UNIVERSITY OF CALIFORNIA SAN DIEGO

System-Level Considerations for Optical Switching in Data Center Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering (Electronic Circuits and Systems)

by

John Alexander Forencich

Committee in charge:

Professor George C. Papen, Chair
Professor Joseph E. Ford
Professor Shayan Mookherjea
Professor George M. Porter
Professor Alex C. Snoeren

2020

The dissertation of John Alexander Forencich is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California San Diego

2020

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I consider myself fortunate to have had the opportunity to work with many talented people over the past few years. I would first like to thank my advisor, George Papen, for his guidance, insight, and support over the course of my graduate career, not to mention his original invitation to work on Mordia as an undergrad, without which I would not be where I am today. I would also like to thank the other members of my committee: Joseph Ford, Shayan Mookherjea, George Porter, and Alex Snoeren for investing their time and effort. Although George Papen is my sole advisor, I have also worked closely with George Porter and Alex Snoeren, who have helped me broaden the scope of my research with their expertise in the systems aspects of computer networking.

To Moein Khazraee, it has been a pleasure working with you on Dastgāh and other FPGA-related projects. Corundum would not be in its current state without all of the bugs found and fixed during integration.

To Laurent Schares, Nicolas Dupuis, Ben Lee, Chris Baks, Dan Kuchta, Fuad Doany, and Valerija Kamchevska, it was a pleasure working with all of you at IBM research. I am thankful for the opportunity to work in the lab there.

To Max Mellette, Rob McGuinness, and Arjun Roy, it was a pleasure working with all of you on RotorNet. To Hannah Grant and Ryan Aguinaldo, it was a pleasure working with you in the chip-scale testing lab. To He Liu, Feng Lu, Rishi Kapoor, and Malveeka Tewari, it was a pleasure working with all of you on REACToR. To Nathan Farrington and Pang-Chen Sun, it was a pleasure working with you on Mordia.

Research in the field of computer science is seldom a solo activity. I would like to thank my numerous collaborators, who are listed next.

Chapter 2, in part, reprints material as it appears in the paper titled: "A dynamically-reconfigurable burst-mode link using a nanosecond photonic switch," published in the *Journal of Lightwave Technology*, 2020, by Alex Forencich, Valerija Kamchevska, Nicolas Dupuis,

Benjamin G. Lee, Christian Baks, George Papen, and Laurent Schares. The dissertation author was the primary researcher and author of this material.

Chapter 3, in part, reprints material that has been submitted for publication in a paper titled: "Corundum: An Open-Source 100-Gbps NIC," submitted to the FCCM conference, by Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. The dissertation author was the primary researcher and author of this material.

VITA

2012	Bachelor of Science in Electrical Engineering, University of California San Diego
2016	Master of Science in Electrical Engineering (Electronic Circuits and Systems), University of California San Diego
2020	Doctor of Philosophy in Electrical Engineering (Electronic Circuits and Systems), University of California San Diego

JOURNAL PUBLICATIONS

Alex Forencich, Valerija Kamchevska, Nicolas Dupuis, Benjamin G. Lee, Christian Baks, George Papen, and Laurent Schares. A dynamically-reconfigurable burst-mode link using a nanosecond photonic switch. *Journal of Lightwave Technology*, 38(6):1330–1340, March 2020.

How Yuan Hwang, Jun Su Lee, Tae Joon Seok, Alex Forencich, Hannah R. Grant, Dylan Knutson, Niels Quack, Sangyoon Han, Richard S. Muller, George C. Papen, Ming C. Wu, and Peter O'Brien. Flip chip packaging of digital silicon photonics MEMS switch for cloud computing and data centre. *IEEE Photonics Journal*, 9(3):1–10, June 2017.

Ryan Aguinaldo, Alex Forencich, Christopher DeRose, Anthony Lentine, Douglas C. Trotter, Yeshaiahu Fainman, George Porter, George Papen, and Shayan Mookherjea. Wideband silicon-photonic thermo-optic switch in a wavelength-division multiplexed ring network. *Optics Express*, 22(7):8205–8218, Apr 2014.

Nathan Farrington, Alex Forencich, George Porter, Pang Chen-Sun, Joseph E. Ford, Yeshaiahu Fainman, George C. Papen, and Amin Vahdat. A multiport microsecond optical circuit switch for data center networking. *IEEE Photonics Technology Letters*, 25(16):1589–1592, Aug 2013.

CONFERENCE PROCEEDINGS

Moein Khazraee, Alex Forencich, George Papen, Alex C. Snoeren, Aaron Schulman, "Dastgāh: Software-defined FPGA SmartNICs", Submitted to *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.

Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: an open-source 100-Gbps NIC. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Fayetteville, AR, May 2020. IEEE.

