.3.2, is similar to a large associative memory that maps each doorbell to a *<target core, queue address, prefetch type>* tuple. The *prefetch type* specifies the queue type (e.g., regular or Virtio) and whether the whole data buffer or a subset of it (e.g., the first few cache lines) needs to be prefetched. The queue address along with the prefetch type is sent to the target core (step 3) to prefetch the required cache lines. Once this is done, the

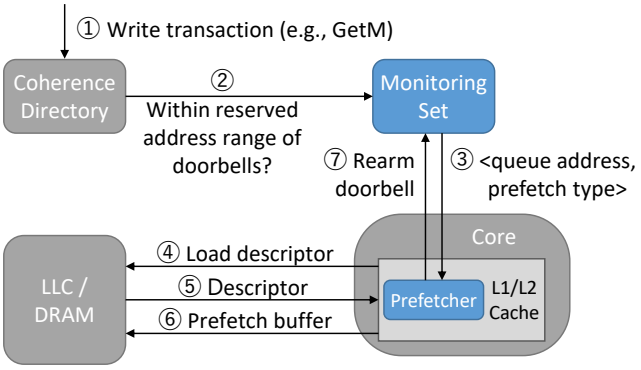


Figure 4.4: Overview of HyperData design (*monitoring set* and *prefetcher*).

entry related to the doorbell is disarmed in the monitoring set. Meanwhile, the prefetcher actively performs prefetching.

HyperData's specialized prefetcher is programmed to know how to interpret the queue and descriptor data structures. Once the prefetcher is triggered by the monitoring set, it loads the appropriate descriptor from the queue data structure (step 4 in Figure 4.4). The descriptor describes the related data item, e.g., the address of the buffer of a network packet and its length. When the prefetcher receives the descriptor (step 5), it extracts the buffer address and length and sends appropriate prefetch requests to the (shared) LLC or DRAM for the required cache lines based on the given prefetch type (e.g., only the packet header or the complete header and payload), as shown in step 6. Note that for complicated queue structures like Virtio, an additional indirection step is required to obtain the buffer address. The prefetcher may repeat steps 4-6 in Figure 4.4 if there are more data items in the queue. Finally, the prefetcher rearms the doorbell in the monitoring set to get triggered for upcoming data item arrivals (step 7). More details on the programming and design of the prefetcher will be provided in Section 4.3.3.

4.3.2 Monitoring Set

Operation. The monitoring set in HyperData is, in principle, a cache-like key–value lookup table in hardware. It enables simultaneous watching of write transactions to all queue doorbells (Figure 4.4). The keys are the tags of the doorbell addresses, which

```
monitoring_set_addr_range(doorbell_addr_range)
for each QID in queue_IDs
    core = core_map[QID]
    queue_addr = queue_addr_map[QID]
    prefetch_type = prefetch_type_map[QID]
    do
        doorbell = allocate_addr(doorbell_addr_range)
    while (monitoring_set_add(doorbell, core,
           queue_addr, prefetch_type) == FAIL)
end
```

Figure 4.5: Initialization of the monitoring set.

are mapped to <*target core, queue address, prefetch type*> tuples. Each entry in the monitoring set also contains a valid/invalid bit and an armed/unarmed bit. The latter is used to temporarily disable monitoring of the related queue doorbell, as described in Section 4.3.1. The monitoring set needs to be set up and filled with the doorbell addresses that are to be watched. This setup is done during the initialization of the data plane or when a new tenant connects. Figure 4.5 shows a code snippet for initializing the monitoring set. Such initialization/configuration is performed in the SDP driver, and since it requires access to physical or kernel memory, it is run in privileged mode.

The *target core* field of each entry in the monitoring set denotes to which core the related queue's data should be prefetched. Each queue is, in fact, coupled to a data plane or tenant thread (or multiple threads in the case of shared queues, as will be discussed in Section 4.3.4). However, the monitoring set needs to be configured with the physical core to which each queue is coupled. Data plane threads are often pinned to particular cores to prevent them from being context-switched, for their activity is critical to the performance of the tenants and overall system. Moreover, in the traditional interrupt-based transport processing in the kernel, interrupts are typically configured to be delivered to particular cores [40]. On the other hand, the mapping of tenants to cores may change during their lifetime. In those cases, the OS or hypervisor should inform the monitoring set of such a change in the thread-to-core mapping. This also applies to data plane threads in another traditional approach, where the kernel thread performs transport processing in the same core as the tenant (user-level) thread [40].

Structure. The monitoring set must hold the address tags of all queue doorbells to be able to monitor them simultaneously and provide a fully associative key–value lookup functionality. However, large fully associative structures are costly in terms of area, latency, and energy. In contrast, set-associative structures (with small associativity) are cheaper but suffer from high conflict rates. As such, we leverage a Cuckoo hash table [175] for building the monitoring set, as in ZCache [187] and HyperPlane [161]. Associativity and ways are decoupled through Cuckoo hashing, thereby providing low conflict rates even with a small number of ways (e.g., 2 or 4) [187].

Cuckoo hash tables exhibit higher insertion complexity than set-associative structures due to the "table walk" process, which may take a number of steps equal to the number of ways [187]. Nevertheless, insertions to the monitoring set happen only during the data plane initialization (Figure 4.5) or when a new tenant connects to the data plane (at second or minute time scales). Therefore, monitoring set insertions are not the common case, but their lookups occur every few microseconds or less. A Cuckoo hash table provides faster and more efficient lookups due to its smaller number of ways in comparison to regular fully or highly associative structures. Note that conflicts between doorbell tags are still possible in a Cuckoo hash table, although they are rare thanks to the table's high "effective" associativity. In the case of a conflict, the address of the conflicting doorbell needs to be reallocated (as in the `do-while` loop in Figure 4.5), so that the doorbell can be inserted in the table conflict-free.

For performing the lookups, the monitoring set snoops the relevant coherence signals at a directory or at a bus (Figure 4.4). Since doorbells are allocated from a restricted address range (Section 4.3.1), the monitoring set only need snoop addresses in this range and the snooping bandwidth is tractable. In the case of distributed directories, the monitoring set must also be banked, attached to individual directory banks. In such cases, the SDP driver must spread doorbell addresses across the banks.

4.3.3 Prefetcher Design

Unlike common prefetchers, which predict what data or instruction blocks should be brought to the cache next, HyperData's prefetcher uses the information already available in the data item descriptors in the queues. As such, it does not need to store any history for generating prefetch requests. Rather, it is implemented using a finite-state machine that discovers the exact memory locations that should be prefetched. To this end, the prefetcher is programmable so that it can be instructed on the layout of the queue and descriptor data structures.

In regular queues, the descriptors are typically stored in a circular array—the *descriptor ring*—in the queue data structure. The entries between the *head* and *tail* of the ring are available for new descriptors to be enqueued. In complex queues like Virtqueues in Virtio, the descriptors are, in fact, written in a descriptor table, and an indirection ring includes the indexes of the descriptors received from the source in the table. As Figure 4.6(a) shows, HyperData's prefetcher has two separate sets of programmable registers to store the offsets of the head, tail, and ring/table in the queue data structure as well as the ring size. Using these registers, the prefetcher is able to traverse the ring from the tail to the head to read the descriptors of enqueued data items. Furthermore, for extracting the target addresses of prefetch requests, the prefetcher has programmable registers to store the offset of the address and length of the corresponding data item buffer in the descriptor data structure. Note that HyperData's prefetcher is programmed at data plane or tenant software initialization. As the software is not expected to interact with more than one type of direct (regular) or indirect (Virtio) queue at a time, only one instance of the programmable registers is sufficient per queue type (the extreme case is where an SDP core interacts with a particular type of regular queues and Virtqueues at the I/O-side or tenant-side, respectively). When tenants are context-switched, the core's prefetcher is reprogrammed upon resumption of the tenant process.

Figure 4.6(b) sheds light on the functionality of the prefetcher. The prefetcher should

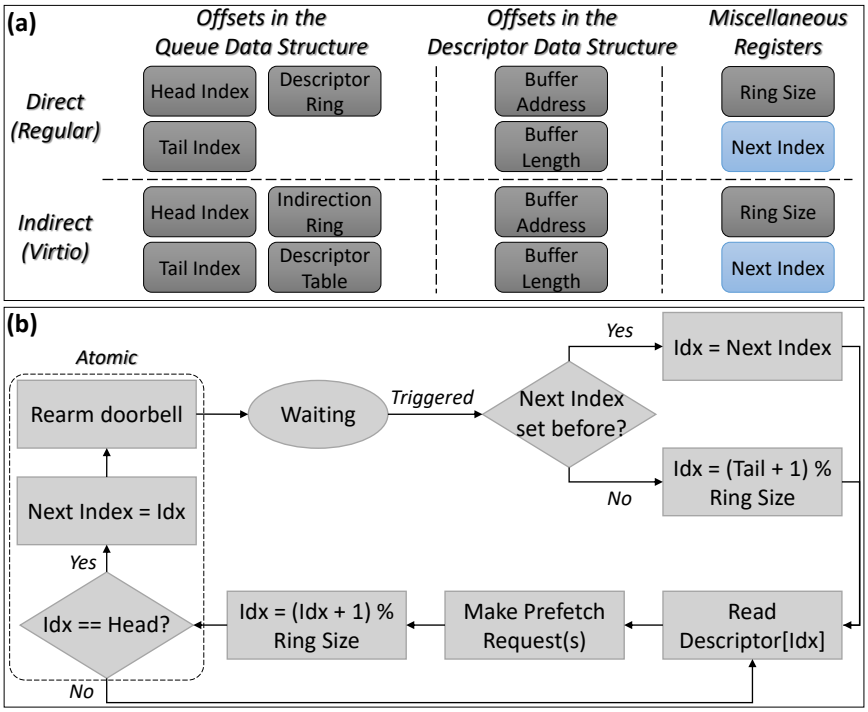


Figure 4.6: HyperData's prefetcher design: (a) Registers that enable traversing the descriptor rings and reading the descriptors; programmable registers are shown by the dark color. (b) States and operations for making prefetch requests.

prefetch data items from the tail to the head (the source enqueues items at the head and advances it). When the prefetcher is triggered by the monitoring set due to a write to the doorbell, it starts to make prefetch requests from the first (oldest) item that has not been prefetched yet. The corresponding descriptor is read, and appropriate prefetch requests are made based on the specified prefetch type, i.e., all the cache lines of the buffer or only the buffer header. In the latter case, only the first two cache lines of the buffer are prefetched, as they typically include all the protocol (e.g., TCP/IP) headers. To avoid making repetitive prefetch requests—e.g., when a new batch arrives, while the previous batch has not been dequeued despite already being prefetched—the prefetcher stores the index of the next to-be-prefetched item of the ring in the *Next Index* register (Figure 4.6(a)). When all the prefetch requests of the enqueued data item(s) are made, the prefetcher rearms the doorbell in the monitoring sets and waits until it gets triggered again. To avoid missing updates to the queue head, checking the head and rearming the monitoring set are performed atomically (the

dashed box in Figure 4.6(b)). Note that the prefetcher makes the prefetch requests through the (private) cache controller. As such, if the cache controller puts back-pressure on the prefetcher—e.g., due to fully occupied request queues or unavailable Miss Status Holding Registers (MSHRs)—the prefetcher may have to stall in the related stage in Figure 4.6(b).

When a core services more than one queue simultaneously, there may be cases where multiple queues are ready or have data items at the same time. The prefetch order must match the service order of ready queues—which, according to the service policy, may be round-robin or priority-based, as discussed in Section 3.3.1. As such, the prefetcher may only prefetch one data item or batch from one ready queue (rather than prefetching all the available items in that queue) and move on to the next ready queue. In such cases, the prefetcher must have multiple instances of the *Next Index* register, one for each ready queue whose data items have not been completely prefetched.

4.3.4 Scale-up Queuing

Load balancing in SDPs is an important challenge. In many modern systems, the load of an I/O source (e.g., a NIC) is distributed through multiple queues to the consumer cores using Receive-Side Scaling (RSS) [40] by applying a hash function on the traffic. Nevertheless, better load balancing can be enabled by scale-up queuing, i.e., sharing a queue (or a set of queues) among multiple cores, thereby improving the overall latency and throughput in SDPs (Section 4.2). In this section, we describe how the baseline design of HyperData can be enhanced to support prefetching for scale-up queuing.

