TOWARDS HIGH-SPEED NETWORKING IN THE POST-MOORE ERA

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Vishal Shrivastav
August 2020

TOWARDS HIGH-SPEED NETWORKING IN THE POST-MOORE ERA

Vishal Shrivastav, Ph.D.
Cornell University 2020

The motivation behind this dissertation stems from two polarizing trends inside modern datacenters—on the one hand, the bandwidth demand and link speed within datacenters keep increasing rapidly as applications keep getting more distributed, and resources (e.g., storage) keep getting disaggregated; but on the other hand, the processing and switching speeds of the underlying networking infrastructure, comprising primarily of general-purpose CPUs and packet switches, is scaling much more slowly due to the slowdown in Moore's law and the end of Dennard scaling. Thus, at the end-hosts, it is becoming increasingly difficult to saturate the high-speed links using CPU-based network stack, and in the network core, it is becoming increasingly difficult to build high-speed switching fabric in a cost and power effective manner using packet switches. To that end, this dissertation looks beyond the general-purpose CPUs and the packet switches as the building blocks for datacenter networks, and instead presents two promising alternatives that demonstrate that it is possible to build high-performance, high-speed end-host network stacks and switching fabrics at low cost and power, even as Moore's law continues to slow down.

First, with regards to the end-host network stack, this dissertation demonstrates the need to complement general-purpose CPUs with domain-specific processors for network processing, to keep up with increasing link speeds. However, one of the key challenges with designing a domain-specific processor is providing the right balance between programmability and performance.

Using packet scheduling as the target network processing application, our main contribution here is a new packet scheduling primitive, called Push-In-Extract-Out (PIEO), and the corresponding hardware architecture implementing the primitive, that together form an idealistic packet scheduler which is *simultaneously* programmable, scalable, and high-speed—PIEO primitive is more expressive (programmable) than any state-of-the-art packet scheduling primitive, and PIEO's hardware architecture could easily scale to 10s of thousands of flows (>30× more scalable than state-of-the-art) while making scheduling decisions in $O(1)$ time (up to 300× faster than a single CPU core).

Second, with regards to the switching fabric, this dissertation demonstrates that by using fast circuit switches (that could reconfigure within nanoseconds) as building blocks, one could build extremely high-speed and high-performance switching fabrics in a cost and power effective manner. The key challenge here is designing a high-speed control plane for the circuit-switched network that could complement the fine-grained reconfiguration technology by being able to schedule the right set of circuits every few nanoseconds. In that regard, our main contribution here is a new control plane architecture for circuit switching which is a stark departure from the traditional (slow) centralized controller-based architectures, and is in fact a fully de-centralized and traffic agnostic design that could schedule high-performing circuit configurations at nanosecond granularity. Further, our design is agnostic to the underlying circuit switching technology, and can operate with electrical, optical, or wireless technologies alike. We demonstrate that the resulting circuit-switched network, called Shoal, can effectively scale to high switching speeds while achieving comparable or better throughput and latency than several state-of-the-art packet-switched network designs at significantly lower power and cost.

Overall, through PIEO and Shoal, this dissertation demonstrates that by effectively leveraging the promise of domain-specific processing and fast circuit switching, one can indeed build efficient high-speed end-host network stacks and switching fabrics even in light of continued slowdown in Moore's law and the end of Dennard scaling.

BIOGRAPHICAL SKETCH

Vishal Shrivastav is a Ph.D. candidate in the department of Computer Science at Cornell University, advised by Hakim Weatherspoon. He holds a Master of Science in Computer Science from Cornell University and a Bachelor of Technology in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur.

Dedicated to my grandmother, the late Renuka Sahay,

for being the kindest, most loving soul that ever lived.

ACKNOWLEDGEMENTS

First, I would like to express my deepest gratitude to my advisor, Hakim Weatherspoon. It is very true that your Ph.D. advisor has the greatest influence on how your graduate study experience would turn out, and so it is no coincidence that I had such am amazing experience as a graduate student with Hakim as my advisor. I am so thankful for all his support and guidance over the years, and for all the independence he gave me with my research. In fact, there is an endless list of things that I could thank Hakim for, so instead I am just going to say this to him – Thank you for everything!

Next, I would like to thank my amazing research collaborators, starting with Rachit Agarwal. I worked with Rachit through most of my graduate studies, and learned a lot from him during that time, especially about how to critique research ideas and how to effectively present and defend my own ideas. I cannot thank him enough for his mentorship. Next, I would also like to express my sincerest gratitude to Hitesh Ballani and Paolo Costa. I worked closely with Hitesh and Paolo for almost two years during the early days of my graduate studies. During this time, they helped me navigate the existing landscape of computer networking research, which really helped kick-start my research career. Finally, I would like to thank the entire computer science faculty at Cornell, for always being so kind to me and for being such great sources of inspiration. In particular, I would like to thank Nate Foster (for securing the NetLab for us, and for providing us with an endless supply of nuts that fed us through the many all-nighters during paper deadlines), Christina Delimitrou (for accepting to be on my thesis committee even after I had already passed my A-Exam), Robbert van Renesse (for always encouraging me since my very first year at Cornell when I worked under him as a T.A.), and Lorenzo Alvisi and Bobby Kleinberg (for always being so supportive and approachable). As I begin my own journey as a Professor in the next few months, I am so glad to have such great role models to look up to. I would also like to take this opportunity to thank the amazing staff at Cornell, especially Becky Stewart, for helping me out with so many administrative issues over the years.

Next, I would like to thank my peers at Cornell for making the entire graduate study experience so enjoyable. During my first two years of graduate school, senior graduate students Han Wang and Ki Suh Lee took me under their wing and guided me through everything, from research to life as a graduate student in general. I cannot thank them enough for all their support and guidance in those early years. I will always cherish my friendship with Praveen Kumar, Saksham Agarwal, Qizhe Cai, and Hardik Soni, for all the great times we shared, both inside and outside of Cornell. And more generally, I would like to thank all the amazing graduate students, too many to name, that I came to know during my time at Cornell. I can only hope that the friendships and connections I forged at Cornell would last a lifetime.

During the summer of 2017 and 2018, I got the opportunity to intern with Microsoft Azure's Host Networking team in Redmond. It turned out to be a great learning experience for me, and also inspired some of the work presented in this dissertation. I would like to thank Microsoft for giving me this opportunity, especially Daniel Firestone, Andrew Putnam, Jitu Padhye, Yibo Zhu, Larry Luo, Norman Lam, and Gautham Popuri.

I would also like to take this opportunity to acknowledge some of the people from my undergraduate years who got me interested in graduate studies in the first place. Back in 2013, I got an opportunity to spend a few months as an undergraduate researcher at University of Toronto, working with Eyal de

Lara. I will always remember this trip as the first time I ever flew in an airplane, but more than that, I also had an amazing time working with all the graduate students in Eyal's group, especially Nilton Bila, Utkarsh Roychoudhury, Akshay Kumar, and Sahil Suneja. Looking back, those few months in Toronto had a big influence over my decision to go for graduate studies. But perhaps I owe the biggest gratitude to my undergraduate research advisor, Partha Pratim Chakrabarti (or PPC, as known to IIT Kgpians). I did undergraduate research under PPC for almost two years, and I would like to thank him for his amazing mentorship and his constant encouragement that finally convinced me to apply for graduate studies.

Finally, I would like to thank (by far) the most important people in my life, my parents and my two sisters, as well as my entire extended family of uncles, aunts, and cousins, for always being there for me and proudly supporting me through all my endeavors. Every little success that I have had so far, and every success that I might enjoy in the future, I owe it all to them. They are my rock and I am so blessed to call them my family.

Overall, graduate school has truly been an experience of a lifetime. And it is all because of the people mentioned above.

TABLE OF CONTENTS

LIST OF FIGURES

PREVIOUSLY PUBLISHED MATERIAL

Chapter 2 in this dissertation is adapted from a previous publication [99] –

Vishal Shrivastav. *Fast, Scalable, and Programmable Packet Scheduler in Hardware*. In Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM), 2019.

Chapter 3 in this dissertation is adapted from a previous publication [101] –

Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. *Shoal: A Network Architecture for Disaggregated Racks*. In Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI), 2019.

CHAPTER 1

INTRODUCTION

The modern datacenter has emerged as the dominant computing platform that powers most of world's consumer online services, financial, military, and scientific application domains. Further, virtually every application and service running inside the datacenter relies on network connectivity. In this dissertation, we study the impact of slowdown in Moore's law and the end of Dennard scaling on datacenter networking, and propose new designs and architectures for high-performance, high-speed networking in the Post-Moore era.

1.1 Background

We begin by giving a brief overview of the modern datacenter network, followed by a discussion of the ongoing trends in transistor scaling, the underlying technology behind the existing datacenter networking infrastructure.

1.1.1 Modern Datacenter Networks

Modern datacenters comprise a cluster of compute nodes or "end-hosts", that can range from an order of few thousands to a million, interconnected using a datacenter-wide network. The primary job of the datacenter network is to effectively facilitate all remote communications between the applications running at the end-hosts, which puts network on the critical path to performance, availability, and even security of applications inside datacenters.

The communication over a datacenter network typically has two components—(a) a *network stack* at the end-host that moves data from the applications onto the wire, and (b) a *switching fabric* that moves data between the end-hosts. We discuss both of them below.

End-host Network Stack. The network stack at each end-host provides the interface to the applications to communicate over the network. Typically, the entire network stack functionality is divided into smaller independent modules called layers. The Open Systems Interconnection (OSI) model introduces seven abstraction layers [133] — Physical layer (the lowest layer in the stack), Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, and Application Layer (the highest layer in the stack). Each layer serves the layer above it and is served by the layer below it.

Further, end-hosts in modern datacenters are built using commodity servers. Each server is equipped with networking hardware infrastructure (Figure 1.1) that implements the end-host network stack, as described below.

General-purpose Central Processing Unit (CPU). Most of the network stack at the end-hosts (from Application Layer through most of the Data Link Layer) is typically implemented on top of general-purpose CPUs. The key advantage of such a design is the ease of programmability, that provides the datacenter operators with the flexibility to rapidly experiment, test, and deploy new network stack functionalities, or update the existing ones. Figure 1.2 shows the typical CPU specifications of commodity servers around the year 2020.

Network Interface Card (NIC). A Network Interface Card (NIC) is an Application-specific Integrated Circuit (ASIC), that typically implements a part

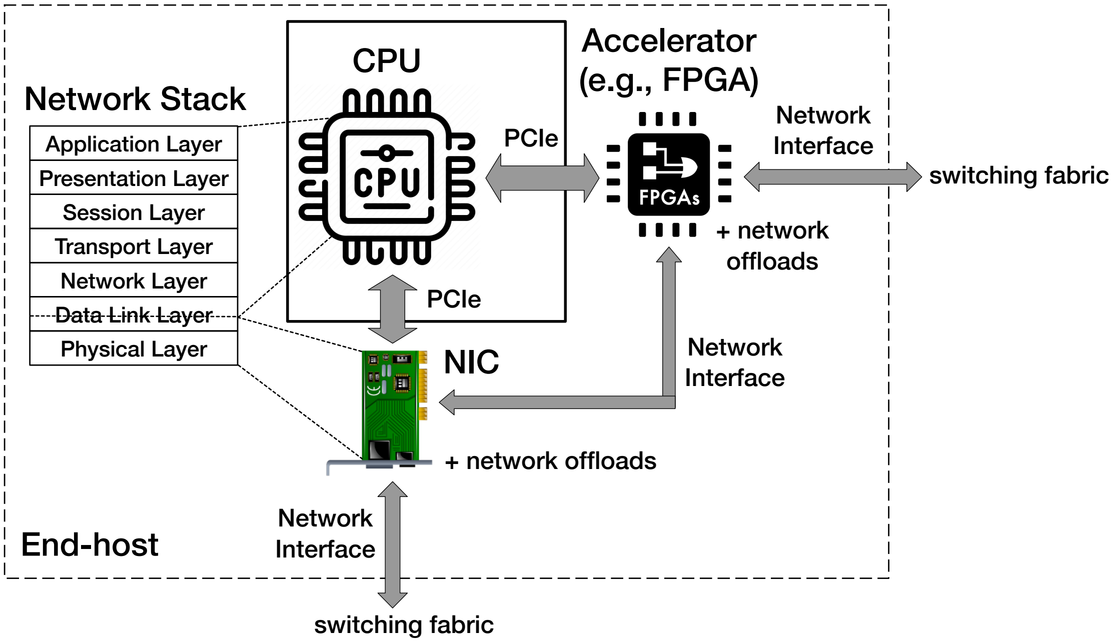


Figure 1.1: End-host networking infrastructure inside modern datacenters, comprising general-purpose CPU, Network Interface Card (NIC), and accelerators such as FPGA. CPU implements majority of the network stack, while NIC implements a part of the Data Link Layer and the Physical Layer. For higher performance, certain network stack functionalities are offloaded to NIC and accelerator. CPU communicates with NIC and accelerator over the Peripheral Component Interconnect Express (PCIe) bus, while NIC and accelerator optionally communicate via their respective network interfaces.

of the Data Link Layer and the entire Physical Layer of the network stack. In addition, modern NICs also implement a few other network stack functionalities offloaded from the CPUs for higher performance, such as Large Send Offload (LSO) [129], Receiver Side Scaling (RSS) [156], Quality-of-Service (QoS), and Remote Direct Memory Access (RDMA) [135]. While offloading functionalities to NICs result in higher performance, a key limitation of this approach is that commodity ASIC NICs at the time of writing this dissertation are mostly fixed-function, thus not allowing users to modify or program new functionalities once the NIC has been manufactured. Figure 1.2 shows the typical specifications of commodity ASIC NICs around the year 2020.

Component	Specifications
General-purpose CPU	Number of cores: 4–56 Clock speed: 1.8–4.5 GHz L2/L3 cache size: 8.25–77 MB Power: 165 W
Network Interface Card	Number of ports: 1–2 Per port speed: 200/100/50/40/25/10 Gbps
FPGA Board	Number of logic elements: 200–3000 K Embedded SRAM size: 52–234 MBits Number of ports: 1–4 Per port speed: 100/50/40/25/10 Gbps
Packet Switch (fixed-function)	Aggregate switching speed: 6.4–25.6 Tbps Number of ports: 32–256 Per port speed: 400/200/100/50/40/25/10 Gbps
Packet Switch (programmable)	Aggregate switching speed: 6.4–12.8 Tbps Number of ports: 32–256 Per port speed: 400/200/100/50/40/25/10 Gbps

Figure 1.2: Typical specifications of networking hardware inside datacenters around the year 2020. CPU specifications are from Intel [117], NIC specifications are from Mellanox [159], FPGA specifications are from Terasic Stratix DE boards [163], fixed-function packet switch specifications are from Broadcom [143], and programmable packet switch specifications are from Barefoot Networks (an Intel company) [149].

Network Accelerator. Modern datacenters also equip end-hosts with network accelerators to offload certain network stack functionalities from the CPU for higher performance. The most prominent network accelerator used in modern datacenters are the Field-programmable Gate Arrays (FPGAs). As an example, in 2018, Microsoft datacenters introduced AccelNet [33], a system that offloaded the virtual switch (vSwitch) from the CPUs onto the FPGAs. The advantage of offloading functionalities onto an FPGA as opposed to a commodity NIC is that FPGAs are programmable, unlike fixed-function commodity NICs. Thus FPGAs give the datacenter operators more flexibility, at the cost of lower performance and power efficiency compared to commodity NICs. Figure 1.2 shows the typical specifications of FPGAs around the year 2020.

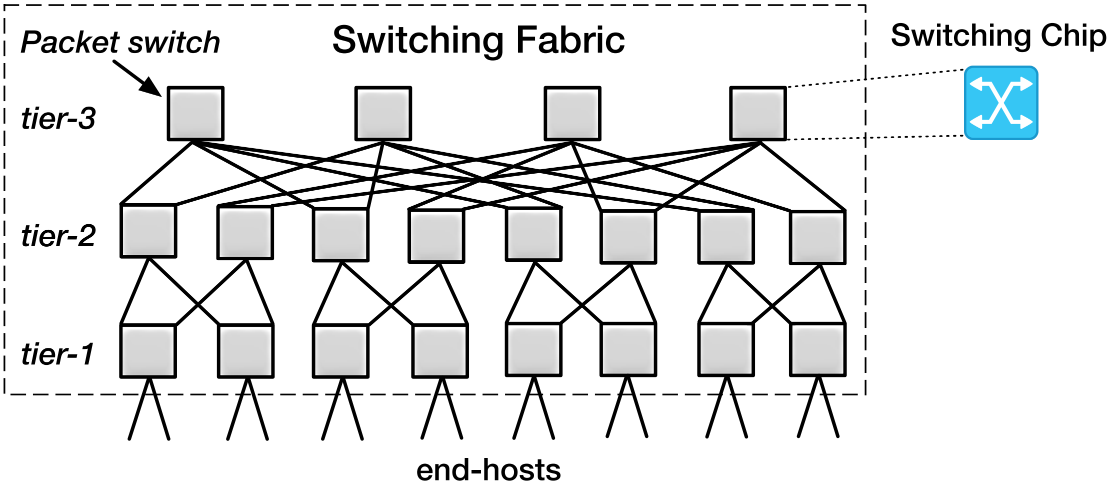


Figure 1.3: A typical packet-switched switching fabric inside modern datacenters. The example comprises 20 packet switches forming a three-tier Fattree topology with full bisection bandwidth connecting 16 end-hosts.

Switching Fabric. The end-hosts inside datacenters are interconnected using a datacenter-wide switching fabric. The switching fabric comprises a collection of *packet switches* connected in some network topology to provide the necessary bisection bandwidth. The most common topology inside modern datacenters is a Folded Clos topology, such as Fattree [1] (Figure 1.3).

Packet Switch. The building blocks of modern datacenter switching fabric are the packet switches. Traditionally, each packet switch comprises a fixed-function switching chip that implements the Ethernet protocol. Figure 1.2 shows the typical specifications of fixed-function packet switches around the year 2020. More recently however, there has been an introduction of *programmable packet switches* inside the datacenters, that comprise a programmable switching chip that allows the users to program custom packet processing protocols, rather than being restricted to the conventional Ethernet protocol. Figure 1.2 shows the typical specifications of programmable packet switches around the year 2020.

1.1.2 Trends in Transistor Scaling

In 1965, Gordon Moore postulated that the number of transistors on a given chip would double every year. He later revised it in 1975 to doubling every two years. This came to be known as the *Moore's law*. Around the same time in 1974, Robert H. Dennard proposed a power scaling law that roughly stated that as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area. This came to be known as *Dennard scaling*. Combined with Moore's law, this meant that the performance per Watt of transistor-based technology would double every 18 months.

For a better part of four decades, Moore's law and Dennard scaling drove the entire transistor-based industry, that powers everything from processors to memory to (solid-state) storage to packet switches—in every technology generation, as the number of transistors on a chip doubled, the circuit became roughly 40% faster, while power consumption (with twice the number of transistors) stayed the same.

However, towards the later half of 2000s, both Moore's law and Dennard scaling started to deviate from their original trends—Dennard scaling ended in 2006, while Moore's law has slowed down significantly since 2010, and is expected to end altogether by as early as 2025 according to some projections [131]. This seismic shift in the transistor scaling trends has had, and continues to have, profound implications for every field in computing. This dissertation focuses specifically on its impact on datacenter networking, and proposes new designs and architectures for networking in the Post-Moore era.

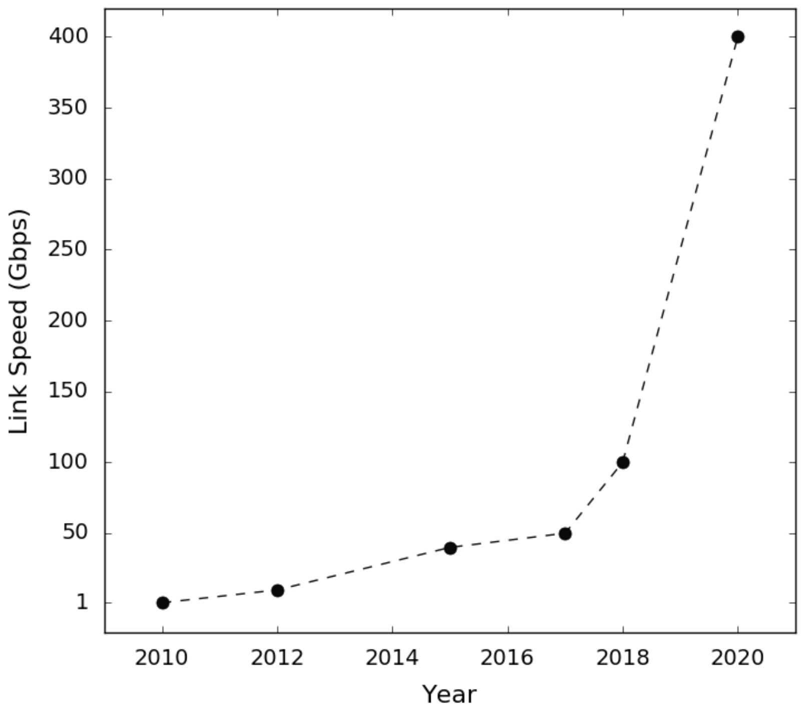


Figure 1.4: Exponential rise in datacenter link speed over the last decade [32].

1.2 Motivation

The number of users on the Internet continues to grow steadily [126], from 1.04 billion in 2005 (16% of the world population in 2005) to 4.15 billion in 2019 (53.6% of the world population in 2019). Popular web applications such as Facebook alone has over 2.6 billion monthly active users as of early 2020 [162]. And almost entirety of all such web applications and services are hosted inside one of the datacenters worldwide. Next, we are also living in a time of massive data explosion, with over 90% of the total data in human history generated in just last few years [158]. According to a market intelligence company [161], the total amount of world's data reached 18 zettabytes in 2018 (a zettabyte is 10^{21} or 2^{70} bytes), and is expected to reach 175 zettabytes by 2025. Further, over 40%

of that data in 2018 was stored and processed inside datacenters worldwide, and by 2025 that number is expected to reach over 65% [161]. Thus, to be able to scale to the large number of users, and process and analyze the large amounts of data, computation inside modern datacenters tends to be massively distributed in nature, i.e., the computation for a given application or service is distributed across multiple end-hosts to overcome the limited processing capacity of a single end-host. A key upshot of such a distributed computation is that all the communication between the end-hosts has to go over the datacenter network connecting the end-hosts. Thus the bandwidth demand from the datacenter network, to effectively facilitate all the remote communications, is rising very rapidly. This demand is further exacerbated by the slowdown in Moore's law and the end of Dennard scaling, that has stagnated the scaling of processing capacity of individual end-hosts, thus forcing the applications to get even more distributed to meet their performance and scalability objectives. In addition, the resources within datacenters (*e.g.,* storage) are also becoming increasingly disaggregated [34], as disaggregation promises to significantly improve the efficiency of datacenters, *e.g.,* via more efficient resource pooling and provisioning, higher compute density, and elastic scaling. This further adds to the rising bandwidth demand. In fact, Google reports that the bandwidth demand inside their datacenters doubles every year [7]. This is also evident from the exponential rise in link speeds inside datacenters, that has increased by 400× during a ten year period of 2010—2020 (virtually doubling every year), as illustrated in Figure 1.4.

Thus overall, there is a pressing need to design a high-speed datacenter networking infrastructure (for both the end-host network stack and the switching fabric) that could effectively scale with the ever-increasing link speeds and bandwidth demand. This is precisely the focus of this dissertation.

1.3 Limitations of Existing Approaches

In this section, we discuss how the existing approaches to designing the end-host network stacks (Section 1.3.1) and switching fabrics (Section 1.3.2) fall short when it comes to scaling to ultra-high link speeds and bandwidth demands.

1.3.1 General-purpose Processor-based Network Stacks

The major portion of the network stack at the end-hosts inside modern datacenters is implemented on top of general-purpose CPUs (Figure 1.1). The key benefit of such a design is the ease of programmability, that allows users the flexibility to program and deploy new algorithms and protocols in software at very small timescales, which is a key enabler for innovations. Further, while the link speeds inside datacenters were relatively slow and CPU processing speeds were scaling exponentially according to Moore's law, a CPU-based network stack was also sufficient to saturate the network access link. However, the slowdown in Moore's law and the end of Dennard scaling has meant that the processing speed of a single CPU core, which once used to double every 18 months, is now doubling every 20 years (Figure 1.5), whereas the link speeds are doubling virtually every year (Figure 1.4). As a result, it is becoming increasingly difficult to saturate the high-speed links using CPU-based network stack [33, 5, 6]. For a concrete example, we consider packet scheduling, a key end-host network stack operation. The job of a packet scheduler is to transmit packets from end-hosts onto the wire in accordance with some scheduling algorithm running at the end-host. Unfortunately, the state-of-the-art CPU-based packet schedulers are barely able to saturate a 10 Gbps link using a single CPU

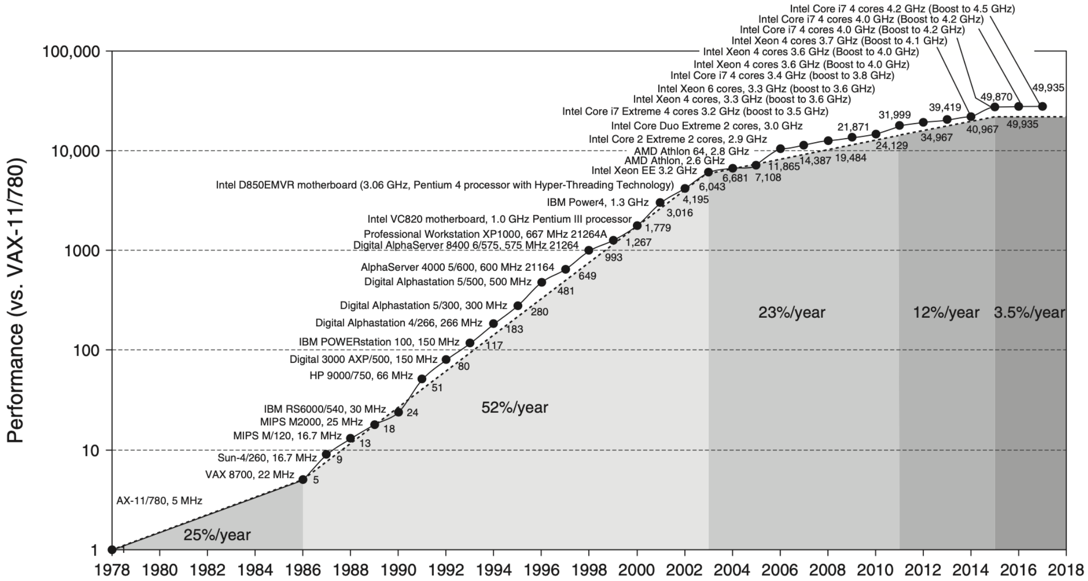


Figure 1.5: Scaling of CPU performance over the years. Since 2018, the performance improvement is just 3.5% per year, or doubling every 20 years. (Image source: Hennessy & Patterson 6e [43])

core [93]. This is clearly not sufficient, given that the link speeds inside datacenters have already reached over 100 Gbps.

Using multiple CPU cores to implement the network stack can potentially improve the performance, but the performance scaling is typically sub-linear with the number of cores due to Amdahl's law [121] and the overheads of stateful operations (e.g., locking overheads and cache coherency). Further, several prior studies [5, 42, 52] have shown that saturating a 100 Gbps or a 400 Gbps link using a CPU-based network stack would require prohibitively large number of cores. Using the example of packet scheduling from above, even with perfect scaling, it would require 10 cores to schedule packets at line rate over a 100 Gbps link, and a whopping 40 cores for a 400 Gbps link. Even the most powerful commodity servers in 2020 only have a few 10s to up to a 100 cores, and to make matters even worse, the multi-core scaling is fundamentally limited by the power consumption, typically forcing only a subset of cores to be poweredon at any given time or forcing the cores to run at slower clock speeds [30]. Thus, allocating a large number of cores for exclusively network processing is not only becoming unsustainable but is also a poor utilization of the general-purpose computation power, which could have otherwise been used to run actual applications. For datacenter operators, this could result in significant loss in revenue [33].

Some recent designs have also explored alternative general-purpose architectures for implementing the network stack, such as Graphics Processing Units (GPUs) [52] and NICs with embedded CPU cores [67]. However, these architectures are just different manifestations of a multi-core architecture described above, and hence ultimately suffer from the same fundamental limitations.

1.3.2 Packet-switched Switching Fabrics

Packet switches are the building blocks of modern datacenter switching fabrics. Each packet switch typically comprises a switching chip responsible for forwarding data at line rate. However, slowdown in transistor and power scaling, due to slowdown in Moore's law and end of Dennard scaling, has also impacted the scaling of switching speeds of packet switches [75]—(i) it has become increasingly difficult to increase the number of I/O pins on a switching chip, thus limiting the number of I/O channels per switching chip, and (ii) it has also become increasingly difficult to increase the rate of serialization and deserialization (SerDes) hardware, thus limiting the data rate per I/O channel. Fortunately, over the last decade (Figure 1.6), by supplementing the slow transistor scaling (40 to 7 nanometer(nm) process shrink over 10 years) with higher-level archi-

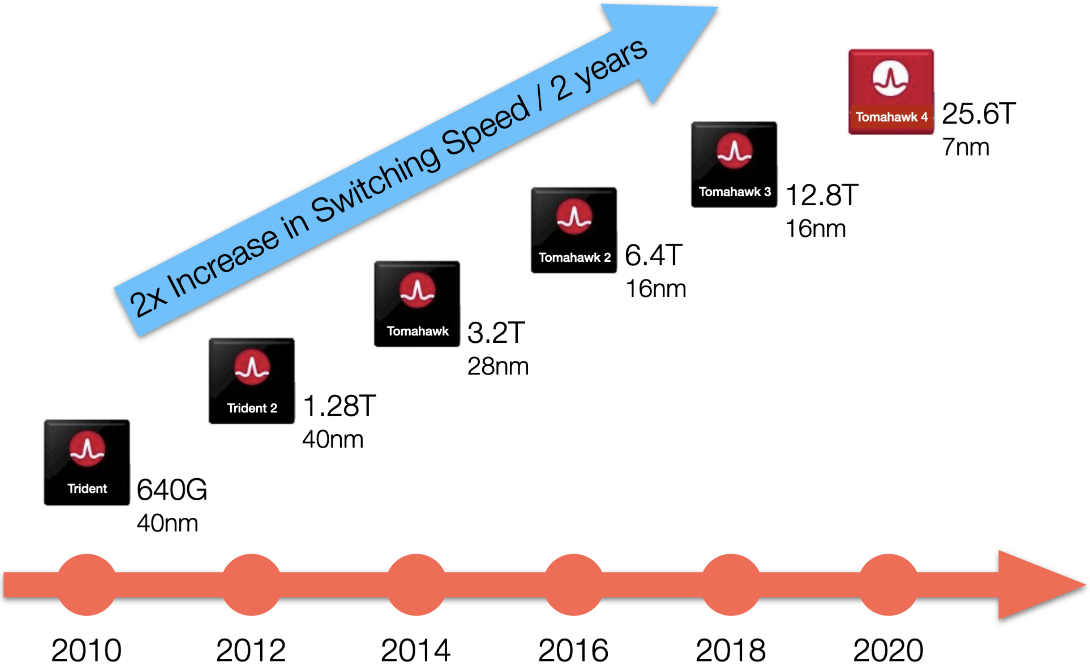
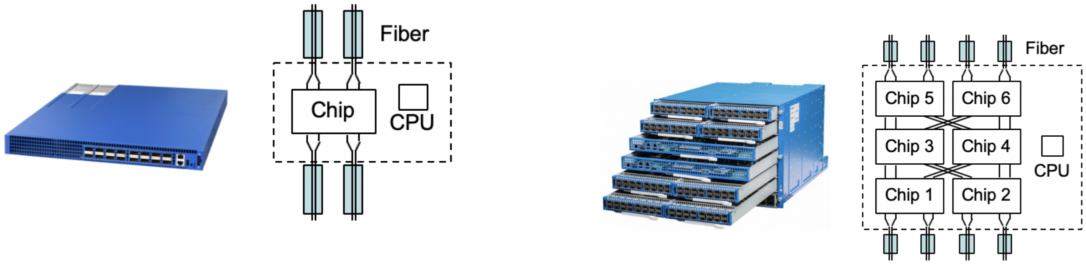


Figure 1.6: Broadcom switching chips double in speed every two years. The most recent generation of Broadcom switching chip (Tomahawk 4) runs at 25.6 Terabits/second and is built using 7 nm process technology.

tectural innovations, leading switch chip manufacturers like Broadcom have managed to double the switching speeds of their packet switches every two years [152]. Unfortunately however, even with the current scaling trend, the increase in switching speed is still not able to keep pace with the rising bandwidth demand (Figure 1.8), and even worse, the current scaling trend of packet switches is only going to get slower in the coming years, as transistor scaling is expected to end altogether [131], and the gains from higher-level architectural innovations (such as pipeline parallelism) tend to hit a wall eventually, partly because of power constraints, as observed with multi-core scaling [30].

Next, a typical datacenter switching fabric comprises a collection of packet switches connected in some network topology, the most common being a Folded Clos topology such as Fattree [1] as illustrated in Figure 1.3. Now given that the switching speeds of individual packet switches are unable to keep pace with the increasing bandwidth demand, scaling the switching fabric to higher bandwidth would require more number of switches, more number of tiers in the topology, more number of fibers, and more number of transceivers, which together translates to increased power, cost, and number of hops through the network (latency). As an illustration, consider a 8,192 end-host network. Assuming 25 Gbps links and 128-port × 25 Gbps switches, one could build a full bisection bandwidth Fattree network using two tiers (4 hops), 192 switching chips, 24 K transceivers, and 12 K fibers. Now suppose the end-host link bandwidth increased to 100 Gbps due to an increased bandwidth demand. If the switching speed of individual packet switches was able to scale perfectly with the bandwidth demand, then we would now have 128-port × 100 Gbps switches, and we could build a full bisection bandwidth Fattree network using the same number of resources as before. However, at the other extreme, if the switching speed of individual switches does not scale with the bandwidth demand, each 128-port × 25 Gbps switch would now operate as 32-port × 100 Gbps switch, and we would now require three tiers (6 hops), 1,280 switching chips, 40 K transceivers, and 20 K fibers to build a full bisection bandwidth Fattree network, resulting in significant cost, power, and latency overhead even for this toy example.

To overcome some of the cost and power overheads of scaling a traditional Fattree topology, that comprises one switching chip per switch box, datacenter operators have started to build a chassis-based Fattree topology where multiple switching chips are integrated into the same switch box and connected using energy-efficient copper backplane traces (Figure 1.7). Such an architecture increases the switching capacity within each box resulting in fewer transceivers and fibers compared to a traditional Fattree, but it comes at the cost of more



(a) Traditional Packet Switch

(b) Chassis Packet Switch

Figure 1.7: A traditional packet switch with one switching chip per switch box versus a chassis packet switch with multiple switching chips per switch box.