Alex Forencich, Valerija Kamchevska, Nicolas Dupuis, Benjamin G. Lee, Christian Baks, George Papen, and Laurent Schares. System-level demonstration of a dynamically reconfigured burst-mode link using a nanosecond Si-photonic switch. In *Optical Fiber Communication Conference (OFC)*, page Th1G.4. Optical Society of America, 2018.

William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, page 267–280, New York, NY, USA, 2017. Association for Computing Machinery.

Hannah R. Grant, Alex Forencich, George Papen, Nicolas Dupuis, Laurent Schares, Russell A. Budd, and Benjamin G. Lee. Bit error rate measurements of a 4×4 Si-photonic switch using synchronous and asynchronous data. In *IEEE Optical Interconnects (OI)*, pages 32–33, May 2016.

Tae Joon Seok, How Yuan Hwang, Jun Su Lee, Alex Forencich, Hannah R. Grant, Dylan Knutson, Niels Quack, Sangyoon Han, Richard S. Muller, Lee Carroll, George C. Papen, Peter O'Brien, and Ming C. Wu. 12×12 packaged digital silicon photonic MEMS switches. In *IEEE Photonics Conference*, pages 629–630, Oct 2016.

He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, NSDI'14, pages 1–15, Berkeley, CA, USA, 2014. USENIX Association.

Ryan Aguinaldo, Alex Forencich, Christopher DeRose, Anthony Lentine, Andrew Starbuck, Yeshaiahu Fainman, George Porter, George Papen, and Shayan Mookherjea. Characterization of a silicon-photonic wideband switch in UCSD's MORDIA ring network. In *IEEE Optical Interconnects (OI)*, pages 102–103, May 2014.

Ryan Aguinaldo, Alex Forencich, Christopher DeRose, Anthony Lentine, Douglas C. Trotter, Andrew Starbuck, Yeshaiahu Fainman, George Porter, George Papen, and Shayan Mookherjea. Energy-efficient, digitally-driven "fat pipe" silicon photonic circuit switch in the UCSD MORDIA data-center network. In *Conference on Lasers and Electro-Optics (CLEO)*, pages 1–2, June 2014.

Nathan Farrington, Alex Forencich, Pang-Chen Sun, Shaya Fainman, Joe Ford, Amin Vahdat, George Porter, and George C Papen. A 10 us hybrid optical-circuit/electrical-packet network for datacenters. In *Optical Fiber Communication Conference (OFC)*, pages OW3H–3. Optical Society of America, 2013.

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. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, page 447–458, New York, NY, USA, 2013. Association for Computing Machinery.

Nathan Farrington, George Porter, Alex Forencich, Joseph Ford, Yeshaiahu Fainman, Amin Vahdat, and George Papen. Optical/electrical hybrid switching for datacenter communications. In *OptoElectronics and Communications Conference (OECC/PS)*, pages 1–2, June 2013.

Nathan Farrington, George Porter, Pang-Chen Sun, Alex Forencich, Joseph Ford, Yeshaiahu Fainman, George Papen, and Amin Vahdat. A demonstration of ultra-low-latency data center optical circuit switching. *SIGCOMM Comput. Commun. Rev.*, 42(4):95–96, August 2012.

ABSTRACT OF THE DISSERTATION

System-Level Considerations for Optical Switching in Data Center Networks

by

John Alexander Forencich

Doctor of Philosophy in Electrical Engineering (Electronic Circuits and Systems)

University of California San Diego, 2020

Professor George C. Papen, Chair

Traditionally, datacenter networks are constructed with electronic packet switches, which forward data in a distributed manner. However, packet switches are running in to serious scaling challenges due to the massive growth in bandwidth requirements within major datacenters. One possible solution to this situation is to use optical circuit switching, which has significant potential for both power and cost savings.

However, circuit switching is a completely different paradigm from packet switching, and as such effectively utilizing circuit switching requires work at all layers in the network, including the network architecture, switches, transceivers, network interfaces, and protocols.

This dissertation validates the thesis that developing a NIC capable of precision admission

control and characterizing its performance can lead to practical sub-microsecond circuit-switched networks at scale. Specifically, this dissertation presents novel contributions to two key system-level issues inherent in utilizing optical circuit switching in datacenter networking applications.

The first contribution of this dissertation quantifies the system-level reconfiguration time of an optically circuit-switched link. This work is critical for understanding how to build optical links using high speed optical switches and how to integrate these links into a datacenter environment. This includes a discussion of system-level reconfiguration time and link-level measurements of an optically switched link, including bit error rate (BER) characterization of a 25 Gbps link utilizing a burst-mode receiver, switched by a nanosecond-scale silicon photonic switch.