As explained in Section 4.3.2, HyperData's monitoring set maintains a mapping of queue doorbells to cores. In scale-up queuing, a queue is serviced by more than one core. Thus, the monitoring set must be enhanced in two ways: (1) support mapping of a doorbell to multiple cores, and (2) choose an appropriate core's prefetcher to trigger when data items are enqueued in the shared queue. As for the first requirement, the monitoring set entries are enhanced to store multiple cores (i.e., *doorbell* $\rightarrow$ *<target cores, queue address, prefetch*

type> mappings). To keep the size of the monitoring set tractable, only a limited number of entries may be designated for queues shared by a maximum number of cores (e.g., 2 or 4 [151, 161]). As for the second requirement, a complex "next-core predictor" may be leveraged, which predicts the next target core based on the patterns of inter-arrival and processing times of data items. However, we note that in the generic context of symmetric multiprocessing [188], the work of a shared queue is distributed to the associated cores in a regular fashion, wherein the first core that becomes available (after processing a previous data item) is the first one that takes care of processing the next data item in the queue. Therefore, we propose to use the Least Recently Used (LRU) approach for determining the target core. We also note that in the context of networking applications, higher level protocols (like TCP/IP) expect ordered delivery of packets in a flow. RSS guarantees such ordering by applying a 4-tuple hash function over source/destination IP addresses and port numbers. While such hashing can be built in the monitoring set to identify the target core responsible for processing a particular flow (which also requires a hardware parser to extract the various packet header fields [69]), we keep the design generic with the LRU approach.

To realize the LRU implementation, the *target cores* related to a doorbell entry in the monitoring set need to be sorted based on the last time they dequeued an item from the corresponding shared queue. The monitoring set triggers the prefetcher of the least recent core. Note that once a core's prefetcher is triggered, unlike the non-shared case (Figure 4.6(b)), it only prefetches one or a fixed batch of data items as the next data item or batch should be processed by a different core. Variability in the processing times of data items, i.e., service times, may result in triggering the prefetcher of a wrong core. However, we will show in Section 4.4.3 that the rate of inaccurate triggering through the LRU approach is low with typical variations in the service time.

The monitoring set requires a mechanism by which it should update the order of the *target cores* associated with a doorbell. We employ a mechanism wherein whenever a core dequeues an item from a shared queue, it notifies the monitoring set accordingly.

Table 4.1: Architectural details of the simulated SDP system.

Core	8-wide issue OoO, 192/32-entry ROB/LSQ
L1 I/D	Private, 32 KB, 64B lines, 4-way SA, 16 MSHRs
LLC	1 MB per core, 64B lines, 16-way SA, 128 MSHRs
CMP	16 cores, directory-based MESI coherence
HyperData	1024-entry, 2-way monitoring set

We propose a new (atomic) CPU instruction, `MON-SET-NOTIFY`, to implement this mechanism. A core dequeuing an item from a shared queue is responsible for executing `MON-SET-NOTIFY(doorbell)`. By executing this instruction, if the Next Index register (Figure 4.6(a)) equals the queue head, meaning no more data items need to be prefetched, the monitoring set is instructed to rearm the doorbell. Otherwise, the monitoring set is instructed to trigger the next core's prefetcher.

4.4 Evaluation

4.4.1 Methodology

We use the gem5 simulator [65] and augment it to model the hardware components of HyperData. We model a 16-core x86-64 CMP system with architectural details as described in Table 4.1. We use an in-house user-level SDP system based upon DPDK [24] that is able to run in the simulator. Our SDP infrastructure closely tracks the performance characteristics of DPDK. The simulated cores run the data plane and tenant software in addition to emulated I/O devices (i.e., I/O sources or sinks).

We evaluate HyperData on both the SDP and tenant cores. The SDP is a dispatching application [161] (*SDP Dispatcher*), which monitors the queues where network traffic is generated by the I/O sources or tenants. The SDP dispatches the network packets to appropriate tenants or I/O sinks, as in Figure 4.1. We also consider a variant (*SDP Vhost*), wherein the SDP runs a simplified Virtio driver to dispatch network packets to/from (virtual) tenants. The tenants run an in-memory key–value store application (*Tenant KV-store*), wherein the requests—i.e., `sets` and `gets` of keys [98]—dispatched by the SDP

(encapsulated in network packets) are processed.

The addresses of data buffers (i.e., network packets) in the generated traffic are both spatially and temporally varied. We define three spatial variations: *Tenant-Concentrated (TC)*, *Tenant-Device-Balanced (TDB)*, and *Device-Concentrated (DC)*. 75%, 50%, and 25% of the generated traffic is sourced from tenants in the TC, TDB, and DC patterns, respectively, and the rest is sourced from the emulated I/O. The tenant-sourced traffic is likely accessed from the LLC by the SDP as the corresponding data buffers have been recently prepared by the tenants. Nevertheless, we pre-prepare data buffers corresponding to the I/O-sourced traffic in order to be accessed from DRAM by the SDP to mimic real-world I/O behavior. The temporal pattern of buffer addresses within each source is set to resemble DPDK in a real system. We extract the address stride distribution of accessed buffers from DPDK (e.g., Figure 4.3(b)) and generate the traffic with similar strides in the emulated I/O or tenants.

4.4.2 Prefetching Performance

We evaluate HyperData by comparing it against the baseline (without any prefetchers) and a stride data prefetcher. Both the HyperData and stride prefetchers are built in the cores' L1 D-caches. We take latency as an application-level performance metric [128]. Latency is measured from the time a packet is enqueued (to either SDP-I/O or SDP-tenant queues) until it is processed by the SDP or tenant. Load is set to be near saturation (i.e., ~100%). Average latency is reported in Figure 4.7 for the different workloads. The stride prefetcher improves the performance by up to 1.23×. HyperData achieves significantly larger performance improvements, i.e., 1.20-2.42×. The stride prefetcher can capture the access pattern within packets (e.g., consecutive cache lines), but it falls short of learning the temporal/spatial access pattern of packets within/across the queues. Nevertheless, HyperData is able to prefetch packets in a timely manner thanks to the notification mechanism of the monitoring set and the already available geometry information of buffers in the descriptors.

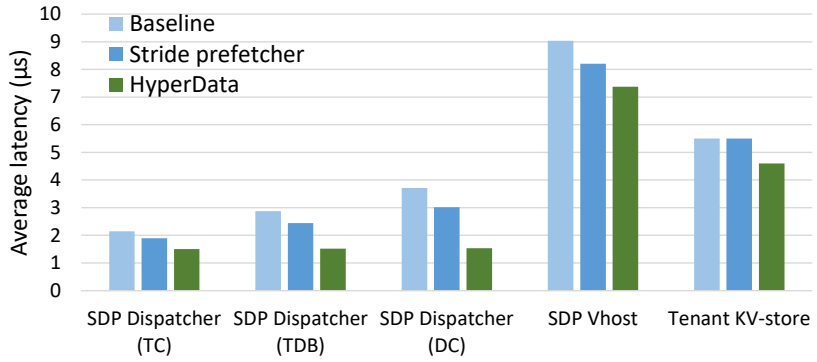


Figure 4.7: Prefetching performance in terms of packet processing latency.

In Figure 4.7, we observe that prefetching brings larger performance improvements in the SDP Dispatcher workloads as compared to the others. This is because computations per packet in SDP Vhost and Tenant KV-store are relatively more complicated than those in the SDP Dispatcher variants; thus, stalls associated with cache misses incurred by accessing packets make up a larger fraction of packet processing time in SDP Dispatcher. Moreover, in SDP Dispatcher, we observe that while the access pattern affects the performance in the baseline and baseline + stride prefetcher configurations (i.e., TC being better than TDB and TDB being better than DC due to more frequent accesses to buffers "warmed" by the tenants), HyperData achieves an almost similar performance with all the patterns as it performs effective prefetching in all three cases.

Figure 4.8 illustrates statistics of LLC accesses by the dequeuer core with the different configurations (normalized over the baseline) and workloads. LLC hits (i.e., L1 misses) and LLC misses are made through program execution in the baseline, whereas with the HyperData and stride prefetchers, prefetching can also result in LLC hits/misses. The ultimate goal of prefetching is that the necessary LLC hits/misses—for accessing the data buffers, in particular—are made by the prefetcher right before the execution. In the baseline, the SDP workloads incur considerable LLC misses. Particularly, LLC misses increase from the TC to TDB to DC patterns in SDP Dispatcher in that a larger portion of the traffic is sourced from I/O (Section 4.4.1). There are no LLC misses in Tenant KV-store because the packets that the tenants process have been recently dispatched and brought to the LLC

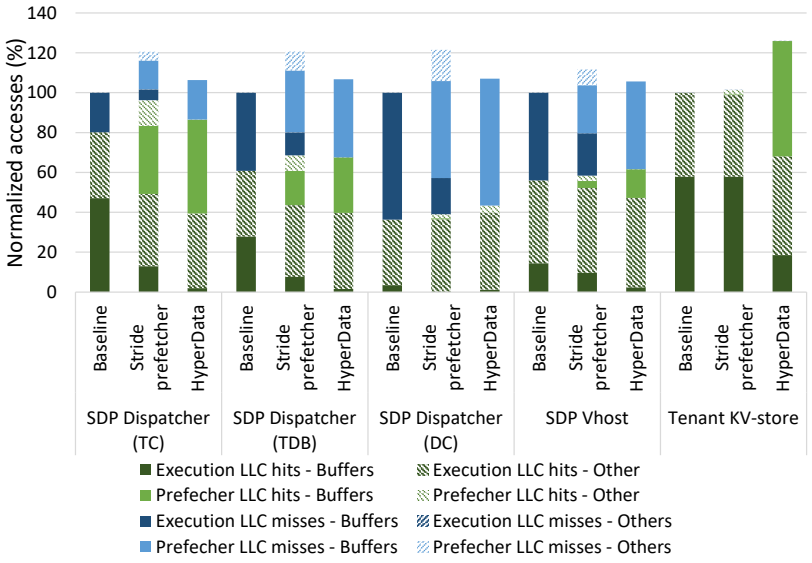


Figure 4.8: LLC hit/miss statistics of the dequeuer core.

(if not already) by the SDP. We observe that HyperData is able to resolve almost all the LLC misses and L1 misses that correspond to accessing the packets in the SDP workloads, whereas the stride prefetcher fails to do so. Note that in the case of Tenant KV-store, while HyperData performs the required prefetches, a portion of data buffers are still accessed from the LLC. This happens because the prefetching rate is high due to LLC-resident (rather than DRAM-resident) buffers, which results in eviction of some of the prefetched buffers from the L1 cache before being accessed. Although HyperData performs targeted prefetching, the overall LLC accesses are more with HyperData in comparison to the baseline because some other useful cache blocks are displaced by the prefetched data buffers, which are brought back to the L1 cache again later. With the stride prefetcher, the overall LLC accesses are even more, but the performance improvements are much smaller as compared to HyperData. We also observe that the stride prefetcher is almost ineffective for Tenant KV-store and does not resolve the execution LLC hits much. This is due to the limited size of the reference prediction table [91], highlighting the limitation of prediction-based prefetchers. On the other hand, HyperData accomplishes effective prefetching with small hardware structures (Section 4.4.4).

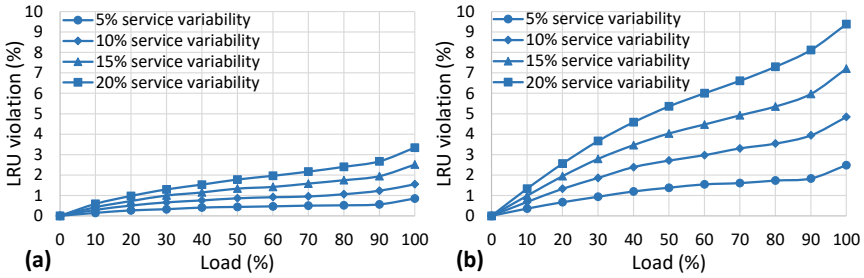


Figure 4.9: The rate of prefetching to an incorrect core using the LRU mechanism with (a) 2 cores, and (b) 4 cores.

4.4.3 Effectiveness with Scale-up Queuing

In this section, we evaluate the effectiveness of the LRU mechanism in determining the appropriate core's prefetcher to trigger when HyperData is used with shared queues, as specified in Section 4.3.4. Variability in processing times of data items (or service times) may result in cores completing the service in an order different from the one they started. This leads to triggering an incorrect core's prefetcher in HyperData.

To quantify LRU violations, we employ discrete-event simulations wherein data items arrive with a Poisson distribution and service rate has a uniform distribution with various deviations. Figure 4.9 shows the LRU violation rate at different loads and service variabilities. Service variability is defined as the range-to-mean ratio of service times. Note that shared queues are aimed for the SDP due to their load balancing merits. Since the SDP needs a small number of cores (e.g., 1-4 [151, 161, 174]), we consider a maximum of 4 sharer cores. As we see in the figure, LRU violations increase by having larger service variabilities or more sharer cores. However, while we mainly observed service variabilities of up to 15% in our experiments (Section 4.4.2), LRU violations are below 10% even with 20% service variability. Thus, the LRU mechanism can retain most of the performance benefits of prefetching when HyperData is used in the scale-up queuing organization.