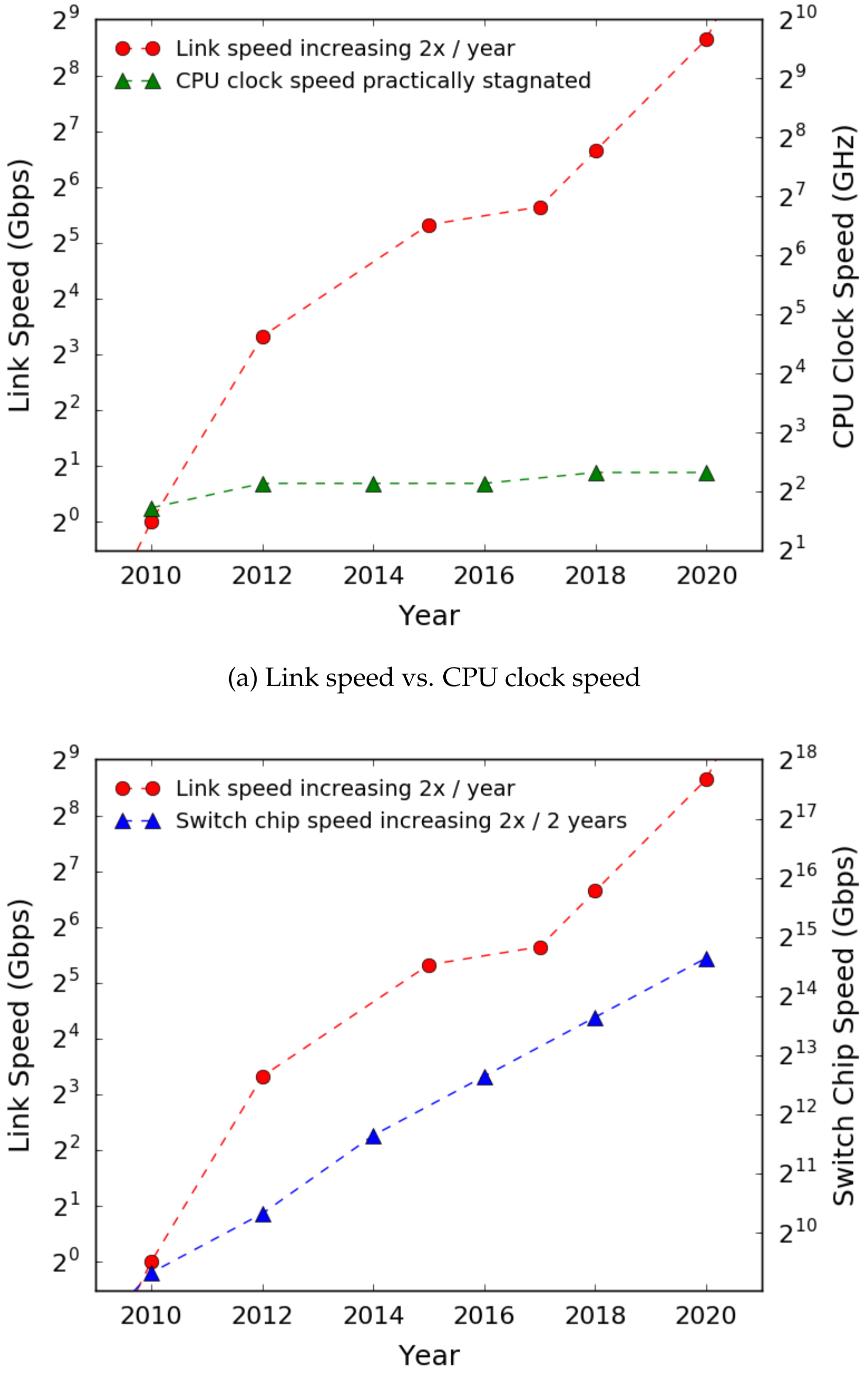
switching chips and more number of hops through the network (latency) [75]. Further, scaling a chassis to higher switching speeds is limited by the power consumption inside a chassis, thus making it difficult to scale to higher speeds.

1.4 Research Question

Figure 1.8 summarizes the growing gap in the scaling of link speed versus the CPU and packet switching speed, as explained in detail in Section 1.2 and Section 1.3. As a result, it is becoming increasingly unsustainable to use general-purpose CPUs and packet switches to build high-speed end-host network stacks and switching fabrics respectively. To that end, this dissertation investigates the following research question —

Research Question # 1 *How to design high-speed end-host network stacks and switching fabrics for datacenters that achieve high performance at low cost and power even in light of the continued slowdown in Moore's law and the end of Dennard scaling?*

We further breakdown this overarching research question into two sub-questions in the following section.



(a) Link speed vs. CPU clock speed

(b) Link speed vs. Switch chip speed

Figure 1.8: Growing gap in the scaling of link speeds versus the CPU and packet switching speeds—link speed is practically doubling every year, whereas packet switch chip speed is doubling every 2 years, while CPU clock speed has practically stagnated (doubling every 20 years).

1.5 Approach, Challenges, and Thesis

Given the limitations of existing solutions (Section 1.3), in this section we describe the approach taken in this dissertation to design high-speed end-host network stacks (Section 1.5.1) and switching fabrics (Section 1.5.2) in Post-Moore era. We conclude this section by stating the thesis in this dissertation (Section 1.5.3).

1.5.1 Domain-specific Processing: Promise and Challenges

Section 1.3.1 described the limitations of using general-purpose processing architectures for building high-speed end-host network stacks. Hence, in this dissertation, we explore the idea of complementing general-purpose processors with domain-specific processors for network processing, as a way to build high-speed end-host network stacks.

A general-purpose processor is characterized by a fixed, general-purpose hardware architecture, that cannot be customized for specific applications. As a result, one has to rely solely on instruction-level optimizations to extract higher performance. An alternative is Field-programmable Gate Array (FPGA), which can be programmed with any custom hardware architecture (general-purpose or otherwise). The ability to customize low-level hardware architecture means that an FPGA-based implementation of an application typically achieves higher performance compared to a general-purpose processor-based implementation, but unfortunately, the performance (and power efficiency) is much lower compared to an Application-specific Integrated Circuit (ASIC)-based implementation [56]. This is because FPGAs rely on a generic reconfigurable substrate to program arbitrary hardware architectures, and unfortunately, this comes with performance overhead—in essence, FPGAs are just another general-purpose platform, albeit with a much lower-level programming abstraction compared to general-purpose processors, and this generality comes at the cost of performance. In fact, in a world where performance is the first-order objective, it has been previously argued [145] that FPGAs are better suited as experimental platforms for prototyping and validating new hardware architectures, prior to porting those architectures to an ASIC (which is a much more time-intensive and irreversible process). As described in Section 1.7, we use FPGAs extensively in this dissertation to prototype and validate new hardware architectures.

On the other hand, a domain-specific processor presents a different set of trade-offs. A domain-specific processor is an Application-specific Integrated Circuit (ASIC), that comprises a custom hardware architecture targeted towards a specific set of applications. The promise of domain-specific processors is that they can potentially achieve much higher performance compared to general-purpose processors and FPGAs, for the target set of applications for which they were designed. The trade-off, of course, is that unlike general-purpose processors and FPGAs, domain-specific processors cannot support arbitrary applications. Thus, domain-specific processors present a fundamental trade-off between the range of applications a processor can support (programmability) and the rate at which it can process those applications (performance).

Unfortunately, even in light of recent advances in programmable network data plane [16, 15, 103], most state-of-the-art domain-specific processors for end-host network processing still tend to be fixed-function, that give up on programmability altogether in favor of performance[1]. Examples include fixed-function ASIC NICs that support a specific congestion control algorithm [187] or a specific packet scheduling algorithm [88] or a handful of other network stack functionalities [129, 156, 135], none of which can be modified or augmented with new functionalities once the NIC has been manufactured. The reason for such a design choice is rooted in the conventional wisdom that it is extremely difficult to achieve both programmability and high-performance. And given that performance is the first-order objective, domain-specific processors make a fair trade-off by compromising on programmability. However, in the fast evolving field of datacenter networking, one needs the flexibility to experiment and adopt new algorithms and protocols without having to re-design the entire processor from scratch, which could take of the order of years. Hence, one cannot completely give up on programmability.

Thus overall, the challenge with realizing the promise of domain-specific processors for network processing lies in designing the hardware architecture for the processor that achieves the right balance between programmability and performance. This dissertation focuses specifically on packet scheduling as the target network processing application. Unfortunately, all state-of-the-art processors for packet scheduling compromise either on programmability or on performance (either speed or scalability) [99].

To that end, this dissertation investigates the following research question —

Research Question # 1.1 *How to design a domain-specific processor for packet scheduling that is simultaneously programmable and high-performance (in terms of both speed and scalability)?*

[1] There are some works done concurrently with this dissertation, most notably Tonic [6], that share our vision of programmable domain-specific processors for end-host network processing.

1.5.2 Circuit Switching: Promise and Challenges

Section 1.3.2 described the limitations of using packet switches for building high-speed switching fabrics. Hence, in this dissertation, we explore the idea of using circuit switches to build high-speed switching fabrics in a cost and power efficient manner.

Unlike packet switches, circuit switches do not make forwarding decisions for each packet (via the packet's header contents). As a result, circuit switches are fundamentally much simpler than packet switches—unlike packet switches, circuit switches have no packet processing pipeline, no packet buffers, and no serialization-deserialization circuit. A circuit switch simply has a *crossbar* used to set-up connections (or circuits) between the switch's input and output ports for forwarding data.

One can use different technologies to build the crossbar for circuit switches, the two most common being the *electrical* and the *optical* technology. An electrical circuit switch is built using the same transistor technology as a packet switch, and hence is fundamentally limited by the transistor scaling trends. But given the simplicity of circuit switches over packet switches, an electrical circuit switch can be far more cost and power efficient than an equivalent packet switch [101]. Thus, one can potentially scale an electrical circuit-switched fabric to higher speeds at significantly lower power and cost than a packet-switched fabric. On the other hand, an optical circuit switch is fundamentally different from an electrical packet or circuit switch. An optical circuit switch is a layer 0 switch—it operates directly on light beams without needing to translate the optical signals into electrical signals and vice-versa. Thus an optical circuit switch is even more cost and power efficient than an electrical circuit switch, e.g., it does not need transceivers at each switch to convert between optical and electrical signals. But more importantly, since optical circuit switches operate directly on light beams, these switches are data rate agnostic, meaning as the link speeds in datacenters scale to higher speeds (e.g., 100 Gbps to 400 Gbps and beyond), an optical switching fabric need not be upgraded. Hence, in summary, a packet-switched fabric can scale to higher speeds with significant increase in cost and power, an electrical circuit-switched fabric can scale to higher speeds with much smaller increase in cost and power compared to a packet-switched fabric, whereas an optical circuit-switched fabric can scale to higher speeds with *zero* increase in cost and power (on account of being data rate agnostic).

Interestingly, the scaling advantages of circuit switches over packet switches have been known for decades. And yet, modern datacenters use packet switches over circuit switches to build their switching fabrics. The reason being that the scaling advantages of circuit switches come at a price—unlike packet switches, circuit switches first need to be configured (i.e., the circuits between input and output ports need to be set-up) prior to sending any data. One can further break down this operation into a *control plane* operation (i.e., deciding on when and what circuits to set-up, or *circuit scheduling*) and a *data plane* operation (i.e., creating/changing the physical circuits as asked by the control plane, or *circuit reconfiguration*). Unfortunately, circuit switches have traditionally been bottlenecked by the data plane reconfiguration delay, that could be of the order of milliseconds [31] to 10s of microseconds [74]. As a result, these switches are unsuitable for carrying low latency traffic or traffic patterns that change unpredictably and at a very fine granularity, all of which are the case with the datacenter traffic [36]. As a result, prior designs [31, 74] have to rely on a *hybrid network*, comprising a circuit-switched fabric carrying the stable bulk traffic and a separate packet-switched fabric carrying the unpredictable low latency traffic. Unfortunately, such a design has several known challenges, including complexity of managing two separate networks, the challenge of precisely dividing the host bandwidth between the two networks (which would depend upon what fraction of the traffic is low latency and what fraction is bulk, something that is a function of the current workload and keeps changing over time), and the challenge of recognizing in the first place what traffic constitutes low latency and what traffic constitutes bulk traffic. Further, by relying on a packet-switched network, the hybrid network is fundamentally limited by the scaling limitations of packet switches (Section 1.3.2). Ideally, we would want an entirely circuit-switched fabric that could efficiently support any arbitrary traffic.

Fortunately, recent advances in the circuit switching technology means that we now have electrical circuit switches commercially available [167], and optical circuit switches available as research prototypes [97, 27, 106, 55, 22, 29, 183, 63], that could reconfigure within nanoseconds. Hence for these switches, the data plane is no longer the bottleneck to supporting unpredictable low latency traffic, and in fact, the bottleneck has shifted to the control plane.

Traditional control planes for circuit-switched networks typically comprise a centralized controller that takes in the current traffic pattern as input and runs some optimization algorithm to figure out the right set of circuits to schedule [31]. While such a design was not a bottleneck for slow reconfigurable circuit switches, unfortunately it is too slow for fast reconfigurable circuit switches, completely neutralizing the advantage of having a fast data plane.

Thus overall, one can potentially use fast circuit switches to build high-speed switching fabrics that could support arbitrary traffic with high-performance at low power and low cost. However, to realize this promise, one needs a fast and high-performance control plane for circuit-switched networks.

To that end, this dissertation investigates the following research question —

Research Question # 1.2 *How to design a control plane for circuit switching that can schedule high-performing circuit configurations at nanosecond granularity?*

1.5.3 Thesis for High-speed Networking in the Post-Moore Era

The central thesis in this dissertation is that by effectively leveraging the promise of domain-specific processing and fast circuit switching, one can indeed build efficient high-speed network stacks and switching fabrics even in light of continued slowdown in Moore's law and the end of Dennard scaling.

The challenge with domain-specific processing is in designing an architecture that achieves the right balance between programmability and performance (Section 1.5.1). This dissertation focuses specifically on a domain-specific processor for packet scheduling. Packet scheduling algorithms assign packets a *time* and a *rank* to decide *when* packets become eligible for scheduling and *in what order* to schedule eligible packets. The thesis in this dissertation is that a processor that simply schedules the "smallest ranked eligible packet" at any given time, is sufficient to program a wide-range of packet scheduling algorithms. Further, by leveraging hardware-level parallelism and indirection techniques, one can also design a fast and scalable architecture for such a processor.

The challenge with fast circuit switching is in designing a control plane that is both fast and high-performance, since the traditional dynamic, traffic-aware centralized solutions are too slow (Section 1.5.2). The thesis in this dissertation is that it is not necessary to have a dynamic, traffic-aware centralized control plane to achieve high performance, and in fact, by carefully choosing a static, pre-defined schedule for circuit scheduling, it is possible to build a fully decentralized, traffic-agnostic control plane that can schedule high-performing circuit configurations at nanosecond granularity.

1.6 Contributions

In this section, we summarize the key contributions of this dissertation that validate the thesis proposed in Section 1.5.3.

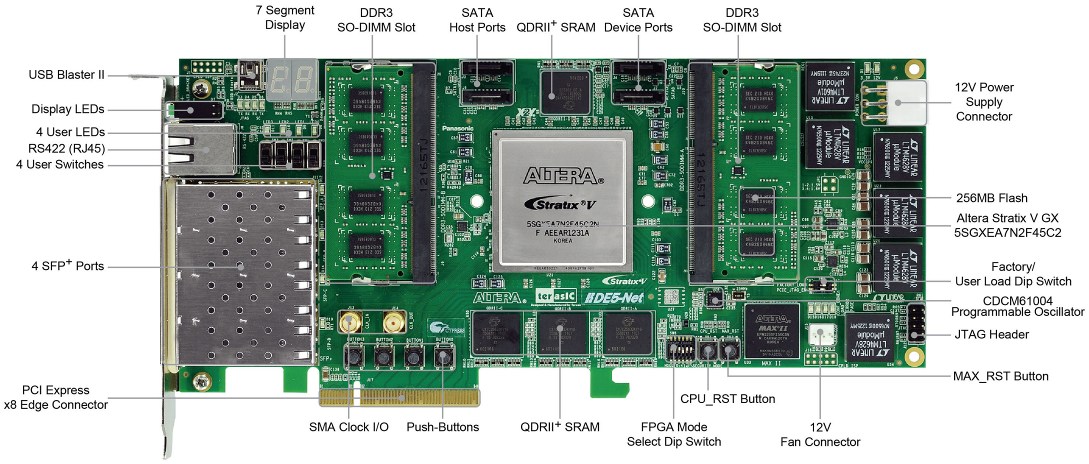
1.6.1 Fast Domain-specific Processor for Packet Scheduling

Packet scheduling is one of the most fundamental network stack operations. Packet scheduling algorithms assign packets a *time* and a *rank* to decide *when* packets become eligible for scheduling and *in what order* to schedule eligible packets onto the wire. Unfortunately, state-of-the-art packet schedulers either compromise on programmability or on performance (either speed or scalability). In PIEO (Chapter 2), we designed and implemented a domain-specific processor for packet scheduling that is *simultaneously* programmable, scalable, and high-speed. For programmability, we introduced a new primitive called Push-In-Extract-Out (PIEO) that can express a wide-range of packet scheduling algorithms by always scheduling the ”smallest ranked eligible packet”. In fact, we show that PIEO primitive is more expressive than any state-of-the-art packet scheduling primitive. Unfortunately, while being expressive, designing an efficient hardware implementation of PIEO is challenging, as it requires maintaining a fully ordered list. Implementing an ordered list in hardware presents a fundamental trade-off between time complexity and hardware resource consumption, which translates into a fundamental trade-off between performance and scalability. To keep up with increasing link speeds, scheduling decisions ideally need to be made in $O(1)$ time, but the best know implementation of an $O(1)$-time ordered list requires $O(N)$ flip-flops and comparators (where N is the list size), which greatly limits the scalability of the design. In PIEO, we improve the $O(N)$ bound on resource consumption for an $O(1)$-time ordered list by an order of complexity to $O(\sqrt{N})$. The result being that PIEO can make scheduling decisions in $O(1)$ time (up to 300× faster than a single CPU core), while also scaling to 10s of thousands of flows (>30× more scalable than state-of-the-art). Looking further, PIEO's design is more general than merely a packet scheduler. In essence, PIEO presents the design of a scalable $O(1)$-time ordered list in hardware. This can be leveraged for high-performance implementations of several key datastructures in hardware, including priority queues [134], key-value stores [128], and balanced search trees [136]. Many applications use one or more of these datastructures, and hence as more applications are offloaded to hardware for higher performance in the Post-Moore era, PIEO can serve as a basic building block for those applications, packet scheduling being one of them.

1.6.2 Fast Circuit-switched Switching Fabric

In Shoal (Chapter 3), we present the design and implementation of a circuit-switched network comprising entirely of fast (nanosecond) reconfigurable circuit switches. The key contribution here is the design and implementation of a control plane for the circuit-switched network that could schedule high-performing circuit configurations at nanosecond granularity. At the core of the design is a new primitive for circuit scheduling that comprises a *static, pre-defined schedule* to determine what circuits to set-up at any given time. By precluding any dynamic traffic-aware decision-making for circuit scheduling, the design ensures that circuits can be scheduled at nanosecond granularity to match the data plane reconfiguration speed. Further, we show that by appropriately configuring the static schedule, one can also extract very high network performance. More specifically, different configurations of the static schedule creates a different virtual network topology, and in Shoal, we configure the schedule to create a *virtual full mesh* topology. Next, we leverage the well-known Valiant Load Balancing (VLB) [171] to route data on top of the virtual full mesh topology that guarantees bounded worst-case network throughput. Unfortunately, such a routing scheme, while bounding the worst-case throughput, could result in congestion at the intermediate hop for certain traffic patterns. To that end, we propose a novel congestion control algorithm, that leverages the static schedule to provide bounded worst-case queuing and fair share of bandwidth for arbitrary traffic. Further, the entire design of Shoal's control plane is agnostic to the underlying circuit-switching technology, and could work with electrical, optical, or wireless technologies alike.

Overall, we demonstrate that Shoal can effectively scale to high switching speeds while achieving comparable or better performance (in terms of both throughput and latency) than several recent packet-switched network designs at significantly lower power and cost.



*Cypress QDRII+ SRAM or functional compatible SRAMS provided by GSI (SigmaQuad-II+) and ISSI (QUADP).

Figure 1.9: A DE5-Net board with Stratix V FPGA.

1.7 Evaluation Methodology

The systems presented in this dissertation propose fundamental changes to the networking hardware, that includes both the switches and the end-host Network Interface Cards (NICs). Hence, to evaluate these systems, we design and implement custom switches and NICs on top of Field-programmable Gate Arrays (FPGAs). In addition, this dissertation also proposes novel network protocols and algorithms to run on top of our custom hardware. To evaluate them, we use a small-scale network prototype built using our FPGA-based switches and NICs, and complement it with a packet-level network simulator for large-scale evaluations. Below we provide an overview of the hardware, prototype, simulator, and workloads used to evaluate the systems in this dissertation. The detailed evaluations can be found in Chapter 2 and Chapter 3.

Hardware. We use DE5-Net boards (Figure 1.9) from Terasic [113] to implement our custom hardware. A DE5-Net board is an FPGA development board with an Altera Stratix V FPGA [154] comprising 234 K Adaptive Logic Modules

(ALMs), 52 MBits SRAM, and four 10 Gbps ports for a total of 40 Gbps interface bandwidth. The code is written in System Verilog [139] and Bluespec System Verilog [118]. We synthesize our design using Quartus Prime software [155].

The DE5 boards are plugged into Peripheral Component Interconnect Express (PCIe) Gen 3.0 slots in a PCIe expansion system (Figure 1.10). The PCIe expansion system is a 4U × 24″ deep × 19″ box from One Stop Systems (OSS) [160], comprising 16 PCIe Gen 3.0 ×8 electrical, ×16 mechanical expansion slots using two OSS-452 backplanes, two 1200 W power supplies, two OSS-HIB38-×16 host and two target interface boards, and two 1 m PCIe cables.

Finally, the PCIe expansion system is connected to a Dell T720 server using the two PCIe cables. The server comprises a dual socket, 2.93 GHz six core Xeon X5670 processor with 12 MB of shared Level 3 (L3) cache, 12 GB of DRAM (6 GB connected to each of the two CPU sockets), 4 PCIe ×8 slots, 1 PCIe Gen 2.0 ×4 slot, and 1 PCIe Gen 2.0 ×16 slot.

The network interface of DE5 boards are connected using Myricom 10 G Small Form-factor Pluggable (SFP+) transceiver modules with Ethernet 10GBase-SR type [150] and 2 m full duplex singlemode fiber cables [151].

Small-scale network prototype. For evaluation, we build a small-scale network comprising eight end-host NICs and six switches using our FPGA-based switches and NICs (Figure 1.10). Two FPGAs are used to implement eight NICs, one per port. Six FPGAs implement six 4-port switches. The switches are connected in a two-tier Fattree topology with full bisection bandwidth.

Large-scale network simulations. For large-scale network evaluations, we complement our small-scale prototype with a packet-level network simulator

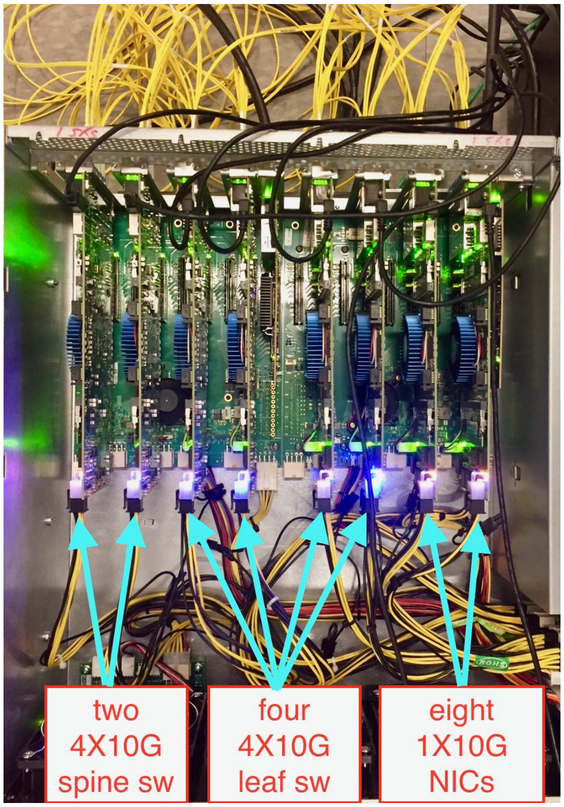


Figure 1.10: Small-scale network for evaluation comprising 8 end-host NICs and 6 switches connected in a two-tier Fattree topology with full bisection bandwidth (also known as leaf-spine topology) connected inside the PCIe expansion system.

written in C. Before running any large-scale simulations, we first cross-validate our simulator with our prototype by running our prototype experiments on the simulator (configured with the same parameters as the prototype) and ensuring that the results match. This gives us confidence in our simulation results. All the system-specific simulation parameters are guided by our prototype.

Workloads. We use a variety of workloads to evaluate our systems. These include a set of custom workloads, used primarily to microbenchmark our systems, followed by more realistic workloads modeled after published datacenter traces [2, 36]. For the sake of robustness, we also evaluate our systems against workloads derived from the emerging disaggregated workload traces [34].

1.8 Source Code Availability

Source code for the systems presented in this dissertation is available online at:
PIEO (Chapter 2): `https://github.com/vishal1303/PIEO-Scheduler`
Shoal (Chapter 3): `https://github.com/vishal1303/Shoal`

- For PIEO, the source code consists of the hardware implementation of PIEO primitive written in System Verilog [139]. The code was synthesized on an Altera Stratix V FPGA [154].
- For Shoal, the source code consists of the hardware implementation of Shoal switch and NIC written in Bluespec System Verilog [118]. The code was synthesized on an Altera Stratix V FPGA [154]. The source code also contains a custom packet-level network simulator written in C.

1.9 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we describe PIEO, a domain-specific processor for high-speed packet scheduling. In Chapter 3, we describe Shoal, a fast circuit-switched switching fabric for datacenters. We survey the related work in Chapter 4, which is followed by future work in Chapter 5. We finally conclude in Chapter 6.

CHAPTER 2

FAST DOMAIN-SPECIFIC PROCESSOR FOR PACKET SCHEDULING : PIEO

In this chapter, we describe the design and implementation of a domain-specific processor of packet scheduling that is programmable, scalable, and high-speed.

2.1 Overview

A packet scheduler enforces a scheduling algorithm or a scheduling policy which specifies *when* and *in what order* to transmit packets on the wire. Implementing a packet scheduler in software gives one the flexibility to quickly experiment with and adopt new scheduling algorithms and policies. However, this flexibility comes at the cost of burning CPU cycles which could have otherwise been used for running applications. In public cloud networks, this translates to loss in revenue [33], as the CPU overhead of packet scheduling [88] takes away from the processing power available to customer VMs. Unfortunately, this issue is only getting worse [5] with the growing mismatch between increase in link speeds and slowdown in the scaling of CPU speeds.

Next, with increasing link speeds, the time budget to make scheduling decisions is also getting smaller, e.g., at 100 Gbps link speeds, to enforce a scheduling policy at Maximum Transmission Unit (MTU) granularity, a scheduling decision needs to be made every 120 ns. To put this in perspective, a single DRAM access takes about a 100 ns. Further, new transport protocols, such as Fastpass [83], QJump [37], and Ethernet TDMA [173], as well as recently proposed circuit-switched designs [101, 66, 174, 74], and protocols that rely on very accurate packet pacing [3, 48], require packets to be transmitted at precise times on the wire, in some cases at nanosecond precision [101]. Meeting these requirements in software is challenging [88, 71, 77, 3, 101], mainly due to non-deterministic software processing jitter, and lack of high resolution software timers.

A domain-specific processor for packet scheduling, implemented in hardware such as a Network Interface Card (NIC), can potentially overcome the limitations of software packet schedulers [88]. However, to retain the flexibility of software packet schedulers, packet scheduler in the hardware must be programmable. Further, multi-tenant cloud networks rely heavily on virtualization, with hundreds of VMs or light-weight containers running on the same physical machine. This can result in tens of thousands of traffic classes or flows per end-host [92, 88]. Hence, the packet scheduler must also be scalable.

State-of-the-art packet schedulers in hardware are based on one of the two scheduling primitives—(i) First-In-First-Out (FIFO), which simply schedules elements in the order of their arrival, and (ii) Push-In-First-Out (PIFO) [104], which provides a priority queue abstraction, by maintaining an ordered list of elements (based on a programmable *rank* function) and always scheduling from the head of the ordered list (”smallest ranked” element). Unfortunately, packet schedulers based on these primitives either compromise on scalability (PIFO-based schedulers), or the ability to express a wide-range of packet scheduling algorithms (FIFO-based schedulers). Further, even a general scheduling primitive like PIFO is not expressive enough to express certain key classes of packet scheduling algorithms (Section 2.2.3). Hence in this dissertation, we propose a new programmable packet scheduler in hardware, which is fast, scalable, and more expressive than state-of-the-art.

To design a programmable packet scheduler, we use the insight that most packet scheduling algorithms have to make two key decisions in the process of scheduling: (i) *when* an element (packet/flow) becomes eligible for scheduling (decided by some programmable *predicate* as a function of time), and (ii) in *what order* to schedule amongst the set of eligible elements (decided by some programmable *rank* function). To that end, at any given time, packet scheduling algorithms first filter the set of elements eligible for scheduling at the time (using the predicate function), and then schedule the smallest ranked element from that set (Section 2.2.2, 2.2.3, 2.4). Hence, most packet scheduling algorithms can be abstracted as the following scheduling policy—At any given time, schedule the ″smallest ranked eligible″ element.

Next, to realize the policy of scheduling the ″smallest ranked eligible″ element, one needs a primitive that provides abstractions for: (i) predicate-based filtering, and (ii) selecting the smallest element within an arbitrary subset of elements. Unfortunately, state-of-the-art priority queue-based primitives such as PIFO do not provide these abstractions. Hence in this dissertation, we propose a new scheduling primitive called *Push-In-Extract-Out (PIEO)* (Section 2.3.1), which can be seen as a generalization of the PIFO primitive. PIEO associates with each element a *rank* and an eligibility *predicate*, both of which can be programmed based on the choice of the scheduling algorithm. Next, PIEO maintains an *ordered list* of elements in the increasing order of rank, by always enqueuing elements at the appropriate position in the list based on the element's rank (″Push-In″ primitive). And finally, for scheduling, PIEO first filters out the subset of elements from the ordered list whose eligibility predicates are true at the time, and then dequeues the element at the smallest index in that subset (”Extract-Out” primitive). Hence, PIEO always schedules the ”smallest ranked eligible” element. Further, we use the insight that for most packet scheduling algorithms, the time an element becomes eligible for scheduling ($t_{eligible}$) can be calculated at enqueue into the ordered list, and the eligibility predicate evaluation for filtering at dequeue usually reduces to ($t_{current} \geq t_{eligible}$). Here t could be any monotonic increasing function of time. This insight enables a very efficient hardware implementation of the PIEO scheduler (Section 2.5). Finally, we present a programming framework for the PIEO scheduler (Section 2.3.2), using which we show that one can express a wide-range of packet scheduling algorithms (Section 2.4).

PIEO primitive maintains an ordered list of elements (”Push-In”), atop which filtering and smallest rank selection happens at dequeue (”Extract-Out”). However, implementing both a fast and scalable ordered list in the hardware is challenging, as it presents a fundamental trade-off between time complexity and hardware resource consumption. E.g., using $O(1)$ comparators and flip-flops, one would need $O(N)$ time to enqueue an element in an ordered list of size N, assuming the list is stored as an array[1] in memory. On the flip side, to enqueue in $O(1)$ time, designs such as PIFO [104] exploit the massive parallelism in hardware by storing the entire list in flip-flops and associating a comparator with each element in the list following the classic *parallel compare-and-shift* architecture [78], thus requiring $O(N)$ flip-flops and comparators, which limits the scalability of such a design [104]. In this dissertation, we present a hardware design (Section 2.5) of an ordered list for the PIEO scheduler which is both fast and scalable. In particular, our design executes both ”Push-In” and ”Extract-

[1] Linked list implementation also has time complexity of $O(N)$. Probabilistic datastructures such as skip lists have avg time complexity of $O(log(N))$, and worst-case complexity of $O(N)$.

Out" primitive operations in $O(1)$ time (four clock cycles), while requiring only $O(\sqrt{N})$ flip-flops and comparators, while the ordered list sits entirely in SRAM.

To demonstrate the feasibility of our hardware design of the PIEO scheduler, we prototype it on an FPGA (Section 2.6.1). Our prototype executes each primitive operation in few tens of nanoseconds (up to 20× faster than a single CPU core) (Section 2.6.2), while also scales to tens of thousands of flows (over 30× more scalable than PIFO) (Section 2.6.3). Further, an ASIC-based implementation of our design is expected to execute each primitive operation in just a few nanoseconds (Section 2.6.2), making it up to 300× faster than a single CPU core. To further evaluate the performance of our prototype, we program it with a two-level hierarchical scheduling algorithm implementing rate limit and fair queue policies. We show that our prototype could very accurately enforce these policies atop FPGAs with 40 Gbps interface bandwidth (Section 2.6.4).

Finally, we discuss the applicability of PIEO beyond packet scheduling. We show that PIEO's design can be readily used to implement key datastructures, such as priority queues and key-value stores, with an O(1) run time complexity while also being scalable (Section 2.7). Overall, PIEO can serve as a key basic building block for hardware-accelerated applications in the Post-Moore era.

2.2 Background

In this section, we provide the relevant background on packet scheduling. We start by describing the packet scheduling model assumed in this dissertation, followed by a survey and analysis of the known scheduling algorithms, and conclude by discussing the state-of-the-art packet scheduling primitives.

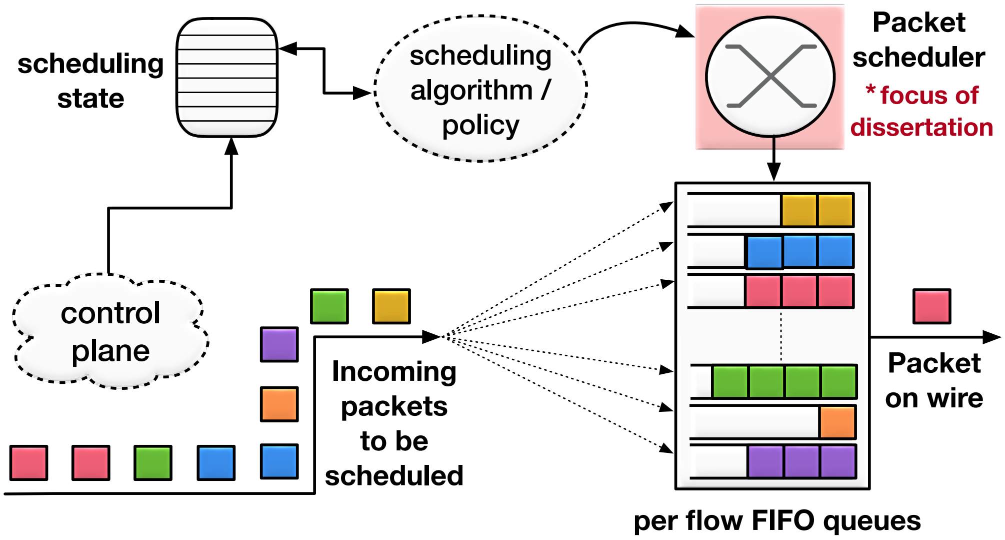


Figure 2.1: A generic packet scheduling model.

2.2.1 Packet Scheduling Model

Figure 2.1 shows the packet scheduling model assumed in this dissertation. Packets ready to be scheduled for transmission are stored in one of the *flow queues* or *traffic classes*[2]. Packets within each flow queue are scheduled in FIFO order, whereas packets across queues are scheduled according to some custom *packet scheduling algorithm or policy*, which specifies when and in what order packets from each queue should be transmitted on the wire. To facilitate scheduling, a custom *scheduling state* is maintained in memory. Typically, this state could also be accessed and configured by the control plane. And finally, a *packet scheduler* is used to express and enforce the chosen scheduling algorithm/policy. The focus of this dissertation is to design an efficient packet scheduler in hardware, which could be programmed to express a wide-range of packet scheduling algorithms/policies, in a fast and scalable manner.

[2] We use *flow queues* and *traffic classes* interchangeably.

2.2.2 Packet Scheduling Algorithms

Most packet scheduling algorithms can be abstracted as the following scheduling policy: *Assign each element (packet/flow) an eligibility predicate and a rank. Whenever the link is idle, among all elements whose predicates are true, schedule the one with the smallest rank.*

The predicate determines *when* an element becomes eligible for scheduling, while rank decides in *what order* to schedule amongst the eligible elements. By appropriately choosing the predicate and rank functions, one can express a wide-range of packet scheduling algorithms (Section 2.4). Further, packet scheduling algorithms can be broadly classified into two key classes:

Work-conserving algorithms. This class of packet scheduling algorithms ensure that the network link is never idle as long as there are outstanding packets to send. Hence, in these algorithms, the eligibility predicate of at least one active element is always true, and whenever the link is idle, the next eligible element in the order of rank is scheduled. Examples of work-conserving packet scheduling algorithms include Deficit Round Robin (DDR) [98], Weighted Fair Queuing (WFQ) [26], Worst-case Fair Weighted Fair Queuing (WF^{2}Q) [11], and Stochastic Fairness Queuing (SFQ) [72].

Non-work conserving algorithms. Under this class of packet scheduling algorithms, the network link can be idle even when there are outstanding packets to send, i.e., the eligibility predicate associated with each active element could all be false at the same time. Non-work conserving packet scheduling algorithms generally specify the *time* to schedule an element, and the eligibility predicate for an element p generally takes the form ($t_{current} \geq p.t_{eligible}$). Non-work conserving algorithms are naturally suited to express *traffic shaping* primitives such as rate limiting and packet pacing [88, 92]. A classic example of a non-work conserving packet scheduling algorithm is Token Bucket [140].

2.2.3 Packet Scheduling Primitives

First-In-First-Out (FIFO). FIFO is the most basic scheduling primitive, which simply schedules elements in the order of their arrival. As a result, FIFO primitive is not able to express a wide-range of packet scheduling algorithms. And yet, FIFO-based schedulers are the most common packet schedulers in hardware, as their simplicity enables both fast and scalable scheduling.

Push-In-First-Out (PIFO) [104]. PIFO primitive assigns each element a programmable *rank* value, and at any given time, schedules the ″smallest ranked″ element. To realize this abstraction, PIFO maintains an ordered list of elements in the increasing order of rank, and supports two primitive operations atop the ordered list—(i) *enqueue(f)*, which inserts an element f into the ordered list, at the position dictated by f's rank, and (ii) *dequeue()*, which extracts the element at the head of the ordered list.

Limitations of PIFO. PIFO fundamentally provides the abstraction of a priority queue, which at any given time, schedules the ″smallest ranked″ element in the entire list. [104] shows that this simple abstraction can express a wide-range of scheduling algorithms which either specify when or specify in what order to schedule each element. However, PIFO's abstraction is not sufficient to express a more general class of scheduling algorithms/policies which specify *both* when and in what order to schedule each element — This class of algo-

```
f.start_time = max(f.finish_time, virtual_time) # if enqueue in empty flow queue
             = f.finish_time                    # if dequeue from flow queue
f.finish_time = f.start_time + L/r
virtual_time(t+x) = max(virtual_time(t)+x, min f in F(f.start_time))
amongst all flows f s.t. (virtual_time(t) >= f.start_time):  # eligibility
     schedule packet from flow with smallest finish_time      # ordering
```

(a)

Transmit length	5	20	10	5	5	10
Packet ID	A	B	C	D	E	F
start time	0	20	2	1	3	5
finish time	10	50	80	100	125	150

(b)

	Ideal Algorithm					
Virtual Time	0	5	15	20	40	45
Eligible Packets	A	C,D,E,F	D,E,F	B,E,F	E,F	F
Scheduling Order	A	C	D	B	E	F

(c)

	Single PIFO ordered by increasing finish time					
Virtual Time	0	20	40	50	55	60
Scheduling Order	A	B	C	D	E	F
	Single PIFO ordered by increasing start time					
Virtual Time	0	5	10	20	25	35
Scheduling Order	A	D	C	E	F	B

(d)

	Two PIFOs: (1) Eligibility PIFO - ordered by increasing start time (2) Rank PIFO - ordered by increasing finish time					
Virtual Time	0	5	10	20	40	45
Eligibility PIFO	D,F,C,E,B	B	B	[ ]	[ ]	[ ]
Rank PIFO	[ ]	C,E,F	E,F	E,F	F	[ ]
Scheduling Order	A	D	C	B	E	F

(e)

Figure 2.2: (a) WF^{2}Q+ algorithm [11]. L is the length of packet at the head of flow queue, r is the rate for flow f, x is the transmission length of current packet being transmitted, and F is the set of back-logged flows. (b) Packets at the head of six different flows in the example system, where packets can be of different sizes (transmission length). We also show start and finish times for each packet. (c) Scheduling order in an ideal run of WF^{2}Q+. (d) and (e) scheduling orders when running WF^{2}Q+ using PIFO.

rithms/policies schedule the smallest ranked element, but only from the set of elements that are eligible for scheduling at the time, which, in principle, could be any arbitrary subset of active elements[3]. Hence, they invariably require a

[3] PIFO could express a special case of such algorithms, where the smallest ranked element in

primitive that supports dynamically filtering a subset of elements at dequeue and then return the smallest ranked element from that subset, something that PIFO primitive does not support. Such complex packet scheduling policies are becoming more common in multi-tenant cloud networks [105], and a classic example of one such algorithm is the Worst-case Fair Weighted Fair Queuing (WF^{2}Q) [12]. WF^{2}Q is the most accurate packet fair queuing algorithm known, making it an ideal choice for implementing fair queue scheduling policies. Further, non-work conserving version of WF^{2}Q can very accurately implement shaping primitives such as rate limiting [88].

WF^{2}Q+[4] (Figure 2.2(a)) tries to schedule a packet whenever the link is idle, which could be at any arbitrary discrete time *t*. However, challenge with WF^{2}Q+ is that the eligibility predicate of any arbitrary subset of elements can become true at *t*, as shown in Figure 2.2(c), and hence scheduling the smallest ranked eligible element at time *t* becomes challenging. In Figure 2.2(d) and (e), we try to express WF^{2}Q+ using PIFO. First, we try using a single PIFO (Figure 2.2(d)). It is easy to see that this is not sufficient—if we order the PIFO by increasing finish times, it results in wrong scheduling order if some arbitrary element becomes eligible before the element at the head, and if we order the PIFO by increasing start times, the right scheduling order could still be violated if multiple elements become eligible at the same time, and the element with the smallest finish time is not at the head of the PIFO. A more promising approach is to use two PIFOs, ordered by increasing start and finish times respectively, and move elements between the two PIFOs as and when the elements become eligible, as demonstrated in Figure 2.2(e). However, this approach is also not sufficient, precisely because an arbitrary number of elements can become eligible at any given time,

the entire list is always eligible at dequeue.

[4] WF^{2}Q+ [11] is the implementation-friendly version of WF^{2}Q [26].

e.g., in Figure 2.2, C,D,E, and F all become eligible at $t = 5$, and ideally C should have been scheduled as C has the smallest finish time amongst the eligible elements. But since the eligibility PIFO is ordered by increasing start time, D is released to rank PIFO before C, resulting in the wrong scheduling order. In general, $O(N)$ elements (N is PIFO size) could become eligible at any given time, which in the worst-case could result in $O(N)$ deviation from the ideal scheduling order for an element.

Further, PIFO primitive does not allow dynamically updating the attributes (such as rank) of an arbitrary element in the ordered list, as required by certain scheduling algorithms, e.g., updating the priority of an element based on it's age to avoid starvation in a strict priority-based scheduling algorithm (Section 2.4.4).

Finally, the hardware design of the ordered list used to implement the PIFO primitive achieves very limited scalability ([104], 2.10a). Hence, PIFO scheduler is also not scalable. In principle, one could use *approximate* datastructures, such as a multi-priority fifo queue [47], a calendar queue [17], a timing wheel [172], or a multi-level feedback queue [8], to implement an approximate version of the PIFO primitive. These datastructures could approximate the behavior of a priority queue or an ordered list in a fast and scalable manner by using multiple FIFO queues. However, by design, they could only express approximate versions of key packet scheduling algorithms [95, 8], invariably resulting in weaker performance guarantees [181]. Further, these datastructures also tend to have several performance-critical configuration parameters, e.g., number of priority levels in a multi-priority fifo queue, or size and number of buckets in a calendar queue, which are not trivial to fine-tune.

Universal Packet Scheduling (UPS) [76]. In the same vein as PIFO, which tries to propose a general packet scheduling primitive, UPS tries to propose a single scheduling algorithm that can emulate all other packet scheduling algorithms. While [76] proves that no such algorithm exists, it also shows that the classical Least Slack Time First (LSTF) [130] algorithm comes close to being universal. However, just like PIFO, LSTF also uses a priority queue abstraction at it's core, as it always schedules the ″smallest slack first″, just as PIFO would schedule the ″smallest rank first″. As a result, LSTF has the same limitations as PIFO discussed above.

2.3 Design

In this section, we describe the PIEO primitive (Section 2.3.1) and present a programming framework to program the PIEO scheduler (Section 2.3.2).

2.3.1 Push-In-Extract-Out (PIEO) Primitive

PIEO primitive assigns each element an eligibility *predicate* and a *rank*, both of which can be programmed based on the choice of the scheduling algorithm, and at any given time, it schedules the ″smallest ranked eligible″ element. To realize this abstraction, PIEO maintains an ordered list of elements in the increasing order of rank, and supports three primitive operations atop the ordered list:

1. **enqueue(f):** This operation inserts an element f into the ordered list, at the position dictated by f's rank. This realizes the ″Push-In″ primitive.

2. **dequeue():** This operation first filters out a subset of elements from the ordered list whose eligibility predicates are true at the time, and then dequeues the element at the smallest index in that subset. Hence, this operation always dequeues the ”smallest ranked eligible” element. If there are multiple eligible elements with the same smallest rank value, then the element which was enqueued first is dequeued. If no eligible element exists, NULL is returned. This realizes the ”Extract-Out” primitive.
3. **dequeue(f):** This operation dequeues a specific element f from the ordered list. If f does not exist, NULL is returned.

While the *enqueue(f)* and *dequeue()* operations are sufficient to schedule elements according to the PIEO primitive, the additional *dequeue(f)* operation provides an added flexibility to asynchronously extract a specific element from the list, to say, dynamically update the scheduling attributes (such as rank) of the element (and then re-enqueue using *enqueue(f)*), as illustrated in Section 2.4.4.

Limitations on complexity of predicate function. PIEO primitive associates a custom predicate with each element, which is evaluated at dequeue to filter a subset of elements. However, the complexity of predicate function is limited by the practical constraints of a fast and scalable packet scheduler. In particular, we want each primitive operation to execute in as few clock cycles as possible to keep up with increasing link speeds, while encode the predicate in as few bits as possible for scalability. Fortunately, for most packet scheduling algorithms, the predicate usually takes the form ($t_{current} \geq t_{eligible}$), where t could be any monotonic increasing function of time, and $t_{eligible}$ is when the element becomes eligible for scheduling and can be calculated at enqueue into the ordered list (Section 2.4). This allows for a fast and parallel evaluation of predicates at

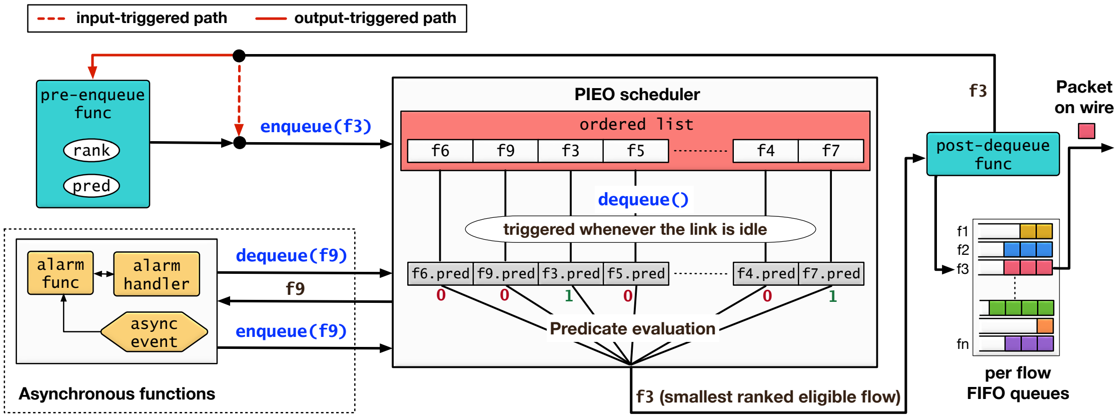


Figure 2.3: PIEO programming framework.

dequeue. Further, one only needs to encode $t_{eligible}$ for each element as the predicate, thus also ensuring a small storage footprint, important for scalability. One can potentially use PIEO primitive with more complex predicate functions for problems where constraints on time and memory are more relaxed.

2.3.2 PIEO Programming Framework

In this section, we describe a framework to program the PIEO scheduler. PIEO scheduler operates on a set of *flow queues*. Each packet is stored in one of the *flow queues*, and that the packets within each flow are scheduled in a FIFO order, as discussed in Section 2.2.1. Hence, each element in the ordered list refers to a flow, and scheduling a flow f results in the transmission of the packet at the head of flow queue f.

The programming framework for PIEO scheduler is shown in Figure 2.3. PIEO scheduler comprises an ordered list datastructure that implements the PIEO primitive (Section 2.3.1), and can be programmed through the *rank* and *predicate* attributes assigned to each element.

Programming functions. In this section, we describe three generic types of functions that the programmers can implement to program the PIEO scheduler. We also describe the default behavior of each function, which the programmers can then extend based on the choice of the scheduling algorithm they intend to program. All the state needed for scheduling can be stored as either per flow or global state, and should be accessible by both the control plane and the programming functions. The control plane can use the memory to store control states, e.g., per-flow rate limit value or QoS priority, while the programming functions can use the memory to store algorithm-specific state, e.g., virtual finish time in WFQ [26].

Pre-Enqueue and Post-Dequeue functions. Pre-Enqueue function takes as argument the flow f to be enqueued into the ordered list, and at the very minimum, assigns f a *rank* and a *predicate* as dictated by the choice of the scheduling algorithm being programmed. Note that while the predicate is assigned during enqueue into the list, it is only evaluated during the process of dequeue.

Post-Dequeue function takes as argument a flow f dequeued from the ordered list, and at the very minimum, transmits the packet at the head of flow queue f and re-enqueues f back into the ordered list if f's queue is not empty after current packet transmission.

While the Post-Dequeue function is always triggered after each *dequeue()* operation on the ordered list, the Pre-Enqueue function can be triggered in two different ways, depending upon whether it is triggered on a packet enqueue into the flow queue (Input-triggered model), or on a packet dequeue from the flow queue (Output-triggered model). We describe the two models in more detail below:

1. **Input-triggered model**: In this model, the Pre-Enqueue function is triggered whenever a packet is enqueued into a flow queue. The behavior of the default implementation of Pre-Enqueue and Post-Dequeue functions under this model is shown below.

```
pre-enqueue-func(flow f):  { # default
  f.curr_pkt.rank = 1
  f.curr_pkt.predicate = True
  if (pkt enqueue into an empty f.queue):
    ordered_list.enqueue(f)
}
post-dequeue-func(flow f):  { # default
  send(f.queue.head)
  if (f.queue not empty):
    f.rank = f.queue.head.rank
    f.predicate = f.queue.head.predicate
    ordered_list.enqueue(f)
}
```

2. **Output-triggered model**: In this model, the Pre-Enqueue function is triggered whenever a packet is dequeued from a flow queue, or a packet is enqueued into an *empty* flow queue. The behavior of the default implementation of Pre-Enqueue and Post-Dequeue functions under this model is shown below.

```
pre-enqueue-func(flow f):  { # default
  f.rank = 1
  f.predicate = True
  ordered_list.enqueue(f)
}
```

```
post-dequeue-func(flow f):  { # default
  send(f.queue.head)
  if (f.queue not empty):
    pre-enqueue-func(f)
}
```

Programmers have the flexibility to choose between the two models. The tradeoff is that while the output-triggered model can provide more precise guarantees for certain shaping policies [104], it also puts the Pre-Enqueue function on the critical path of scheduling, which means the complexity of rank and predicate calculations would affect the overall scheduling rate.

Alarm function and handler. The ability to asynchronously enqueue and dequeue specific elements to/from the ordered list using the *enqueue(f)* and *dequeue(f)* operations gives programmers the ability to define custom *events* which could trigger a custom *alarm function* that can asynchronously enqueue or dequeue a particular flow in or out of the ordered list. Programmers can also define a custom *alarm handler* function to operate upon the dequeued flow. By default, these functions are disabled in PIEO.

```
async_event e = NULL # default
alarm-func(async_event e):  {} # default
alarm-handler(flow f):  {} # default
```

Implementing programming functions. While we present a programming framework for the PIEO scheduler, the dissertation does not focus on a specific programming language to implement the programming functions, that are in turn used to program the PIEO scheduler. This would depend upon the underlying architecture of the hardware device. E.g., for FPGA devices, one could use languages such as System Verilog [139], Bluespec [118], or OpenCL [132] to implement the programming functions. In our FPGA prototype, we use System Verilog to implement the programming functions (Section 2.6.4). For ASIC hardware devices with RMT [16] architecture, one could potentially adapt one of the domain-specific languages for programmable switches such as P4 [15] or Domino [103] (used to program the PIFO scheduler). We leave the exploration of new domain-specific languages (and compilers) for new programmable hardware architectures on top of which one might implement the PIEO scheduler as an avenue for future work.

2.4 The Expressiveness of PIEO

In this section, we use the PIEO primitive and the programming framework described in Section 2.3 to express a wide-range of packet scheduling algorithms. Without loss of generality, the pseudo code for the programming functions presented in this section assumes the output-triggered model described in Section 2.3.2.

2.4.1 Work-conserving Algorithms

Deficit Round Robin (DRR) [98]. DRR schedules flows in a round-robin fashion. DRR is a credit-based algorithm, where once a flow is scheduled, DRR keeps transmitting packets from that flow until the flow runs out of credit (`f.deficit_counter`).

```
post-dequeue-func(flow f): {
  f.deficit_counter += f.quanta
  while (f.queue not empty
      & f.deficit_counter ≥ size(f.queue.head)):
    f.deficit_counter -= size(f.queue.head)
    send(f.queue.head)
  if (f.queue empty): f.deficit_counter = 0
  else: pre-enqueue-func(f)
}
other functions: default as described in Section 2.3.2
```

Weighted Fair Queuing (WFQ) [26]. WFQ calculates a virtual finish time for each packet in a flow, and at any given time, schedules the flow whose head packet has the smallest finish time value.

```
pre-enqueue-func(flow f): {
  r = Link_Rate / f.weight      # rate for flow f
  f.finish_time = max(f.finish_time, virtual_time)
            + (size(f.queue.head) / r)
  f.rank = f.finish_time
  f.predicate = True
  ordered_list.enqueue(f)
}
post-dequeue-func(flow f): {
  virtual_time += (size(f.queue.head) / Link_Rate)
  rest is default as described in Section 2.3.2
}
other functions: default as described in Section 2.3.2
```

Worst-case Fair Weighted Fair Queuing (WF^{2}Q+) [11]. WF^{2}Q+ calculates a virtual start and finish time for each packet in a flow, and at any given time, schedules the flow whose head packet has the smallest finish time value amongst all the flows whose head packet has the start time less than or equal to current virtual time.

```
pre-enqueue-func(flow f): {
  calculate f.start_time as in Figure 2.2(a)
  calculate f.finish_time as in Figure 2.2(a)
  f.rank = f.finish_time
  f.predicate = (virtual_time$_{(at\ deq)}$ $\geq$ f.start_time)
  ordered_list.enqueue(f)
}
post-dequeue-func(flow f): {
  calculate virtual_time as in Figure 2.2(a)
  rest is default as described in Section 2.3.2
}
other functions: default as described in Section 2.3.2
```

2.4.2 Non-work conserving Algorithms

Token Bucket (TB) [140]. Token Bucket is a classic non-work conserving algorithm, and is used by several real-world systems for rate limiting the flows. Each flow is assigned some tokens, based on the target rate for that flow. Token Bucket schedules packets from eligible flows, i.e., flows with enough tokens, or else defers the scheduling of the flow to some future time by when the flow has gathered enough tokens.

```
pre-enqueue-func(flow f): {
  f.tokens += f.rate * (now - f.last_time)
  if (f.tokens > f.burst_threshold):
    f.tokens = f.burst_threshold
  if (size(f.queue.head) ≤ f.tokens):
    send_time = now
  else:
    send_time = now + (size(f.queue.head) - f.tokens) / f.rate
  f.tokens -= size(f.queue.head)
  f.last_time = now
  f.rank = send_time
  f.predicate = (wall_clock_time$_{\text{(at deq)}}$ ≥ send_time)
  ordered_list.enqueue(f)
}
other functions: default as described in Section 2.3.2
```

Rate-controlled Static-Priority Queuing (RCSP) [184]. This class of algorithms shape the traffic in each flow by assigning an eligibility time to each packet within the flow, and at any given time, schedules the highest priority flow amongst all the flows with an eligible packet at the head of the queue.

```
pre-enqueue-func(flow f): {
  send_time = f.queue.head.time
  f.rank = f.priority
  f.predicate = (wall_clock_time$_{\text{(at deq)}}$ ≥ send_time)
  ordered_list.enqueue(f)
}
other functions: default as described in Section 2.3.2
```

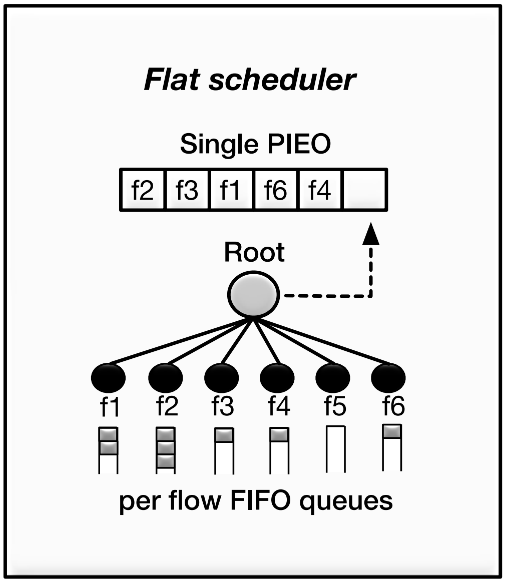


Figure 2.4: Classic flat PIEO scheduler.

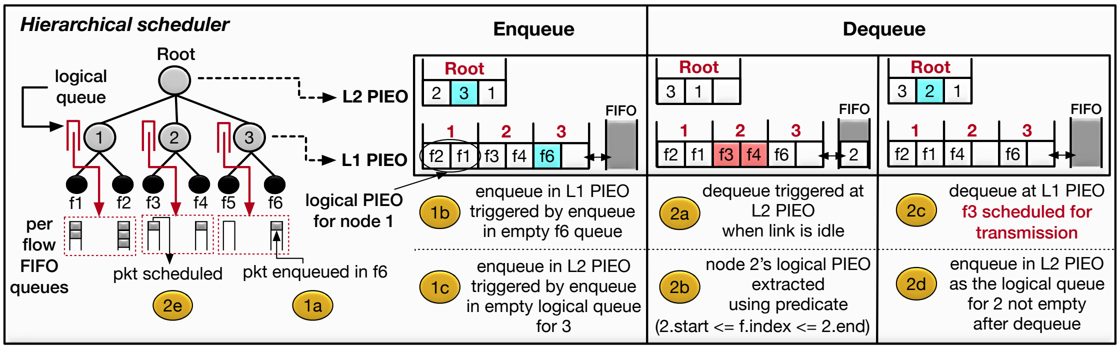

Hierarchical scheduler
Root
logical queue
L2 PIEO
L1 PIEO
1
2
3
f1
f2
f3
f4
f5
f6
per flow FIFO queues
pkt scheduled
pkt enqueued in f6
2e
1a
logical PIEO for node 1
Enqueue
Dequeue
Root
2
3
1
FIFO
f2
f1
f3
f4
f6
1b
enqueue in L1 PIEO triggered by enqueue in empty f6 queue
1c
enqueue in L2 PIEO triggered by enqueue in empty logical queue for 3
Root
3
1
2
2a
dequeue triggered at L2 PIEO when link is idle
2b
node 2's logical PIEO extracted using predicate (2.start <= f.index <= 2.end)
Root
3
2
1
2c
dequeue at L1 PIEO f3 scheduled for transmission
2d
enqueue in L2 PIEO as the logical queue for 2 not empty after dequeue

Figure 2.5: Hierarchical packet scheduling in PIEO.

2.4.3 Hierarchical Scheduling

So far we have only discussed flat scheduling (Figure 2.4). However, in practice, it is often desirable to group flows into a hierarchy of classes, e.g., a two-level hierarchy comprising a group of VMs, with a group of flows within each VM. In this example, the link bandwidth can be shared amongst the VMs using some scheduling policy, e.g., a rate limit for each VM, while one can use a different scheduling policy to schedule the flows within each VM, e.g., fair queuing. In general, we can represent such hierarchies using a tree structure, as shown in

Figure 2.5, where the leaf nodes represent the flows, while the non-leaf nodes represent higher-level classes, such as VMs. Unfortunately, since each non-leaf node can implement it's own custom scheduling policy to schedule it's children, a single PIEO is not sufficient to express hierarchical scheduling policies. However, we can support hierarchical scheduling using multiple PIEOs.

We associate each node in the tree (except the root node) with a queue—for leaf nodes, these are per flow FIFO queues storing the packets, while for non-leaf nodes, these are *logical queues*, which are references to the set of child queues for that node. Next, we associate each non-leaf node with a *logical PIEO*, which schedules the node's children. Since PIEO allows us to filter a subset of elements from an ordered list using a predicate, all the nodes at the same level of hierarchy can share the same physical PIEO, which can then be logically partitioned into a set of logical PIEOs, one for each node at the same level in the hierarchy, with the size of each logical PIEO equal to number of corresponding node's children. Next, each non-leaf node p maintains the start and end indices of it's logical PIEO. We use that to extend (AND boolean operation) the existing eligibility predicate of each of it's child element, f, with ($p.start \leq f.index \leq p.end$), thus allowing one to extract the ordered list of elements in p's logical PIEO (p's children) from the physical PIEO.

Enqueue in each level happens independently and is triggered by the same conditions as for flat scheduling (Section 2.3.2). Dequeue always starts at the root PIEO, and propagates down to the lower levels in the tree hierarchy. Each lower level PIEO is associated with a FIFO to store the dequeued ids from the parent level. Dequeue at a level i is triggered whenever the corresponding FIFO in not empty. The logical PIEO corresponding to node *fifo.head* is then extracted, and the smallest ranked eligible element in the logical PIEO is dequeued and put into the FIFO at level $i-1$, until we reach the lowest non-leaf level, at which point the dequeued element (a leaf node representing a flow) is scheduled for transmission. All this is demonstrated in Figure 2.5.

Finally, to support $n-$level hierarchical scheduling with arbitrary tree topologies, we need n physical PIEOs. We map this hierarchy to the hardware as an array of n independent physical PIEOs with a FIFO as the interface between any two consecutive PIEOs in the arraylist (Figure 2.5).

2.4.4 Asynchronous Scheduling

Starvation avoidance in strict priority scheduling. A common way to avoid starvation of lower priority flows in a strict priority-based scheduling algorithm is to periodically increase the priority of the flow being starved. This is generally triggered whenever a flow has spent time larger than some threshold without being scheduled. Assuming flows are ranked by their priority in PIEO, one can define an alarm function and handler that can asynchronously update the starving flow's priority to avoid starvation.

```
async_event e = (curr_time - f.age ≥ threshold)
alarm-func(async_event e): ordered_list.dequeue(f)
alarm-handler(flow f): {
  f.age = curr_time
  f.priority = f.priority - 1
  pre-enqueue-func(f)
}
```

Scheduling based on asynchronous network feedback. Certain datacenter protocols such as [177, 165] can result in change of a flow's rank or eligibility based on some asynchronous feedback from the network. E.g., in D^3 [177], already scheduled flows can be quenched asynchronously based on the feedback received from the network.

```
async_event e = receipt of pause or resume feedback
alarm-func(async_event e): {
  if (recvd pause feedback for flow f):
    f.block = True
    ordered_list.dequeue(f)
  if (recvd resume feedback for flow f):
    f.block = False
    pre-enqueue-func(f)
}
alarm-handler(flow f): default as described in Section 2.3.2
```

2.4.5 Priority Scheduling

Several scheduling algorithms assign a *priority* to each element, and schedule elements in the order of their priority. Examples include Shortest Job First (SJF) [137], Shortest Remaining Time First (SRTF) [138], Least Slack Time First (LSTF) [130], and Earliest Deadline First (EDF) [123]. Such algorithms can be expressed using a priority queue datastructure. One can easily emulate a priority queue using PIEO, by setting the rank of each element as equal to it's priority value, and setting the eligibility predicate of each element as true.

2.5 Hardware Architecture

> *"All problems in computer science can be solved by another level of indirection."*
>
> *— David Wheeler*

In this section, we describe the hardware architecture of PIEO scheduler.

2.5.1 Hardware Model

We assume that the target hardware device is equipped with SRAM. Further, we assume that SRAM is divided into multiple blocks, and each SRAM block comprises independent access ports. Such a memory layout is very common in hardware devices, e.g., Stratix V FPGA [154] used in our prototype (Section 2.6.1) comprise ~2500 dual-port SRAM blocks of size 20 KBits each, where each SRAM block has access latency of one clock cycle.

2.5.2 Architecture and Key Complexity Results

PIEO scheduler comprises an ordered list, that supports three primitive operations (Section 2.3.1)—*enqueue(f)*, *dequeue()*, and *dequeue(f)*. However, implementing an ordered list in hardware presents a fundamental trade-off between time complexity and hardware resource consumption. To keep up with the increasing link speeds, we want to execute each primitive operation atop the ordered list in $O(1)$ time. However, to achieve this, state-of-the-art designs such as PIFO [104] require parallel access to each element in the list using the classic *parallel compare-and-shift* architecture [78], and hence have to store the entire list

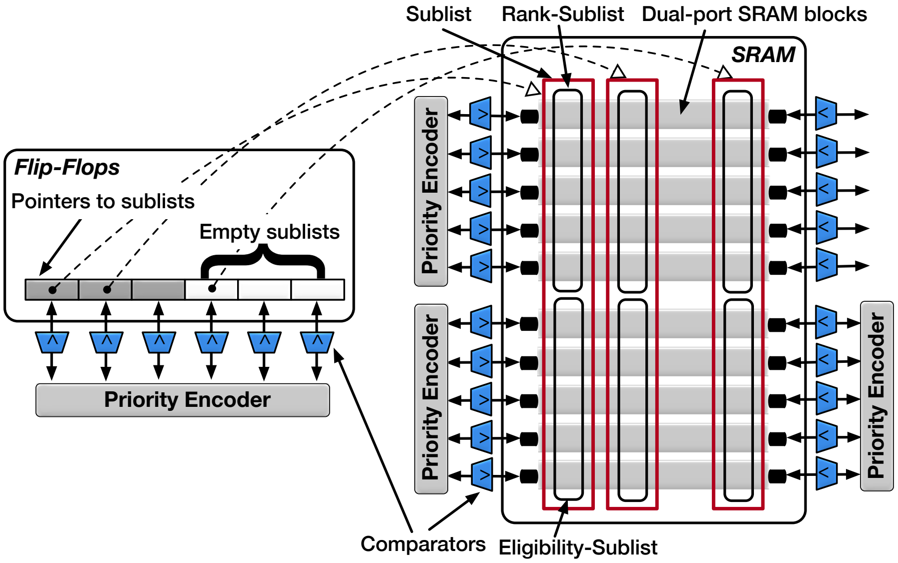


Figure 2.6: PIEO scheduler hardware architecture. A priority encoder takes as input a bit vector and returns the smallest index containing 1.

in flip-flops (as opposed to a more scalable memory technology such as SRAM), and associate a comparator with each element. Thus, such a design requires $O(N)$ flip-flops and comparators for a list of size N, and with the slowdown in transistor scaling, this limits the scalability of such a design.

In this dissertation, we present a design of the ordered list that still executes primitive operations in $O(1)$ time, but only needs to access and compare $O(\sqrt{N})$ elements in parallel, while the ordered list sits entirely in SRAM. The key insight we use is to store and access the ordered list using one level of indirection (Figure 2.6). More specifically, the ordered list is stored as an array (of size $2\sqrt{N}$) of *sublists* in SRAM, where each sublist is of size $\sqrt{N}$ elements. Elements within each sublist are ordered by both increasing rank (*Rank-Sublist*), and increasing order of eligibility time (*Eligibility-Sublist*). We stripe the elements of each sublist across $O(\sqrt{N})$ dual-port SRAM blocks, which allows us to access two entire sublists in one clock cycle. Next, we maintain an array (of size $2\sqrt{N}$) in flip-flops, which stores the pointers to the sublists, with sublists in the array ordered by increasing value of the smallest rank within each sublist. Thus, by sweeping across the sublists in the order they appear in the pointer array, one can get the entire list of elements in the order of increasing rank.

Enqueue and dequeue operations proceed in two steps—First, we figure out the right sublist to enqueue into or dequeue from, using parallel comparisons and priority encoding on the pointer array, and then extract the corresponding sublist from SRAM. Second, we use parallel comparisons and priority encoding on the extracted sublist to figure out the position within the sublist to enqueue/dequeue the element, and then write back the updated sublist to SRAM. Thus, with this design, we only require $O(\sqrt{N})$ flip-flops and comparators, unlike $O(N)$ in PIFO, at the cost of few extra clock cycles to execute each primitive operation (Section 2.5.3) and 2× SRAM overhead (INVARIANT 1). We evaluate these trade-offs in Section 2.6.1.

Key Complexity Results.

The key complexity results of our hardware design are summarized below.

- **Time complexity.** Each PIEO primitive operation (*enqueue(f), dequeue(), and dequeue(f)*) executes in exactly 4 clock cycles, regardless of the number of elements in PIEO.
- **Processing resource complexity.** PIEO of size N elements requires $O(\sqrt{N})$ comparators and flip-flops.
- **Memory resource complexity.** PIEO of size N elements requires $O(\sqrt{N})$ SRAM blocks and a total of $2N$ elements worth of SRAM space.