4.4.4 Overhead Analysis

The hardware components of HyperData consist of a monitoring set and per-core prefetchers (Section 4.3). The area overhead of HyperData is dominated by the monitoring set. The overhead is, in fact, directly proportional to the maximum number of queues that HyperData may monitor. State-of-the-art SDPs and hyper-tenant systems should effectively handle ~1k I/O or tenant queues [161]. Therefore, we consider a 1024-entry monitoring set in our overhead analysis.

Assuming a 2-way monitoring set, 48-bit physical addresses, 128-byte doorbells, and allocating doorbells and queues from a restricted 1-GB address range, total overhead of the monitoring set is 6.25KB. Note that the monitoring set is a centralized structure, and the amortized per-core overhead is even smaller. For example, the area overhead for a 16-core chip (as in Section 4.4.1) is only 0.39KB per core. If all the 1024 queues can be shared among a maximum of 4 cores, the area overhead is 0.58KB per core, considering the expanded monitoring set entries and LRU circuitry (Section 4.3.4). Nevertheless, we can decrease this overhead by designating a limited number of entries for shared queues. If, for instance, 64 queues out of the 1024 queues can be shared among at most 4 cores, the area overhead would be just 0.40KB per core.

4.5 Related Work

Data locality in SDPs. Effective data movement in SDPs, i.e., bringing data to the locality where they are processed with the smallest/fewest transfers as possible, is key to the efficiency and performance of SDPs. With compute-capable I/O devices—such as smart NICs/SSDs [97, 152] and accelerators [133, 178]—data processing can be done at the device. As such, data plane operations can be completely/partially *offloaded* from the CPU, and unnecessary trips of data to/from DRAM over the I/O interconnect (e.g., the PCIe bus) are avoided. However, when data plane operations are *onloaded* to the CPU (Figure 4.1), data

should be within easy reach of the cores. In multi-socket servers, NUMA-aware memory allocation [192], particularly for I/O devices, prevents costly cross-socket data movements. Furthermore, data caching, which is our focus in designing HyperData, can significantly affect the processing latency and throughput. Datacenter applications [95, 120, 198], in general, and SDPs [104, 204, 208], in particular, have large data footprints and incur numerous misses at different layers of the cache hierarchy. As such, cache locality is an important concern in the development of data plane software [64, 87, 113]. Additionally, designs like DDIO [27] and NeBula [204] leverage hardware support for steering I/O data directly into the cache hierarchy. In contrast, HyperData takes a system-level (rather than I/O-driven) approach and is compatible with commodity I/O devices. HyperData can prefetch the data to the L1 caches in both core–device and core–core data communications, and supports complex queue structures such as Virtqueues and scaled-up (shared) queues.

Prefetching techniques. Prefetching is an essential practice in hiding the long latency of memory accesses. A large class of prefetchers seek to predict future memory references based on a program's memory access pattern—e.g., strided, temporal, and spatial patterns [59, 91]. Many modern workloads, such as graph traversal and sparse-matrix linear algebra, demonstrate irregular access patterns. Prior work has proposed prefetchers for such workloads using programmer or compiler assistance [55, 206] or other hardware solutions [217, 220]. Nevertheless, dynamic allocation/deallocation of data buffers in SDPs—which depends on various factors like the load of the system and implementation of transport protocols and I/O drivers—makes predicting the corresponding memory references extremely difficult (Section 4.2). Likewise, software prefetching [73] is unlikely to be effective due to issuing prefetch requests in a static, load-agnostic manner. On the other hand, HyperData explores exact data buffers that need to be prefetched by reading data item descriptors (Section 4.3), similar to run-ahead prefetchers [76, 145, 166]. This class of prefetchers leverage spare core resources or a helper thread to discover long-latency memory accesses ahead of execution. In contrast, HyperData relies on a simple finite-state machine

for address discovery (Section 4.3.3).

4.6 Conclusion

In this chapter, we were motivated by the fact that accessing I/O or tenant data from the L1 cache is key to the performance of SDP systems. Prediction-based prefetching is not suitable for SDPs because consumer cores lack an appropriate data arrival notification mechanism. Moreover, the access pattern of data buffers is highly complex and varies with system conditions as well as the implementation of the SDP and I/O drivers. We proposed HyperData to tackle these issues and accelerate data transfer in SDPs. HyperData performs targeted prefetching, i.e., bringing *exact* cache lines of corresponding data buffers to an appropriate cores' L1 cache, using the data item descriptors in the SDP–I/O or SDP–tenant queues. HyperData is designed to support complex queues used in Virtio and scale-up organizations. Our evaluation results show that HyperData improves processing latency by up to 2.42$\times$ in a simulated state-of-the-art SDP system with small area overhead. We also showed that with scale-up queuing, wherein one of the sharer cores needs to be selected to perform prefetching, at least 90% of performance improvements of the single-core cases can be retained using the LRU mechanism.

CHAPTER V

Conclusion

5.1 Summary

Software data planes coordinate data communication of tenants of a datacenter system with each other and/or I/O devices. State-of-the-art software data planes leverage shared-memory queues and spin-polling cores for transport processing and data transfer. In this dissertation, I discussed the issues of current software data planes and provided hardware-software solutions for them, as summarized below.

First, I started with characterization of software data planes in a real system in Chapter II. I pinpointed inefficiencies of spin-polling (overheads, useless work, adverse effect on co-running hyperthreads, etc.). I also demonstrated that spin-polling lacks queue scalability—due to processor cache capacity constraints—and core scalability—due to operations rate limits of PCIe and LLC. Furthermore, I illustrated inefficiencies of spin-polling in the scale-up queuing organization (i.e., sharing one or more queues among multiple cores), which render such an organization impractical in spite of its theoretical merits.

Next, I proposed the *HyperPlane* accelerator in Chapter III, which replaces spin-polling as a notification mechanism in software data planes. HyperPlane allows a core to not iterate on empty queues, halt when all queues are empty, and efficiently share a queue with other cores. HyperPlane comprises a programming model, based on the `QWAIT` instruction, and a microarchitecture, composed of a *monitoring set* and a *ready set*. Thanks to the monitoring

set, which watches for work/data arrival in the queues, `QWAIT` acts like a "multi-address `MWAIT`". The ready set tracks the ready queues and enables `QWAIT` to return the ID of a ready queue based on a particular service policy. Despite incurring minor power and area overheads, I showed that HyperPlane significantly improves the performance and energy efficiency over a spin-polling–based counterpart. I also illustrated that the strong potentials of scale-up queuing are unleashed through HyperPlane.

Finally, in Chapter IV, I proposed the *HyperData* accelerator, which enhances data transfer in software data planes based on prefetching. The goal of designing HyperData is to prefetch the necessary part of data items to the closest proximity of target data plane or tenant cores, i.e., L1 caches, at the right time and support various queues formats (i.e., regular, Virtio, and scale-up). Because the access pattern is too complicated—hence, hard to predict—in software data planes (due to the highly dynamic allocation/deallocation of corresponding data buffers), HyperData is designed to discover the "exact" memory locations that must be prefetched. HyperData is composed of a *specialized prefetcher*, which performs address discovery and issues prefetch requests, and a *system-level monitoring set*, which tracks data arrival and triggers the prefetcher of an appropriate core. I showed that, with a small area overhead, HyperData substantially improves the performance and efficiency of a modern software data plane.

All in all, HyperPlane and HyperData compose a full-fledged suite of accelerators for software data planes. Note that the basic functionality and structure of the monitoring set—i.e., watching write transactions to queues and looking up address tags in a table—are essentially the same in both these accelerators. Therefore, deploying HyperPlane and HyperData together not only brings their combined benefits but also amortizes their (yet small) overheads.

5.2 Future Research

Deployment of the accelerators. The use of the accelerators introduced in this dissertation, i.e., HyperPlane and HyperData, can be explored further. While these accelerators were originally designed to be deployed at the Chip Multi-Processor (CMP), they may also be deployed as peripherals, making them easier to be incorporated in the near-future systems.

HyperPlane's components, i.e., the monitoring and ready sets, can be fully deployed as a peripheral. However, although HyperData's monitoring set can be similarly deployed as a peripheral, the logic of the per-core prefetchers in HyperData must be built in the cores' private caches. In addition to not consuming the CMP area, deploying HyperData's monitoring set as a peripheral enables more complex designs for the logic used for choosing which core's prefetcher must be triggered in the case of multi-consumer queues, e.g., based on inter-arrival/service time prediction or network flow calculation (Section 4.3.4).

The monitoring set in both HyperPlane and HyperData must be able to observe the relevant cache coherence signals. As such, for its deployment as a peripheral, the peripheral interconnect needs to be cache-coherent. Technologies like Intel QPI/UPI [30, 31] and Arm AMBA ACE [18] have long since enabled cache-coherent multi-CMP products. Recently, cache-coherent interconnects for accelerators have received significant attention through standards like CCIX [22] and CXL [23]. This holds promise of faster deployment of prototypes of accelerators, including HyperPlane and HyperData.

This deployment scenario needs to be investigated in detail in future work. Additionally, questions regarding core–accelerator communication must be addressed, for example: How would the proposed instructions in this dissertation (like `QWAIT`) change? How much would the performance benefits, i.e., speed-ups, get affected by the physical distance of the core and the accelerator?

Specialized data plane cores. Data plane software is, in fact, a tax that has to be paid because there is still no hardware-transport system that provides comparable generality, flexibility, and centralization to operational data planes (Section 3.6). Cloud providers

wish to allocate as much chip area as possible for running customer workloads (such as VMs). Therefore, designing a specialized data plane core that might replace two (or more) server-class cores (like Intel Xeon) or require less area than a conventional core while achieving the same I/O throughput/latency or better performance/memory isolation is an appealing direction for future work.

The specialized data plane core should incorporate the features introduced in this dissertation, i.e., accelerated notification and data buffer prefetching. Furthermore, it should be optimized for operations heavily used in data planes, such as encryption/decryption, flow table lookup, data (e.g., network packet) encapsulation/decapsulation. Data planes also demand efficient ways of performing memory copies because: (1) they do more `memcpy()` than other software, (2) they copy across protection domains/address spaces, and (3) they often move scatter-gather buffers in addition to contiguous buffers. Interestingly, one of the ideas we were considering at the beginning of the projects of this dissertation was a tightly integrated, asynchronous, virtual-memory–aware `memcpy()` engine. A bit later, the Intel Data Streaming Accelerator (DSA) [14] was introduced, which has the mentioned features. Future work may incorporate such optimized data movement features in the specialized data plane core, or enhance existing software data planes using accelerators like the Intel DSA.

APPENDIX

APPENDIX A

Characterization of Unnecessary Computations in Web Applications *

A.1 Introduction

Web applications play an important role in the daily life of many people, and they are widely used in both desktop and mobile environments for various purposes such as online shopping, navigation, and video streaming. In the main body of this dissertation, we focused on enhancing data transport in datacenter systems, which in turn benefits the users of Web applications. However, user experience also depends largely on client-side computations of Web applications, which we discuss in this chapter. Web pages are getting more and more complicated in order to provide content with a visually rich user experience. Although desktop and mobile processors have been constantly advancing in recent years, the quality of service delivered to Web users, especially in the mobile platform, is not satisfying yet as they may experience delays in showing the content of Web pages [7]. This is due to the fact that Web browsers are complex programs, which must process multiple languages (i.e., HTML, CSS, and JavaScript) and manage a wide variety of network transactions.

The quality of user experience depends on how fast the content of a Web page is displayed and how smooth one view transitions to another. In particular, both application

* Published in the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (*ISPASS'19*) [103]

designers (e.g., designers of Web browsers) and Web developers (i.e., Web page designers) should be aware that users' satisfaction relies on three distinct metrics: page load time, response time to user input, and animation smoothness [17]. Among these metrics, page load time is the most important one. In a study on more than 10,000 mobile Web domains [7], it was found that mobile websites load in 19 seconds on average with a 3G network and in 14 seconds on average with a 4G network. It was also observed that 53% of users left their browsing sessions if pages took longer than 3 seconds to load. This shows how deeply Web page load time affects user experience and highlights the need for performance improvement of Web applications.

Considerable effort has been put into improving the performance of Web applications both in academia and industry. Commercial Web browsers are continuously improved by leveraging complicated algorithms [5, 34, 41] and utilizing GPUs as accelerators [25, 26]. Web developers are also provided with advanced libraries and design tools [49, 50] for carefully managing services and ordering the resources. Prior academic work has tried to optimize Web browsers in different ways. [173] and [216] target Web page load time by prefetching and caching of resources and reordering of resources, respectively. Other proposals include enhancing or parallelizing the JavaScript engine [53, 124, 153, 154], proper scheduling of CPU cores [177, 193, 226, 227, 229], and designing specialized hardware [76, 79, 228].