2.5.3 Implementation

In SRAM, PIEO maintains an array (of size $2\sqrt{N}$) of sublists, called *Sublist-Array*. Each sublist in the array is of size $\sqrt{N}$. Further, each sublist comprises two ordered sublists—*Rank-Sublist* and *Eligibility-Sublist*. Each element in *Rank-Sublist* comprises three attributes:

1. **flow_id:** The flow id of the element.
2. **rank:** The *rank* value assigned to the element by the enqueue function.
3. **send_time:** This encodes the eligibility predicate assigned to the element by the enqueue function. PIEO exploits the fact that most scheduling algorithms use eligibility predicate of the form (*curr_time* ≥ *send_time*) (Section 2.4), where *send_time* is the time the element becomes eligible for scheduling. Thus, the eligibility predicate in PIEO is encoded using a single *send_time* value for each element. Predicate that is always true is encoded by assigning *send_time* to 0, and predicate that is always false is encoded by assigning *send_time* to ∞.

The *Rank-Sublist* is ordered by increasing *rank* values. Further, corresponding to each *Rank-Sublist*, there is an *Eligibility-Sublist* of the same size, which maintains a copy of the *send_time* attribute from it's corresponding *Rank-Sublist*. *Eligibility-Sublist* is ordered by increasing *send_time* values.

In flip-flops, PIEO maintains an array of size $2\sqrt{N}$, called *Ordered-Sublist-Array*, where each entry in the array points to a sublist in the *Sublist-Array*. More specifically, each entry in the *Ordered-Sublist-Array* comprises three attributes:

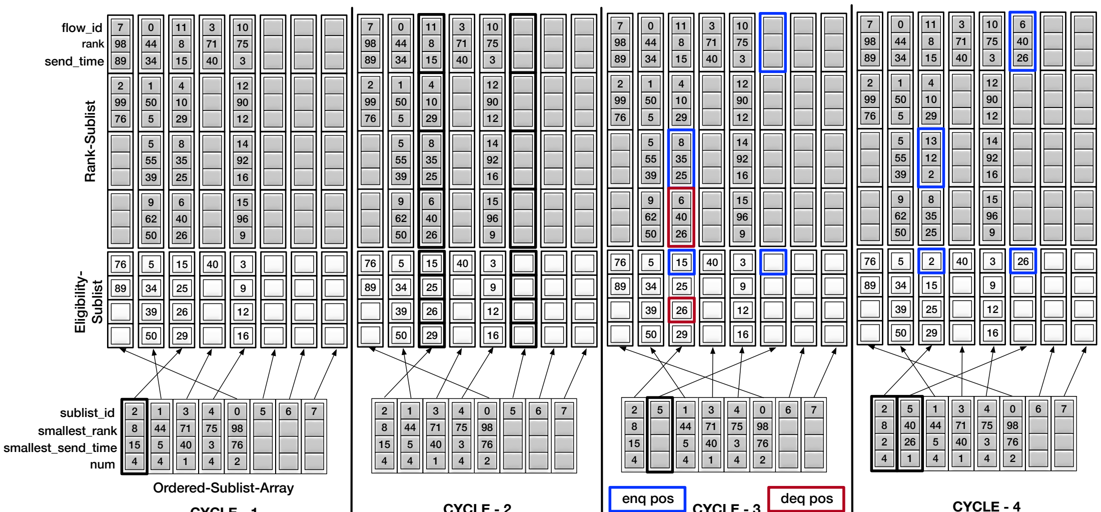
1. **sublist_id:** This is the index (pointer) into the *Sublist-Array*, pointing to sublist *Sublist-Array[sublist_id]*.
2. **smallest_rank:** This is the smallest *rank* value in the sublist *Sublist-Array[sublist_id]*, i.e., *Sublist-Array[sublist_id]*.*Rank-Sublist[0]*.*rank*.
3. **smallest_send_time:** This is the smallest *send_time* value in the sublist *Sublist-Array[sublist_id]*, i.e., *Sublist-Array[sublist_id]*.*Eligibility-Sublist[0]*.
4. **num:** This stores the current number of elements in *Sublist-Array[sublist_id]*.

Ordered-Sublist-Array is ordered by increasing *smallest_rank* value. Further, *Ordered-Sublist-Array* is dynamically partitioned into two sections, as shown in Figure 2.6—the section on the left points to sublists which are not empty, while the section on the right points to all the currently empty sublists.

By stitching together sublists in the order they appear in the *Ordered-Sublist-Array*, one can get the entire list of elements in PIEO. We call this list *Global-Ordered-List*. *Global-Ordered-List* is ordered by increasing *rank* value. Logically, enqueue and dequeue operations happen on top of the *Global-Ordered-List*.

Enqueue(f). The enqueue operation inserts element *f* into the *Global-Ordered-List*. It ensures that after every enqueue operation, the resulting *Global-Ordered-List* is ordered by increasing *rank* value. This is implemented in hardware as follows:

- **Cycle 1:** In this cycle, we select the sublist to enqueue *f* into, using the parallel compare operation (*Ordered-Sublist-Array[i]*.*smallest_rank* > *f*.*rank*). We feed the resulting bit-vector into a priority encoder, which outputs index *j*. Sublist *S* pointed by *Ordered-Sublist-Array[j-1]* is selected for enqueue.



CYCLE - 1
Enqueue request f = [13, 12, 2]
Sublist 2 selected for enq.

CYCLE - 2
Sublist 2 read from SRAM.
Since Sublist 2 is full, and
so is Sublist 1, empty list 5 is
also read from SRAM.

CYCLE - 3
Priority encoder used to figure out
positions to enq/deq.
Ordered-Sublist-Array re-arranged
by shifting Sublist 5 to right of 2.

CYCLE - 4
Elements enq/deq at right positions.
Sublists 2 and 5 written back to SRAM.
Ordered-Sublist-Array entries
for Sublists 2 and 5 updated.

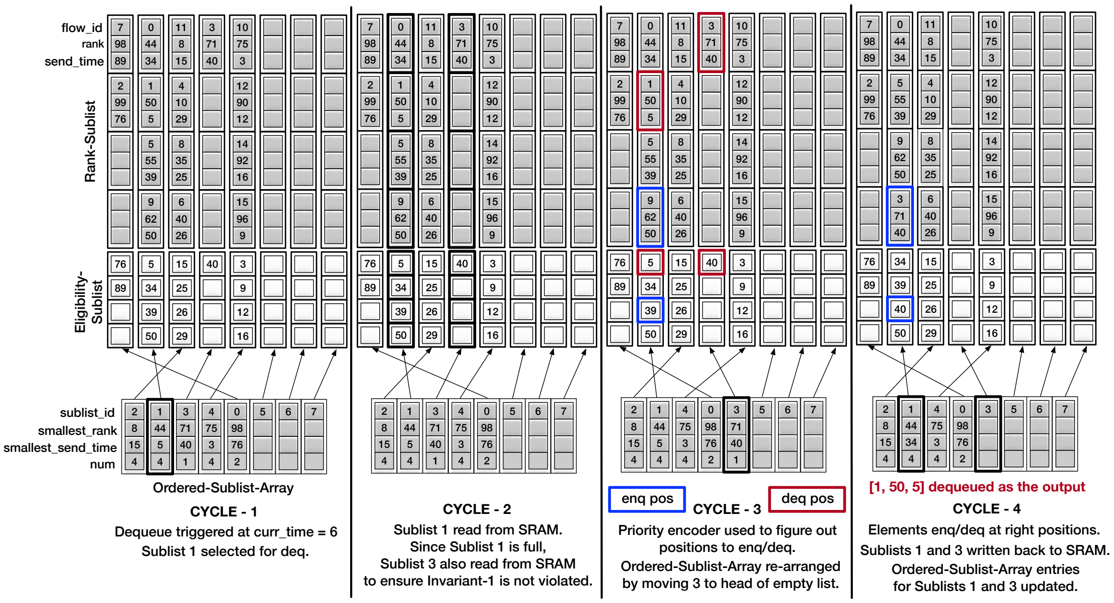
Figure 2.7: An example enqueue into the PIEO ordered list of size 16 elements (8 sublists each of size 4).

- **Cycle 2:** In this cycle, we read the sublist S from SRAM. In case S was full, the enqueue operation would push out an existing element in S. Hence, we also read an additional sublist S' to store the pushed out element. Typically, S' would be the sublist to the immediate right of S in the *Ordered-Sublist-Array*, provided it is not full. But in case it is full, we read an empty sublist as S', to add the ejected element from S, and move that sublist to the immediate right of S in the next cycle, thus preserving the list ordering while also avoiding a potential chain-reaction of shifting the tail element of one sublist to the head of next (which would result in worst-case $O(\sqrt{N})$ SRAM accesses). We explain in the next cycle how indirection allows us to (logically) shift sublists around within one clock cycle.
- **Cycle 3:** In this cycle, we figure out the position to enqueue within sublist S, by running priority encoders on top of bit vectors returned by parallel compare operations (S.*Rank-Sublist[i].rank* > *f.rank*), and (S**.***Eligibility-Sublist[i]* > *f.send_time*) respectively In case S was full, the tail element

in *S.Rank-Sublist* will be moved to the head of *S′.Rank-Sublist*, while we use parallel compare operation (*S′.Eligibility-Sublist[i]* > *S[tail].send_time*), followed by priority encoding, to figure out the position to enqueue the *send_time* value of the element moving from S into the eligibility sublist within S'. In case S' was initially empty, we also re-arrange the *Ordered-Sublist-Array* by shifting S' to the immediate right of S. This shifting can be done efficiently in one clock cycle since we do not shift the physical sublist in SRAM, but only the pointer to the sublist stored in *Ordered-Sublist-Array*, which, in turn, is stored in flip-flops, which allows for parallel shifting in one clock cycle. This further highlights the benefit of indirection.

- **Cycle 4:** In this cycle, we enqueue/dequeue respective elements at the positions output from the last cycle, and write back S (and S') to the SRAM. We also update the *Ordered-Sublist-Array* entries for S and S' with the new values of (i) *num*, (ii) *smallest_rank* (read from the corresponding *Rank-Sublist*), and (iii) *smallest_send_time* (read from the corresponding *Eligibility-Sublist*).

Invariant 1 (Bounding the number of sublists) The key to ensuring $O(1)$ enqueue time is choosing a new empty sublist for enqueue whenever both the sublist to which the new element is to be enqueued and the sublist to it's immediate right in the *Ordered-Sublist-Array* are full. This avoids the chain-reaction of shifting the tail element of one sublist to the head of next (which would result in worst-case $O(\sqrt{N})$ SRAM accesses), at the cost of memory fragmentation (Figure 2.7). However, an upshot of this design is the invariant that there cannot be two consecutive partially full sublists in the *Ordered-Sublist-Array*. As a consequence, to store N elements using $\sqrt{N}$-sized sublists, one would require at most $2\sqrt{N}$ sublists (2× SRAM overhead). For a detailed proof, refer to Appendix A.



CYCLE - 1
Dequeue triggered at curr_time = 6
Sublist 1 selected for deq.

CYCLE - 2
Sublist 1 read from SRAM.
Since Sublist 1 is full,
Sublist 3 also read from SRAM
to ensure Invariant-1 is not violated.

CYCLE - 3
Priority encoder used to figure out
positions to enq/deq.
Ordered-Sublist-Array re-arranged
by moving 3 to head of empty list.

[1, 50, 5] dequeued as the output

CYCLE - 4
Elements enq/deq at right positions.
Sublists 1 and 3 written back to SRAM.
Ordered-Sublist-Array entries
for Sublists 1 and 3 updated.

Figure 2.8: An example dequeue from the PIEO ordered list of size 16 elements (8 sublists each of size 4).

Dequeue(). This operation returns the ”smallest ranked eligible” element in *Global-Ordered-List*. It is implemented as follows:

- **Cycle 1:** In this cycle, we select the sublist that contains the ”smallest ranked eligible” element. For this, we use the priority encoder to extract the sublist at the smallest index in the *Ordered-Sublist-Array* that satisfies the predicate (*curr_time* ≥ *Ordered-Sublist-Array[i]***.***smallest_send_time*). We call it *S*. The predicate ensures that *S* will have at least one eligible element, and since the *Ordered-Sublist-Array* is ordered by increasing *smallest_rank* value, the ”smallest ranked eligible” element in the entire list is guaranteed to be in *S*.
- **Cycle 2:** In this cycle, we read the sublist *S* from SRAM. In case *S* was full, after a dequeue it would be partially full. So, to ensure INVARIANT 1 is not violated, we read another sublist, either to the immediate left of *S* in the *Ordered-Sublist-Array*, or to it’s immediate right, whichever is not full, and

choose either in case both of them are not full. We call it S'. If both left and right sublists are full, we only read S, as in that case even a partially full S would not violate INVARIANT 1.

- **Cycle 3:** In this cycle, we figure out the position to dequeue the "smallest ranked eligible" element from S. For that, we use the priority encoder that outputs the smallest index *idx* in *S.Rank-Sublist* satisfying the predicate (*curr_time* ≥ *S.Rank-Sublist[i].send_time*). The "smallest ranked eligible" element to be dequeued and returned as the final output of *dequeue()* operation is *S.Rank-Sublist[idx]*. Further, in the case S was full, we move an element from S' to S, to ensure that S remains full even after dequeue, hence ensuring that INVARIANT 1 will not be violated. The element to be moved, *e*, is deterministically added to either the head (if S' is to the left of S) or to the tail (if S' is to the right of S) of *S.Rank-Sublist* in the next cycle. However, we have to rely on priority encoding to figure out the position to dequeue *e.send_time* from *S′.Eligibility-Sublist*, and the corresponding position in *S.Eligibility-Sublist* where it would be enqueued. For that, we use parallel compare operations (*S′.Eligibility-Sublist[i]* == *e.send_time*) and (*S.Eligibility-Sublist[i]* > *e.send_time*) respectively Finally, in case either S or S' becomes empty after dequeue, we re-arrange the *Ordered-Sublist-Array* by shifting S or S' to the head of the logical partition of empty sublists.

- **Cycle 4:** In this cycle, we enqueue/dequeue respective elements at the positions output from the last cycle, and write back S (and S') to the SRAM. We also update the *Ordered-Sublist-Array* entries for S and S' with the new values of (i) *num*, (ii) *smallest_rank* (read from the corresponding *Rank-Sublist*), and (iii) *smallest_send_time* (read from the corresponding *Eligibility-Sublist*).

Dequeue(f). This operation dequeues a specific element f from the *Global-Ordered-List*. PIEO keeps track of the sublist id that each element (flow) within the *Global-Ordered-List* is stored at, as part of the flow state, and updates this information after each primitive operation. In cycle 1, we use that information to select the sublist storing f, and then repeat cycles 2-4 from the *dequeue()* operation, with a modification in cycle 3 where we use the predicate (f == *S.Rank-Sublist[i].flow_id*) to figure out the index *idx* of the element in *S.Rank-Sublist* to be dequeued and returned as the final output.

2.6 Evaluation

In this section, we evaluate the performance of our FPGA prototype across three metrics—(i) Scalability, (ii) Scheduling rate, and (iii) Programmability. To serve as the baseline, we synthesized the open-source PIFO implementation [142] atop our FPGA. We use 16-bit rank and predicate fields, same as in PIFO implementation.

2.6.1 Prototype

We prototyped PIEO on a Terasic DE5-Net board [113] comprising an Altera Stratix V [154] FPGA with 234 K Adaptive Logic Modules (ALMs), 52 MBits (6.5 MB) SRAM, and four 10 Gbps ports for a total of 40 Gbps interface bandwidth. Our prototype is written in System Verilog [139] (~1300 LOCs).

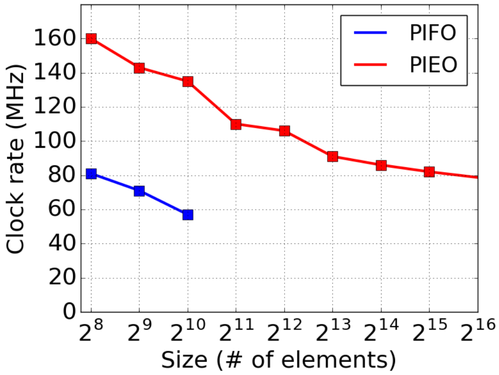


Figure 2.9: Clock rates achieved by the scheduler circuit.

2.6.2 Prototype Experiments: Scheduling Rate

In this section, we evaluate the rate at which PIEO scheduler can make the scheduling decisions. Scheduling rate is typically a function of: (a) the clock rate of the scheduler circuit, and (b) number of cycles needed to execute each primitive operation[5]. Each primitive operation in PIEO takes 4 clock cycles and Figure 2.9 shows the clock rate of PIEO circuit against increasing PIEO size. The clock rate naturally decreases with increasing circuit complexity, but even at 80 MHz and assuming a non-pipelined design, one can execute a PIEO primitive operation every 50 ns, which is sufficient to schedule MTU-sized packets at 200 Gbps line rate. A recent work [93] on software packet scheduler demonstrated that a single CPU core was only able to schedule MTU-sized packets at 10 Gbps line rate, thus making our prototype 20× faster.

PIEO's scheduling rate can be further improved by pipelining the primitive operations. In a fully pipelined design, one could execute one primitive operation every clock cycle. However, PIEO's design is limited by the number of SRAM access ports. As such, the memory stages of each primitive oper-

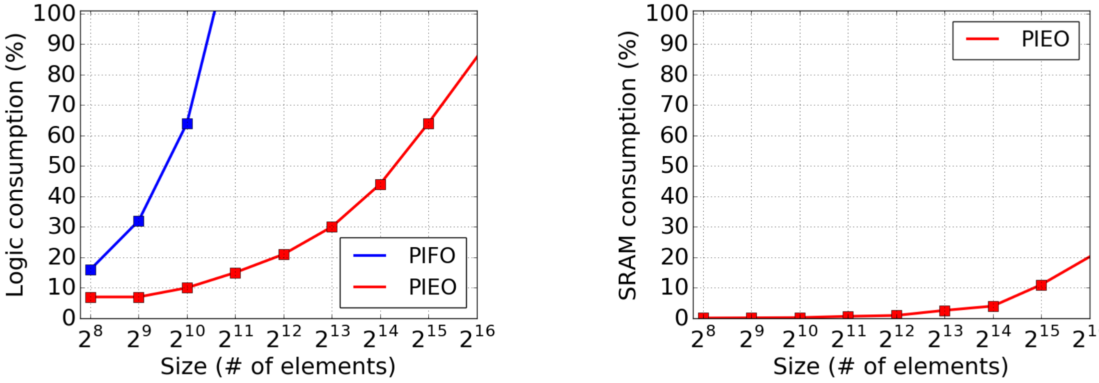
[5] For output-triggered model (Section 2.3.2), the complexity of Pre-Enqueue function also affects the scheduling rate, as explained in Section 2.3.2.

ation (cycle 2 and 4) uses both the available access ports of dual-port SRAM to read/write two sublists. Hence, memory stages of different primitive operations cannot be executed in parallel, thus preventing a fully pipelined design. In principle, by carefully scheduling the primitive operations, one can still achieve some degree of pipelining, resulting in a better scheduling rate than a non-pipelined design. However, for simplicity, our prototype only implements the non-pipelined design, and all the analysis and results in the dissertation are for a non-pipelined design.

Further, the clock rates achieved by PIEO is a function of both the PIEO design and the hardware device used to run the design. We expect our design to run at much higher clock rates on more powerful FPGAs [153], but even more importantly, on an ASIC hardware, as ASIC-based implementations tend to be more performant than an equivalent FPGA-based implementation of the same design [56]. To back this using a concrete example, we note that PIFO's design on top of our FPGA was clocked at 57 MHz, as opposed to 1 GHz on an ASIC hardware as shown in [104]. At 1 GHz clock rate, each primitive operation in PIEO would only take 4 ns, making PIEO 300× faster than a single CPU core.

PIEO vs. PIFO trade-offs. Unlike PIEO, PIFO's hardware design can be fully pipelined, partly because it does not access SRAM at all, and hence PIFO scheduler can schedule at a higher rate than PIEO. This is the price we pay in PIEO's hardware design to achieve order of magnitude more scalability than PIFO. Alternatively, one can also implement PIEO primitive using PIFO's hardware design[6] and achieve more expressibility than PIFO without compromising on scheduling rate, albeit at a much smaller scale. We, however, argue that the

[6] Porting PIEO primitive to PIFO's hardware design is trivial despite PIEO supporting a more complex dequeue function, because the kind of predicates used in PIEO implementation can be evaluated in parallel in flip-flops in one clock cycle.



(a) Logic consumption

(b) Memory consumption

Figure 2.10: Percentage of logic modules (ALMs) consumed (out of 234 K) and Percentage of SRAM consumed (out of 6.5 MB).

trade-off made in PIEO's hardware design is a good trade-off to make, as PIEO's design is still extremely fast and can schedule in tens of nanoseconds (even less at higher clock rates), while also scale to tens of thousands of flows, which is critical in multi-tenant cloud networks [88, 92].

2.6.3 Prototype Experiments: Scalability

In this section, we evaluate how the logic and memory resources consumed by PIEO's design scale with the size of the PIEO scheduler. We report the percentage of available Adaptive Logic Modules (ALMs) consumed to implement the combinational and sequential logic (Figure 2.10(a)), and the percentage of available SRAM consumed to store the ordered list (Figure 2.10(b)). Further, we compare it against the PIFO implementation [142].

The baseline PIFO implementation consumes 64% of the available logic modules to implement a PIFO scheduler of size 1 K elements, and this scales linearly with the size of PIFO, meaning we can't fit a PIFO with 2 K elements or

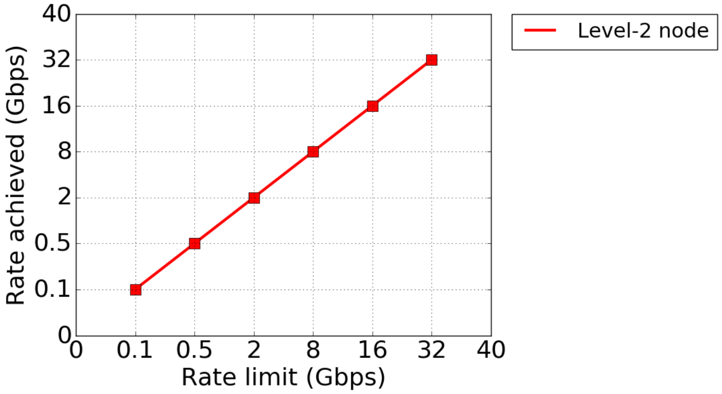


Figure 2.11: Rate limit enforcement in PIEO prototype.

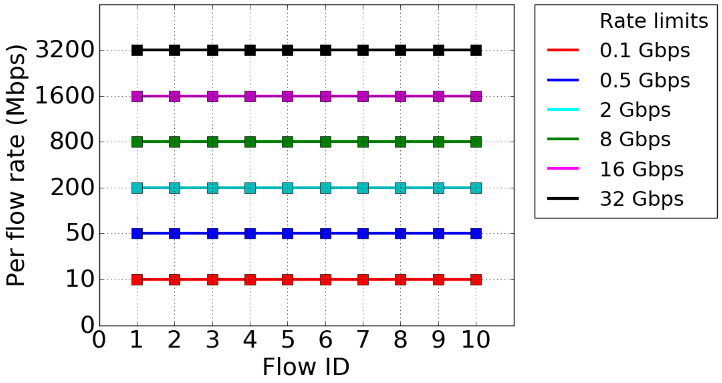


Figure 2.12: Fair queue enforcement in PIEO prototype.

more on our FPGA. In contrast, the logic consumption for PIEO increases sub-linearly (as the square root function), and we can easily fit a PIEO scheduler with 30 K elements on our FPGA. This is a direct consequence of PIEO's design, which unlike PIFO, exploits the memory hierarchy available in hardware to efficiently distribute storage and processing across SRAM and flip-flops. Of course, this scalability comes at the cost of 2× SRAM overhead as discussed in INVARIANT 1. However, even with 2× SRAM overhead, the total SRAM consumption for PIEO's implementation is fairly modest as shown in Figure 2.10(b).

2.6.4 Prototype Experiments: Programmability

We show in Section 2.4 that one can express a wide-range of scheduling algorithms using the PIEO primitive. In this section, we program two such algorithms, namely Token Bucket (Section 2.4.2) and WF^{2}Q+ (Section 2.4.1), atop our FPGA prototype using System Verilog as the programming language. The two chosen algorithms implement two of the most widely used scheduling policies in practice, namely rate limiting and fair queuing.

We program a two-level hierarchical scheduler using our prototype, with ten nodes at level-2 and ten flows within each node, for a total of 100 flows. We implement packet generators, one per flow, on the FPGA to simulate the flows. The link speed is 40 Gbps, and we schedule at MTU granularity. For experiments, we assign varying rate limit values to each node at level-2 in the hierarchy, and enforce it using the Token Bucket algorithm. The rate limit value of a particular node at level-2 is then shared fairly across all it's ten flows using WF^{2}Q+ algorithm. In Figure 2.11, we sample a random level-2 node, and show that PIEO scheduler very accurately enforces the rate limit on that node. Further, in Figure 2.12, we show that for each rate limit value assigned to the chosen level-2 node, PIEO scheduler very accurately enforces fair queuing across all the flows within that level-2 node.

2.7 Discussion

In this section, we discuss the applicability of PIEO's hardware design beyond packet scheduling. In particular, we discuss how PIEO can be used to implement key datastructures such as priority queues [134] and key-value (dictionary) stores [122] in hardware, thus providing a potential building block for designing hardware-accelerated applications in the Post-Moore era.

2.7.1 PIEO as a Generic Priority Queue

One can readily use PIEO as a generic $O(1)$-time priority queue, by setting the *rank* attribute to the priority value, and setting the *predicate* to be always True. Thus, one can use PIEO instead of the traditional heap-based implementation of the priority queue, and improve the time complexity from $O(log(N))$ to $O(1)$, where N is the size of the queue.

2.7.2 PIEO as an Abstract Dictionary Datatype

PIEO primitive can also be viewed as an *abstract dictionary data type* [122], which maintains a collection of (*key, value*) pairs, indexed by *key*, and allows operations such as search, insert, delete and update. PIEO presents an extremely efficient implementation of the dictionary data type in hardware, which can do all the above mentioned operations in $O(1)$ time, while also being scalable. Further, it can also very efficiently support certain other key dictionary operations considered traditionally challenging, such as filtering a set of keys within a range, as PIEO implementation described in Section 2.5 can be naturally extended to support predicates of the form $a \leq key \leq b$. Thus, PIEO presents an attractive alternative to the traditional hardware implementations of the dictionary data type, such as hashtables [127] and search trees [136].

2.8 Summary

In this chapter, we presented a new packet scheduling primitive, called *Push-In-Extract-Out (PIEO)*, which could express a wide-range of packet scheduling algorithms. PIEO assigns each element a rank and an eligibility predicate, both of which could be programmed based on the choice of the scheduling algorithm, and at any given time, schedules the ″smallest ranked eligible″ element. We presented a fast and scalable hardware architecture of PIEO scheduler, and prototyped it on an FPGA. Our prototype could schedule in tens of nanoseconds, while also scale to tens of thousands of flows. Thus overall, PIEO scheduler is simultaneously programmable, scalable, and high-speed.

CHAPTER 3

FAST CIRCUIT-SWITCHED SWITCHING FABRIC : SHOAL

In this chapter, we describe the design and implementation of a datacenter switching fabric that can scale to high switching speeds in a cost and power efficient manner while also achieving high-performance.

3.1 Overview

Traditionally, datacenter switching fabrics are built using a collection of low port-count packet switches arranged in some network topology, the most common being a Folded Clos topology such as Fattree [1]. The topology is designed in a way that can provide the necessary bisection bandwidth needed by the applications running inside the datacenters. Further, when coupled with sate-of-the-art network protocols [4, 187, 41, 2], the packet-switched fabric can be extremely high performant in terms of both high throughput and low latency. However, the slowdown in Moore's law combined with the rising bandwidth demand means that the switching speed of packet switches is no longer able to keep pace with the bandwidth demand (Section 1.3.2). As a result, to meet the required bisection bandwidth, one would require more number of switches, more number of tiers in the topology, more number of fibers, and more number of transceivers, resulting in increased power, cost, and number of hops through the network (latency).

The limitations of packet-switched networks have already prompted network designs that leverage circuit switches in datacenters [31, 175, 20, 87, 39, 185, 38, 35, 74, 59]. Such switches can be optical or electrical, and the fact that they operate at the physical layer with no buffers, no arbitration and no packet inspection mechanisms means they can be cheaper and more power efficient than an equivalent packet switch (Section 3.6). These benefits are particularly magnified for optical circuit switches on account of being data rate agnostic, and hence, unlike packet switches, can scale to arbitrary switching speeds with no increase in cost and power. However, achieving low latency is still challenging as traditional circuit switches have reconfiguration delays of the order of few microseconds [74] to even milliseconds [31]. Such a solution, thus, either compromises on performance or still relies on a separate packet-switched network to handle latency-sensitive traffic. In summary, existing network solutions for high-speed switching fabric either compromise on power and cost efficiency (packet-switched) or on performance (slow circuit-switched).

In this chapter, we show that it is possible to design a high-speed switching fabric network that is cost and power efficient while achieving performance comparable to packet-switched networks. Our work is motivated by fast circuit switches that can be reconfigured in a few to tens of nanoseconds while still being power and cost efficient. These are available commercially [167] as well as research prototypes [59, 19, 55, 29, 183, 63, 106, 27]. Unfortunately, it is not sufficient to simply take existing circuit-switch-based architectures and upgrade their switches from slow to fast reconfigurable circuit switches, as these architectures were designed under the assumption of slow reconfiguration times. In particular, these solutions rely either on a centralized controller to reconfigure the switches [59, 20, 87, 39, 185, 38], which would be infeasible at a nanosecond scale, or on a scheduler-less design with a large congestion control loop [74], which prevents taking advantage of fast reconfiguration speeds.

We present Shoal, a power and cost efficient yet performant high-speed switching fabric for datacenters built using fast circuit switches. Shoal proposes a new control plane for circuit scheduling which reconfigures the fabric using a *static, pre-defined schedule* that connects each pair of end-hosts at an equal rate (Section 3.3.4). This avoids the need for a slow centralized scheduler. To accommodate dynamic traffic patterns atop a static schedule, traffic from each end-host is uniformly distributed across all end-hosts which then forward it to the destination; a form of detour routing (Section 3.3.5). Such *traffic agnostic scheduling*, first proposed by Chang et al. [18] as an extension of Valiant's method [171], obviates the complexity and latency associated with centralized schedulers while guaranteeing the worst-case network throughput across *any* traffic pattern [18]. Such scheduling, however, requires that all end-hosts are connected through what looks like a single circuit switch. To achieve this, Shoal's fabric either uses a single large port-count circuit switch or many low port-count circuit switches connected in a non-blocking Clos topology, such as a Fattree [1] with full bisection bandwidth, that are reconfigured synchronously to operate like a single circuit switch (Section 3.3.2).

The detour-based routing used in Shoal can lead to congestion at the intermediate end-hosts for certain traffic patterns. To handle that, we devised an efficient congestion control mechanism to run atop Shoal's fabric (Section 3.3.6). Congestion control in Shoal is particularly challenging to achieve due to high multi-pathing—traffic between a pair of end-hosts is routed through all end-hosts. Shoal leverages the observation that the static schedule creates a periodic connection between every pair of end-hosts to implement an efficient backpressure-based congestion control, amenable to efficient hardware implementation.

To verify the feasibility of our design, we implemented an FPGA-based NIC and circuit switch (Section 3.5). Our implementation achieves small reconfiguration delay (6.4 ns) for the circuit switch and is a faithful implementation of our entire design including the scheduling and the congestion control mechanisms.

Further, we incorporated the NIC and the switch implementation into an end-to-end small-scale prototype that comprises six FPGA-based circuit switches forming a two-tier Fattree topology with full bisection bandwidth connecting eight FPGA-based NICs as end-hosts (Figure 1.10, Section 3.7.1). Experiments on this small-scale prototype shows that Shoal offers high throughput and low latency (Section 3.7.2); yet our analysis indicates that its power can be 71% lower than an equivalent packet-switched network at a lower cost (Section 3.6). Using a cross-validated simulator (Section 3.7.3), we show that Shoal's properties hold at scale too. Across datacenter-like workloads, Shoal achieves comparable or higher performance than a packet-switched network using state-of-the-art protocols [187, 41, 2], with improved tail latency (up to 2× lower as compared to NDP [41]) (Section 3.7.5). Further, through simulations based on real traces [34], we also demonstrate that Shoal can cater to the demands of emerging disaggregated workloads (Section 3.7.6).

3.2 Background

We first consider how conventional datacenter switching fabric designs fall short when it comes to building high-performance high-speed switching fabrics in a cost and power efficient manner.

3.2.1 Strawman Design 1: Packet Switching

A Folded Clos topology of packet switches, such as Fattree [1], is the most common way of building datacenter switching fabrics. The structure of a Fattree can be characterized by the number of tiers of switch chips that it requires, and how the chips are packaged into boxes. These design choices then dictate the number of hops a packet must traverse, as well as the number of fiber links and optical transceivers required to connect the fabric. Each additional tier incurs cost, power, latency, and cabling complexity. However, given that packet switching speeds are no longer able to keep pace with the rising bandwidth demand due to slowdown in Moore's law, scaling out a Fattree topology with more tiers to meet the necessary bandwidth demand is inevitable, resulting in high cost and power overhead (Section 1.3.2).

The disadvantages of traditional Fattree designs have led to chassis-based Fattree topologies in which multiple switch chips are integrated into a common box, known as a chassis, and connected using energy-efficient copper backplane traces. As a result, this architecture requires fewer optical transceivers and fibers compared to traditional Fattree topology. However, this comes at the cost of more switching chips and the number of hops (higher latency) through the network. Further, chassis power consumption also presents a scaling challenge as network speeds increase.

3.2.2 Strawman Design 2: Direct-connect Network

Motivated by the observation that building a high-speed switching fabric in the Post-Moore era using packet switches would result in high power and cost overheads, practitioners have attempted to integrate several very low-port (typically four or six ports) packet switches in the System-on-Chip (SoC) of the microserver. Thus, instead of building a topology of packet switches, the microservers can be connected to each other using direct-connect topologies prevalent in High Performance Computing (HPC) and Super-computing systems, e.g., a 3D torus [157, 82]. This design significantly reduces the overall network power consumption as the additional logic per SoC is small. However, a key drawback of direct-connect networks is that they have a static topology which cannot be reconfigured based on current traffic pattern. Hence their performance is workload dependent—for dynamically changing workloads such as datacenter workloads, it results in routing traffic across several end-hosts, which hurts network throughput and latency (Section 3.7.5) and complicates routing and congestion control [25].

3.2.3 Circuit Switching

These strawmans lead to the question whether packet-switched networks are well-suited to build high-speed switching fabrics in the Post-Moore era. On the upside, packet-switched networks offer excellent performance and allow the network core to be loosely coupled with the end-host's network stack. Traditionally, this has been a good trade-off—as long as packet switching speeds were able to scale to the bandwidth demands, the cost and power overheads were not a concern yet loose coupling allowed the core network technologies to evolve independent of the end-host's network stack. This also allowed the network to be asynchronous, which helps scaling. This trade-off, however, does not hold up now when switching speeds of packet switches are unable to keep pace with the rising bandwidth demand due to the slowdown in Moore's law and the end of Dennard scaling.

Instead, we argue that a circuit-switched network offers a different set of trade-offs that are more suited to building high-speed switching fabrics in a cost and power efficient manner. Compared to a packet switch, circuit switches can draw less power and be cheaper due to their simplicity, and these gains could grow with future optical switches (Section 3.6). Thus, they can scale to high switching speeds in a highly cost and power efficient manner. On the flip side, circuit switching does necessitate a tight coupling where all nodes are synchronized and traffic is explicitly scheduled. Further, past solutions with slow circuit switches have had to rely on a separate packet-switched network to support low latency workloads which increases complexity and hurts network manageability (Section 1.5.2). Using fast circuit switches has the potential to support both low latency and bulk traffic effectively, but requires a fast control plane for circuit scheduling, which is challenging. In this chapter, we show that these challenges can be solved at the scale of a datacenter and it is feasible to build a high-speed datacenter switching fabric that is cost and power efficient while achieving performance on par with a packet-switched network.

3.3 Design

Shoal is a circuit-switched network comprising a switching fabric built entirely using fast reconfigurable circuit switches that is tightly coupled with the end-host's network stack. In this section, we describe the design of Shoal.

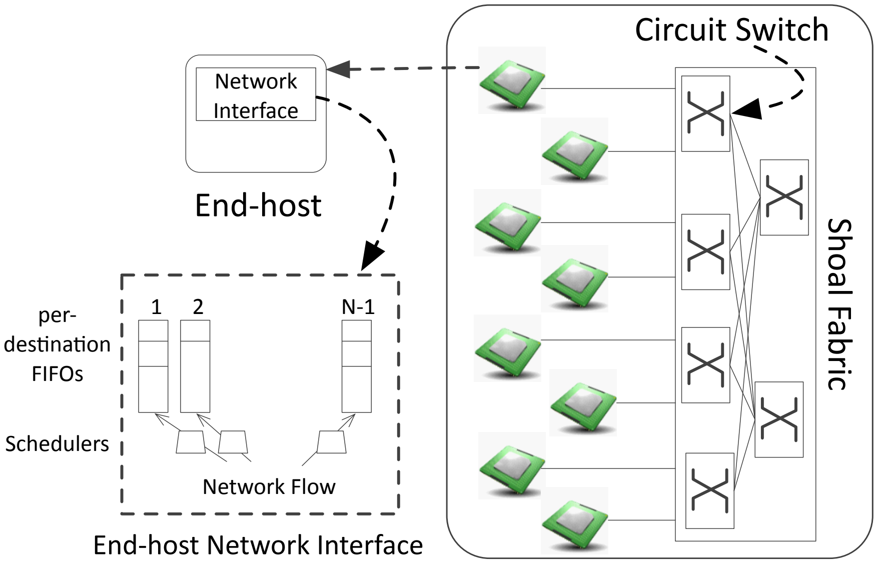


Figure 3.1: Shoal architecture.

3.3.1 Design Overview

Shoal's architecture is shown in Figure 3.1. Each end-host in the network is equipped with a network interface connecting it to the Shoal fabric. The fabric comprises a hierarchical collection of smaller circuit switches, electrical or optical, that are reconfigured synchronously. Hence, the fabric operates like a single, giant circuit switch (Section 3.3.2). The use of a circuit-switched fabric means that we need a control plane to schedule it. One possible approach is to schedule it *on-demand*, i.e., connect end-hosts depending on the current traffic matrix. However, such on-demand scheduling requires complicated scheduling algorithms and demand estimation, and would make it hard to meet low-latency constraints.

Instead, Shoal uses *traffic agnostic* scheduling [18]. Specifically, each circuit switch forwards fixed-sized packets or "*cells*" between its ports based on a predefined "schedule". These per-switch schedules, when taken together, yield a schedule for the fabric which dictates when different end-host pairs are connected to each other. The schedule for individual switches is chosen such that the fabric's schedule provides equal rate connectivity between each pair of end-hosts (Section 3.3.4). To accommodate any traffic pattern atop the equal rate connectivity offered by the fabric, each end-host spreads its traffic uniformly across all other end-hosts, which then forward it to the destination (Section 3.3.5).

The second mechanism implemented in Shoal's network stack is a congestion control technique that ensures that network flows converge to their fair rates, while bounding the maximum queuing at all end-hosts (Section 3.3.6). Our main insight here is that the periodic connection of end-hosts by the fabric enables a simple backpressure-based congestion control amenable to hardware implementation. One of the main challenges in implementing backpressure-based mechanisms over multi-hop networks is instability for dynamic traffic [50]. In Shoal, we restrict the backpressure mechanism to a *single hop*, avoiding the instability issue altogether.

In the following sections, we describe the aforementioned mechanisms in more detail.

3.3.2 Shoal Switching Fabric

Shoal's switching fabric connects the end-hosts through what looks like a single circuit. This can be achieved either using a single high port-count circuit switch connecting all the end-hosts or using multiple low port-count circuit switches forming a non-blocking Clos topology, such as Fattree [1] with full bisection bandwidth (Figure 3.2). All the switches in the fabric reconfigure their circuits

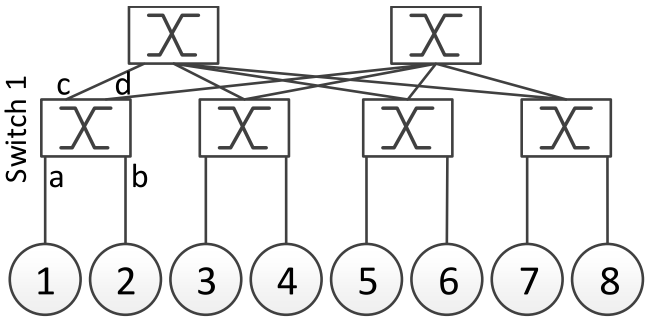


Figure 3.2: A non-blocking Clos topology of circuit switches. This particular topology is also known as two-tier full bisection bandwidth Fattree topology or a leaf-spine topology.

synchronously (according to the logic described in Section 3.3.4), thus ensuring that the entire fabric operates as a single circuit switch. The fabric periodically connects each pair of end-hosts according to a static, pre-defined schedule (Section 3.3.4). End-hosts currently connected by the fabric exchange packets (Section 3.3.3), which are always routed through the highest tier of the topology, even if the highest common ancestor switch of the two end-hosts is in a lower tier (like end-hosts 1 and 2 in the figure). Since the topology is non-blocking, this does not impact network throughput. It ensures, however, that the distance between any two end-hosts is the same which, in turn, aids network-wide time synchronization (Section 3.3.7).

3.3.3 Shoal Data Plane

Shoal's data plane is illustrated in Figure 3.3. Shoal's mechanisms operate at the data link layer (layer 2) of the network stack. End-hosts send and receive fixed-sized cells. Packets received from higher layers are thus fragmented into cells at the source end-host and reassembled at the destination. Each cell has a header (Section 3.3.7) that contains the routing and other control information.

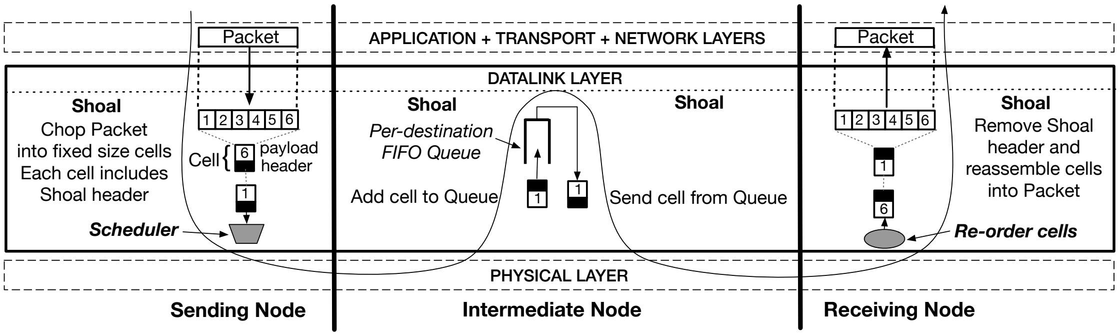


Figure 3.3: Shoal data plane.

Cells in Shoal are routed to their final destination via *at most* one intermediate end-host (Section 3.3.5). Thus each end-host in Shoal has dual responsibility—(i) transmit its local cells to an intermediate end-host (on the way to the final destination), and (ii) forward detouring remote cells received from other end-hosts to their final destinations.

To that end, each end-host implements a *scheduler* which is configured by Shoal's congestion control algorithm (Section 3.3.6). The scheduler decides *when* and *which* local cells to forward to an intermediate end-host. Further, each end-host also maintains a set of FIFO queues, one per destination, to store the remote cells—cells arriving at an intermediate end-host are are put into the FIFO queue corresponding to their final destination. These per-destination FIFO queues are served whenever the fabric's schedule connects the intermediate end-host to the corresponding destination.

Finally, when the fabric's schedule connects end-host i to end-host j, the former *always* transmits a cell; the cell at the head of the queue $i \rightarrow j$ is transmitted, otherwise an empty cell is sent. This ensures that each end-host periodically receives a cell from every other end-host, which enables implementing an efficient backpressure-based congestion control (Section 3.3.6) and simple failure detection (Section 3.4.4).

End-host \ Time slot	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	1
3	4	5	6	7	8	1	2
4	5	6	7	8	1	2	3
5	6	7	8	1	2	3	4
6	7	8	1	2	3	4	5
7	8	1	2	3	4	5	6
8	1	2	3	4	5	6	7

Figure 3.4: Fabric schedule for a network with 8 end-hosts (see Figure 3.2 for the physical topology).

Port \ Time slot	1	2	3	4	5	6	7
a	d	c	d	c	d	c	d
b	c	d	c	d	c	d	c
c	a	b	a	b	a	b	a
d	b	a	b	a	b	a	b

Figure 3.5: Switch 1's schedule (see Figure 3.2 for the physical topology).

3.3.4 Shoal Control Plane: Static Schedule & Virtual Topology

Shoal uses a static, pre-defined schedule to reconfigure the fabric such that the end-hosts are connected at an equal rate. Figure 3.4 shows an example schedule with $N = 8$ end-hosts. Thus, in a network with N end-hosts, each pair of end-hosts is directly connected by the fabric once every $N - 1$ time slots, where a slot refers to the cell transmission time.

Further, we decompose the schedule of the overall fabric into the schedule for each constituent circuit switch. Consider the example fabric shown in Figure 3.2. Figure 3.4 shows the schedule for this fabric while Figure 3.5 shows the schedule for switch 1. Each switch's schedule is contention-free, i.e., at a given instant, any port is connected to only one port. This allows the switch to

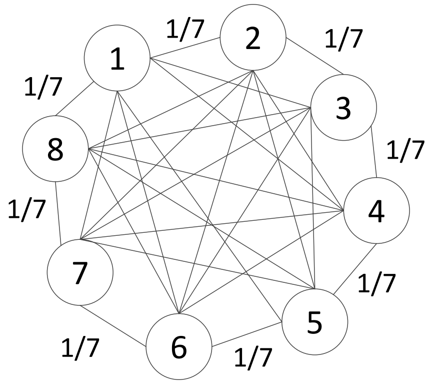


Figure 3.6: Shoal's virtual full mesh topology with 8 end-hosts (see Figure 3.2 for the physical topology).

do away with any buffers and any mechanisms for packet inspection or packet arbitration. Further, each switch's schedule guarantees that if every switch reconfigures synchronously, we would get the same end-to-end connections as dictated by the overall fabric schedule. This way Shoal's fabric operates as a single, giant circuit switch.

Next, the periodic reconfiguration of Shoal's fabric means that the entire network can be seen as a *virtual full mesh* topology with virtual links between each pair of end-hosts. E.g., consider a network with 8 end-hosts whose schedule is shown in Figure 3.4. Since each end-host is connected to every other end-host $1/7^{th}$ of the time, the network provides the illusion of a full mesh with virtual links whose capacity is $1/7^{th}$ of each end-host's total network bandwidth. Figure 3.6 shows this virtual topology achieved by the fabric.

3.3.5 Shoal Control Plane: Routing

To route cells on top of the virtual full mesh topology, Shoal leverages the well known Valiant Load Balancing (VLB) [171], by routing each cell via a random intermediate end-host—cells sourced by an end-host, irrespective of their destination, are sent to the next end-host the source is connected to by the fabric's schedule. The intermediate end-host then forwards the cell to the final destination. Thus, each cell is detoured through at most one intermediate end-host.

The key aspect of this design is that it converts any arbitrary traffic pattern into a uniform traffic pattern. This guarantees the worst-case throughput across *any* traffic pattern [18]—Shoal's network throughput can be at most 2× worse than that achieved by a hypothetical, network-wide ideal packet switch. To compensate for this throughput reduction due to detouring, we double the aggregate bisection bandwidth of the fabric for Shoal. This is a good trade-off as circuit switches are expected to be cheaper and hence, adding fabric bandwidth is inexpensive; in Section 3.6, the cost of the resulting network is still estimated to be lower than the cost of a traditional packet-switched network.

3.3.6 Shoal Control Plane: Congestion Control

The detouring mechanism described in Section 3.3.5 can lead to congestion at the intermediate end-host when there are multiple sources sending data to the same destination, also known as the *incast* traffic pattern. To handle this, Shoal relies on a congestion control mechanism. We begin with a discussion of the implications of Shoal's virtual topology and routing for congestion control, followed by the details of our design.

High Multi-pathing. As described in Section 3.3.4, the periodic reconfiguration of Shoal's fabric means that the entire network can be seen as a virtual full mesh topology with virtual links between each pair of end-hosts.

Shoal's use of detouring means that each end-host's traffic is routed uniformly through all the end-hosts on their way to their destination, resulting in very high multi-pathing. In contrast, the Transmission Control Protocol (TCP) suite of protocols, including protocols tailored for datacenters [2, 178] and recent protocols for Remote Direct Memory Access (RDMA) networks [187, 77] only use a single path. Even multi-path extensions like MPTCP [89] target scenarios with tens of paths, which is orders of magnitude less than in Shoal.

Design insights. Shoal's congestion control design is based on three key insights. First, we leverage the fact that the fabric in a network with N end-hosts directly connects each pair of end-hosts once every $N-1$ time slots. We refer to this interval as an *epoch*. This means that, when the queues at an intermediate end-hosts grow, it can send a timely backpressure signal to the sender. As we detail below, the periodic nature of this signal coupled with careful design of how a sender reacts to it allows us to bound the queue size across all end-hosts.

Second, achieving per-flow fairness with backpressure mechanisms is challenging [187], especially in multi-path scenarios. In Shoal, a *flow* refers to all layer 2 packets being exchanged between a pair of end-hosts. For network traffic, this includes all transport connections between the end-hosts. For storage traffic, this includes all I/O between them. Each flow comprises $N-1$ *subflows*, one corresponding to each intermediate end-host. Shoal achieves fairness across flows by leveraging the fact that each flow comprises an equal number of subflows that are routed uniformly across a symmetric network topology, so we can achieve per-flow fairness by ensuring per-subflow fairness. We thus treat each subflow independently and aim to determine their fair sending rates. The mechanism can also be extended to other flow sharing policies (Section 3.8.2).

Finally, each subflow traverses two virtual links, either of which can be the bottleneck. E.g., a subflow $i \to j \to k$ can either be bottlenecked at the virtual link between end-hosts i and j, or between end-hosts j and k. Shoal maintains a queue, Q_{ij}, at end-host i to store cells destined to end-host j. We use the length of the queue Q_{ij} as an indication of the load on the virtual link between end-hosts i and j. Note that the end-host sourcing the traffic, i, can observe the size of the local queue Q_{ij}. It, however, also needs to obtain information about the size of the remote queue Q_{jk} that resides at end-host j. This is achieved using the congestion control mechanism described below.

Congestion control mechanism. We use a subflow from source i to destination k through intermediate end-host j, $i \to j \to k$, as a running example to explain Shoal's congestion control. When end-host i sends a cell to end-host j, it records the subflow that the cell belongs to. Similarly, when end-host j receives the cell, it records the index k of the queue that the cell is added to. The next time end-host j is connected to end-host i, it embeds the current length of queue Q_{jk} into the cell header:

$$rate\ limit\ feedback_{ji} = len(Q_{jk}) \tag{3.1}$$

Each pair of end-hosts in the network exchange a cell every epoch, even if there is no actual traffic to be sent. Thus, when end-host i sends a cell to end-host j, it gets feedback regarding the relevant queue at j within the next epoch. Let us assume that end-host i receives this feedback at time T. At time t ($\geq T$), it knows the instantaneous length of its local queue to end-host j, $Q_{ij}(t)$, and a sample of the length of the remote queue between end-hosts j and k, $Q_{jk}(T)$. The fair sending rate for a subflow is governed by the most bottlenecked link on its path, i.e., the link with the maximum queuing. As a result, the next cell for this subflow should only be sent after both the queues have had time to drain, i.e., at least, $max(len(Q_{ij}(T)), len(Q_{jk}(T)))$ epochs have passed since the feedback was received. To achieve this, end-host i releases a cell for this subflow into its local queue for j only when the current length of the queue, after accounting for the time since the last feedback, exceeds the size of the remote queue Q_{jk}, i.e., a cell is released into Q_{ij} at time t when,

$$len(Q_{ij}(t)) + (t - T) \geq len(Q_{jk}(T)) \tag{3.2}$$

Thus, when a new cell is released into the queue at its source, the previous cell in that queue is guaranteed to have been sent to the remote queue while the previous cell in the remote queue is guaranteed to have been sent to the destination. This ensures the *invariant* that at any given time a subflow has at most one cell each in both the queue at its source and the queue at its intermediate end-host. As a consequence, at any given time, the size of each queue Q_{ij} is bounded by:

$$len(Q_{ij}) \leq outcast\ degree(i) + incast\ degree(j) \tag{3.3}$$

Here, *outcast degree*(i) equals the number of end-hosts i is sourcing traffic to, and *incast degree*(j) equals the number of end-hosts sourcing traffic destined to j. Thus, this mechanism ensures that, for each virtual link, Shoal performs *fair queuing* at cell granularity across all the subflows sharing that link. This, in turn, results in a tighter distribution of flow completion times. Note that while Shoal's basic design assumes a single traffic class for the flows, it can be easily extended to support multiple traffic classes (Section 3.8.2).

While Equation 3.3 bounds the queue size, it also highlights one of the challenges of detouring: network latency experienced by a cell, while bounded, is impacted by cross-traffic — traffic from remote end-hosts at the cell's source end-host and traffic from local end-host at the cell's intermediate end-host. To reduce this impact of detouring, we introduce following optimizations:

Reducing cell latency at the intermediate end-host. In addition to queue Q_{jk}, end-host j also maintains a ready queue R_{jk}. Instead of adding cells to Q_{jk} from local flows that satisfy Equation 3.2, Shoal adds the corresponding flow ids into the ready queue R_{jk}. Thus,

$$len(R_{jk}) \leq outcast\ degree(j) \qquad \leq N - 1 \tag{3.4}$$

Shoal then scans the local flow ids in R_{jk}, and adds the corresponding cells into the queue Q_{jk} such that at any given time there is at most one local cell in Q_{jk}. Thus Equation 3.3 changes to:

$$len(Q_{jk}) \leq 1 + incast\ degree(k) \qquad \leq N \tag{3.5}$$

However, to ensure that the rate limit feedback accounts for the local subflows, Equation 3.1 needs to be updated accordingly:

$$rate\ limit\ feedback_{ji} = len(Q_{jk}) + len(R_{jk}) - 1 \tag{3.6}$$

The network is thus still shared fairly, while simultaneously reducing the impact of local traffic on the latency of remote cells — the latency experienced by a remote cell at any intermediate end-host is determined only by the incast degree of cell's destination.

Reducing cell latency at the source end-host. While Equation 3.5 reduces the impact of detouring at the intermediate end-host, at the source end-host, *i*, the latency for a local cell in Q_{ij} is governed by incast degree of intermediate end-host *j*. To reduce the impact of cross traffic (i.e., non-local traffic), Shoal selectively adds cells from a new flow to queue Q_{ij} only if $len(Q_{ij}) \leq 2^{\text{age}}$, where age is measured in epochs since the flow started. Thus, for the first few epochs, cells will be released to queues over virtual links with low contention, and afterwards will quickly converge to uniform load balancing using all virtual links after a max of $log(N)$ epochs. This achieves uniform load balancing for long flows, and hence preserves Shoal's throughput bounds, while reducing completion time for short flows.

The impact of these optimizations is evaluated in Figure 3.15.

Bounded queuing. Equation 3.5 guarantees that at any given time, the size of each queue Q_{ij} at end-host i is bounded by the instantaneous number of flows destined to destination$j$ plus one, with at most one cell per flow. This queue bound can be used to determine the maximum buffering needed at each end-host's network interface to accommodate even the worst-case traffic pattern of all-to-one incast. Importantly, since Shoal accesses a queue only once every epoch for transmission, and assuming the access latency of off-chip memory is less than an epoch, Shoal only needs to buffer one cell from each queue Q_{ij} on the on-chip memory, resulting in $N - 1$ total cells.

3.3.7 Shoal Control Plane: Configuration Parameters

In this section, we describe the various configuration parameters in Shoal's design. Shoal operates in a time-slotted fashion. Slots are separated by a "guard band" during which the switches are reconfigured. The guard band also accounts for any errors in synchronization. Further, each cell carries a header that encodes both the routing and congestion control information.

Cell header. Each cell in Shoal carries a header that has eight attributes:

- `source id`: the id of the end-host sourcing the cell.
- `destination id`: the id the end-host the cell is destined to.

- `sequence number`: a unique (increasing) number assigned to each cell, used primarily for ordering the cells in the correct sequence in the event of re-ordering in the network.
- `rate limit feedback`: encodes the rate limit feedback, as described in Section 3.3.6, used for Shoal congestion control.
- `start-of-packet`: a binary attribute which is set to 1 if the cell happens to be the first cell in a layer 2 packet, otherwise set to 0.
- `end-of-packet`: a binary attribute which is set to 1 if the cell happens to be the last cell in a layer 2 packet, otherwise set to 0.
- `last-cell-dropped`: a binary attribute which is set to 1 in a cell going from end-host i to end-host j, if the last cell from end-host j to end-host i was dropped at i, otherwise set to 0.
- `Checksum`: a checksum on the header fields, used to detect and recover from cell corruption due to bit flips.

Circuit switch reconfiguration. Shoal uses fast reconfigurable circuit switches. For example, our prototype implements an FPGA-based circuit switch that can be reconfigured in 6.4 ns (Section 3.5.1). Electrical circuit switches with fast reconfiguration are also commercially available [167] while fast optical circuit switches with nanosecond-reconfiguration time have also been demonstrated [55, 29, 183, 63, 22, 27, 106].

Time synchronization. Shoal's slotted operation requires that all end-hosts and switches are time synchronized, i.e., they agree on when a slot begins and ends. Synchronizing wide-area networks is hard, primarily because of high propagation delay and the variability in it. In contrast, fine-grained datacenter-wide synchronization is tractable due to their size and ability to control every node in the network. Protocols like Datacenter Time Protocol (DTP) [58] have shown to achieve bounded nanosecond-level synchronization precision across an entire datacenter. Furthermore, datacenter networks can be constructed with tight tolerances to aid synchronization. For example, if all links are the same length with a tolerance of ± 2 cm, the propagation delay would vary by a maximum of 0.2 ns. Small physical distance also mitigates the impact of temperature variations that could lead to variable propagation delay.

Shoal leverages the WhiteRabbit synchronization technique [80, 57, 65, 90] to achieve synchronization with bit-level precision. WhiteRabbit has been shown to achieve sub-50 picoseconds of synchronization precision [90]. The main idea is to couple frequency synchronization with a time synchronization protocol such as Precision Time Protocol (PTP) [108] or Datacenter Time Protocol (DTP) [58] (Section 3.7.1).

Frequency synchronization is achieved by distributing a global clock to all the nodes (end-hosts and switches) in the network. This global clock is generally derived from one of the nodes, designated as the clock master. The clock can be distributed explicitly, or implicitly through Synchronized Ethernet (SyncE) [166] whereby nodes derive a clock from the data they receive and use this clock for their transmissions.

In Shoal, time synchronization protocol needs to run only between the end-hosts (and not the switches). At bootstrap, each switch's circuits are configured according to their respective schedule's configuration at time slot 1 (e.g., Figure 3.5) and they do not change. End-hosts then start running the time synchronization protocol. Once all the end-hosts are synchronized to a desired level of precision, they send a "start" signal to the switches, followed by actual data according to the fabric schedule (Figure 3.4). Switches on receiving the signal start reconfiguring their circuits according to their respective schedules (Figure 3.5).

Slot size and guard band. Overall, the guard band size is the sum of the reconfiguration delay, variability in propagation and the precision of synchronization. Given the guard band size, the slot size can be configured to balance the trade-off between latency and throughput: a smaller slot reduces epoch size resulting in smaller latency, yet it imposes higher guard band overhead resulting in smaller duty cycle and hence lower throughput.

Epoch size and multiple channels. In Shoal, two end-hosts exchange cells at the interval of an epoch. Therefore, each queue drains at the rate of one cell per epoch, meaning a smaller epoch size results in smaller queuing delay. We can reduce the epoch size by taking advantage of the fact that network links comprise multiple I/O channels. E.g., 100 Gbps links actually comprise four 25 Gbps channels, which can be switched independently. Thus, in Shoal, each channel is used to send cells to a quarter of the end-hosts in parallel. Given a fixed slot size (as determined based on guard band size), this shrinks the epoch size by a quarter ($epoch = \frac{N-1\ slots}{num\ of\ channels}$). Finally, the actual cell size is determined by the slot size and channel speed, e.g., a slot size of 20.5 ns (without guard band) will correspond to 64 B cells over a 25 Gbps channel.

3.4 Handling Practical Concerns

In this section, we discuss some of the practical concerns around Shoal's design and describe how we handle those concerns.

3.4.1 Clock and Data Recovery (CDR)

A key challenge for any network relying on fast circuit switches is that each switch needs to be able to receive traffic from different senders at each time slot. This requires that, at each time slot, the incoming bits are sampled appropriately so as to achieve error-free reception. The sampling is done by the Clock and Data Recovery (CDR) circuitry at the receiver. However, we note that this is a problem only when using layer 0 circuit switches that operate at the raw physical layer, e.g., when using an optical circuit switch. Such a switch imposes no latency overhead but requires very fast CDR at the receiver in order to achieve a reasonable guard band. Recent work has shown that sub-nanosecond CDR is achievable in datacenter settings [24].

Electrical circuit switch can also operate at layer 1 [167]. When a circuit is established between ports $i \rightarrow j$, the switch retimes data received on port i before sending it to port j. With such switches, each link in the network is a point-to-point link and thus, fast CDR is not needed. Each switch, however, does need to be equipped with a small buffer to account for any differences in the clocks associated with ports i and j. For Shoal, only a few bits worth of buffering is required since the entire network is frequency synchronized and the buffer is only needed to absorb any clock jitter.

3.4.2 Accounting for Propagation Delay

The propagation delay within a datacenter is not negligible as compared to the transmission time of a cell. This means that a cell sent at time slot t will not be received within the same slot at the receiver. More generally, say that the cell

	Time slot					
End-host	1	2	3	4	5	6
1	2	3	4	7	6	5
2	3	4	5	1	7	6
3	4	5	6	2	1	7
4	5	6	7	3	2	1
5	6	7	1	4	3	2
6	7	1	2	5	4	3
7	1	2	3	6	5	4

(a) Fabric schedule for odd *N*

	Time slot							
End-host	1	2	3	4	5	6	7	8
1	2	3	4	5	8	7	6	
2	3	4	5	6	1	8	7	
3	4	5	6	7	2	1	8	
4	5	6	7	8	3	2	1	
5	6	7	8		4	3	2	1
6	7	8	1		5	4	3	2
7	8	1	2		6	5	4	3
8	1	2	3		7	6	5	4

(b) Fabric schedule for even *N*

Figure 3.7: Fabric schedule for a network with $N = 7$ end-hosts (odd N) and $N = 8$ end-hosts (even N) after accounting for propagation delay. For the original fabric schedule with 8 end-hosts refer to Figure 3.4.

is received at slot $t + k$. For the feedback mechanism described in Section 3.3.6 to work optimally in the face of propagation delay, there should be at least k slots from the time end-host i transmits to end-host j and the time end-host j transmits to i, as j needs to know the destination of the last cell that i sent to j to send back the right rate limit feedback.

Fortunately, we can easily re-arrange any cyclic permutation schedule, such as in Figure 3.4, to ensure this property, as long as k is less than half the number of slots in an epoch—when number of end-hosts, N, is odd, this constraint can be easily accommodated by inverting the order of the last $\frac{N-1}{2}$ columns in the schedule. This would ensure that there are exactly $\frac{N-1}{2}$ slots between the slot in which i communicates with j and the one in which j communicates with i. This is shown in Figure 3.7(a). For even N, this is slightly more complicated as it requires to introduce an additional empty slot per end-host to satisfy this requirement as demonstrated in Figure 3.7(b), consequently resulting in a throughput reduction by a factor of $\frac{1}{N}$ for each end-host, which is negligible for any practical value of N.

3.4.3 Cell Re-ordering and Re-assembly

The `sequence number` field in each cell's header is used to assemble cells in-order at the destination. Note that Shoal's congestion control is resilient to re-ordering, as it operates at the granularity of individual subflows with a congestion window of size 1. Once the cells have been reordered, the `start-of-packet` and `end-of-packet` fields in the cell header are used to figure out the packet boundary, and the cells within each packet boundary are then assembled together to re-construct the original packet.

3.4.4 Failures

In this section, we discuss how Shoal deals with failures. In particular, we focus on two kinds of failures—(i) cell corruption, and (ii) component failures.

Cell corruption. Shoal uses the `Checksum` field in the cell header to check for cell corruption due to bit flips. If end-host j receives a corrupted cell from end-host i, it first extracts the header fields corresponding to congestion control, namely the `rate limit feedback` and `last-cell-dropped`, and then discards the cell. It also signals the sender i that the cell was dropped by setting the `last-cell-dropped` field to 1 in the header of the next cell sent to end-host i. End-host i then re-transmits the last cell it sent to end-host j. Since the `last-cell-dropped` field in the corrupted cell might also have been corrupted, end-host j takes the conservative approach of assuming the field was set to 1 and re-transmits the last cell it sent to end-host i. Further, if the `rate limit feedback` field also happens to be corrupted, the queue bound as described in Section 3.3.6 might get violated and there could be tail drops in the worst case. Shoal again uses the `last-cell-dropped` field in the cell header to notify the sender of any tail drop.

Component failures. To detect component failures, such as a switch or an end-host or a link failure, Shoal relies on the fact that an end-host sends a cell to every other end-host in the network, even if there is no traffic to send, once every epoch. A path refers to the set of links and switches through which an end-host *j* sends a cell to some other end-host *i* once every epoch.

When an end-host *i* does not receive a cell from an end-host *j* in its corresponding slot, either due to path failures or end-host failure, it conservatively infers that end-host *j* has failed, and i) stops sending any further cells to *j*, ii) notifies other end-hosts that it can no longer communicate with *j*, so other end-hosts stop forwarding cells destined to *j* via *i*, iii) forwards the last cell (if it happens to be *i*'s local cell) it sent to *j* via some other end-host, and iv) discards all the outstanding cells it was supposed to forward to *j*. Shoal relies on a higher layer end-to-end transport protocol to recover from the loss of those outstanding cells. Finally, in case of an actual end-host failure, Shoal again relies on the transport protocol to recover all the cells that were queued to be forwarded at the failed end-host. If the failed node (switch or end-host) was the primary clock reference for synchronization, another node needs to take over and remaining nodes switch to it as the new reference. International Telecommunication Union (ITU) standard for Synchronized Ethernet (SyncE) [166] already supports this.

Note that the failure detection mechanism is symmetric—when end-host *i* infers that end-host *j* has failed, it immediately stops sending cells to *j*, causing *j* to infer that *i* has failed, and hence immediately stop sending cells to *i*. This ensures that Shoal's congestion control (Section 3.3.6) is robust to failures.

One of the consequences of Shoal's design is if an end-host can no longer "directly" communicate with some other end-host, either because the other end-host or the path to it has failed, it hurts Shoal's throughput as the corresponding slot is marked as failed and hence goes unused. We evaluate network performance against fraction of failed end-hosts in Section 3.7.5.

3.5 Hardware Implementation

In this section, we discuss our FPGA-based implementation of Shoal's switch and NIC. Our implementation is written in Bluespec System Verilog [118] (~1,000 LOCs). We synthesize our implementation on a Terasic DE5-Net board [113] comprising an Altera Stratix V FPGA [154] with 234 K Adaptive Logic Modules (ALMs), 52 MBits SRAM, and four 10 Gbps ports. Our design runs at a clock speed of 156.25 MHz, thus each clock cycle is 6.4 ns.

3.5.1 Switch Implementation

Our circuit switch operates at layer 1, i.e., data traversing the switch is routed through the Physical (PHY) layer at the ingress and egress ports (Figure 3.8). The mapping between the ingress and egress ports varies at every time slot according to the static schedule. This mapping is implemented using p different p:1 multiplexers, where p is the number of ports in the switch. The control signals to these multiplexers are driven by p registers, one per multiplexer. In each time slot, all the p registers are configured in parallel according to the schedule. Hence, the switch reconfiguration delay is simply the time it takes to update

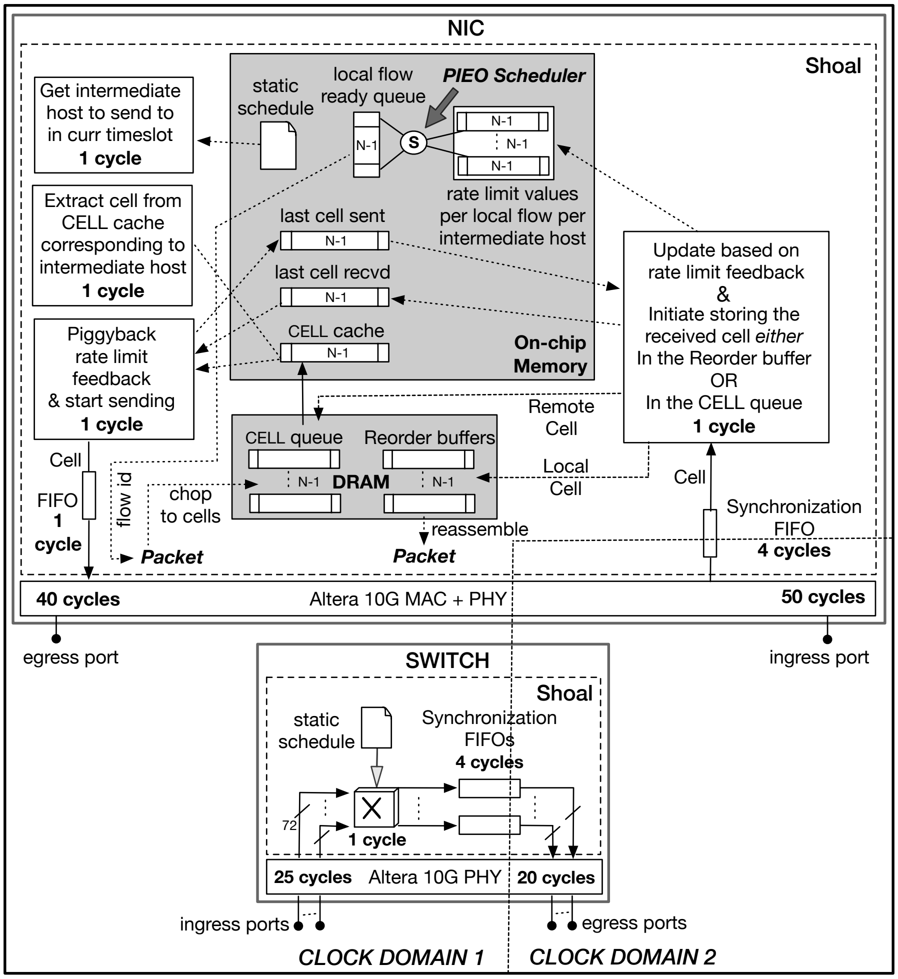


Figure 3.8: Switch and NIC implementation with the latency of each block. Clock cycle is 6.4 ns. N = number of end-hosts. The NIC scheduler is implemented using PIEO (Chapter 2).

the registers, which can be done in one clock cycle. Our switch is driven by the clock that drives the interface to PHY. The interface to 10G Ethernet PHY (XGMII) runs at 156.25 MHz, resulting in reconfiguration delay of 6.4 ns for our switch. However, for higher link speeds the clock frequency can be higher, for e.g., at 50 Gbps, the interface to PHY (LGMII) runs at 390.625 MHz [164], yielding a reconfiguration delay of 2.5 ns.