In this chapter, we argue that in current Web applications—Web browsers in particular—there exists unnecessary computations, which are completely or most likely wasted. These unnecessary computations are caused by processing codes that are never used, pitfalls in the design of Web applications, or producing output that is never or most likely not noticed or used by the user. More details regarding potential sources of unnecessary computations are provided in Section A.2. Next, we develop a profiler that effectively identifies portions of Web browser computations that are important to the user (e.g., generating display pixels and network outputs), and analyzes the computations that do not belong to this portion

(e.g., the unnecessary computations). The unnecessary computations are either completely useless, or done at improper time, so that they could be deferred to a later time when they are actually needed. Therefore, the designed profiler could be leveraged to both identify wasted computations and also reveal opportunities to optimize performance and energy efficiency of Web applications.

Our profiler is based on dynamic backward program slicing, and it works on the instruction and memory traces collected while a Web browser renders a Web page. The main slicing criteria are the pixels buffer at points where it contains the final values of pixels that are going to be put on the device display. While going backwards, the profiler identifies instructions whose execution has any effect on the values stored in the pixels buffer. Therefore, the instructions that do not belong to the calculated slice do not have anything to do with what is shown to the user. As an alternative to pixels buffer, system calls could be leveraged to define broader slicing criteria (Section A.4.3), so that the profiler determines what instructions have any impact on the values communicated with I/O, including the network, display monitor, and audio device.

The profiling results show that only 45% of dynamically executed instructions on average contribute to the value of pixels in the process of rendering the Web pages in our benchmarks. We provide details of slicing percentage in important threads of the rendering process of the browser under test (Google Chromium). Moreover, by analyzing the the instructions which do not belong to the pixel-based slice (i.e., 55% of all instructions), we categorize potentially unnecessary computations and show that the most notable category is processing of JavaScript codes.

In the remaining sections of this chapter, we first provide background on how Web browsers render Web pages and what the potential sources of unnecessary computations are. Next, in Section A.3, the design of the backward-slicing–based profiler is presented. We introduce the evaluation methodology in Section A.4 and describe how we leverage the profiler to identify unnecessary computations of different benchmarks. Then, we present

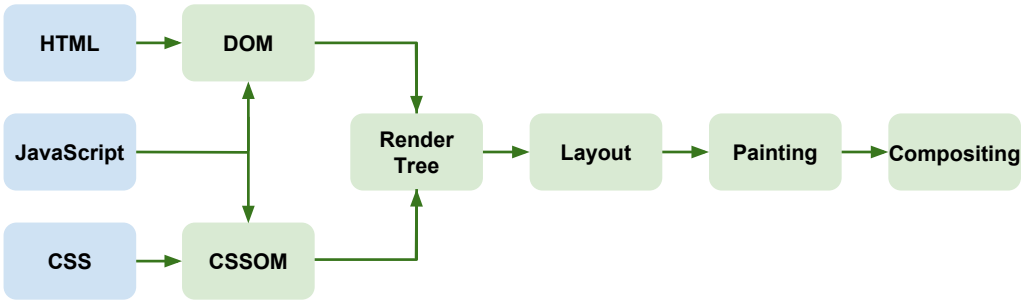


Figure A.1: Rendering pipeline of a Web browser.

and discuss the results in Section A.5. Finally, the chapter is concluded in Section A.7.

A.2 Background and Motivation

A.2.1 Rendering Pipeline of Web Browsers

For rendering a Web page, browsers follow a number of steps called the *rendering pipeline*. Figure A.1 shows an overview of this pipeline, which is described below:

- First, the browser starts parsing an HTML file and generates a tree named the Document Object Model (DOM). This tree defines the hierarchical relationship between all the different elements available in the HTML file.
- Next, CSS files are parsed and a tree called CSS Object Model (CSSOM) is constructed. CSS files are complementary to the HTML file and define the exact style of the different elements in the HTML file.
- In the next step, the required JavaScript codes are executed which can arbitrarily modify or update the object model trees.
- After running JavaScript codes, the browser merges the updated DOM and CSSOM and generates a new tree which then gets trimmed down to only contain objects that include visual context to the user. The resulting tree is called the Render Tree.
- Next, the exact position and size of different elements, which may be grouped in different layers, are computed in the layout stage. Then, the required graphical commands are generated in the paint stage, and according to the relative order of the layers computed in

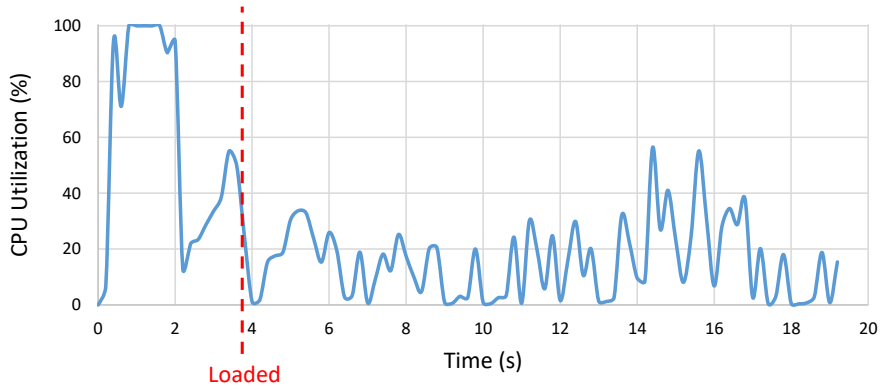


Figure A.2: CPU utilization by the main thread of the tab process while browsing *amazon.com*.

the compositing stage, the final view of the Web page is rendered in the user's display.

Note that the pipeline outlined above describes how a Web page is rendered during both load time and also the time when the page is modified based on user interactions (e.g., opening a menu) or dynamics of the page (e.g., an animation). However, the computations of load time are much more intensive because the whole page is rendered from the ground up, while once it is completely loaded, changes made to the page by user interactions or dynamics only affect a few elements of the page. To illustrate this behavior, Figure A.2 shows the percentage of CPU utilization in a fairly short browsing session, where the *amazon.com* website is loaded, the user scrolls down and up a little bit, clicks to see the next two photos in a photo roll, and finally opens a menu. The utilization percentage corresponds to the main thread of the tab process, in which the most critical computations, such as calculation of styles and execution of JavaScript code, are performed. Note that compositing is done in a separate thread (more details about the architecture of the Chromium browser are provided in Section A.5).

A.2.2 Unnecessary Computations in Web Browsers

In the rendering pipeline of Web browsers, there may be unnecessary computations. We categorize them into three main groups:

Unused JavaScript and CSS codes. There are various JavaScript and CSS libraries

Table A.1: Unused JavaScript and CSS code bytes.

Website		Amazon	Bing	Google Maps
Only Load	Unused bytes	955 KB	103 KB	1.9 MB
	Total bytes	1.6 MB	199 KB	3.9 MB
	Percentage	58%	52%	49%
Load and Browse	Unused bytes	882 KB	82.5 KB	2.0 MB
	Total bytes	1.6 MB	206 KB	4.6 MB
	Percentage	54%	40%	43%

that Web developers tend to use—such as jQuery [33], Bootstrap [21], and React [37]—in order to reduce development time. Not all these codes, when imported, are really used, meaning that processing them is a useless computation. Table A.1 shows the percentage of unused JavaScript and CSS code bytes after loading three different websites–that is, Amazon, Bing, and Google Maps–and also after browsing them for 30 seconds in a typical way. As can be seen, about 40-60% of JavaScript and CSS codes are unused, and even by browsing the websites, not all these codes are used. Moreover, in the case of Bing and Google Maps, more code bytes are downloaded while browsing, which adds to the total bytes, and may add to the number of unused bytes, as compared to the load time.

Browser design pitfalls. Web browser designers have been constantly trying to improve the performance of Web browsers by leveraging complicated methods and algorithms. Although the improvement in the performance of Web browsers could be easily observed by comparing their earlier versions to their state-of-the-art ones, there are a number of optimizations, some of which are done speculatively, that have not been fully verified to work all the time or in the common case. For example, in the compositing algorithm of the Chrome browser [2], multiple elements of the page are grouped together as different layers, and to avoid repainting their contents, each layer has its own backing store/cache. However, this is expensive in terms of memory requirements; moreover, the computations and memory space related to the layers that are only rendered once and will not be required to be repainted (e.g., because they are always on top of other layers or they are always invisible) are wasted. The compositing algorithm of Chrome blindly accepts these overheads and potentially unnecessary computations. Other examples include multi-threaded rasterization, which may

invalidate some pixel-based optimizations done at the early stages of the rendering pipeline [34], and the JavaScript JIT compiler deoptimizations, which are done because of wrong assumptions of the compiler about object types [196].

Imperceptible computations. A Web page consists of many layers, which may overlap each other, and elements, which may never be noticed or utilized by the users. For example, a layer that is overlapped by another layer may most likely remain invisible while the user interacts with the Web page. Similarly, a button element that is placed at the bottom of the page may never be clicked by the user. Therefore, the calculation of their styles and layouts, or compilation of the JavaScript code that corresponds to their event handlers (e.g., the code for handling the *onclick* event) is imperceptible to the user. Existence of Web analytics tools that could even track user clicks and scrolls enlightens the fact that not all the elements in the Web page have the same importance level.

A.2.3 Detection of Unnecessary Computations

A program slice contains instructions whose execution affects the values of a set of variables at a specific point in the program execution. The pair *(program point, set of variables)* is called *slicing criterion* [209]. Program slicing is typically done by starting from the *program point* given by the *slicing criterion* and going backwards toward the beginning of the program. Hence, this is called backward program slicing. Program slicing could be done either statically or dynamically. In static program slicing, no assumption is made on program inputs, while in the dynamic approach, slicing is done on the dynamic instruction trace of a sample execution. Static program slicing is less precise in that it has to make conservative assumptions on program inputs. Thus, our choice for the profiler is dynamic program slicing.

A profiler based on dynamic backward program slicing can theoretically identify all wasted computations mentioned in Section A.2.2. If the slicing criteria are defined in a way to include all the "necessary" variables at exact program points, execution of whatever

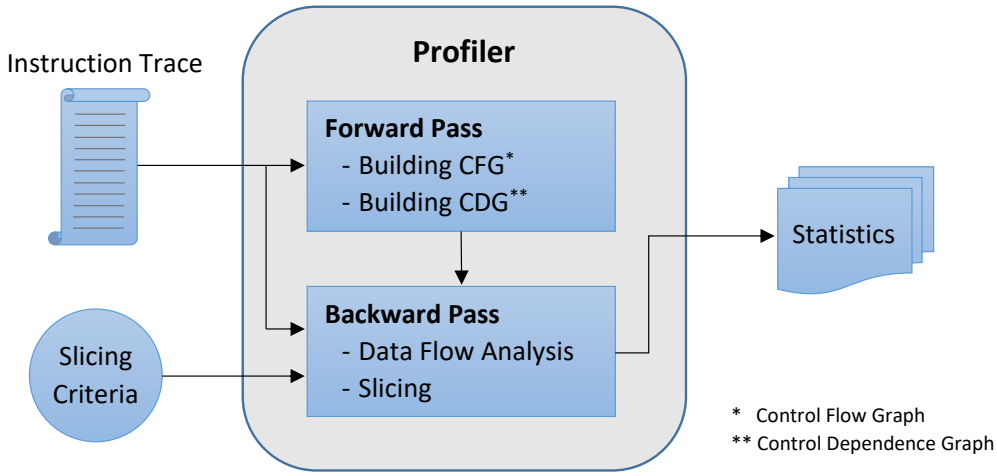


Figure A.3: Profiler design overview.

instructions that are not part of the calculated slice is unnecessary. However, these necessary variables should be carefully specified, which may not be practical or even possible. If such criteria intuitively cover what the user cares about—that is, visual contents shown to them and page objects with which they interact—the computations related to processing unused JavaScript and CSS codes, layers that are invisible, and page elements that are not important to the user will be discovered.

In the next section, we describe our slicing-based profiler, and then in Section A.4, we explain how slicing criteria are chosen to effectively identify unnecessary computations.

A.3 Profiler Design

The profiler implemented and used in this chapter is based on dynamic backward program slicing. Figure A.3 shows an overview of the profiler design and how it works. The profiler performs dynamic backward program slicing on a trace of dynamically executed machine instructions. In other words, it does not do slicing at the C/C++ source code level; rather, it tracks back machine-level instructions from the end of the instruction trace to the beginning and marks each instruction as being part of the slice or not based on the slicing criteria as it goes backwards. The slicing criteria essentially determine what the target

variables (i.e., memory locations) are at what points in the instruction trace. The output of the profiler includes statistics about the calculated slice, such as distribution of instructions of the slice among all instructions at function-level or thread-level. Note that unlike other slicers that only focus on a specific aspect of a Web application, such as JavaScript [219], our profiler treats the browser as a whole program rendering a page.