The transmit and receive paths of the switch are located in two separate clock domains: the transmit path is driven by the clock distributed across all the nodes (switches and NICs) in the network, while the receive path is driven by the clock recovered from the incoming bits. To safely execute the Clock Domain Crossing (CDC), we use synchronization FIFOs.

The total port-to-port latency of our switch is 50 cycles (320 ns). The latency is dominated by the PHY layer (45 cycles), whereas switching (1 cycle) and synchronization FIFO (4 cycles) account for only 5 cycles.

In terms of resource consumption, assuming 64-port switches, our switch implementation consumes 2% of ALMs and 0.2% of SRAM.

3.5.2 Network Interface Card (NIC) Implementation

Figure 3.8 shows the routing and congestion control pipelines implemented in Shoal's NIC. Each NIC maintains a cell cache on the on-chip memory, of size $N-1$ cells, which stores the next cell to forward per intermediate end-host. Remaining cells sit in the DRAM.

The backpressure-based mechanism underpinning Shoal's congestion control (Section 3.3.6) is implemented using two vectors of size $N-1$ each, that record the last cell sent (received) to (from) each intermediate end-host, and a $(N-1) \times (N-1)$ matrix that stores the rate limit feedback received from each intermediate end-host for each active local flow. A *packet scheduler* uses these data structures to schedule local cells into a ready queue in accordance with the logic described in Section 3.3.6. We use PIEO (Chapter 2) to implement the scheduler.

NIC latency is dominated by the Physical (PHY) and Medium Access Control (MAC) layers (in total 40 cycles on the transmit path and 50 on receive),

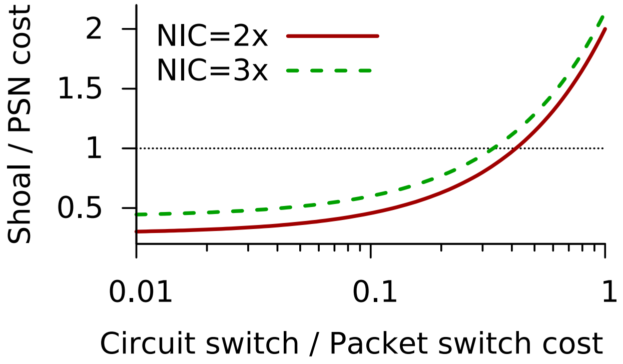


Figure 3.9: Relative cost of Shoal network vs. packet switch network (PSN).

with the routing and congestion control logic only adding 4 and 5 cycles on the transmit and receive paths, respectively. Thus, Shoal's additional mechanisms impose low overhead.

In terms of resource consumption, assuming a 500 end-host network, our NIC implementation consumes 84% of ALMs and 13% of SRAM. We also did a power analysis to quantify the overhead of Shoal's additional NIC functionalities over a commercial NIC. We leverage the study done in [56] to translate the power consumption of our FPGA-based design into an equivalent ASIC design. Assuming a 500 end-host network, this results in Shoal NIC's on-chip extra functionality – routing and congestion control – consuming up to 11% of the power consumed by commercial ASIC NICs.

3.6 Power and Cost Implications

We now compare the power and cost of a Shoal network to that of a packet-switched network (PSN). Along with the performance evaluation in Section 3.7, we demonstrate that for 71% lower power and an estimated cost reduction of up to 40%, Shoal's circuit-switched fabric can reduce tail latency by up to 2× as compared to state-of-the-art protocols such as NDP [41] atop a PSN.

We analyze a network with 512 end-hosts. For a PSN, we consider state-of-the-art packet switches [109, 111], which support 64 ports at 50 Gbps and consume a maximum of 350 W [115]. End-hosts have 50 Gbps NICs with copper cables (i.e., no opto-electronic transceivers) and connecting them using a full-bisection bandwidth Fattree topology requires 24 such switches. For Shoal, extrapolating from state-of-the-art circuit switches [167], we estimate that a 64-port × 50 Gbps circuit switch would consume 38.5 W. To compensate for the throughput overhead of detouring, each end-host is equipped with 100 Gbps links. So the Shoal network has 48 circuit switches. Based on current System-on-Chip (SoC) trends [116], we expect the NIC to be integrated with the CPU on a single SoC and to benefit from the same 10× reduction in power consumption. Given a typical power consumption of 12.4 W for state-of-the-art 100 Gbps NICs [110], this would lead to an estimated power consumption of 1.37 W for Shoal's NIC (including 11% overhead (Section 3.5.2)) and of 0.62 W for PSN's. Thus, total power consumption of the Shoal network is 2.55 KW, 71% lower than PSN (8.72 KW).

Quantifying the cost of the Shoal network is harder as it requires determining the at-volume cost of circuit switches. Circuit switches can be electrical or optical; electrical circuit switches are commercially used in scenarios like HDTV [167] and are capable of fast switching while fast optical switches only exist as research prototypes [55, 29, 183, 63, 27, 106, 97]. Thus, instead of focusing on absolute costs, we ask: *how cheap would circuit switches need to be, relative to equivalent packet switches, for Shoal to offer cost benefits over PSN?* We assume

Shoal NICs cost between 2 and 3× PSN NICs to account for the 2× bandwidth and extra functionality they provide. Figure 3.9 shows how the relative cost of the Shoal network varies as a function of the relative cost of circuit switches to packet switches. A Shoal network would cost the same as a PSN as long as circuit switches are 33.3–41.6% the cost of packet switches while providing power and performance gains. If the cost circuit switches was 9.6–18.3% of the cost of packet switches, then Shoal would offer a 40% cost reduction. While absolute costs are hard to compare as they also depend on several non-technical factors, the analysis below and our estimations based on hardware costs indicate that at-volume circuit switches could cost as low as 15% of equivalent packet switches. We next present a technical analysis to back this estimate.

The lack of buffering, arbitration and packet inspection in circuit switches means that they are fundamentally simpler than packet switches which should mean lower cost. For electrical switches, designers often use the switch package area as a first-order approximation of switch cost—the actual chip area dictates yield during fabrication and therefore fabrication cost while the total package area dictates the assembly and packaging cost. In state-of-the-art packet switches 50% of the total area is attributed to memory, 20% to packet processing logic while 30% is due to serial I/O (SerDes) [146]. Electrical circuit switches, by contrast, have no memory and packet processing, so the first two components have negligible contribution. While the amount of I/O bandwidth in a circuit switch remains the same, the actual SerDes is much smaller because they are only retiming the signals, instead of serializing and deserializing data from a high-rate serial channel to lower-rate parallel channels. Even assuming the SerDes are only halved in size, the total packaged area for circuit switching could be as low as 15% that of a packet switch. This agrees with our earlier estimate of the relative costs. Overall, this analysis indicates that, with volume manufacturing, the relative cost of electrical circuit switches can be low enough for Shoal to simultaneously reduce both power and cost as compared to PSN.

Looking ahead, optical circuit switches hold even more promise: as bandwidth increases beyond 100 Gbps per channel, copper transmission becomes noise limited even at intra-rack distances and optical transmission becomes necessary [112]. Optical circuit switches further reduce cost and power because they obviate the need for expensive transceivers for opto-electronic conversions. However, a few technical challenges need to be solved for optical switches to be used in Shoal [9]. For example, while several technologies being studied in the optics community can achieve nanosecond switching, practical demonstrations have been limited to 64–128 ports [22]. Another longstanding challenge is to achieve fast CDR at layer 0 although recent work has shown the feasibility of such CDR within 625 ps [24].

3.7 Evaluation

In this section, we evaluate Shoal using a small-scale FPGA-based prototype and large-scale simulations.

3.7.1 Prototype

We first evaluate Shoal using a small prototype comprising eight end-hosts. Our prototype comprises eight Terasic DE5-Net boards [113], each with an Altera Stratix V FPGA [154] and four 10 Gbps ports. Two FPGAs are used to implement

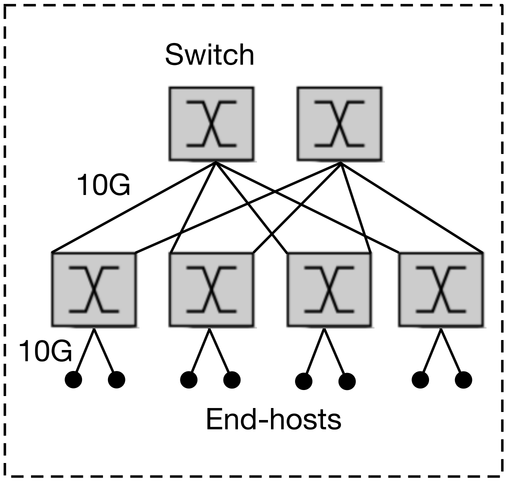


Figure 3.10: Shoal prototype's topology.

eight NICs, one per port. The remaining six FPGAs implement six 4-port circuit switches. The switches are connected in a two-tier Fattree topology with full bisection bandwidth, as shown in Figure 3.10. We connect all eight FPGAs to a PCIe expansion system which is then connected to a Dell T710 server. We distribute the global PCIe clock to the Phase-locked Loop (PLL) circuit running on each FPGA. Thus all the local clocks derived from the respective PLL on each FPGA are frequency synchronized. For time synchronization we use Datacenter Time Protocol (DTP) [100].

Guard band. Our prototype achieves synchronization precision of less than a clock cycle. Further, the switch reconfiguration delay is one clock cycle (Section 3.5.1), and all wires are of same length. Hence a guard band of one clock cycle (6.4 ns) is sufficient.

Slot size. To keep the guard band overhead to around 10%, we select a slot size of 12 clock cycles (76.8 ns). This includes 1 cycle of guard band overhead and 24 B (3 cycles) of Altera MAC overhead. Thus the usable slot size equals 8 cycles (51.2 ns), i.e., 64 B cells at 10 Gbps link speed. The epoch size equals 0.53 us.

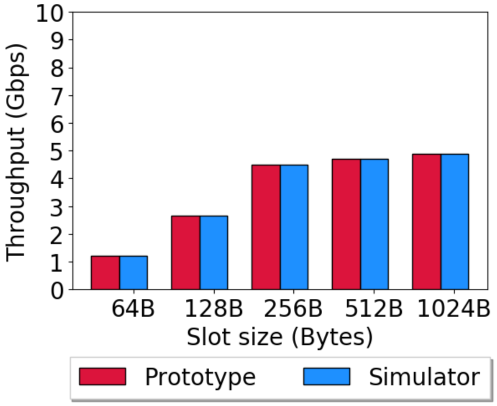


Figure 3.11: [Prototype] Average destination throughput for full permutation matrix.

Next, we used the prototype to verify that our implementation achieves throughput and latency in accordance with the design. We also use it to cross-validate our simulator which, in turn, is used for a large-scale evaluation regarding the viability and benefits of a real world deployment of Shoal.

3.7.2 Prototype Experiments: Throughput and Latency

Throughput. We consider a permutation matrix with $N = 8$ flows: each end-host starts a single long-running flow to another random end-host such that each end-host has exactly one incoming and outgoing flow. For throughput, this is the worst-case traffic matrix. In Figure 3.11, we show performance in terms of *destination throughput*, measured as the amount of "useful" cells (i.e., excluding the cells to forward and the empty ones) received by each destination. For full permutation matrix, the throughput for Shoal is expected to converge to ~50% of the ideal throughput. Interestingly, however, the throughput is significantly lower for smaller slot sizes, and it converges towards 50% only for larger slot sizes. This is an artifact of the small scale of our prototype, which causes the host-to-host cell propagation latency (1.57 us: 40 ns of wire propagation latency + 3×320 ns of switching latency (dominated mostly by Altera 10G PHY latency) for three switches along the path + 576 ns of Altera 10G MAC and PHY latency at the two end end-hosts (Figure 3.8)) to be higher than the epoch size (0.53 us for 64 B cells). The problem is that Shoal's congestion control mechanism prevents an end-host from sending its next cell to an intermediate end-host until it has received feedback from it. Therefore, if the cell propagation latency spans multiple epochs, the overall throughput suffers as senders cannot fully utilize their outgoing bandwidth. As the slot size increases, the ratio between the cell propagation latency and the epoch size decreases, and this explains why in our prototype the throughput improves with larger slots. In practice, however, even for modest-sized networks, this issue will not occur as the cell propagation latency will be much smaller than the epoch size, and can be easily accommodated in the schedule as explained in Section 3.4.2.

Finally, we measured the queue size for a permutation workload with different number of flows. With only one flow in the system, the maximum queue occupancy is one cell. As more flows are added, the queues grow up to a maximum of two cells. This is consistent with the queue length analysis in Section 3.3.6 because each end-host is sourcing at most one flow and it is receiving at most one.

Latency. We consider all-to-one incast, where seven end-hosts each send 448 B of data (seven 64 B cells) to the same destination at the same time. Figure 3.12 shows the distribution of flow completion time (FCT) of all seven flows. The queue at each end-host corresponding to the destination end-host grows up to a maximum of 7 (Equation 3.5). This results in maximum FCT of 6.9 us : 3.76 us

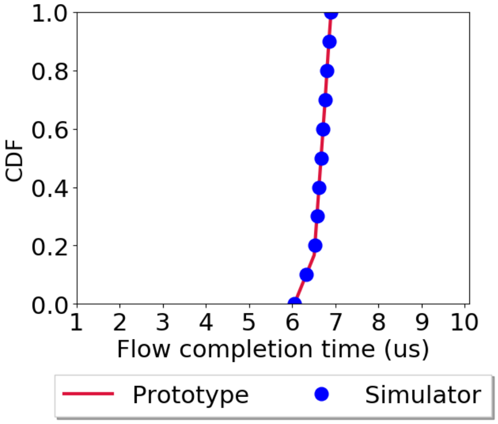


Figure 3.12: [Prototype] Flow completion time for 7:1 synchronized incast.

of queuing delay plus 2×1.57 us of propagation latency. Also note that the difference between the fastest and slowest flow is fairly small (6.05 us vs. 6.9 us), highlighting Shoal's fair queuing.

3.7.3 Simulation

We complement the prototype experiments in Section 3.7.2 with large-scale simulations to investigate the scalability of Shoal.

We use a custom packet-level network simulator written in C that was cross-validated against our prototype. We simulate a network with 512 end-hosts, where each end-host is equipped with an interface bandwidth of 100 Gbps and connected using a full bisection bandwidth Fattree topology of circuit switches.

Guard band. We assume a guard band of 2.75 ns, based on a 2.5 ns switch reconfiguration delay (Section 3.5.1) and 0.25 ns to account for any variability in propagation (0.2 ns) and synchronization imprecision (50 ps), as discussed in Section 3.3.7.

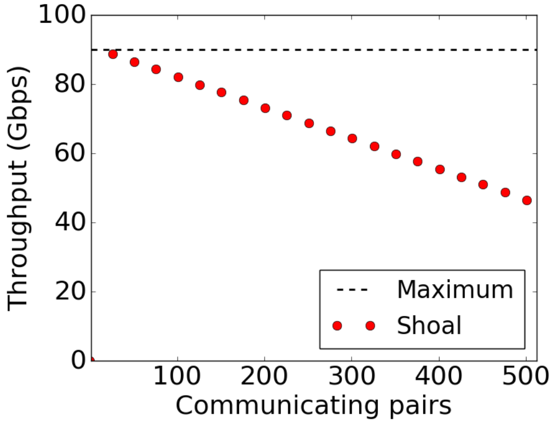


Figure 3.13: Average destination throughput vs. size of permutation matrix.

Slot size. To keep the guard band overhead to around 10% (resulting in max throughput of 90 Gbps), we select the slot size of 23.25 ns. This results in 20.5 ns of usable slot size. As explained in Section 3.3.7, we use the fact that existing 100 Gbps links comprise 4×25 Gbps channels, resulting in 4 parallel uplinks and an epoch size of 2.9 us. Finally, usable slot size of 20.5 ns translates to 64 B cells at 25 Gbps channel speed.

3.7.4 Simulation Experiments: Microbenchmarks

We start with a set of microbenchmarks to verify that the behavior observed in our prototype holds at large scale too. In particular, the microbenchmarks evaluate the throughput, fairness, and latency properties of Shoal.

Throughput. In Figure 3.13, we plot the average destination throughput, as defined in Section 3.7.2, for the permutation traffic matrix: each communicating end-host sends and receives one flow. We vary the number of communicating pairs from 1 to 512. As there is no contention at any of the source and destination end-hosts, the ideal destination throughput equals the maximum

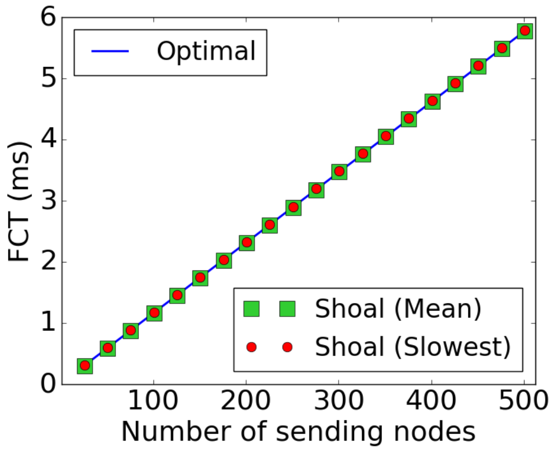


Figure 3.14: Flow completion times against synchronized short flow incast.

interface bandwidth. However, for Shoal, as the number of communicating pairs increases, so does the amount of detouring traffic, resulting in the expected throughput trend: it starts from the peak value for a single flow and then monotonically decreases until it halves when all pairs are communicating (full permutation traffic matrix).

Fairness. To verify Shoal's fairness, we ran several workloads comprising a variable number of flows from 50 to 1,024 with randomly selected sources and destinations. We compared the throughput achieved by each flow against its ideal throughput computed using the max-min water-filling algorithm [13]. Across all workloads, 99% of the flows achieve a throughput within 10% of the ideal one. This shows that, despite the simplicity of its mechanisms, Shoal closely approximates max-min fairness.

Latency under Incast. We evaluate Shoal under incast, the most challenging traffic pattern for low latency. A set of end-hosts send a small flow of size 130 KB each, to the same destination at the same time. In Figure 3.14, we plot the flow completion time (FCT) of the slowest flow as well as the mean completion time,

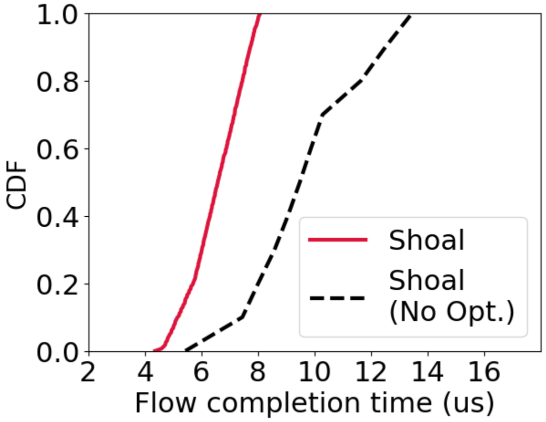


Figure 3.15: Reduced impact of detouring on latency via optimizations in Section 3.3.6.

against increasing number of sending end-hosts. As expected, at each intermediate end-host, the queue corresponding to the destination end-host grows linearly with increasing number of sending end-hosts, but bounded by the incast degree of the destination (Equation 3.5). Hence the FCT for the slowest flow increases linearly and is also the optimal maximum FCT under such incast. The mean completion time coincides with the slowest flow's FCT, thus highlighting Shoal's fair queuing.

Reducing the impact of detouring on network latency. To show that the optimizations described in Section 3.3.6 indeed improve the network latency, we choose two end-hosts, Host-511 (source) and Host-0 (destination), to exchange short flows (20 KB) at regular intervals, and generate random background traffic amongst the remaining end-hosts. We plot the distribution of flow completion time (FCT) of short flows exchanged between end-hosts 511 and 0 in Figure 3.15. The optimizations described in Section 3.3.6 to reduce the cell latency at the source and the intermediate end-host enable Shoal to achieve much smaller and predictable flow completion time for target short flows—at the source, Shoal selectively adds cells to local queues where there is low contention, and at the intermediate end-host the queue length is bounded to two regardless of the cross-traffic, as the incast degree of Host-0 is one (Equation 3.5). However, without the optimizations, the cross-traffic due to detouring significantly increases the flow completion time of target short flows.

3.7.5 Simulation Experiments: Datacenter Workloads

We next investigate the performance of Shoal in dynamic settings, using more realistic datacenter workloads.

Workload. We generate a workload modeled after published datacenter traces [2, 36]. Flow sizes are heavy tailed, drawn from a Pareto distribution with shape parameter 1.05 and mean 100 KB [3, 4]. Flows arrive according to a Poisson process and each simulation ends when one million flows have completed. Flow sources and destinations are chosen with uniform probability across all end-hosts (we will evaluate the performance of Shoal against skewed workloads in Section 3.7.6).

Network load. We define network load $L = \frac{F}{R \cdot N \cdot \tau}$ where F is the mean flow size, R is the per-host bandwidth, N is the number of end-hosts, and τ is the mean inter-arrival flow time, e.g., $L = 1$ means that, on average, there are N active flows in the network.

Evaluation metric. We evaluate Shoal based on the average and 99.9 percentile flow completion time (FCT) for short flows (flow size $\leq$ 100 KB) and average goodput (i.e., throughput after accounting for the cell header overhead) for long flows (flow size $\geq$ 1 MB).

Baseline 1: Direct-connect network. We start with comparing Shoal against a network using a direct-connect topology. We arranged the 512 end-hosts into a 3D torus. As with the Shoal network, we assume an aggregate end-host bandwidth of 100 Gbps. We use R2C2 [25] for congestion control. For all values of load, Shoal consistently outperforms the direct-connect network up to a factor of 14.9 for tail FCT for short flows (respectively a factor of 4.8 for average goodput for long flows). This is due to the multi-hop nature of direct-connect topologies; it significantly increases the end-to-end latency as queuing can occur at any hop. Further, end-host bandwidth is also used to forward traffic originating several hops away, which reduces the overall throughput. This does not occur in Shoal as packets only traverse one hop and the congestion control guarantees bounded queues.

Baseline 2: Packet-switched network. Now we compare Shoal against a packet-switched network (PSN) with 512 end-hosts, that connects the end-hosts using a Fattree topology with full bisection bandwidth. The interface bandwidth of each end-host is 50 Gbps. Thus Shoal is provisioned with 2× bandwidth, to compensate for the throughput overhead of detouring packets. Note that despite the extra bandwidth, Shoal's power is still estimated to be lower than that of PSN with a comparable or lower cost (Section 3.6). As baselines, we use DCTCP [2], NDP [41], and DCQCN [187] as the three state-of-the-art packet-switched network designs. The baselines are based on the simulator used in [41]. We assume 1500 B packets for all of them. DCTCP and DCQCN use standard Equal-cost Multi-path (ECMP) routing, with the congestion window size of 35 packets and queue size of 100 packets. NDP uses packet spraying for routing with initial window size of 35 packets and queue size of 12 packets.

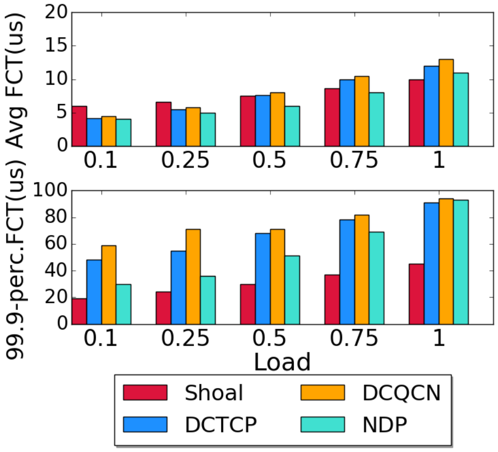


Figure 3.16: Flow completion time (short flows ≤100 KB) vs. traffic load.

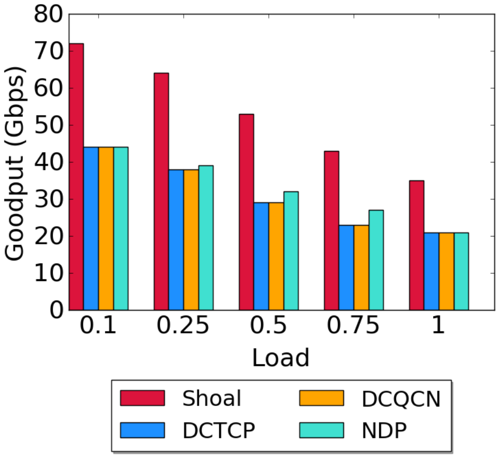


Figure 3.17: Average flow goodput (long flows ≥1 MB) vs. traffic load.

As shown in Figure 3.16, at low to moderate load, Shoal exhibits an average FCT comparable to DCQCN and DCTCP and slightly higher than NDP. This increase is a consequence of the use of detouring due to the static schedule. However, Shoal outperforms DCTCP and DCQCN in terms of tail FCT for short flows by a factor of 3× at low load and 2× at high load (respectively outperforms NDP by a factor of 1.5× and 2×). The reason for this is three-fold: i) 2× bandwidth per end-host in Shoal reduces the serialization delay, ii) selectively adding cells from new flows to local queues with low contention reduces queuing delay at the source, and iii) Shoal's congestion control ensures small and

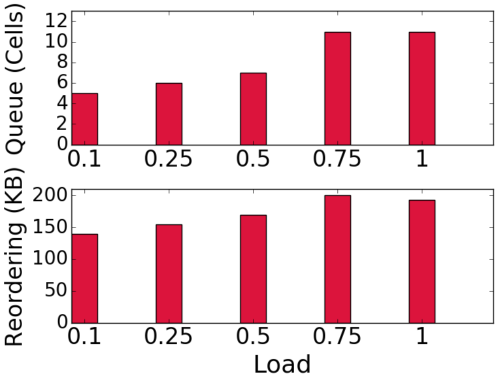


Figure 3.18: Max queue size and max cell re-ordering vs. traffic load.

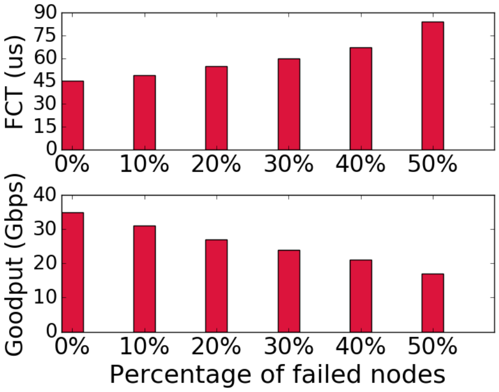


Figure 3.19: Short flow 99.9 percentile FCT and long flow average goodput vs. failures.

bounded queuing at intermediate end-hosts, thus reducing the queuing delay at intermediate end-hosts. Shoal also outperforms all three baselines in terms of long flow goodput (Figure 3.17) by a factor of 1.7×, even at high load. This is primarily because each Shoal end-host is equipped with more bandwidth.

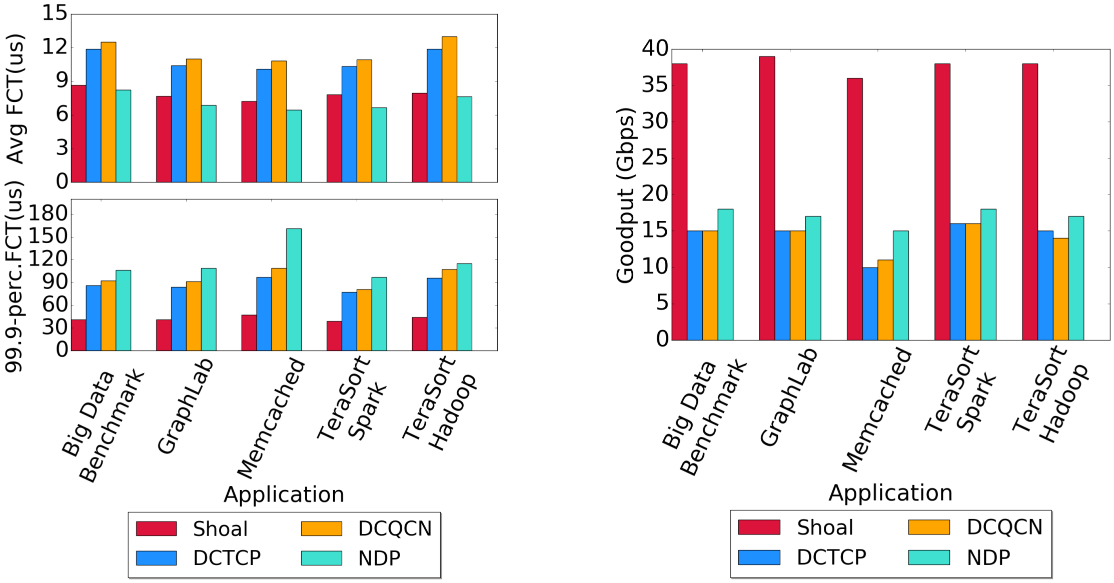
Queuing and re-ordering. To validate our claim that Shoal operates with very small queues, we plot the maximum queuing observed across all end-hosts in Figure 3.18. Even at high load, the maximum queue size is 11 cells (704 B) and the maximum aggregate queue per end-host is 336 cells (21 KB). Maximum cell re-ordering within a flow across all end-hosts and across all values of load is 200 KB (Figure 3.18).

Failures. We now focus our attention to the impact of failures. We ran the same workload as in the previous experiments ($L = 1$) but at the beginning of each experiment we fail an increasing fraction of end-hosts (up to 50%). As expected, the goodput decreases linearly (2× worse for 50% failure rate, Figure 3.19) because the slots corresponding to the failed end-hosts are wasted. We can alleviate this with a more sophisticated mechanism that, on detecting long-term failures, updates the schedule of both end-hosts and switches to discount the failed end-hosts. FCT also increases with increasing failed end-hosts, as the number of paths along which cells from a flow can be sent is reduced, resulting in higher subflow collision and increased queuing. However, Figure 3.19 shows that even for high failure rate the increase in completion time is rather marginal, e.g., 1.5× for a 40% failure rate (respectively 1.2× for 20% failure rate).

Impact of epoch size on network latency. Larger epoch size results in higher latency (Section 3.3.7). In the first experiment, we reduced the number of channels from 4×25 Gbps to 2×50 Gbps, thus doubling the epoch size. This increased tail FCT by 1.26× at low load (respectively 1.15× at high load). In the second experiment, we kept the number of channels constant at 4, and increased the number of end-hosts to 1,024, again doubling the epoch size. In this case, tail FCT at low load grew by 1.28× (respectively 1.2× at high load). This shows that Shoal's performance degrades slowly with increasing number of end-hosts.

3.7.6 Simulation Experiments: Disaggregated Workloads

Finally, we evaluate the performance of Shoal on disaggregated workloads, based on recently published traces [34]. These traces comprise a variety of real-



(a) Flow completion time

(b) Average flow goodput

Figure 3.20: Flow completion time (short flows ≤100 KB) and average flow goodput (long flows ≥1 MB) for different applications with disaggregated workload.

world applications, including batch processing, graph processing, interactive queries, and relational queries. To generate the workloads, we mapped each end-host to one of the server resources (CPU, memory, and storage) and created flows between them following the distributions observed in these traces. This yielded a much more skewed workload than the one in Section 3.7.5 with more than 84% of the flows being generated among a third of the end-hosts.

Figure 3.20 shows the results for all the six applications, assuming a mean inter-arrival time of 12.65 ns. Shoal significantly outperforms the baselines in terms of both the tail FCT for short flows (factor of 2× or more) and average goodput for long flows (factor of 2.5×). As explained in Section 3.7.5, this is due to higher bandwidth provisioning in Shoal in combination with its highly effective scheduling and congestion control mechanisms (resulting in maximum queue size of just 10 cells across all applications).

These results show the versatility of Shoal and its ability to carry different types of traffic, including disaggregated workloads, with high throughput and low latency.

3.8 Discussion

3.8.1 Running Applications on top of Shoal

Shoal is a link layer architecture with support for congestion control. Running applications on top of Shoal, however, requires a transport layer providing application multiplexing, reliability, and flow control. One option would be to re-use an existing transport layer such as Transmission Control Protocol (TCP), although the impact of the interaction of its congestion-control mechanism with Shoal's remains an open question. Another approach would be to design a Shoal-specific transport layer. This would be significantly simpler than existing transport layers as the complexity of congestion control is already handled by Shoal. We leave the exploration of these options for future work.

3.8.2 Quality-of-Service on top of Shoal

By default, Shoal assumes that each cell belongs to the same traffic class, and schedules them using a single per destination FIFO queue. However, Shoal can be easily extended to support multiple traffic classes by maintaining multiple per destination FIFO queues, one per traffic class, and scheduling cells from the FIFOs in the order of their priority. Note that in this case the queue bound

(Section 3.3.6) will only hold for the highest priority traffic, while the lower priority traffic can see tail drops. To notify the sender of the tail drop, the end-host sets the `last-cell-dropped` field to 1 in the header of the next cell it sends to the sender. And finally, the rate limit feedback for a single traffic class in Equation 3.6 is updated for multiple traffic classes as follows—for a cell in traffic class c,

$$\mathit{rate\ limit\ feedback}^{c}_{ji} = \sum_{p} [\mathit{len}(Q^{p}_{jk}) + \mathit{len}(R^{p}_{jk}) - 1] \quad \forall\, p \;\; s.t.\ \mathit{priority}(p) \geq \mathit{priority}(c)$$

3.9 Summary

In this chapter, we presented Shoal, a switching fabric for datacenters comprising entirely of fast reconfigurable circuit switches. The fabric operates like a giant network-wide switch with a static, pre-defined schedule that creates a virtual full mesh topology. Shoal achieves bounded network throughput by detouring the network traffic uniformly, and implements backpressure-based congestion control which achieves fairness and bounded queuing. Our FPGA-based prototype achieves good performance and illustrates that Shoal's mechanisms are amenable to hardware implementation. Overall, we show that Shoal can effectively scale to high switching speeds in a cost and power efficient manner while achieving high throughput and low latency across diverse workloads.

CHAPTER 4

RELATED WORK

4.1 High-speed End-host Network Stacks

Traditionally network stacks have been implemented inside the Kernel at the end-hosts. This provides ease of protection, isolation, and resource sharing. However, as the performance requirements inside datacenters have become more stringent, such as microsecond tail latency and high packet rates, traditional Kernel-based network stacks have proven to be inadequate. This has prompted three broad alternative approaches:

Kernel optimized network stacks. Several proposals have been made to improve the existing Kernel-based network stacks. Affinity-accept [84] reduces overheads by ensuring all processing for a network flow is affinitized to the same core. Zero-copy sockets [144] improve performance by reducing the number of data copies. Busy polling driver mode [45] in Linux Kernel improves latency at the cost of CPU utilization. FastSocket [64] improves the performance for short-lived connections by eliminating various lock contentions in the entire stack. MegaPipe [40] and StackMap [182] propose new APIs to achieve zero copy and improve I/O multiplexing, at the cost of requiring application modifications.