Traditionally, program slicers perform slicing on a program dependence graph, which is a combination of the data dependence graph and control dependence graph [209]. In our profiler, we construct the control dependence graph in a forward pass, as displayed in Figure A.3. However, we do not explicitly construct a data dependence graph. As will be explained in Section A.3.2, data dependencies are discovered through a liveness analysis meanwhile the profiler goes backwards and performs slicing. Since the input trace contains exact memory addresses accessed by the browser, the profiler does not suffer from the memory aliasing problem in capturing data dependencies.

In the rest of this section, we go over the details of the forward and backward passes. Then in Section A.4, we describe how the slicing criteria should be chosen so that the unnecessary computations of a trace collected while a Web browser renders a Web page are effectively identified.

A.3.1 Forward Pass

In a single forward pass, the profiler first builds a Control Flow Graph (CFG) for each function/procedure from the trace of dynamically executed instructions. Boundaries of functions/procedures are identified through matching call and return instructions. Note that since the profiler works on machine-level instructions, it is necessary to build the CFGs from the trace of dynamic instructions in that the target(s) of indirect branches could not be found statically (i.e., from the instruction opcode). Also, all CFGs have their own specific *entry* and *exit* nodes.

In the next step, the Control Dependence Graph (CDG) of the instructions is built. CDG

shows on what branches each instruction is dependent. For building the CDG, we first need to determine the postdominators of each instruction. In a CFG, a node *n* postdominates a node *m* if and only if every directed path from *m* to *exit* contains *n*. Algorithms for computing postdominators of each node in a CFG and subsequently, computing the CDG are not very complicated, and could be derived from basic compiler books and articles [54, 96]. Note that the calculated CDG could be stored in stable storage, so that it can be re-used multiple times in the backward pass for different slicing criteria.

A.3.2 Backward Pass

In the backward pass, data dependence analysis and slicing are done concurrently through liveness analysis. Conceptually, in our slicing method, there is a set of live variables, which is updated based on two distinct factors: slicing criteria and operation of instructions. As Figure A.3 illustrates, slicing criteria—which are pairs of *(program point, set of variables)* (Section A.2.2)—are given to the backward pass analyzer of the profiler as input. When the profiler reaches to any *program point* specified in a slicing criterion, it puts the corresponding *set of variables* into the live set.

The second factor, based on which the live variables set may be updated, is operation of instructions, which also determines whether or not instructions should be part of the slice. If an instruction writes into a variable that is a member of the live variables set, that variable is taken out of the live variables set, and variables which are read by the instruction, if any, are put into the live variables set. Moreover, the instruction becomes part of the slice. As an example, if the slicer reaches the pseudo-instruction `c = a + b`, and `c` is a member of live variables set, it removes `c` from it, puts `a` and `b` into it, and finally puts this instruction into the slice.

Control dependencies also play an important role in putting instructions into the slice or not. When an instruction becomes part of the slice based on the described liveness analysis above, all branches on which this instruction is dependent should also be put into the slice.

Therefore, these branches are put into a pending list, so that when the backward pass reaches a branch in the pending list, it is put into the slice. Moreover, the way branches update the live variables set differs from how regular instructions do so in the way described in the previous paragraph: when a branch must become part of the slice, its condition variable is put into the live variables set. For example, when the profiler reaches the pseudo-instruction `if (c)` (`c` is the condition variable) which is in the pending branch list, `c` is put into the live variables set, and the branch is put into the slice and removed from the pending branch list.

In practice and at machine-level instructions, variables are, in fact, registers and memory locations. Therefore, in a single-threaded program, the live variables set actually consists of a live memory set and a live registers set. On the other hand, Web applications are typically multi-threaded programs, and thus, it is required that our profiler also works for multi-threaded programs. The profiler assumes that even for a multi-threaded program, it is given a single instruction trace, which means that it requires that different threads are executed sequentially during the instruction trace collection phase. This makes the design of the profiler simpler because there is no need to handle synchronization between threads, and data dependence of instructions of different threads through shared memory can be easily identified by the liveness analysis described above. Finally, since the architectural context of the CPU changes when it switches the execution between threads, the profiler needs to keep a separate live registers set for each thread. Note that we should not have separate live memory sets for different threads because each thread has a distinct address space for local memory (i.e., heap and stack).

A.4 Evaluation Methodology

In this section, we utilize the proposed profiler to identify unnecessary computations in rendering real websites. We implemented the profiler in C++ based on the descriptions in the previous section. Our test Web browser is Google Chromium, which is an open-

source program [45]. For collecting instruction traces, we attach Intel's dynamic binary instrumentation tool, that is, Pin [32], to a specific tab of Chromium (each Chromium tab has its own separate process). Using a Pin tool written by us, we obtain the required information about the execution of instructions and store it in stable storage. In the rest of this section, we first explain the details of our Pin tool. Then, we describe the benchmarks and how slicing criteria are designated.

A.4.1 Dynamic Binary Instrumentation

Pin [32] is Intel's dynamic binary instrumentation tool, which can inspect and even manipulate dynamically executed instructions using only the program binary. The task of instrumentation and inspection/manipulation could be customized through writing Pin tools.

We wrote a Pin tool that collects static and dynamic information about the executed instructions. Static information includes the required data that could be extracted from the instruction opcodes, such as whether an instruction is a call, return, or direct/indirect conditional/unconditional branch, and which registers it accesses. Dynamic information includes data that are available at runtime, such as the addresses of memory locations accessed by an instruction, the ID of the thread where it is executed, and the system call number if the instruction is `syscall`.

System calls need special attention. Pin only instruments user-level code and does not inspect operating system instructions. System calls may change the value stored in registers and memory, thereby affecting the procedure of our liveness analysis. In order to solve this issue, we determined the record of all system calls that Chromium executes. We looked in the Linux kernel manual to understand how each of these system calls manipulate memory. For example, the syntax of `sendto` system call is as follows:

```
ssize_t sendto(int sockfd, const void *buf,
               size_t len, int flags,
               const struct sockaddr *dest_addr,
```

```
socklen_t addrlen);
```

When our Pin tool reaches a `sendto` system call, it indicates in the trace file that memory locations pointed by `buf` and `dest_addr` are read accesses. How registers are manipulated by a system call is specified in a CPU's ABI (Application Binary Interface). Our profiler takes care of this issue based on the standard specified in the Intel's x86-64 (i.e., AMD64) ABI, which is the processor architecture used in our experiments.

A.4.2 Benchmarks

We use the Chromium browser, as was briefly mentioned earlier, to generate real-world benchmarks. We collected four instruction trace sets from different websites: Amazon in desktop view, Amazon in emulated mobile view, Google Maps, and Bing. We chose these three websites because their appearance and user interface totally differ from each other. Moreover, the desktop and mobile views of Amazon are considerably different. The instruction traces of the first three benchmarks include the load time of the corresponding websites (i.e., Amazon and Google Maps); that is, the trace is collected from entering the URL to when the Web page is completely loaded. However, the last benchmark, i.e., Bing, includes the instructions of loading the Web page and browsing it in a typical way. The browsing is composed of several user actions: opening and closing the top right menu, clicking on a button to roll the news pane in the bottom of the page, and typing a term in the search bar.

In Chromium, each tab is actually a separate process composed of multiple threads. Before starting to collect the instruction trace of a tab of Chromium, we set affinity of the corresponding process to one, so that all the threads of that process are sequentially executed on only one CPU core. This requirement, as explained in Section A.3.2, is imposed by our profiler. Next, we attach our Pin tool to the tab's process to start collecting the trace of instructions, and we enter the URL of a website. Benchmarks are generated using Chromium v58 that was run on an Ubuntu 14.04 desktop with 8 GB of RAM and an Intel Xeon E31230

CPU; note that Pin only supports Intel CPUs.

As will be explained later in this section, for the slicing criteria that we use, we need to know the address of pixels buffer and the points in the trace at which they contain values that are going to be put on the screen. In order to achieve this knowledge, we studied the source code of Chromium and found the point in the code (which is inside the `RasterBufferProvider::PlaybackToMemory` function) where the final value of pixels (i.e., bitmaps) are written into a special buffer which corresponds to a tile of the screen (tiles are typically squares of 256×256 pixels). We put a unique instruction marker, that is, "`xchg %r13w, %r13w`", in a proper point in this function. We also modified the code of this function so that whenever Chromium executes it, the address of the tile buffer and its size are stored in an external file. This file and also the special instruction marker are, in fact, a set of slicing criteria provided to the profiler.

A.4.3 Choice of Slicing Criteria for Web Applications

As mentioned in Section A.2.2, in order for our profiler to effectively discover unnecessary computations of a Web application, slicing criteria should be carefully designated. Ideally, slicing criteria should contain all variables at exact program points that are somehow valuable and important to the user. Defining such criteria is a difficult task because relating user satisfaction in all possible executions to machine-level variables may not be practical or even possible. Therefore, we try to designate slicing criteria that closely match the ideal case. In this work, we use two types of slicing criteria: pixels buffer and system calls.

Pixels buffer. We define our first set of slicing criteria as the values of the pixels buffer that are shown to the user during rendering the page. The values of pixels of the display containing the Web page are actually the endpoint result of the application computations. Therefore, whatever that does not have any visible effect by no means—such as unused JavaScript and CSS codes, invisible layers, and page elements located at the very end of the page that are not shown on the first view of the Web page—will not be part of the calculated

slice.

System calls. System calls are, in fact, means by which a process communicates with the outside world, including the network and display monitor. Therefore, we define our second set of slicing criteria as the values used by any system calls. Note that the slice computed by this set of slicing criteria must be inclusive of that of the pixel-based criteria, and the reason that we also use such criteria is to capture important computations to the user that do not have any visual effect, such as bank transactions through the network or audio playback.

Both types of slicing criteria described above are browser-independent. Particularly, in the case of pixels buffer, we only need to locate in the browser's source code where this buffer is filled with the final value of the pixels. In other words, how the values stored in the pixels buffer are calculated, which may differ from one browser to another, does not affect the way the profiler performs slicing.

For the benchmark related to a complete browsing session—that is, loading and browsing the Web page for a while—the instructions that do not belong to the calculated slice through either of the mentioned types of slicing criteria specify computations that were not necessary for rendering the page in that particular session. On the other hand, such instructions for the benchmarks that only contain loading a Web page denote either computations that are unnecessary (similar to the complete browsing session case), or computations that would be useful if the user started browsing the page, e.g., computations that are responsible for preparing the state of the application for the interactions of the user with the page which do not have any visible effect at load time (such as pre-compiling JavaScript code that would be fired as soon as the user starts interacting with the page). Our results, however, show that the latter item includes a very small percentage of instructions, and almost all the instructions that do not belong to the calculated slice in the benchmarks that only contain the load time could be treated in a similar way to the benchmark containing both loading and browsing the page.

Table A.2: Slicing statistics of pixel-based approach for all instructions and important threads.

Threads	Amazon (desktop view) Load		Amazon (mobile view) Load		Google Maps Load		Bing Load + Browse	
	Pixels slice	Total instructions	Pixels slice	Total instructions	Pixels slice	Total instructions	Pixels slice	Total instructions
All	**46%**	6,217 M	**43%**	2,861 M	**47%**	4,238 M	**43%**	10,494 M
Main	52%	2,173 M	59%	764 M	61%	1,382 M	44%	3,499 M
Compositor	34%	1,711 M	35%	1,135 M	35%	1,698 M	34%	3,702 M
Rasterizer 1	55%	199 M	14%	76 M	78%	32 M	71%	617 M
Rasterizer 2	60%	66 M	13%	88 M	74%	29 M	52%	345 M
Rasterizer 3	54%	191 M	-	-	-	-	-	-

A.5 Results and Discussion

In this section, we present the output results of our profiler regarding doing pixel-based slicing on the collected instruction traces from different websites. Our results show that slicing based on either pixels buffer or system calls leads to almost the same slice. Hence, only results of pixel-based slicing are presented and discussed.

A.5.1 Calculated Slice

Table A.2 contains the statistics of the pixel-based slicing approach. The results show that the pixels slice is, on average, composed of **45%** of dynamically executed instructions in the four different benchmarks, which is an interestingly small percentage number. This implies that there is a good opportunity to identify useless computations in more than 50% of instructions. Note that in the Amazon benchmarks, the length of the trace in the mobile view (2.9 billion instructions) is so much smaller than that of the trace in the desktop view (6.2 billion instructions), which is because the first view of the Amazon Web page is much simpler in mobile displays as compared to desktop displays.

For the Bing benchmark, we also performed backward slicing starting from the time when the page was completely loaded back to the beginning time, which is composed of 1.7 billion instructions. The total slicing percentage for this experiment is 49.8%. On the other hand, when slicing is done starting from the end of the full trace, i.e., when the browsing session is complete, 50.6% of instructions that correspond to the load time are part of the calculated slice. This implies that browsing the Web page only makes about 1% more

instructions of load time become useful.

Table A.2 also includes statistics of three important thread types: main thread, compositor, and rasterizers. The main thread is mainly responsible for processing HTML, CSS, and JavaScript codes. The compositor thread handles the order of the layers containing the elements of the Web page and is also in charge of handling user inputs and animations. User inputs that do not cause any major change to the rendered page, such as scrolling, are handled in the compositor thread, but for other inputs, such as a mouse click to open a menu, the compositor thread notifies the main thread to render the changes. Moreover, the compositor thread also notifies the main thread when a new animation frame must be rendered. Chromium might launch a different number of rasterizer threads for each website. These light-weight threads translate graphical objects (e.g., lines and circles) into pixels. In our benchmarks, Amazon with desktop view had three rasterizer threads, while other benchmarks had only two rasterizers.

The slicing percentage of the compositor thread is almost the same across all the benchmarks, while that of the main and rasterizer threads varies and is website-specific. This is reasonable because HTML, CSS, and JavaScript codes of different websites, which are processed by the main thread, are not the same, and what will finally be rasterized and displayed on the screen completely depends on the website content. On the other hand, the responsibilities of the compositor thread are not dependent on the details of the website content. Calculating the correct order of the layers and determining whether or not they are visible; handling user inputs and forwarding them to the main thread if necessary; and notifying the main thread to render a new animation frame are generic, website-independent tasks performed by the compositor thread.

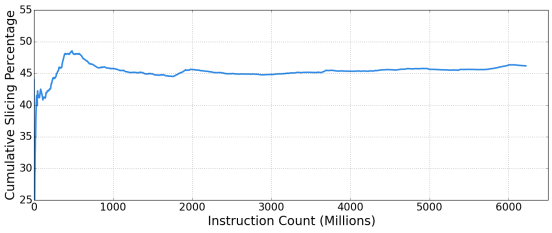
In the Amazon benchmark with mobile view, the slicing percentage of the rasterizer threads is very small. Note that for this benchmark, we emulated a mobile display using the Developers Tool of Chromium. The emulated display has a 360×640 resolution, which does not actually contain a large number of pixels. Therefore, these threads' effort to rasterize the

content seems to be not quite useful as it is reflected on a few pixels.

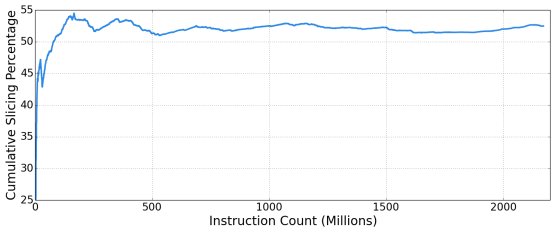
The slicing percentage of the compositor thread in all the benchmarks is also small. As mentioned in Section A.2.2, in the compositing algorithm of Chrome/Chromium, a backing store/cache is specified to each layer, either when the layer is visible or not, so that if the order of layers changes and some layers become visible, the correct content is displayed quickly. While this idea may bring performance, it may also lead to useless computations in case of the backing stores whose contents are never used because some layers are fully or partially overlapped during the whole browsing session. The low slicing percentage of the compositor thread indicates that more smart compositing algorithms could provide both performance and energy efficiency.

Figure A.4 shows how the slicing percentage changes in the backward pass for the pixel-based slicing criteria on different benchmarks. The x-axis in these charts shows the progress in the backward pass; therefore, the starting point on the x-axis corresponds to the time when the Web page is loaded or the browsing session is done, and the last point is related to the time when the Web page URL is entered. The y-axis shows the percentage of instructions of the slice for a specific point on the x-axis (aggregated from the starting point) in the instructions analyzed up to that point. The results are shown both for the instructions of all threads and also for the instructions of only the main thread. We can see that the changes in the overall slicing percentage of all threads in the backward pass is almost constant in large intervals. This implies that the distribution of instructions of the slices among all instructions is fairly even overall. However, the range of changes in the slicing percentage of the main thread is more in contrast to all threads. This means that computation regions that do or do not contribute to the pixel values are more conspicuous in the main thread as compared to other threads. It is also interesting to notice that for the main thread in the Bing benchmark (Figure A.4h), there are some points where the slicing percentage suddenly increases (i.e., $x = 400$, $x = 1100$, and $x = 1800$), and then there is a gradual decrease in it. These points correspond to the user interactions that make the main

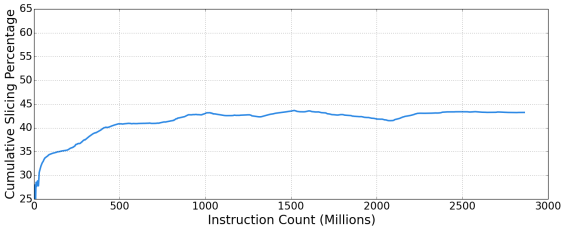
thread render the imposed changes, such as rolling the news pane. Moreover, near the end of the chart (i.e., $x = 3000$), there is another considerable increase in the slicing percentage, which is related to loading the page. All in all, whenever rendering or re-rendering happens, the overall slicing percentage increases in that it leads to changes in the pixel values.



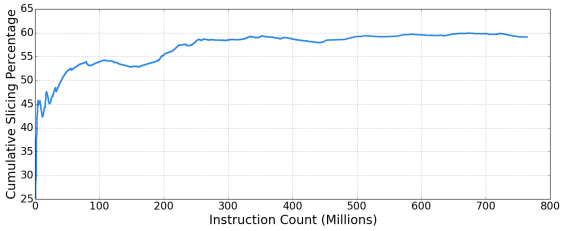
(a) Loading Amazon (desktop view): all threads



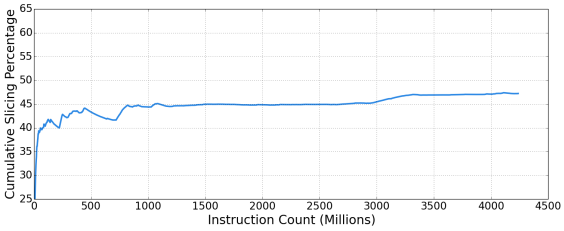
(b) Loading Amazon (desktop view): main thread



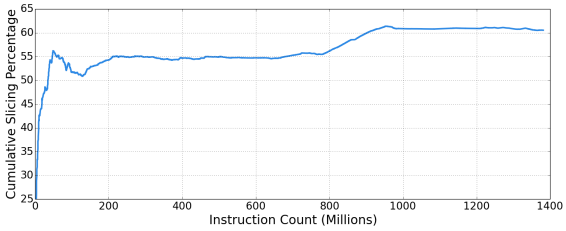
(c) Loading Amazon (mobile view): all threads



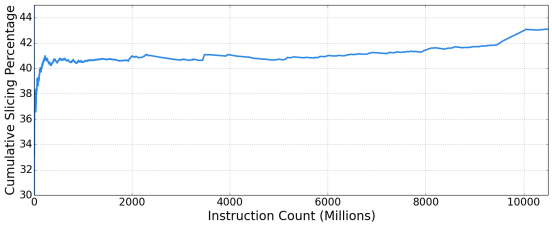
(d) Loading Amazon (mobile view): main thread



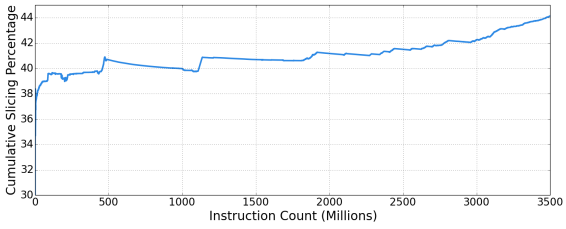
(e) Loading Google Maps: all threads



(f) Loading Google Maps: main thread



(g) Loading and browsing Bing: all threads



(h) Loading and browsing Bing: main thread

Figure A.4: Changes of slicing percentage over the backward pass. $x = 0$ indicates the Web page is loaded or the browsing session is done, and the last point on the x-axis corresponds to entering the Web page URL.

A.5.2 Categorization of Unnecessary Computations

Now that the slice of instructions that determine the value of pixels is calculated, we categorize unnecessary computations by analyzing the instructions that are not part of the calculated slice (~55% of all instructions). We closely examined the functions that each dynamically executed instruction belongs to using the symbol table stored in the application binary and used the namespace of the functions as the basis for categorization.

The categories of potentially unnecessary instructions by this namespace analysis are: JavaScript, Debugging, Inter-Process Communication (IPC), Multi-threading, Compositing, Graphics, CSS, and Other. Note that when compiling the Chromium source code, all debugging options were turned off, and the Debugging category reflects the default debugging mechanisms built in Chromium. IPC corresponds to the communication of the tab process with browser's main process. In Chromium, there is a single main process which manages the views of different tabs and other things such as browser extensions. Each process in Chromium is multi-threaded, and the Multi-threading category mainly consists of PThread code, which enables thread communication and synchronization. The Compositing category relates to the operations of the compositor thread, which is also the last stage shown in Figure A.1. The Graphics category basically corresponds to the Paint stage of the rendering pipeline (Figure A.1), and the CSS category is related to style and layout calculation in the rendering pipeline. The Other category mainly consists of event scheduling; note that all threads in Chromium are event-driven in nature, and event scheduling deals with managing an event queue, which holds events that should be executed.

Distribution of the categories of potentially unnecessary instructions through the namespace analysis is illustrated in Figure A.5. Note that through this methodology, not all instructions could be categorized because not all functions have a specific namespace. The results shown in this figure include 74%, 59%, 53%, and 61% of the Amazon in desktop view, Amazon in mobile view, Google Maps, and Bing benchmarks, respectively.

Figure A.5 shows that most of the potentially unnecessary instructions belong to the first

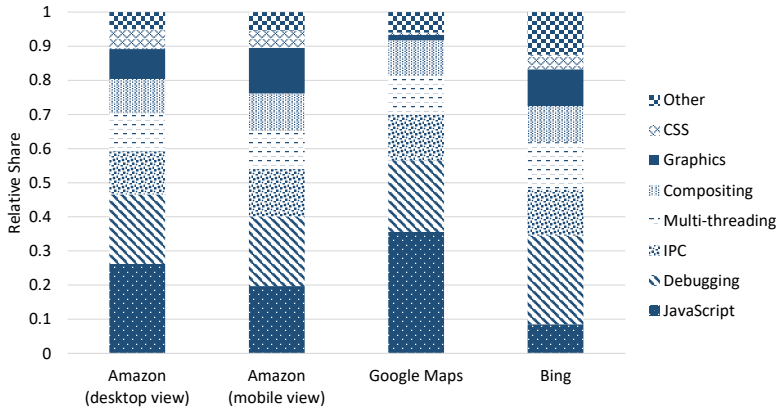


Figure A.5: Categorization of potentially unnecessary computations and their distribution through analysis of instructions that do not belong to the pixel-based slice.

three categories, which are JavaScript, Debugging, and IPC. Presence of JavaScript in this list is not surprising. Also, it is reasonable that debugging codes are detected as unnecessary in that their execution has nothing to do with what is displayed on the screen. However, the IPC category needs more inspection because execution of instructions belonging to this category might have useful effect on the browser's main process; this is left as future work. It is interesting that in the Bing benchmark, which includes both loading and browsing the page, the JavaScript category has a smaller share as compared to other benchmarks, which only include loading the page. This implies that, generally, loading is the most intensive time in terms of processing JavaScript codes, not all of which are useful in a browsing session. Therefore, deferring processing of JavaScript codes to a time when they are really needed could provide better performance in Web applications. It is also worth mentioning that because of the noticeable presence of the Multi-threading category in Figure A.5, and also because the share of the Other category, which mainly has to do with event scheduling, increases by browsing the page, assignment of tasks to different threads and scheduling mechanism of Chromium need reconsideration.

A.6 Related Work

A.6.1 Workload Characterization of Web Applications

Prior work on characterization of Web applications mainly focused on JavaScript [180, 182, 191]. In contrast, in this work, we essentially characterize the whole JavaScript and rendering engines and determine computations that are useful for users. [180] and [182] characterize dynamic behavior of JavaScript workloads in terms of functions and objects, events and event handlers, and memory allocation. [180] concludes that JavaScript behavior of real Web applications and available benchmarks differ, and the benchmarks are not representative of real-world websites. [182] points out common misunderstandings of the behavior of JavaScript programs mainly caused by the available benchmarks. As a result, benchmarks inspired by real user actions have been developed [42, 181].