User space network stacks. Motivated by the fact that Kernel optimized network stacks still inherit several Kernel overheads, several proposals have been made to bypass the Kernel and implement the network stack in the user space. Examples include SandStorm [70], mTCP [49], Seastar [120], F-Stack [114], Lib-

VMA [141], OpenOnload [169], DBL [147], LOS [44], and TAS [53]. While user space network stacks improve upon the performance of Kernel-based network stacks, it comes at the cost of lack of protection and isolation, e.g., application bugs and crashes can corrupt the networking stack and impact other workloads. Motivated by this observation, IX [10] and Arrakis [85] were proposed. The key idea is to separate the control plane (used to provide protection and isolation) from the high-performance data plane (used for network processing). In addition, there have been several high performance packet I/O frameworks proposed, e.g., Netmap [91], NetSlice [69], Intel DPDK [119], that allow direct access to NIC queues in user space, and are used by many of the aforementioned systems.

Hardware offloaded network stacks. As link speeds inside datacenters increase to 100 Gbps and beyond, software network stacks, both Kernel-based and user space, start to hit performance bottlenecks. As a result, recently several systems have been proposed that offload either part or whole of the network stack on to the hardware such as a NIC. Examples include RoCE NICs [168], AccelNet [33], FreeFlow [54], SocksDirect [60], AccelTCP [79], and Tonic [6]. This dissertation presented PIEO, which offloads packet scheduling, a key network stack functionality, to the NIC for higher performance.

4.2 High-speed Programmable Network Data Plane

The hardware implementations of the network data plane have traditionally been fixed-function as it was believed that a programmable architecture would compromise on the stringent performance requirements, such as line rate forwarding. However, more recently there have been several data plane architectures proposed that achieve both programmability and high-performance. RMT [16] proposes a reconfigurable match-action table architecture for programmable packet header processing, that has already been implemented on a wide-range of targets including software [94], FPGA [176, 46], and ASIC [148]. dRMT [23] improves on the resource efficiency of RMT by disaggregating memory and compute resources. FlowBlaze [86] and Domino [103] extend the RMT architecture to support stateful packet processing. Marple [81] presents a programmable architecture for network performance monitoring on the switches. Tonic [6] proposes a programmable architecture for the transport layer, while PIFO [104], AFQ [95], and CalQ [96] propose programmable architectures for packet scheduling. This dissertation presented PIEO, which is a new programmable architecture for packet scheduling that is more expressive than any state-of-the-art solutions, while also being highly scalable and high-speed. Somewhat complementary to our work, Loom [105] proposes a flexible packet scheduling framework in the NIC, using a new abstraction for expressing scheduling policies, and an efficient OS/NIC interface for scheduling, while leveraging the PIFO scheduler for enforcing the scheduling policies. In principle, systems like Loom can use PIEO scheduler instead of PIFO and achieve more expressibility and scalability.

4.3 Switching Fabric Topologies and Technologies

Several topologies have been investigated for datacenter switching fabric, both at datacenter-scale and rack-scale. The most widely used topology in datacenters is a Folded Clos topology, such as Fattree [1], built using commodity Ethernet switches. There have also been proposals for a flat topology of switches, such as Jellyfish [102], LongHop [107], Slimfly [14], and Xpander [170]. These topologies promise higher capacity fabrics for the same cost as a Clos network. However, there are sizable differences in performance even across flat topologies [51]. In particular, flat topologies that emulate a good expander graph [124], such as Jellyfish and Xpander, achieve much higher performance. At rack-scale, there have been proposals for direct-connect topologies whereby each end-host is connected directly to a small subset of other end-hosts through point-to-point links, such as a 3D Torus topology [157, 82]. Such topologies are extremely cost and power efficient, as they do not rely on expensive power-hungry packet switches, and have also been shown to achieve high-performance for long-running traffic.

The topologies discussed above are all statically wired topologies, and hence they cannot be altered dynamically. Motivated by the fact that the datacenter traffic can be skewed and unpredictable over time, there have also been proposals for dynamic topologies that leverage flexible electrical [59, 19], optical [31, 175, 20, 87, 39, 185, 35] or wireless [38] connections to operate like a circuit switch and dynamically alter the topology in response to the current traffic pattern. However, these solutions typically rely on a centralized controller to schedule traffic. This imposes significant communication and computation overhead and requires accurate demand estimation.

In 2002, load balanced switches [18] were proposed as a way to obviate arbitration in monolithic packet switches. This inspired the design of Shoal and a related work done concurrently with this dissertation called RotorNet [74], both of which operate like a load balanced switch, except instead of using an explicit intermediate stage as in the original proposal, Shoal and RotorNet detour cells through other end-hosts in the network. Furthermore, while the original proposal was for a single monolithic packet switch, both Shoal and RotorNet scale the idea to an entire switching fabric comprising circuit switches. However, RotorNet leverages circuit switches with high reconfiguration delay (20 μs) and hence, requires a separate packet-switched network for low-latency traffic. It also does not ensure bounded queuing. In contrast, Shoal leverages circuit switches with nanosecond reconfiguration, and proposes a novel congestion control that achieves high throughput, low latency, fairness, and bounded queuing for all traffic atop a purely circuit-switched fabric. A more recent work, called Opera [73], builds upon the design of Shoal and RotorNet by allowing packets from low-latency traffic to take multiple hops to the destination, as opposed to at most one hop as in Shoal and RotorNet, while all other traffic is still routed via at most one hop to the destination. This design reduces the wait time for the low-latency traffic at the intermediate nodes, and has been shown to achieve better end-to-end latency. However, Opera relies on the heavy-tailed traffic distribution [36] within datacenters to unlock its benefits, whereas Shoal and RotorNet are agnostic to the traffic distribution.

4.4 Congestion Control via Tight Host-Fabric Coupling

Unlike the Internet, datacenter networks are characterized with smaller scale, structured topologies, and easily customizable networking hardware. All this has led to an ongoing trend of increased coupling between the congestion control algorithms running at the end-hosts and the datacenter switching fabric. Examples include congestion control protocols such as DCTCP [2] and DCQCN [187] that use Explicit Congestion Notification (ECN) [125] from the switches to compute the sending rate for flows, pFabric [4] that uses in-network scheduling at the switches to reduce the average flow completion time, NDP [41] that uses the payload cut mechanism [21] inside the switches for an early packet drop notification, and RCP [28] and HPCC [62] that use queuing information at the switches to compute the sending rate for flows. This dissertation presented Shoal, which is an extreme design point in this direction as the coupling of its congestion control to its fabric achieves bounded queuing (unlike above protocols) and fairness despite very high multi-pathing.

CHAPTER 5

FUTURE WORK

5.1 High-speed Programmable Architectures for Middleboxes

Traditionally network middleboxes are implemented either on fixed-function hardware devices (compromising on flexibility) or on commodity servers (compromising on performance, such as latency). A high-speed programmable hardware architecture for middleboxes can potentially provide both flexibility and high-performance. This also opens up several interesting research directions.

Programmable architectures for Network Functions (NFs). Recent programmable hardware architectures for network data plane have focused primarily on packet header processing [16, 86] and packet scheduling [104, 99]. In PIEO, we presented one such architecture for packet scheduling. However, network middleboxes implement a much wider range of functionalities or Network Functions (NFs), including security (e.g., firewalls), performance (e.g., load balancing, Wide-area Network (WAN) optimizers), and support for new applications and protocols (e.g., Network Address Translation (NAT), Transport Layer Security (TLS) proxies). Further, many of these functionalities are stateful in nature. This calls for new computation models and hardware architectures that could express a wide-range of stateful NFs. Going a step further, it would also be interesting to explore the appropriate division of labor across the specialized programmable hardware and general-purpose CPUs for NFs.

Virtualizing programmable hardware middleboxes. While commodity server-based middleboxes have performance overheads, they do allow for Network

Function Virtualization (NFV) where multiple NFs are consolidated on the same machine and leverage known virtualization techniques to achieve both resource and performance isolation. NFV has been shown to significantly simplify the development and deployment of middleboxes. An interesting research direction is to adapt the known virtualization techniques for commodity servers (and potentially propose new virtualization abstractions and techniques) for specialized programmable hardware middleboxes, so that one does not have to compromise on the benefits of NFV in the quest for higher performance.

Offloading NFs to programmable switches. The presence of programmable switches in the network present an opportunity for different ways to implement NFs. Switches sit at a fundamentally different vantage point in the network, and state-of-the-art programmable switches are capable of performing non-trivial in-network computations at low latency and line rate. However, these switches are typically resource constrained. Thus, it would be interesting to explore if offloading a part or whole of an NF to switches could result in better performance, security, etc., without violating the resource constraints on the switches. This should also provide key insights into the capabilities and limitations of current programmable switches, that could guide the design of next-generation programmable switch architectures for in-network computations.

Programming framework for middleboxes. Finally, as we design new programmable architectures for middleboxes, we would also need new domain-specific languages and compilers to program those devices. Further, the vision of distributing NFs across heterogeneous architectures, ranging from specialized programmable hardware middleboxes, general-purpose CPUs, and programmable switches, calls for new languages and compiler designs that could express and automatically compile and distribute a given NF across multiple hardware targets.

5.2 Low Latency Optical Circuit Switching

Optical circuit switching promises unlimited bandwidth scaling at low cost and power. However, achieving low latency over an optical circuit-switched network still remains a challenge.

Designing low latency data plane for optical circuit switching. State-of-the-art commercial optical circuit switches reconfigure in the order of few 10s of microseconds, and have ~1000 ports. We need to improve on both these fronts for next-generation optical switches—lower reconfiguration delay is key to achieving low latency, while higher port count would connect more end-hosts using a flat topology, which would preclude signal degradation due to multiple hops and the need for more powerful and costly transceivers to recover the signal. However, there are two key challenges in building a fast reconfigurable, high port-count optical switch, namely fast Clock and Data Recovery (CDR) circuit and scalable crossbar technology. While there have been several advancements made on both these fronts over the last decade [24, 97, 27, 106, 55, 22, 29, 183, 63], including nanosecond reconfigurable optical switches, unfortunately those designs still exist as research prototypes, and scaling those designs to datacenter scale still remains an open challenge.

Designing low latency control plane for optical circuit switching. In Shoal, we presented the idea of using static schedules to create virtual topologies as a way to build fast and high-performing control plane for circuit switching. However, Shoal explores only one such virtual topology, namely, a full mesh topology. Fundamentally, one can configure the entries of the static schedule to create arbitrary virtual topologies, and each virtual topology presents a trade-off between worst-case throughput and worst-case latency. An interesting direction going forward is to explore the entire space of virtual topologies to better understand the throughput-latency trade-offs. Going one step further, one can explore designs with multiple virtual topologies operating in parallel, each serving different classes of applications, e.g., high throughput applications vs. low latency applications. Such a design is a natural fit for datacenters, that run a mixture of traffic with different performance objectives [37].

Programmable Hybrid Electro-Optical Networks. In-network computing has been shown to significantly improve the performance of several key applications [179, 180, 68, 186, 61]. However, an optically circuit-switched network precludes a possibility of in-network computation altogether. An interesting research direction is to build a hybrid network comprising both the programmable packet switches and optical circuit switches, that can potentially allow in-network computations while also achieve high network performance at low cost and power. While there have been a few proposals for hybrid networks in the past, they primarily optimize for network performance, i.e., throughput and latency, and do not consider programmability or in-network computation as an objective. A programmable hybrid network would add in-network computation as a first-order objective on top of the traditional network performance objectives, and would call for re-visiting several of the conventional network design choices for the right distribution of packet and circuit switches in the fabric, topology design, routing, and congestion control.

CHAPTER 6
CONCLUSION

The slowdown in Moore's law and the end of Dennard scaling has had, and continues to have, a profound impact on every field in computing that relies on transistor-based technology. This dissertation focuses specifically on its impact on datacenter networking. In particular, we note that it has become increasingly difficult to scale the speeds of general-purpose processors and packet switches, the two building blocks of the modern datacenter networking infrastructure. As a consequence, general-purpose processor-based end-host network stacks and packet switch-based switching fabrics are no longer able to keep pace with the ever-increasing link speeds and bandwidth demand inside modern datacenters. Thus, there is a pressing need for a grounds-up re-thinking of the design of datacenter networking infrastructure in the Post-Moore era.

To that end, this dissertation looks beyond the general-purpose processors and packet switches as the building blocks for datacenter networks, and instead explores the potential of using domain-specific processors and fast (nanosecond) reconfigurable circuit switches for building high-speed end-host network stacks and switching fabrics. In that pursuit, we identify two fundamental challenges—(i) designing hardware architecture for domain-specific processors that achieves the right balance between programmability and performance, and (ii) designing fast control plane for circuit switching that can schedule high-performing circuit configurations at nanosecond granularity. To address these challenges, this dissertation presents two systems, PIEO and Shoal. PIEO presents the design and implementation of a domain-specific processor for packet scheduling that is simultaneously programmable and high-performance (both scalable and high-speed). While Shoal presents the design and implementation of a fast control plane for circuit switching that enables a circuit-switched network that could effectively scale to high speeds while achieving comparable or better performance than several recent packet-switched network designs at significantly lower power and cost.

Overall, this dissertation demonstrates that by effectively leveraging the promise of domain-specific processing and fast circuit switching, through novel networking primitives and hardware architectures as presented in PIEO and Shoal, one can indeed build high-performing, high-speed end-host network stacks and switching fabrics, even in light of continued slowdown in Moore's law and the end of Dennard scaling.

APPENDIX A

PROOF OF INVARIANT 1 AND BOUND ON THE NUMBER OF SUBLISTS IN PIEO

INVARIANT 1 in Section 2.5.3, which states that there can never be two consecutive partially full sublists in the *Ordered-Sublist-Array*, ensures that the number of sublists needed in PIEO's hardware design to store N elements is bounded by $2\sqrt{N}$, where each sublist can store a maximum of $\sqrt{N}$ elements. So, we have:

PROPERTY 1: There can never be two consecutive partially full sublists in the *Ordered-Sublist-Array*.

Below we provide a formal proof of—(1) PROPERTY 1 is an invariant, i.e., it always holds, and (2) INVARIANT 1 bounds the number of sublists to $2\sqrt{N}$.

Claim 1: PROPERTY 1 is an invariant, i.e., it always holds.

Proof. This can be proved by induction. First note that after each operation (enqueue or dequeue) in PIEO, every sublist will be in one of the three states—full, empty, or partially full. Further, the *Ordered-Sublist-Array* is logically partitioned into a series of non-empty sublists (full or partially full), followed by a series of empty sublists (Figure 2.6). We will refer to the two partitions as *non-empty partition* and *empty partition* respectively. At the start, i.e., before any operations, all the sublists are empty, and hence PROPERTY 1 holds. This proves the base of induction. Next, we show that if PROPERTY 1 holds after the k^{th} operation, then it will also hold after the $(k+1)^{th}$ operation.

- **Case 1:** $(k+1)^{th}$ **operation is an enqueue operation, i.e., enqueue(f):** If the sublist in which we need to enqueue the new element, S, is initially partially full, then after enqueue, it will either be still partially full or become full. In either case, it cannot result in consecutive partially full sublists.

Instead, if S is full and is also the rightmost sublist in the *non-empty partition*, then we enqueue the new element into S, and enqueue the ejected element from S into the sublist to the immediate right of S, which is guaranteed to be empty as S is the rightmost sublist in the *non-empty partition*. After this operation, we have a new partially full sublist, S', to the immediate right of S, but it does not violate PROPERTY 1, as the sublist to the immediate left of S', i.e., S, is full, and the sublist to its immediate right is empty as S' is now the rightmost sublist in the *non-empty-partition*.

Instead, if S is full, and the sublist to its immediate right, S_{right}, is partially full, then we enqueue the new element into S, and enqueue the ejected element from S into S_{right}. Thus, after the operation, the state of S remains full as before, while the state of S_{right} either remains partially full or changes to full, neither of which could result in consecutive partially full sublists.

Finally, if S is full, and so is S_{right}, then we enqueue the new element into S, and enqueue the ejected element from S into a new sublist that we move from the *empty partition* into the *non-empty partition* and place to the immediate right of S (in cycle 3 and 4 of the enqueue operation (Section 2.5.3)). After this operation, we have a new partially full sublist, S', to the immediate right of S, but it does not violate PROPERTY 1, as the sublist to the immediate left of S', i.e., S, is full, and the sublist to its immediate right, i.e., S_{right} is also full.

Hence, under all possible scenarios, if PROPERTY 1 holds before an enqueue operation, it will also hold after the enqueue operation.

- **Case 2:** $(k + 1)^{th}$ **operation is a dequeue operation, i.e., dequeue() or dequeue(f):** If the sublist from which we need to dequeue an element, S, is partially full, then after dequeue, S will either be still partially full, or become empty. In case S becomes empty, we move it to the *empty partition*. As a result, if there was a sublist to the immediate left of S, S_{left}, and a sublist to the immediate right of S, S_{right}, then S_{left} and S_{right} would now become consecutive sublists in the *Ordered-Sublist-Array*. However, this would not violate PROPERTY 1, as since S was partially full before the dequeue operation, neither S_{left} nor S_{right} could have been partially full. Hence, even if they become consecutive sublists after this operation, we still won't have two consecutive partially full sublists.

  Instead, if S is full, then after dequeue, S would become partially full. Now if both S_{left} and S_{right} are also full, then even a partially full S will not violate PROPERTY 1. However, if either S_{left} or S_{right} or both are partially full, then this operation would result in the violation of PROPERTY 1. To avoid this, we choose one of the partially full neighbors of S, either S_{left} or S_{right}, and after dequeue from S, we move an element from the chosen neighbor to S (in cycle 3 and 4 of the dequeue operation (Section 2.5.3)). This ensures that S remains full even after dequeue, thus never violating PROPERTY 1. However, the neighbor from which we moved the element into S could become empty after the operation, and we would need to move it to the *empty partition*. However, this would also not violate PROPERTY 1, using the same argument as used in the above paragraph.

  Hence, under all possible scenarios, if PROPERTY 1 holds before a dequeue operation, it will also hold after the dequeue operation.

Claim 2: INVARIANT 1 ensures that $2\sqrt{N}$ sublists, of size $\sqrt{N}$ each, are sufficient to store N elements in PIEO.

Proof. Let us assume that we need x sublists, of size $\sqrt{N}$ each, to store N elements in PIEO. Then, each of the x sublists will have at least one element. Further, since INVARIANT 1 holds, hence at least $\frac{x-1}{2}$ of the sublists must be full (if x is odd), and respectively $\frac{x}{2}$ (if x is even), to ensure there are no two consecutive partially full sublists.

Assuming x is odd, there must be at least $(\sqrt{N}\frac{x-1}{2} + \frac{x+1}{2})$ elements in PIEO.

$$\sqrt{N}\frac{x-1}{2} + \frac{x+1}{2} \leq number\ of\ elements\ in\ PIEO \tag{A.1}$$

Further, the number of elements in PIEO is bounded by N, i.e.,

$$number\ of\ elements\ in\ PIEO \leq N \tag{A.2}$$

Hence, from Equation A.1 and Equation A.2, we have,

$$\sqrt{N}\frac{x-1}{2} + \frac{x+1}{2} \leq N \tag{A.3}$$

On solving for x, we get,

$$\begin{aligned} x &\leq \frac{2N+\sqrt{N}-1}{\sqrt{N}+1} \\ &< \frac{2N+\sqrt{N}-1}{\sqrt{N}} \\ &< 2\sqrt{N}+(1-\frac{1}{\sqrt{N}}) \\ &< 2\sqrt{N} \quad since\ the\ second\ term\ is\ < 1 \end{aligned} \tag{A.4}$$

Similarly, assuming x is even, Equation A.3 would change to,

$$\sqrt{N}\frac{x}{2} + \frac{x}{2} \leq N \tag{A.5}$$

And on solving for x, we would get,

$$\begin{aligned} x &\leq \frac{2N}{\sqrt{N}+1} \\ &< \frac{2N}{\sqrt{N}} \\ &< 2\sqrt{N} \end{aligned} \tag{A.6}$$

Hence, either way, we get $x < 2\sqrt{N}$, i.e., $2\sqrt{N}$ sublists are sufficient to store N elements in PIEO.

APPENDIX B

GLOSSARY OF TERMS

- **10GBase-SR.** 10GBase-SR (SR stands for ″short range″) is a port type for optical fibers used for communication over short distances. It uses 850 nanometer lasers to deliver serialized data at a line rate of 10.3125 Gigabaud. See also *Port*, *Serialization-Deserialization (SerDes)*, *Serial Communication Channel*, *Fiber (optical)*.

- **3D Torus.** In networking, a torus is a network topology for connecting compute nodes. A 3D torus is a three-dimensional torus where nodes are imagined connected in the shape of a rectangular prism, with each node connected with its 6 neighbors. See also *Network Topology*, *Compute Node*.

- **Adaptive Logic Module (ALM).** The Adaptive Logic Module (ALM) is the basic building block of the FPGAs used in this dissertation. ALMs are used to implement the digital circuit logic on top of FPGAs. See also *Field-programmable Gate Array (FPGA)*.

- **Amdahl's Law.** The speedup of a program using multiple processors in parallel is limited by the time needed for the sequential fraction of the program. In particular, if a fraction P of the overall computation can be sped up through parallelism by a factor of S, then the overall speedup of the entire computation, including the fraction $(1 - P)$ that cannot be sped up (parallelized), is $\frac{1}{(1-P)+\frac{P}{S}}$.

- **Application-specific Integrated Circuit (ASIC).** An Application-specific Integrated Circuit (ASIC) is an integrated circuit (IC) chip customized for a particular use, rather than intended for general-purpose use. See also *Integrated Circuit (IC)*.

- **Application Programming Interface (API).** An application programming interface is a computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc.

- **Backplane Trace.** The backplane trace is a printed circuit board containing connections (slots) for other boards and allows for communication between all connected boards.

- **Backpressure.** In computer networking, backpressure refers to the resistance or force opposing the desired flow of data through the network. This typically happens if the rate of outgoing data from the network is not able to keep pace with the rate of incoming data into the network.

- **Bandwidth.** Bandwidth is the maximum rate of data transfer across a network.

- **Bisection Bandwidth.** If the network is bisected into two partitions, the bisection bandwidth of a network topology is the bandwidth available between the two partitions. Bisection should be done in such a way that the bandwidth between two partitions is minimum. Bisection bandwidth gives the true bandwidth available in the entire system. Bisection bandwidth accounts for the bottleneck bandwidth of the entire network. Therefore bisection bandwidth represents bandwidth characteristics of the network better than any other metric. See also *Bandwidth*, *Network Topology*.

- **Cache.** A smaller, faster memory which stores copies of the data from a subset of main memory (DRAM) locations for faster access. Caches are generally divided into different levels—Level 1 (L1) cache is the closest to the processor and also the fastest, followed by Level 2 (L2) cache, and then Level 3 (L3) cache. See also *Dynamic Random Memory Access (DRAM)*.

- **Capacitor.** A capacitor is a device that stores electrical energy in an electric field.

- **Checksum.** A checksum is a value used to verify the integrity of data transfer. In other words, it is a sum that checks the validity of data. Checksums are typically used to compare two sets of data to make sure they are the same.

- **Circuit Reconfiguration.** It refers to the process of setting up the physical connections between the input and output ports of a circuit switch. See also *Port*, *Circuit Switching*, *Circuit Switch*, *Circuit Scheduling*.

- **Circuit Scheduling.** It refers to the process of deciding what connections to set up between the input and output ports of a circuit switch. See also *Port*, *Circuit Switching*, *Circuit Switch*, *Circuit Reconfiguration*.

- **Circuit Switch.** A circuit switch is a special kind of switch that does no processing on the data flowing through the switch, nor does it buffer any data. It simply forwards data between its input and output ports via a crossbar. However, it does require circuit scheduling and circuit reconfiguration. A circuit switch can be electrical or optical. See also *Switch*, *Forwarding*, *Crossbar*, *Circuit Switching*, *Circuit Scheduling*, *Circuit Reconfiguration*, *Electrical Circuit Switch*, *Optical Circuit Switch*, *Packet Switch*.

- **Circuit Switching.** Circuit switching is a method of implementing a communication network in which two communicating nodes establish a dedicated communication channel (circuit) through the network before the nodes may communicate. See also *Switching*, *Communication Channel*, *Packet Switching*.

- **Clock Domain Crossing (CDC).** In digital electronics, clock domain crossing refers to the traversal of a signal in a digital circuit from one clock domain into another. See also *Retiming*, *Synchronization FIFO*.

- **Clock and Data Recovery (CDR).** In serial communication of digital data, clock and data recovery is the process of extracting timing information from a serial data stream to allow the receiving circuit to decode the transmitted data. See also *Serial Communication Channel*.

- **Clos Network.** In the field of communications, a Clos network is a kind of multistage switching network that has an input stage, an output stage, and some number of middle stages. Clos network represents a theoretical idealization of practical, multistage switching systems. See also *Switching*, *Folded Clos Network*.

- **Cloud Computing.** Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user.

- **Cloud Network.** Cloud network is referred to a computer network that exists within or is part of a cloud computing infrastructure. It is a computer network that provides network interconnectivity between cloud based or cloud enabled application, services and solutions. See also *Cloud Computing*.

- **Commodity.** A mass-produced unspecialized product for which there is demand, but which is supplied without qualitative differentiation across a market. It is a fungible product, meaning the same irrespective of who produces it.

- **Communication Channel.** A communication channel refers either to a physical transmission medium such as a wire, or to a logical connection over a multiplexed medium such as a radio channel in telecommunications and computer networking.

- **Comparator.** A digital circuit that takes two numbers as input in binary form and determines whether one number is greater than, less than or equal to the other number.

- **Compiler.** A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language).

- **Compute Node.** Compute nodes are the machines on which applications and services run.

- **Congestion Control.** Congestion control controls the flow of data transmission in a communications network in order to prevent senders from overwhelming the network. See also *Flow Control*.

- **Container.** In computing, a container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

- **Control Plane.** In computer networking, control plane is responsible for the management and configuration of how data is forwarded through the network. See also *Forwarding*, *Data Plane*.

- **Crossbar.** In computer networking, a crossbar is a switching element with N inputs and M outputs where, typically, $N = M$. The crossbar can simultaneously transport signals on any of the N inputs to any of the M outputs as long as multiple signals do not compete for the same input or output port. See also *Switching*, *Port*.

- **Data Plane.** In computer networking, data plane is responsible for actual forwarding of data through the network. See also *Forwarding*, *Control Plane*.

- **Datacenter.** A datacenter is a building, dedicated space within a building, or a group of buildings used to house computer systems and associated components, such as communications and storage systems.

- **Datastructure.** A datastructure is a specialized format for organizing, processing, retrieving and storing data.

- **Dennard Scaling.** A power scaling law that roughly states that as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area. It ended around 2006. See also *Transistor*.

- **Die.** In electronics, a die is a small block of semiconducting material on which a given digital circuit is fabricated. Typically several dies are batched together on a single wafer. See also *Semiconductor*, *Wafer*.

- **Direct Memory Access (DMA).** Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems (such as NICs) to access main system memory (DRAM) independent of the CPU. See also *Network Interface Card (NIC), Dynamic Random Access Memory (DRAM), Remote Direct Memory Access (RDMA)*.

- **Direct-connect Topology.** A network topology where end-hosts are connected directly to one another without needing a switching fabric. See also *Network Topology*, *End-host*, *Switching Fabric*.

- **Domain-specific Language.** A computer programming language specialized to a particular application domain.

- **Domain-specific Processor.** An application-specific integrated circuit (ASIC) designed for a specific set of application domain. See also *Application-specific Integrated Circuit (ASIC)*, *General-purpose Processor*.

- **Dynamic Random Access Memory (DRAM).** A type of random access memory (RAM) that stores each bit of data in a memory cell consisting of a tiny capacitor and a transistor. See also *Random Access Memory (RAM)*, *Capacitor*, *Transistor*.

- **Electrical Circuit Switch.** A circuit switch that operates on electrical signals. See also *Circuit Switch*, *Optical Circuit Switch*.

- **End-host.** An end-host is a computer or other device sitting at the edge of a computer network.

- **Equal-cost Multi-path (ECMP).** Equal-cost multi-path (ECMP) is a routing strategy where packet destined to a single destination can be forwarded over multiple ”best paths” which tie for top place in routing metric calculations. See also *Routing*, *Forwarding*.

- **Ethernet.** Ethernet is a family of protocols for the data link layer and the physical layer of the network stack. See also *Network Stack*.

- **Fair Queuing.** Fair queuing is a family of scheduling algorithms designed to achieve fairness when a limited resource is shared. See also *Packet Scheduling*, *Fairness*.

- **Fairness.** Fairness is a metric used to determine whether users or applications are receiving a fair share of system resources.

- **Fast Reconfigurable Circuit Switch.** In the context of this dissertation, a fast reconfigurable circuit switch can do circuit reconfiguration within a few nanoseconds. See also *Circuit Switch*, *Circuit Reconfiguration*, *Slow Reconfigurable Circuit Switch*.

- **Fiber (optical).** A fiber is a long, thin strand of very pure glass about the diameter of a human hair. They are arranged in bundles called optical cables and used to transmit light signals over long distances.

- **Field-programmable Gate Array (FPGA).** Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects. They can be configured to implement arbitrary digital circuits. See also *Semiconductor*.

- **FIFO.** A First-In-First-Out queue.

- **Firewall.** In computing, a firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules. A firewall typically establishes a barrier between a trusted internal network and untrusted external network, such as the Internet.

- **Fixed-function Hardware.** A digital device whose logic cannot be changed once the device has been designed and manufactured.

- **Flip-flop.** In electronics, a flip-flop is a circuit that has two stable states and can be used to store state information.

- **Flow.** A sequence of packets from a source to a destination. See also *Packet*.

- **Flow Control.** Flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver. See also *Congestion Control*.

- **Folded Clos Network.** A Folded Clos network is the one-sided version of the Clos network: it basically merges the corresponding input and output stages into one stage. See also *Clos Network*.

- **Forwarding.** The act of relaying packets from one network segment to another by nodes in a computer network. See also *Packet*, *Network Segment*.

- **Frequency Synchronization.** The state in which the clock frequencies of different clocks match. See also *Time Synchronization*.

- **Full Bisection Bandwidth.** If the network is bisected into two partitions, the aggregate bandwidth of the hosts in one section equals the bisection bandwidth. See also *Bisection Bandwidth*.

- **Full Duplex.** A full duplex device is capable of simultaneous, bi-directional network data transmissions.

- **Full Mesh Topology.** Full mesh topology occurs when every node has a link connecting it to every other node in a network. See also *Network Topology*.

- **General-purpose Processor.** A processor that is not designed for a specific application domain, but rather can be programmed to support a variety of applications. See also *Domain-specific Processor*.

- **Graphics Processing Unit (GPU).** A massively multi-core processor designed for fast parallel processing of large blocks of data.

- **Hardware Offload.** Moving certain applications from CPU to hardware, such as a NIC, for higher performance. See also *Network Interface Card (NIC)*.

- **Hardware-accelerated.** Accelerating applications via hardware offload. See also *Hardware Offload*.

- **Header.** The portion of a packet that carries control information, typically used for routing the packet. See also *Packet*, *Packet Switching*, *Routing*, *Payload*.

- **Hybrid Network.** In the context of this dissertation, a hybrid network refers to a network comprising a combination of packet and circuit switches. See also *Packet Switch*, *Circuit Switch*.

- **I/O.** In computing, input/output or I/O is the communication between a device, such as a computer, and the outside world, possibly a human or another device.

- **I/O Channel.** An I/O channel is an independent hardware component that co-ordinates all I/O between a set of devices. See also *I/O*.

- **I/O Pin.** An I/O pin is a form of I/O channel that is both an Input and an Output and can be used as both at the same time. Input pins are used to read the outside world and Output pins are used to control the outside world. See also *I/O*, *I/O Channel*.

- **In-network Computation.** It refers to computation done inside network switches. See also *Switch*.

- **In-network Scheduling.** It refers to packet scheduling inside network switches. See also *Packet Scheduling*, *Switch*.

- **Integrated Circuit (IC).** An integrated circuit (also referred to as an IC, a chip, or a microchip) is a set of electronic circuits on one small flat piece (or ”chip”) of semiconductor material that is normally silicon. See also *Semiconductor*.

- **IP Address.** An Internet Protocol (IP) address is a numerical label assigned to each device connected to a computer network.

- **Kernel.** The kernel is a computer program at the core of a computer’s operating system with complete control over everything in the system. It facilitates interactions between hardware and software components.

- **Latency.** In computer networking, latency refers to the time it takes for a piece of data (such as a packet) to go from one point (source) to another point (destination) over a communication channel. See also *Packet*, *Communication Channel*.

- **LGMII.** 50 gigabit media-independent interface (LGMII) is a standard defined in IEEE 802.3 for connecting full duplex 50 Gigabit Ethernet (50GbE) ports to each other and to other electronic devices. See also *Full Duplex*, *Ethernet*, *Port*, *XGMII*.

- **Link.** The link is the physical or logical network component used to interconnect nodes in the network. *E.g.,* optical fiber. See also *Fiber (optical)*.

- **Load Balancing.** Load balancing refers to the process of distributing a set of inputs over a set of resources, with the aim of making the overall system more efficient.

- **Maximum Transmission Unit (MTU).** In computer networking, the maximum transmission unit (MTU) is the size of the largest protocol data unit (PDU) that can be communicated in a single transaction of the protocol. For Ethernet, the MTU value is typically 1500 bytes. See also *Protocol Data Unit (PDU)*, *Ethernet*.

- **Medium Access Control (MAC).** The medium access control sublayer is the layer that controls the hardware responsible for interaction with the wired, optical or wireless transmission medium. The MAC sublayer is part of the data link layer in the network stack. See also *Network Stack*

- **Middlebox.** A middlebox or network appliance is a computer networking device that transforms, inspects, filters, and manipulates traffic for purposes other than packet forwarding. See also *Forwarding*.

- **Moore's Law.** An empirical law that postulated that the number of transistors on a chip would double every year. Moore's law has slowed down significantly since 2010, and is expected to end altogether soon.

- **Multi-tenant Cloud.** A multi-tenant cloud is a cloud computing architecture that allows customers to share computing resources. See also *Cloud Computing*.

- **Network Accelerator.** A device used to accelerate network functions. See also *Network Function*, *Hardware-accelerated*, *Hardware Offload*.

- **Network Address Translation (NAT).** Network address translation is a method of remapping an IP address space into another by modifying network address information in the header of packets while they are in transit across a traffic routing device. See also *IP Address*, *Routing*.

- **Network Function.** A functional building block within a network infrastructure, which has well-defined external interfaces and a well-defined functional behavior.

- **Network Function Virtualization (NFV).** NFV is a network architecture concept that virtualizes entire classes of network functions into building blocks that may connect, or chain together, to create communication services. See also *Virtualization*, *Network Function*.

- **Network Interface Card (NIC).** A hardware device that connects the end-host network stack to the switching fabric. See also *End-host*, *Network Stack*, *Switching Fabric*.

- **Network Segment.** A portion of a computer network wherein every device communicates using the same physical layer in the network stack. See also *Network Stack*.

- **Network Stack.** The network stack is an implementation of a computer networking protocol suite or protocol family. It typically comprises seven abstraction layers.