A.6.2 Performance Optimization of Web Applications

Many techniques have been proposed in prior work to improve performance of Web applications targeting various components of them. These techniques mainly enhance the JavaScript engine or improve the load time of Web pages.

JavaScript. Much prior work has focused on improving the JavaScript JIT compiler and execution engine. [53] enhances object type prediction of a JavaScript compiler by decoupling prototypes and method bindings from the object type. [124] uses server-side profiling to reduce deoptimizations done at client-side JavaScript engines. WebAssembly [107] is low-level, high-performance code compiled from C/C++ which could be utilized in Web applications through specific JavaScript APIs. Prior work also tried to bring parallelization to the JavaScript engine. [154] proposes offloading runtime checks of the JavaScript JIT compiler to a separate thread. [153] tries to parallelize loops in compute-intensive JavaScript applications.

Web page load time. The load time of Web pages has also received lots of attention in

prior work due to its high impact on user experience. [148] proposes a coupled design of a server, which decomposes Web pages into sub pages on-the-fly, and a Web browser, that processes the sub pages in parallel. [146] leverages a machine learning model to predict future Web accesses of a user and prefetch the Web content. [173] decreases the load time of Web pages by caching and re-using JavaScript objects across browsing sessions. [72] and [216] dynamically reprioritize the content of a Web page to improve the load time of the Web page and sooner deliver resources that are critical to user experience.

A.6.3 Energy-efficient Mobile Web Applications

Energy efficiency of Web applications is a critical matter in mobile devices such as smartphones. Prior work mainly focused on frequency/voltage scaling of heterogeneous multiprocessors [177, 193, 226, 227, 229]. In [227], statistical models are achieved to estimate the time and energy consumption of loading Web pages based on their characteristics—such as, number of HTML tags, number of CSS rules, and content size. Based on these models, proper frequency/voltage of Arm big.LITTLE cores [19] are found after parsing the Web page. [177] characterizes the energy consumed in different processes and threads of a Web browser and proposes several power management policies on heterogeneous multiprocessor platforms. [226] and [229] propose energy-efficient schedulers of a heterogeneous mobile architecture based on the QoS requirements of users, which is, respectively, determined by automatic reasoning based on intensity and latency, and two novel CSS language extensions provided for Web developers.

A.6.4 Architectural Support for Web Applications

Due to widespread use of Web applications, prior work also proposed specialized hardware and architectures for them. [228] identifies fine-grained parallelism in applying styles to HTML elements and proposes a specialized hardware unit for it. It also proposes a specific cache for the document object model tree since its content is heavily re-used while

rendering a Web page. In [76], a specialized prefetcher is designed that takes advantage of long latency cache misses to bring to cache data and instructions required for future events that are in the event queue. [79] accelerates JavaScript object accesses through a hardware table similar to a branch target buffer.

A.7 Conclusion

The performance of today's Web applications is often unsatisfactory to users, and in this chapter, we argued one of the reasons for it is that there are unnecessary computations occurring in Web applications which could be avoided or scheduled in a better way. We designed a profiler that effectively identifies computations that are important to the user. To the best of our knowledge, this is the first work that quantitatively characterizes unnecessary computations of Web applications. The profiler detects instructions contributing to what is shown to the user on the device display during rendering a Web page. We showed that only 45% of dynamically executed instructions in the rendering process of the browser under test are useful for calculating the value of the pixels displayed to the user on average. By analyzing the rest of the instructions, we revealed inefficiencies of the Web browser (e.g., the compositing algorithm) and provided a categorization of computations that are either completely wasted or could be deferred to a more appropriate time (e.g., compiling a piece of JavaScript code when it is really needed), thereby providing opportunities for higher performance or reduced energy consumption.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[2] Compositing in Blink/Webcore. https://bit.ly/3g5orcv, 2014.

[3] Network Functions Virtualization. http://www.hp.com/hpinfo/newsroom/press_kits/2014/MWC/White_Paper_NFV.pdf, 2014.

[4] Integrating SDN into the Data Center. https://www.juniper.net/assets/es/es/local/pdf/whitepapers/2000542-en.pdf, 2015.

[5] Using Request Idle Callback. https://tinyurl.com/ybyseoo5, 2015.

[6] Intel 64 and IA-32 Architectures Software Developer's Manual, September 2016. Volume 3A: System programming guide, part 2.

[7] The need for mobile speed (DoubleClick by Google). https://bit.ly/3peqkb4, 2016.

[8] IEEE 802.3bs-2017: 200 Gbps and 400 Gbps Ethernet. http://www.ieee802.org/3/bs/, 2017.

[9] Powering 8K Video for Next-Generation IP Broadcasting. https://www.mellanox.com/related-docs/whitepapers/WP_Mellanox_VMA.pdf, 2017.

[10] High Performance Computing on AWS Redefines What is Possible. https://d1.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf, 2018.

[11] Reaching the Summit with InifiBand. https://www.mellanox.com/related-docs/solutions/hpc/CS_ORNL_Summit_InfiniBand.pdf, 2018.

[12] Cloud-Ready High Performance Computing. https://www.suse.com/media/white-paper/cloud_ready_high_performance_computing_taking_hpc_to_the_clouds_wp.pdf, 2019.

[13] Intel 64 and IA-32 Architectures Software Developer's Manual, October 2019. Volume 2B: Instruction Set Reference.

[14] Intel Data Streaming Accelerator (DSA). https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator, 2019.

[15] The Future of HPC Cloud Computing. https://services.google.com/fh/files/misc/gcp_hyperion_tech_spotlight_aug_2019.pdf, 2019.

[16] Cloud-Native Networking for a 5G Era. https://www.abiresearch.com/blogs/2020/04/06/cloud-native-networking-5g-era/, 2020.

[17] Measure Performance with the RAIL Model. https://web.dev/rail/, 2020.

[18] Arm AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/e/ACE-Protocol-Specification, 2021.

[19] Arm big.LITTLE Technology. https://www.arm.com/why-arm/technologies/big-little, 2021.

[20] Arm Cache Stashing. https://bit.ly/3e3a3Ak, 2021.

[21] Bootstrap. http://getbootstrap.com/, 2021.

[22] Cache Coherent Interconnect for Accelerators (CCIX). https://www.ccixconsortium.com/, 2021.

[23] Compute Express Link (CXL). https://www.computeexpresslink.org/, 2021.

[24] Data Plane Development Kit (DPDK). https://www.dpdk.org/, 2021.

[25] Firefox's Hardware Acceleration. https://support.mozilla.org/en-US/kb/performance-settings, 2021.

[26] GPU Accelerated Compositing in Chrome. https://tinyurl.com/no64sem, 2021.

[27] Intel Data Direct I/O (DDIO) Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html/, 2021.

[28] Intel Intelligent Storage Acceleration Library (ISA-L). https://software.intel.com/en-us/isa-l, 2021.

[29] Intel Optane Technology. http://www.intel.com/optane/, 2021.

[30] Intel QuickPath Interconnect (QPI). https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, 2021.

[31] Intel Ultra Path Interconnect (UPI). https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html, 2021.

[32] Intel's Pin. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2021.

[33] jQuery. https://jquery.com/, 2021.

[34] Multi-threaded Rasterization in Chrome. https://tinyurl.com/yaksfwz8, 2021.

[35] Open vSwitch with DPDK. http://docs.openvswitch.org/en/latest/intro/install/dpdk/, 2021.

[36] Pktgen: Traffic Generator Powered by DPDK. http://git.dpdk.org/apps/pktgen-dpdk/, 2021.

[37] React. https://reactjs.org/, 2021.

[38] Samsung SmartSSD Computational Storage Drive. https://samsungatfirst.com/smartssd/, 2021.

[39] Samsung Z-SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/, 2021.

[40] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2021.

[41] Speculative Parsing. https://tinyurl.com/yxta9zyu, 2021.

[42] Speedometer 2.0. https://browserbench.org/Speedometer2.0/, 2021.

[43] Storage Performance Development Kit (SPDK). https://spdk.io/, 2021.

[44] T6 Crypto Offload. https://www.chelsio.com/crypto-offload/, 2021.

[45] The Chromium Web Browser. https://www.chromium.org/, 2021.

[46] The Geekbench Benchmark Suite. https://www.geekbench.com/, 2021.

[47] The Go Programming Language. https://golang.org/, 2021.

[48] urdma: User-space Software RDMA. https://github.com/zrlio/urdma/, 2021.

[49] Vue.js: The Progressive JavaScript Framework. https://vuejs.org/, 2021.

[50] Webpack: Build System and Module Bundler. https://webpack.js.org/, 2021.

[51] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. *Remote Memory in the Age of Fast Networks*, page 121–127. Association for Computing Machinery, New York, NY, USA, 2017.

[53] Wonsun Ahn, Jiho Choi, Thomas Shull, María J Garzarán, and Josep Torrellas. Improving JavaScript Performance by Deconstructing the Type System. *ACM SIGPLAN Notices*, 49(6):496–507, 2014.

[54] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[55] Sam Ainsworth and Timothy M. Jones. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 578–592, New York, NY, USA, 2018. Association for Computing Machinery.

[56] Alaa R. Alameldeen and David A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[57] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. Memory Hierarchy for Web Search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656, Feb 2018.

[58] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. AsmDB: Understanding and Mitigating Front-end Stalls in Warehouse-scale Computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 462–473, New York, NY, USA, 2019. ACM.

[59] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Evaluation of Hardware Data Prefetchers on Server Processors. *ACM Comput. Surv.*, 52(3), June 2019.

[60] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[62] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, December 2007.

[63] Andrew Beaumont-Smith and Cheng-Chew Lim. Parallel Prefix Adder Design. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ARITH '01, page 218, USA, 2001. IEEE Computer Society.

[64] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[65] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[66] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt. Performance Analysis of System Overheads in TCP/IP Workloads. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 218–230, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Nathan L. Binkert, Ali G. Saidi, and Steven K. Reinhardt. Integrated Network Interfaces for High-bandwidth TCP/IP. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 315–324, New York, NY, USA, 2006. ACM.

[68] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[69] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[70] Anne Bracy, Kshitij Doshi, and Quinn Jacobson. Disintermediated Active Communication. *IEEE Computer Architecture Letters*, 5(2):15–15, 2006.

[71] Richard P Brent and Hsiang T Kung. A Regular Layout for Parallel Adders. *IEEE transactions on Computers*, (3):260–264, 1982.

[72] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 439–453, Oakland, CA, May 2015. USENIX Association.

[73] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 40–52, New York, NY, USA, 1991. Association for Computing Machinery.

[74] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[75] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[76] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. Accelerating Asynchronous Programs through Event Sneak Peek. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 642–654, 2015.

[77] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[78] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the Killer Microsecond. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 627–640. IEEE Press, 2018.

[79] Jiho Choi, Thomas Shull, Maria J Garzaran, and Josep Torrellas. Shortcut: Architectural Support for Fast Object Access in Scripting Languages. *ACM SIGARCH Computer Architecture News*, 45(2):494–506, 2017.

[80] Shihabur Rahman Chowdhury, Mohammad A Salahuddin, Noura Limam, and Raouf Boutaba. Re-Architecting NFV Ecosystem with Microservices: State of the Art and Research Challenges. *IEEE Network*, 33(3):168–176, 2019.

[81] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 939–953, New York, NY, USA, 2019. ACM.

[82] Jaewoong Chung and Karin Strauss. User-Level Interrupt Mechanism for Multi-Core Architectures, August 28 2012. US Patent 8,255,603.

[83] Travis Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical report, 1993.

[84] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 35–48, New York, NY, USA, 2019. ACM.

[85] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[86] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 482–493, New York, NY, USA, 2007. Association for Computing Machinery.

[87] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, April 2018. USENIX Association.

[88] Gautham K. Dorai and Donald Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, page 30, USA, 2002. IEEE Computer Society.

[89] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. HIP: Hybrid Interrupt-Polling for the Network Interface. *SIGOPS Oper. Syst. Rev.*, 35(4):50–60, October 2001.

[90] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking Energy. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 393–406, Denver, CO, June 2016. USENIX Association.

[91] Babak Falsafi and Thomas F Wenisch. A Primer on Hardware Prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67, 2014.

[92] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic Routing Encapsulation (GRE). Technical report, RFC 2784, March, 2000.

[93] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[94] H. Fatih Ugurdag and Onur Baskirt. Fast Parallel Prefix Logic Circuits for n2n Round-Robin Arbitration. *Microelectron. J.*, 43(8):573–581, August 2012.

[95] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[96] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[97] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[98] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 124, 2004.

[99] Sheila Frankel, R Glenn, and S Kelly. The AES-CBC Cipher Algorithm and Its Use with IPsec. 2003.

[100] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[101] Tanmay Gangwani, Adam Morrison, and Josep Torrellas. CASPAR: Breaking Serialization in Lock-Free Multicore Synchronization. In *Proceedings of the Twenty-First*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, pages 789–804, New York, NY, USA, 2016. ACM.

[102] Younghwan Go, Muhammad Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 83–96, USA, 2017. USENIX Association.

[103] Hossein Golestani, Scott Mahlke, and Satish Narayanasamy. Characterization of Unnecessary Computations in Web Applications. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 11–21, 2019.

[104] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F. Wenisch. Software Data Planes: You Can't Always Spin to Win. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 337–350, New York, NY, USA, 2019. Association for Computing Machinery.

[105] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A Case for Unlimited Watchpoints. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 159–172, New York, NY, USA, 2012. ACM.

[106] Pankaj Gupta and Nick McKeown. Designing and Implementing a Fast Crossbar Scheduler. *IEEE Micro*, 19(1):20–28, January 1999.

[107] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[108] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 195–206, New York, NY, USA, 2010. Association for Computing Machinery.

[109] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[110] Wanja Hofer, Daniel Lohmann, and Wolfgang Schroder-Preikschat. Sleepy Sloth: Threads as Interrupts as Threads. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, RTSS '11, page 67–77, USA, 2011. IEEE Computer Society.

[111] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 17–33, Renton, WA, April 2018. USENIX Association.

[112] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 50–59, Washington, DC, USA, 2005. IEEE Computer Society.

[113] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, 2014. USENIX Association.

[114] Xiaowei Jiang, Yan Solihin, Li Zhao, and Ravishankar Iyer. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 169–180, Washington, DC, USA, 2009. IEEE Computer Society.

[115] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[116] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, page 170–180, New York, NY, USA, 1997. Association for Computing Machinery.

[117] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[118] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[119] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[120] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[121] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM.

[122] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald 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)*, pages 171–186, Renton, WA, April 2018. USENIX Association.

[123] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[124] Madhukar N Kedlaya, Behnam Robatmili, and Ben Hardekopf. Server-Side Type Profiling for Optimizing Client-Side JavaScript Engines. *ACM SIGPLAN Notices*, 51(2):140–153, 2015.

[125] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, Boston, MA, February 2019. USENIX Association.

[126] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, pages 41–45, Berkeley, CA, USA, 2016. USENIX Association.

[127] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, New York, NY, USA, 2017. ACM.

[128] Maciek Konstantynowicz, Patrick Lu, and Shrikant M Shah. Benchmarking and Analysis of Software Data Planes, 2017.

[129] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server Engineering Insights for Large-Scale Online Services. *IEEE Micro*, 30(4):8–19, July 2010.

[130] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Ingens: Huge Page Support for the OS and Hypervisor. *SIGOPS Oper. Syst. Rev.*, 51(1):83–93, September 2017.

[131] Alexey Lavrov and David Wentzlaff. HyperTRIO: Hyper-Tenant Translation of I/O Addresses. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 487–500, 2020.

[132] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[133] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 591–605, New York, NY, USA, 2020. Association for Computing Machinery.

[134] J. Li and B. Li. Erasure Coding for Cloud Storage Systems: A Survey. *Tsinghua Science and Technology*, 18(3):259–272, June 2013.

[135] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[136] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 469–480, New York, NY, USA, 2009. Association for Computing Machinery.

[137] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. Spin Detection Hardware for Improved Management of Multithreaded Systems. *IEEE Trans. Parallel Distrib. Syst.*, 17(6):508–521, June 2006.

[138] Guangdeng Liao and Laxmi Bhuyan. Performance Measurement of an Integrated NIC Architecture with 10GbE. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects*, HOTI '09, pages 52–59, Washington, DC, USA, 2009. IEEE Computer Society.

[139] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[140] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium*

on Computer Architecture, ISCA '09, page 267–278, New York, NY, USA, 2009. Association for Computing Machinery.

[141] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[142] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 318–333, New York, NY, USA, 2019. ACM.

[143] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving Resource Efficiency at Scale with Heracles. *ACM Trans. Comput. Syst.*, 34(2):6:1–6:33, May 2016.

[144] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 65–76, Boston, MA, 2012. USENIX.

[145] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC® CMP Processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, page 93–104, USA, 2005. IEEE Computer Society.

[146] Dimitrios Lymberopoulos, Oriana Riva, Karin Strauss, Akshay Mittal, and Alexandros Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. *ACM SIGARCH Computer Architecture News*, 40(1):1–12, 2012.

[147] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, page 165–171, USA, 1994. IEEE Computer Society.

[148] HaoHui Mai, Shuo Tang, Samuel T. King, Calin Cascaval, and Pablo Montesinos. A Case for Parallelizing Web Pages. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*, Berkeley, CA, June 2012. USENIX Association.

[149] Howard Mao. Hardware Acceleration for Memory to Memory Copies. Master's thesis, EECS Department, University of California, Berkeley, Jan 2017.

[150] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, August 2014.

[151] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve

Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[152] Pankaj Mehra. Samsung smartSSD: Accelerating Data-Rich Applications. *Proceedings of the Flash Memory Summit*, 2019.

[153] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 87–98, 2011.

[154] Mojtaba Mehrara and Scott Mahlke. Dynamically Accelerating Client-side Web Applications through Decoupled Execution. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 74–84, 2011.

[155] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 205–216, New York, NY, USA, 2009. ACM.

[156] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power Management of Online Data-intensive Services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 319–330, New York, NY, USA, 2011. ACM.

[157] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, page 106–113, New York, NY, USA, 1991. Association for Computing Machinery.

[158] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch. Express-Lane Scheduling and Multithreading to Minimize the Tail Latency of Microservices. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 194–199, June 2019.

[159] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing Server Efficiency in the Face of Killer Microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198, Feb 2019.

[160] A. Mirhosseini and T. F. Wenisch. The Queuing-First Approach for Tail Management of Interactive Services. *IEEE Micro*, 39(4):55–64, July 2019.

[161] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes. In *2020*

53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 852–867, 2020.

[162] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 207–219. IEEE, 2020.

[163] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 254–265, Feb 2006.

[164] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, page 247–258, New York, NY, USA, 1996. Association for Computing Machinery.

[165] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. *HP laboratories*, 27:28, 2009.

[166] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 129–140, 2003.

[167] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[168] Vijay Nagarajan and Rajiv Gupta. ECMon: Exposing Cache Events for Monitoring. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 349–360, New York, NY, USA, 2009. Association for Computing Machinery.

[169] Dmitry Namiot and Manfred Sneps-Sneppe. On Micro-services Architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[170] Gilbert Neiger and Rajesh M Sankaran. Delivering Interrupts to User-Level Applications, March 20 2018. US Patent 9,921,984.

[171] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. ASPLOS '14, page 3–18, New York, NY, USA, 2014. Association for Computing Machinery.

[172] Fabian Oboril and Mehdi B. Tahoori. ExtraTime: Modeling and Analysis of Wearout Due to Transistor Aging at Microarchitecture-level. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[173] JinSeok Oh and Soo-Mook Moon. Snapshot-based Loading-Time Acceleration for Web Applications. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 179–189, 2015.

[174] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[175] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[176] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[177] Nadja Peters, Sangyoung Park, Samarjit Chakraborty, Benedikt Meurer, Hannes Payer, and Daniel Clifford. Web Browser Workload Characterization for Power Management on HMP Platforms. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2016.

[178] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1203–1216, New York, NY, USA, 2020. Association for Computing Machinery.

[179] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[180] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *USENIX Conference on Web Application Development (WebApps 10)*. USENIX Association, June 2010.

[181] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 677–694, 2011.

[182] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 1–12, 2010.

[183] Marcin Rogawski, Ekawat Homsirikamol, and Kris Gaj. A Novel Modular Adder for One Thousand Bits and More Using Fast Carry Chains of Modern FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[184] P. Rogers. Heterogeneous System Architecture Overview. In *2013 IEEE Hot Chips 25 Symposium (HCS)*, pages 1–41, Aug 2013.

[185] Rusty Russell. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[186] Frederick Ryckbosch, Stijn Polfliet, and Lieven Eeckhout. Trends in Server Energy Proportionality. *Computer*, 44(9):69–72, September 2011.

[187] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, page 187–198, USA, 2010. IEEE Computer Society.

[188] Curt Schimmel. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley Reading, 1994.

[189] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up Latencies for Processor Idle States on Current x86 Processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.

[190] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[191] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[192] Lance Shelton. High Performance I/O with NUMA Systems in Linux. *Linux Foundation Event*, 2013.

[193] Davesh Shingari, Akhil Arunkumar, Benjamin Gaudette, Sarma Vrudhula, and Carole-Jean Wu. DORA: Optimizing Smartphone Energy Efficiency and Web Browser Performance under Interference. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 64–75, 2018.

[194] Robert T Short, John M Parchem, and David N Cutler. Method and Apparatus for Reducing the Rate of Interrupts by Generating a Single Interrupt for a Group of Events, January 13 1998. US Patent 5,708,814.

[195] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[196] Gabriel Southern and Jose Renau. Overhead of Deoptimization Checks in the V8 JavaScript Engine. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[197] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[198] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 513–526, New York, NY, USA, 2019. ACM.

[199] Akshitha Sriraman and Thomas F Wenisch. μSuite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[200] Akshitha Sriraman and Thomas F. Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[201] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján. FastPath: Towards Wire-Speed NVMe SSDs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 170–1707, Aug 2018.

[202] Karin Strauss and Jaewoong Chung. Flexible Notification Mechanism for User-Level Interrupts, October 9 2012. US Patent 8,285,904.

[203] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 85–96, New York, NY, USA, 2004. Association for Computing Machinery.

[204] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The NEBULA RPC-Optimized Architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 199–212, Los Alamitos, CA, USA, jun 2020. IEEE Computer Society.

[205] E. Takeda, Y. Nakagome, H. Kume, and S. Asai. New hot-carrier injection and device degradation in submicron MOSFETs. *IEE Proceedings I - Solid-State and Electron Devices*, 130(3):144–150, June 1983.

[206] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667, 2021.

[207] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, Oakland, CA, February 2018. USENIX Association.

[208] Shelby Thomas, Geoffrey M. Voelker, and George Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[209] Frank Tip. A Survey of Program Slicing Techniques. Technical report, Amsterdam, The Netherlands, 1994.

[210] Abhishek Tiwari and Josep Torrellas. Facelift: Hiding and Slowing Down Aging in Multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 129–140, Washington, DC, USA, 2008. IEEE Computer Society.

[211] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, Renton, WA, April 2018. USENIX Association.

[212] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 191–202, New York, NY, USA, 1996. ACM.

[213] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[214] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. Detecting Transient Bottlenecks in n-Tier Applications Through Fine-Grained Analysis. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 31–40, Washington, DC, USA, 2013. IEEE Computer Society.

[215] Wenping Wang, Shengqi Yang, Sarvesh Bhardwaj, Sarma Vrudhula, Frank Liu, and Yu Cao. The Impact of NBTI Effect on Combinational Circuit: Modeling, Simulation, and Analysis. *IEEE Trans. Very Large Scale Integr. Syst.*, 18(2):173–183, February 2010.

[216] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up Web Page Loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 109–122, Santa Clara, CA, March 2016. USENIX Association.

[217] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient Metadata Management for Irregular Data Prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 449–461, New York, NY, USA, 2019. Association for Computing Machinery.

[218] Z. Yang, C. Liu, Y. Zhou, X. Liu, and G. Cao. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76, Nov 2018.

[219] Jiabin Ye, Cheng Zhang, Lei Ma, Haibo Yu, and Jianjun Zhao. Efficient and Precise Dynamic Slicing for Client-Side JavaScript Programs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 449–459, 2016.

[220] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect Memory Prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 178–190, New York, NY, USA, 2015. Association for Computing Machinery.

[221] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.

[222] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 601–614, New York, NY, USA, 2019. Association for Computing Machinery.

[223] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 73–80, New York, NY, USA, 2019. Association for Computing Machinery.

[224] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. IWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st*

Annual International Symposium on Computer Architecture, ISCA '04, page 224, USA, 2004. IEEE Computer Society.

[225] Yanqi Zhou and David Wentzlaff. MITTS: Memory Inter-arrival Time Traffic Shaping. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 532–544, Piscataway, NJ, USA, 2016. IEEE Press.

[226] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-Based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, 2015.

[227] Yuhao Zhu and Vijay Janapa Reddi. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, 2013.

[228] Yuhao Zhu and Vijay Janapa Reddi. WebCore: Architectural Support for Mobile Web Browsing. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 541–552, 2014.

[229] Yuhao Zhu and Vijay Janapa Reddi. GreenWeb: Language Extensions for Energy-Efficient Mobile Web Computing. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–160, 2016.