  - **Layer 1 (Physical Layer)** The physical layer (PHY) is the lowest layer in the network stack. PHY consists of the basic hardware transmission technologies needed to transmit raw bits over a physical link connecting network nodes.

  - **Layer 2 (Data Link Layer)** The data link layer is the protocol layer which transfers data between adjacent network nodes in a network or between nodes on the same network segment. See also *Network Segment*.

  - **Layer 3 (Network Layer)** The network layer is responsible for routing data from source to the destination end-host through the network, including routing through intermediate routers. See also *Routing*.

  - **Layer 4 (Transport Layer)** The transport layer provides transparent transfer of data between endpoints, providing data transfer services to the upper layers. The transfer could be reliable or unreliable.

  - **Layer 5 (Session Layer)** The session layer controls the dialogues (connections) between computers. It establishes, manages and terminates the connections between the local and remote application.

  - **Layer 6 (Presentation Layer)** The presentation layer establishes context between application-layer entities, in which the application-layer entities may use different syntax and semantics if the presentation service provides a mapping between them.

  - **Layer 7 (Application Layer)** The application layer is the closest to the end applications, and typically identifies the identity and availability of communication partners for an application with data to communicate.

- **Non-blocking Network.** In a non-blocking network, the nodes are interconnected in such a way that any unused input-output pair can be connected by a path through unused nodes, no matter what other paths exist at the time.

- **Off-chip Memory.** Memory which is not part of the processor chip. *E.g.,* DRAM. See also *Dynamic Random Access Memory (DRAM)*.

- **On-chip Memory.** Memory which is part of the processor chip. *E.g.,* SRAM. See also *Static Random Access Memory (SRAM)*.

- **Open Systems Interconnection (OSI) Model.** The Open Systems Interconnection model is a conceptual model that characterizes and standardizes the communication functions of a communication or computing system without regard to its underlying internal structure and technology.

- **Optical Circuit Switch.** A circuit switch that operates on optical signals. See also *Circuit Switch*, *Electrical Circuit Switch*.

- **Ordered List.** A sorted list of elements.

- **Packet.** A packet is a formatted unit of data carried by a packet-switched network. A packet consists of control information (header) and user data (payload). See also *Packet Switching*, *Header*, *Payload*.

- **Packet Pacing.** A set of techniques to make the pattern of packet transmission more uniform (or less bursty). See also *Packet*.

- **Packet Scheduling.** Deciding and enforcing when and in what order to transmit packets onto the wire. See also *Packet*.

- **Packet Spraying.** A routing strategy in which packets from a flow are sent uniformly over all available paths. See also *Routing*, *Packet*, *Flow*.

- **Packet Switch.** A packet switch is a special kind of switch that forwards packets by inspecting and processing each packet going through it. See also *Switch*, *Packet*, *Forwarding*, *Circuit Switch*.

- **Packet Switching.** A mode of data transmission in which a message is broken into a number of parts which are sent independently, over whatever route is optimum for each packet, and reassembled at the destination. See also *Switching*, *Packet*, *Circuit Switching*.

- **Parallel Communication Channel.** In data transmission, parallel communication channel is a type of communication channel that conveys multiple bits simultaneously. See also *Communication Channel*, *Serial Communication Channel*.

- **Pareto Distribution.** Pareto distribution is a skewed, heavy-tailed distribution that is sometimes used to model that distribution of incomes. The basis of the distribution is that a high proportion of a population has low income while only a few people have very high incomes.

- **Payload.** The portion of the packet that carries user data. See also *Packet*, *Packet Switching*, *Header*.

- **Peripheral Component Interconnect Express (PCIe).** PCIe is a high-speed serial computer expansion bus standard for connecting high-speed peripheral devices to the computer.

- **Phase-locked Loop (PLL).** A phase-locked loop is a control system that generates an output signal whose phase is related to the phase of an input signal.

- **Poisson Process.** A Poisson Process is a model for a series of discrete event where the average time between events is known, but the exact timing of events is random . The arrival of an event is independent of the event before (waiting time between events is memoryless).

- **Port.** A physical port is a specialized outlet on a piece of equipment (such as a switch or a NIC) to which a cable connects. See also *Switch*, *Network Interface Card (NIC)*.

- **Programmable Switch.** A packet switch that can be programmed with different packet processing protocols. See also *Packet*, *Packet Switch*.

- **Propagation Delay.** In computer networks, propagation delay is the amount of time it takes for the head of the signal to travel from the sender to the receiver over a communication channel, such as a wire. See also *Communication Channel*.

- **Protocol Data Unit (PDU).** A protocol data unit (PDU) is a single unit of information transmitted among peer entities of a computer network. A PDU is composed of protocol-specific control information and user data.

- **Proxy.** In computer networking, a proxy is an application or appliance that acts as an intermediary for requests from clients seeking resources from servers that provide those resources.

- **Public Cloud.** The public cloud is defined as computing services offered by third-party providers over the public Internet, making them available to anyone who wants to use or purchase them. See also *Cloud Computing*.

- **Quality-of-Service (QoS).** Quality of service is the description or measurement of the overall performance of a service, particularly the performance seen by the users.

- **Random Access Memory (RAM).** Random-access memory is a form of computer memory that can be read and changed in any order. See also *Dynamic Random Access Memory (DRAM)*, *Static Random Access Memory (SRAM)*.

- **Rate Limiting.** Rate limiting is used to control the amount of incoming and outgoing traffic to or from a network.

- **Reconfiguration Speed.** The time a circuit switch takes to do circuit reconfiguration. See also *Circuit Switch*, *Circuit Reconfiguration*, *Fast Reconfigurable Circuit Switch*, *Slow Reconfigurable Circuit Switch*.

- **Remote Direct Memory Access (RDMA).** In computing, remote direct memory access is a direct memory access from the memory of one computer into that of a remote computer without involving either one's CPUs. See also *Direct Memory Access (DMA)*.

- **Resource Disaggregation.** Resource disaggregation separates hardware resources (such as CPU, memory, storage) into network-attached, standalone devices that applications access over the network.

- **Retiming.** In the context of this dissertation, retiming refers to modulating the signal delay for a safe clock domain crossing. See also *Clock Domain Crossing (CDC)*, *Synchronization FIFO*.

- **Routing.** Routing is the process of selecting a path for traffic in a network or between or across multiple networks.

- **Semiconductor.** A solid substance that has a conductivity between that of an insulator and that of most metals, either due to the addition of an impurity or because of temperature effects. Devices made of semiconductors, notably silicon, are essential components of most electronic circuits.

- **Serial Communication Channel.** In data transmission, serial communication channel is a type of communication channel that conveys data one bit at a time. See also *Communication Channel*, *Parallel Communication Channel*.

- **Serialization-Deserialization (SerDes).** A SerDes is a functional block used in high-speed communication to convert data between serial and parallel channels in each direction. See also *Serial Communication Channel*, *Parallel Communication Channel*.

- **Singlemode Fiber.** In optical fiber technology, singlemode fiber is optical fiber that is designed for the transmission of a single ray or mode of light as a carrier and is used for long-distance signal transmission. See also *Fiber (optical)*.

- **Slow Reconfigurable Circuit Switch.** In the context of this dissertation, a slow reconfigurable circuit switch does circuit reconfiguration within a few microseconds to a few milliseconds. See also *Circuit Switch*, *Circuit Reconfiguration*, *Fast Reconfigurable Circuit Switch*.

- **Small Form-factor Pluggable (SFP+).** A type of transceiver that can support data rates up to 16 Gbps. See also *Transceiver*.

- **Static Random Access Memory (SRAM).** A type of random-access memory (RAM) that uses flip-flops to store each bit. See also *Random Access Memory (RAM)*, *Flip-flop*.

- **Switching.** Switching is the process to forward data coming in from one port to a port leading towards the destination. See also *Forwarding*, *Port*, *Packet Switching*, *Circuit Switching*.

- **Switch.** A network device used for switching. See also *Switching*, *Packet Switch*, *Circuit Switch*.

- **Switching Chip.** An application-specific integrated circuit (ASIC) that implements the switching logic inside a packet switch. See also *Application-specific Integrated Circuit (ASIC)*, *Switching*, *Packet Switch*.

- **Switching Fabric.** A collection of switches connected in some network topology. See also *Switch*, *Network Topology*.

- **Switching Speed.** The aggregate rate (across all ports) at which a switch can forward data. See also *Port*, *Switch*, *Forwarding*.

- **Synchronization FIFO.** A special kind of First-In-First-Out (FIFO) queue used for signal retiming for a safe clock domain crossing. See also *Clock Domain Crossing (CDC)*, *Retiming*.

- **Synchronization Precision.** In time synchronization, it is the degree to which clock values of two clocks are synchronized, or the maximum offset between any two clock values. See also *Time Synchronization*.

- **System-on-Chip (SoC).** A system on a chip is an integrated circuit that integrates all or most components of a computer or other electronic system on a single chip. See also *Integrated Circuit (IC)*.

- **Throughput.** In computer networking, throughput is the rate of successful data delivery over a communication channel. See also *Communication Channel*.

- **Time Synchronization.** The state in which the clock values of different clocks match. See also *Frequency Synchronization*.

- **Network Topology.** Network topology is the arrangement of the elements of a communication network (*e.g.,* switches). See also *Switch*.

- **Traffic Class.** A category of computer network traffic. Generally a traffic classification module categorizes computer network traffic according to various parameters into a number of traffic classes. Each resulting traffic class can be treated differently to provide different Quality-of-Service. See also *Quality-of-Service (QoS)*.

- **Traffic Matrix.** A traffic matrix represents the volume of traffic between all possible pairs of sources and destinations.

- **Transceiver.** In computer networking, a transceiver is a hardware device designed to connect computers or electronic devices within a network, allowing them to transmit and receive messages.

- **Transistor.** A transistor is a semiconductor device used to amplify or switch electronic signals and electrical power. See also *Semiconductor*.

- **Transmission Control Protocol (TCP).** TCP (Transmission Control Protocol) is a transport layer standard in the network stack that defines how to establish and maintain a network connection through which application programs can exchange data. See also *Network Stack*.

- **Transmission Time.** It is the time from the first bit until the last bit of a message has left the transmitting node.

- **Transport Layer Security (TLS).** Transport Layer Security (TLS) is a cryptographic protocol designed to provide communications security over a computer network.

- **User space.** User space is the set of memory locations in which user processes (i.e., everything other than the kernel) run. See also *Kernel*.

- **Virtual Machine (VM).** In computing, a virtual machine is an emulation of a physical computer system. A single physical computer system can typically run multiple VMs.

- **Virtual Switch (vSwitch).** A virtual switch (vSwitch) is an application that allows communication between virtual machines. See also *Virtual Machine (VM)*.

- **Virtualization.** In computing, virtualization refers to the act of creating a virtual version of something, including virtual computer hardware platforms, storage devices, and computer network resources.

- **Wafer.** In electronics, a wafer is a thin slice of semiconductor, such as silicon, used for the fabrication of integrated circuits. See also *Semiconductor*, *Integrated Circuit (IC)*.

- **Wide-area Network (WAN).** A computer network in which the computers connected may be far apart, generally having a radius of half a mile or more.

- **XGMII.** 10 gigabit media-independent interface (XGMII) is a standard defined in IEEE 802.3 for connecting full duplex 10 Gigabit Ethernet (10GbE) ports to each other and to other electronic devices. See also *Full Duplex*, *Ethernet*, *Port*, *LGMII*.

- **Yield (for chip fabrication).** Yield is a quantitative measure of the quality of a semiconductor process. It is the fraction of dies on the yielding wafers that are not discarded during the manufacturing process. See also *Semiconductor*, *Die*, *Wafer*.

BIBLIOGRAPHY

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. *A Scalable, Commodity Data Center Network Architecture*. SIGCOMM, 2008.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. *Data Center TCP (DCTCP)*. SIGCOMM, 2010.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. *Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center*. NSDI, 2012.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. *pFabric: Minimal Near-optimal Datacenter Transport*. SIGCOMM, 2013.

[5] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. *HotCocoa: Hardware Congestion Control Abstractions*. HotNets, 2017.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. *Enabling Programmable Transport Protocols in High-Speed NICs*. NSDI, 2020.

[7] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Holzle, Stephen Stuart, and Amin Vahdat. *Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network*. SIGCOMM, 2015.

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. *Information-Agnostic Flow Scheduling for Commodity Data Centers*. NSDI, 2015.

[9] Hitesh Ballani, Paolo Costa, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, and Hugh Williams. *Bridging the Last Mile for Optical Switching in Data Centers*. Optical Fiber Communication Conference (OFC), 2018.

[10] 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*. OSDI, 2014.

[11] Jon C. R. Bennett and Hui Zhang. *Hierarchical Packet Fair Queueing Algorithms*. SIGCOMM, 1996.

[12] Jon C. R. Bennett and Hui Zhang. *WF^2Q: Worst-case Fair Weighted Fair Queueing*. INFOCOM, 1996.

[13] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1987.

[14] Maciej Besta and Torsten Hoefler. *Slim Fly: A Cost Effective Low-Diameter Network Topology*. SC, 2014.

[15] 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 CCR, 2014.

[16] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. *Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN*. SIGCOMM, 2013.

[17] Randy Brown. *Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem*. Communications of the ACM, 1988.

[18] Cheng-Shang Chang, Duan-Shin Lee, and Yi-Shean Jou. *Load Balanced Birkhoff-von Neumann Switches, Part I: One-stage Buffering*. Computer Communications, 2002.

[19] Andromachi Chatzieleftheriou, Sergey Legtchenko, Hugh Williams, and Antony Rowstron. *Larry: Practical Network Reconfigurability in the Data Center*. NSDI, 2018.

[20] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. *OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility*. NSDI, 2012.

[21] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. *Catch the whole lot in an action: Rapid precise packet loss notification in data centers*. NSDI, 2014.

[22] Q. Cheng, A. Wonfor, J. L. Wei, R. V. Penty, and I. H. White. *Demonstration of the feasibility of large-port-count optical switching using a hybrid Mach-Zehnder interferometer-semiconductor optical amplifier switch module in a recirculating loop*. Optics Letters, 2014.

[23] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. *dRMT: Disaggregated Programmable Switching*. SIGCOMM, 2017.

[24] Kari Clark, Hitesh Ballani, Polina Bayvel, Daniel Cletheroe, Thomas Gerard, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, Philip Watts, Hugh Williams, Georgios Zervas, Paolo Costa, and Zhixin Liu. *Sub-Nanosecond Clock and Data Recovery in an Optically-Switched Data Centre Network*. European Conference on Optical Communication (ECOC), Post-deadline paper, 2018.

[25] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. *R2C2: A Network Stack for Rack-scale Computers*. SIGCOMM, 2015.

[26] Alan Demers, Srinivasan Keshav, and Scott Shenker. *Analysis and Simulation of a Fair Queueing Algorithm*. SIGCOMM, 1989.

[27] M. Ding, A. Wonfor, Q. Cheng, R. V. Penty, and I. H. White. *Scalable, Low-Power-Penalty Nanosecond Reconfigurable Hybrid Optical Switches for Data Centre Networks*. Conference on Lasers and Electro-Optics (CLEO), 2017.

[28] Nandita Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Doctoral Dissertation, Stanford University, 2007.

[29] Vincenzo Eramo and Marco Listanti. *Power Consumption in Bufferless Optical Packet Switches in SOA Technology*. Journal of Optical Communications and Networking, 2009.

[30] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. *Dark Silicon and the End of Multicore Scaling*. ISCA, 2011.

[31] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. *Helios: a hybrid electrical/optical switch architecture for modular data centers*. SIGCOMM, 2010.

[32] Daniel Firestone. *Hardware-Accelerated Networks at Scale in the Cloud*. Microsoft Azure Networking at Workshop on Kernel-Bypass Networks at SIGCOMM, 2017.

[33] 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*. NSDI, 2018.

[34] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. *Network Requirements for Resource Disaggregation*. OSDI, 2016.

[35] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. *ProjecToR: Agile Reconfigurable Data Center Interconnect*. SIGCOMM, 2016.

[36] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. *VL2: A Scalable and Flexible Data Center Network*. SIGCOMM, 2009.

[37] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. *Queues Don't Matter When You Can JUMP Them!* NSDI, 2015.

[38] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. *Augmenting Data Center Networks with Multi-gigabit Wireless Links*. SIGCOMM, 2011.

[39] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. *FireFly: A Re-*

configurable Wireless Data Center Fabric Using Free-space Optics. SIGCOMM, 2014.

[40] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. *MegaPipe: A New Programming Interface for Scalable Network I/O*. OSDI, 2012.

[41] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew Moore, Gianni Antichi, and Marcin Wojcik. *Re-architecting datacenter networks and stacks for low latency and high performance*. SIGCOMM, 2017.

[42] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. *AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks*. SIGCOMM, 2016.

[43] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach Sixth Edition*. Morgan Kaufmann, 2019.

[44] Yukai Huang, Jinkun Geng, Du Lin, Bin Wang, Junfeng Li, Ruilin Ling, and Dan Li. *LOS: A High Performance and Compatible User-level Network Operating System*. Asia-Pacific Workshop on Networking, 2017.

[45] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. *Direct Cache Access for High Bandwidth Network I/O*. ISCA, 2005.

[46] Stephen Ibanez, Gordon J. Brebner, Nick McKeown, and Noa Zilberman. *The P4->NetFPGA Workflow for Line-Rate Packet Processing*. FPGA, 2019.

[47] IEEE. *Standard for local and metropolitan area networks, Virtual Bridged Local Area Networks.* IEEE Standard 802.11Q-2005, 2005.

[48] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. *Silo: Predictable Message Latency in the Cloud*. SIGCOMM, 2015.

[49] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. *mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems*. NSDI, 2014.

[50] Bo Ji, Changhee Joo, and Ness B. Shroff. *Exploring the inefficiency and instability of Back-Pressure algorithms*. INFOCOM, 2013.

[51] Sangeetha Abdu Jyothi, Ankit Singla, Brighten Godfrey, and Alexandra Kolla. *Measuring and Understanding Throughput of Network Topologies*. SC, 2016.

[52] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. *Raising the Bar for Using GPUs in Software Packet Processing*. NSDI, 2015.

[53] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr.Sharma, Arvind Krishnamurthy, and Thomas Anderson. *TAS: TCP Acceleration as an OS Service*. EuroSys, 2019.

[54] 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*. NSDI, 2019.

[55] Jin Hwan Kim, Won Sik Kwon, Hyub Lee, Kyung-Soo Kim, and Soohyun Kim. *A novel method to acquire ring-down interferograms using a double-looped mach-zehnder interferometer*. Conference on Lasers and Electro-Optics (CLEO), 2014.

[56] Ian Kuon and Jonathan Rose. *Measuring the Gap Between FPGAs and ASICs*. Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2007.

[57] Maciej Lapinski, Thomasz Wlostowki, Javier Serrano, and Pablo Alvarez. *White Rabbit: a PTP Application for Robust Sub-nanosecond Synchronization*. Symposium on Precision Clock Synchronization for Measurement Control and Communication, 2011.

[58] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*. SIGCOMM, 2016.

[59] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. *XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers*. NSDI, 2016.

[60] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. *SocksDirect: Datacenter Sockets can be Fast and Compatible*. SIGCOMM, 2019.

[61] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. *Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering*. OSDI, 2016.

[62] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. *HPCC: High Precision Congestion Control*. SIGCOMM, 2019.

[63] Odile Liboiron-Ladouceur, Assaf Shacham, Benjamin A. Small, Benjamin G. Lee, Howard Wang, Caroline P. Lai, Aleksandr Biberman, and Keren Bergman. *The Data Vortex Optical Packet Switched Interconnection Network*. Journal of Lightwave Technology, 2008.

[64] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. *Scalable kernel TCP design and implementation for short- lived connections*. ASPLOS, 2016.

[65] Maciej Lipinski, Tomasz Wlostowski, Javier Serrano, Pablo Alvarez, Juan David Gonzalez Cobas, Alessandro Rubini, and Pedro Moreira. *Performance results of the first White Rabbit installation for CNGS time transfer*. Symposium on Precision Clock Synchronization for Measurement Control and Communication, 2012.

[66] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. *Scheduling techniques for hybrid circuit/packet networks*. CoNEXT, 2015.

[67] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. *Offloading Distributed Applications onto SmartNICs using iPipe*. SIGCOMM, 2019.

[68] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. *DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching*. FAST, 2019.

[69] Tudor Marian, Ki Suh Lee, and Hakim Weatherspoon. *NetSlices: Scalable multi-core packet processing in user-space*. ANCS, 2012.

[70] Ilias Marinos, Robert N. M. Watson, and Mark Handley. *Network stack specialization for performance*. SIGCOMM CCR, 2014.

[71] Rob McGuinness and George Porter. *Evaluating the Performance of Software NICs for 100-Gb/s Datacenter Traffic Control*. ANCS, 2018.

[72] Paul E. McKenney. *Stochastic Fairness Queuing*. INFOCOM, 1990.

[73] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. *Expanding across time to deliver bandwidth efficiency and low latency*. NSDI, 2020.

[74] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. *RotorNet: A Scalable, Low-complexity, Optical Datacenter Network*. SIGCOMM, 2017.

[75] William M. Mellette, Alex C. Snoeren, and George Porter. *P-FatTree: A Multi-channel Datacenter Network Topology*. HotNets, 2016.

[76] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. *Universal Packet Scheduling*. NSDI, 2016.

[77] Radhika Mittal, Terry Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. *TIMELY: RTT-based Congestion Control for the Datacenter*. SIGCOMM, 2015.

[78] Sung-Whan Moon, Jennifer Rexford, and Kang G. Shin. *Scalable hardware priority queue architectures for high-speed packet switches*. Transactions on Computers, 2000.

[79] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. *AccelTCP: Accelerating Network Applications with Stateful TCP Offloading*. NSDI, 2020.

[80] Pedro Moreira, Javier Serrano, Tomasz Wlostowski, Patrick Loschmidt, and Georg Gaderer. *White Rabbit: Sub-Nanosecond Timing Distribution over Ethernet*. Symposium on Precision Clock Synchronization for Measurement Control and Communication, 2009.

[81] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. *Language-Directed Hardware Design for Network Performance Monitoring*. SIGCOMM, 2018.

[82] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. *Scale-out NUMA*. ASPLOS, 2014.

[83] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. *Fastpass: A Centralized "Zero-queue" Datacenter Network*. SIGCOMM, 2014.

[84] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris. *Improving network connection locality on multicore systems*. EuroSys, 2012.

[85] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson. *Arrakis: The Operating System is the Control Plane*. OSDI, 2014.

[86] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. *FlowBlaze: Stateful Packet Processing in Hardware*. NSDI, 2019.

[87] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. *Integrating Microsecond Circuit Switching into the Data Center*. SIGCOMM, 2013.

[88] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. *SENIC: Scalable NIC for End-Host Rate Limiting*. NSDI, 2014.

[89] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. *Improving Datacenter Performance and Robustness with Multipath TCP*. SIGCOMM, 2011.

[90] Mattia Rizzi, Maciej Lipinski, Tomasz Wlostowski, Javier Serrano, Grzegorz Daniluk, Paolo Ferrari, and Stefano Rinaldi. *White Rabbit Clock Characteristics*. Symposium on Precision Clock Synchronization for Measurement Control and Communication, 2016.

[91] Luigi Rizzo. *Netmap: a novel framework for fast packet I/O*. ATC, 2012.

[92] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. *Carousel: Scalable Traffic Shaping at End Hosts*. SIGCOMM, 2017.

[93] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. *Eiffel: Efficient and Flexible Software Packet Scheduling*. NSDI, 2019.

[94] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. *PISCES: A Programmable, Protocol-Independent Software Switch*. SIGCOMM, 2016.

[95] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. *Approximating Fair Queueing on Reconfigurable Switches*. NSDI, 2018.

[96] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. *Programmable Calendar Queues for High-speed Packet Scheduling*. NSDI, 2020.

[97] Kai Shi, Sophie Lange, Istvan Haller, Daniel Cletheroe, Raphael Behrendt, Benn Thomsen, Fotini Karinou, Krzysztof Jozwik, Paolo Costa, and Hitesh Ballani. *System Demonstration of Nanosecond Wavelength Switching with Burst-mode PAM4 Transceiver*. European Conference on Optical Communication (ECOC), 2019.

[98] M. Shreedhar and George Varghese. *Efficient Fair Queuing using Deficit Round Robin*. Transactions on Networking, 1996.

[99] Vishal Shrivastav. *Fast, Scalable, and Programmable Packet Scheduler in Hardware*. SIGCOMM, 2019.

[100] Vishal Shrivastav, Ki Suh Lee, Han Wang, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*. Transactions on Networking, 2019.

[101] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. *Shoal: A Network Architecture for Disaggregated Racks*. NSDI, 2019.

[102] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. *Jellyfish: Networking Data Centers Randomly*. NSDI, 2012.

[103] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown,

and Steve Licking. *Packet Transactions: High-Level Programming for Line-Rate Switches*. SIGCOMM, 2016.

[104] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. *Programmable Packet Scheduling at Line Rate*. SIGCOMM, 2016.

[105] Brent Stephens, Aditya Akella, and Michael Swift. *Loom: Flexible and Efficient NIC Packet Scheduling*. NSDI, 2019.

[106] Francesco Testa and Lorenzo Pavesi. *Optical Switching in Next Generation Data Centers*. Springer, 2017.

[107] Ratko V. Tomic. *Optimal Networks from Error Correcting Codes*. ANCS, 2013.

[108] `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757`. *IEEE Standard 1588-2008*. IEEE, 2008.

[109] `http://bit.ly/2cZXrwo`.
Mellanox Comes Out Swinging With 100G Spectrum Ethernet. The Next Platform Blog, 2015.

[110] `http://bit.ly/2flbOuZ`.
Mellanox ConnectX-4 Ethernet Adapter Card. Mellanox, 2017.

[111] `http://bit.ly/2wPnJHI`.
Broadcom BCM56960 series (Tomahawk). Broadcom, 2020.

[112] `http://bit.ly/2xsIhaU`.
What's the Difference Between Optical and Electrical Technology for 100-Gbit/s Connectivity in Future Systems? Electronic Design, 2014.

[113] `http://de5-net.terasic.com.tw`.
DE5-Net FPGA Development Kit. Terasic, 2020.

[114] `http://f-stack.org/`.
High-Performance Network Framework Based on DPDK. Tencent Cloud, 2020.

[115] `http://format.com.pl/site/wp-content/uploads/2015/09/pb_sn2700.pdf`. *32-port Non-blocking 100GbE Open Ethernet Switch System*. Mellanox Spectrum, 2015.

[116] http://intel.ly/2hrn1hR.
Intel® Xeon Processor D-1500 Family. Intel, 2020.

[117] https://ark.intel.com/content/www/us/en/ark/products/codename/124664/cascade-lake.html#@Server.
Intel Cascade Lake Processors for Servers. Intel, 2020.

[118] https://bluespec.com/.
BSV High-Level HDL. Bluespec, 2020.

[119] https://dpdk.org/.
Data plane development kit. Intel, 2014.

[120] http://seastar.io.
Seastar: High-Performance Server-side Application Framework. ScyllaDB, 2020.

[121] https://en.wikipedia.org/wiki/Amdahl%27s_law.
Amdahl's Law. Wikipedia, 2020.

[122] https://en.wikipedia.org/wiki/Associative_array.
Abstract Dictionary Data Type. Wikipedia, 2020.

[123] https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.
Earliest Deadline First. Wikipedia, 2020.

[124] https://en.wikipedia.org/wiki/Expander_graph.
Expander graph. Wikipedia, 2020.

[125] https://en.wikipedia.org/wiki/Explicit_Congestion_Notification.
Explicit Congestion Notification. Wikipedia, 2020.

[126] https://en.wikipedia.org/wiki/Global_Internet_usage.
Global Internet Usage. Wikipedia, 2020.

[127] https://en.wikipedia.org/wiki/Hash_table.
Hash Table. Wikipedia, 2020.

[128] https://en.wikipedia.org/wiki/Key\OT1\textendashvalue_database.
Key-value Store. Wikipedia, 2020.

[129] https://en.wikipedia.org/wiki/Large_send_offload.
Large send offload. Wikipedia, 2020.

[130] https://en.wikipedia.org/wiki/Least_slack_time_scheduling.
Least Slack Time First. Wikipedia, 2020.

[131] https://en.wikipedia.org/wiki/Moore%27s_law#Forecasts_and_roadmaps.
Forecast of the end of Moore's law. Wikipedia, 2020.

[132] https://en.wikipedia.org/wiki/OpenCL.
openCL. Wikipedia, 2020.

[133] https://en.wikipedia.org/wiki/OSI_model.
OSI Model. Wikipedia, 2020.

[134] https://en.wikipedia.org/wiki/Priority_queue.
Priority Queue. Wikipedia, 2020.

[135] https://en.wikipedia.org/wiki/Remote_direct_memory_access.
Remote direct memory access. Wikipedia, 2020.

[136] https://en.wikipedia.org/wiki/Search_tree.
Search Tree. Wikipedia, 2020.

[137] https://en.wikipedia.org/wiki/Shortest_job_next.
Shortest Job First. Wikipedia, 2020.

[138] https://en.wikipedia.org/wiki/Shortest_remaining_time.
Shortest Remaining Time First. Wikipedia, 2020.

[139] https://en.wikipedia.org/wiki/SystemVerilog.
System Verilog. Wikipedia, 2020.

[140] https://en.wikipedia.org/wiki/Token_bucket.
Token Bucket. Wikipedia, 2020.

[141] https://github.com/mellanox/libvma.
Messaging Accelerator (VMA). GitHub, 2020.

[142] https://github.com/programmable-scheduling/pifo-hardware/blob/master/src/rtl/design/pifo.v.
PIFO implementation. GitHub, 2020.

[143] https://investors.broadcom.com/news-releases/news-release-details/broadcom-ships-tomahawk-4-industrys-highest-bandwidth-ethernet.
Broadcom Tomahawk. Broadcom, 2019.

[144] https://lwn.net/Articles/726917/.
Zero-copy networking. Jonathan Corbet, 2017.

[145] https://numato.com/blog/differences-between-fpga-and-asics/.
FPGA Vs ASIC: Differences Between Them And Which One To Use? Numato Lab, 2018.

[146] https://p4.org/assets/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf.
Data Plane Programming at Terabit speeds. Barefoot Networks (an Intel company), 2017.

[147] https://www.ariacybersecurity.com/network-adapters/software/dbl/.
Myricom DBL. CSP Inc., 2020.

[148] https://www.barefootnetworks.com.
Tofino Switch. Barefoot Networks (an Intel company), 2020.

[149] https://www.barefootnetworks.com/products/brief-tofino-2/.
Tofino Switches. Barefoot Networks (an Intel company), 2020.

[150] https://www.cdwg.com/product/Myricom-Myri-10G-SFP-transceiver-module-10-Gigabit-Ethernet/1818816?pfm=srh.
Myricom Myri-10G - SFP+ transceiver module - 10 GigE. CDW-G, 2020.

[151] https://www.cdwg.com/product/Tripp-Lite-2M-Duplex-Singlemode-Fiber-8.3-125-Patch-Cable-LC-LC-6ft/717519?
Tripp Lite 2M Duplex Singlemode Fiber. CDW-G, 2020.

[152] https://www.eetimes.com/broadcom-ships-25-6tbps-switch-on-single-7nm-chip/.
Broadcom Ships 25.6Tbps Switch on Single 7nm Chip. Electronic Engineering Times (EE Times), 2019.

[153] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf.
Stratix 10 FPGA. Intel, 2018.

[154] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf.
Stratix V FPGA. Intel, 2015.

[155] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html.
Quartus Prime Software. Intel, 2020.

[156] https://www.kernel.org/doc/Documentation/networking/scaling.txt. *Receive side scaling*. Kernel.org, 2020.

[157] https://www.labs.hpe.com/the-machine.
The Machine. HP Labs, 2020.

[158] https://www.mediapost.com/publications/article/291358/90-of-todays-data-created-in-two-years.html.
90% Of Today's Data Created In Two Years. MediaPost, 2016.

[159] https://www.mellanox.com/products/ethernet/connectx-smartnic. *Mellanox ConnectX Ethernet Adapters*. Mellanox, 2020.

[160] https://www.onestopsystems.com/product/gen3-4u-value-16-slot-expansion-system.
PCIe expansion system. One Stop Systems, 2020.

[161] https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.
The Digitization of the World From Edge to Core. IDC, 2018.

[162] https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. *Number of monthly active Facebook users worldwide*. Statista, 2020.

[163] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=163#Category164. *DE Board Series (Stratix)*. Terasic, 2020.

[164] https://www.xilinx.com/support/documentation/ip_documentation/l_ethernet/v1_0/pg211-50g-ethernet.pdf. *40G/50G High Speed Ethernet Subsystem v1.0*. Xilinx, 2016.

[165] http://www.ieee802.org/1/pages/802.1bb.html. *Priority-based Flow Control*. IEEE DCB. 802.1Qbb, 2011.

[166] http://www.itu.int/rec/T-REC-G.8262. *ITU-T Rec. G.8262*. International Telecommunication Union, 2008.

[167] http://www.macom.com/products/product-detail/M21605/. *M21605 Crosspoint Switch*. Macom, 2020.

[168] http://www.mellanox.com/page/products_dyn?product_family=79&mtag=roce. *RDMA and RoCE for Ethernet Network Efficiency Performance*. Mellanox, 2020.

[169] http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf. *Introduction to OpenOnload: Building Application Transparency and Protocol Conformance into Application Acceleration Middleware*. Solarflare Communications, 2011.

[170] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. *Xpander: Towards Optimal-Performance Datacenters*. CoNext, 2016.

[171] Leslie G. Valiant and Gordon J. Brebner. *Universal schemes for parallel communication*. STOC, 1981.

[172] George Varghese and Anthony Lauck. *Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility*. SOSP, 1987.

[173] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. *Practical TDMA for datacenter ethernet*. EuroSys, 2012.

[174] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. *Costly circuits, submodular schedules and approximate caratheodory theorems*. SIGMETRICS, 2016.

[175] Guohui Wang, David G. Andersen, Michael Kaminsky, Michael Kozuch, T. S. Eugene Ng, Konstantina Papagiannaki, and Michael Ryan. *c-Through: Part-time Optics in Data Centers*. SIGCOMM, 2010.

[176] Han Wang, Robert Soule, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. *P4FPGA: A Rapid Prototyping Framework for P4*. SOSR, 2017.

[177] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. *Better Never Than Late: Meeting Deadlines in Datacenter Networks*. SIGCOMM, 2011.

[178] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. *ICTCP: Incast Congestion Control for TCP in Data Center Networks*. CoNEXT, 2010.

[179] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. *NetChain: Scale-Free Sub-RTT Coordination*. NSDI, 2018.

[180] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. *NetCache: Balancing Key-Value Stores with Fast In-Network Caching*. SOSP, 2017.

[181] Jun Xu and Richard J. Lipton. *On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms*. SIGCOMM, 2002.

[182] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. *StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs*. ATC, 2016.

[183] Xiaohui Ye, Yawei Yin, S. J. B. Yoo, Paul Mejia, Roberto Proietti, and Venkatesh Akella. *DOS - A scalable optical switch for datacenters*. ANCS, 2010.

[184] Hui Zhang and Domenico Ferrari. *Rate-Controlled Service Disciplines*. Journal of High Speed Networks, 1994.

[185] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. *Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers*. SIGCOMM, 2012.

[186] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. *Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection*. VLDB, 2019.

[187] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. *Congestion Control for Large-Scale RDMA Deployments*. SIGCOMM, 2015.