The second contribution of this dissertation is the development of a network interface controller (NIC) that can precisely control the injection of packets into an optically-switched network. This work describes and quantifies the performance of a novel high-performance, open-source, FPGA-based NIC called Corundum. This NIC is designed to precisely control the injection of packets from multiple queues into a circuit-switched network using a hardware-based scheduler. The platform provides the flexibility to implement high precision time synchronization as well as perform link-level characterization including BER measurements. The development of such a network interface can lead to practical sub-microsecond circuit-switched networks at scale.

Chapter 1

Introduction

The web applications and cloud services that underpin the modern Internet are provided by massive distributed systems running on thousands of computers inside of hyperscale datacenters. These datacenters are built and operated by service providers such as Google, Microsoft, Facebook, Amazon, and many others. A modern datacenter can contain tens to hundreds of thousands of hosts, interconnected by a high-performance network.

Traditionally, datacenter networks are constructed with electronic packet switches. Packet switches operate individually on each packet, making routing decisions and forwarding each packet accordingly. One major advantage of packet switches is that they are distributed: no centralized scheduler or similar is required to control traffic in the network, making it easy for very large networks to be constructed with packet switches. State-of-the-art electronic packet switches are built around highly-integrated switch ASICs as this is currently only way to provide sufficient bandwidth to interconnect all of the ports.

However, packet switches are running in to serious scaling challenges. Bandwidth demand and the number of hosts inside of datacenter networks is increasing much faster than switch ASIC bandwidth. Additionally, the number of ports on a single switch ASIC is limited, so using packet switches in modern datacenter networks involves building many layers of switches in order to

provide sufficient ports and sufficient bandwidth. Due to the link rates and distances involved, many of the connections between packet switches are optical, which requires the use of large numbers of power-hungry optical transceivers. As a result, as network bandwidth increases, more and more cost and power must be devoted to the network.

One possible solution to this situation is to use optical circuit switching. Optical circuit switches operate by making and breaking connections at the physical layer. As a result, the operation of such a switch is independent of the line rate and modulation format, enabling switching of very large bandwidths per port. Additionally, optical switches consume a fraction of the power of an electrical switch of similar port count and bandwidth. Since optical switches operate in the optical domain, they also reduce the number of required optical transceivers in the network. Therefore, utilizing optical circuit switching inside a datacenter network has significant potential for both power and cost savings.

However, circuit switching is a completely different paradigm from packet switching. In a packet-switched network, hosts can send data at any time, and the packet switches route it accordingly, storing data temporarily in the case of contention. In a circuit-switched network, the switches form dedicated connections between end points in the network, and data transmissions must be coordinated so that data follows the correct paths through the switches. Effectively utilizing circuit switching requires work at all layers in the network—not just the network architecture and the design of the circuit switches, but also the design of the transceivers, network interfaces, and protocols.

This dissertation validates the thesis that developing a NIC capable of precision admission control and characterizing its performance can lead to practical sub-microsecond circuit-switched networks at scale. Specifically, this dissertation presents novel contributions to two key system-level issues inherent in using optical circuit switching in datacenter networking applications. The first issue is quantifying the system-level reconfiguration time of an optically circuit-switched link. This work is critical for understanding how to build optical links using high speed optical

switches and how to integrate these links into a datacenter environment.

This work is presented in Chapter 2 and is based on a recently published paper [19]. That chapter describes novel link-level measurements on an optically switched link, including bit error rate (BER) characterization of a 25 Gbps link utilizing a burst-mode receiver, switched by a nanosecond scale silicon photonic switch. Key parameters that affect system-level performance are quantified, including switching time, receiver lock time, and guard intervals.

The second key issue addressed in this dissertation is the development of a network interface controller (NIC) that can precisely control the injection of packets into one or more optically-switched networks.

This work is presented in Chapter 3, and describes and quantifies the performance of an FPGA-based NIC called Corundum. This NIC is designed to precisely control the injection of packets from multiple queues into a circuit-switched network under the control of a hardware-based scheduler. The work described in this chapter has been accepted for publication [20].

At a high-level, Corundum is a high-performance, open-source, FPGA-based NIC development platform that can support the development of network protocols and architectures, including practical optically-switched networks. Corundum provides the ability to precisely control packet transmission into the network on microsecond timescales, enabling the use of microsecond-scale optical switches in an otherwise standard datacenter environment. The platform provides the flexibility to implement high precision time synchronization as well as perform link-level characterization including BER measurements similar to those described in Chapter 2. The ability to quantify the low-level physical layer characteristics of an optically-switched link and provide precision transmission control over thousands of queues all within a common hardware platform is a fundamental research contribution of this dissertation. The development of such a network interface can lead to practical sub-microsecond circuit-switched networks at scale.

1.1 Background

The motivation for the work presented in this dissertation is perhaps best understood by providing some historical background on the networking research conducted by our research group. This background will also highlight my other research contributions, which are not explicitly discussed in this dissertation.

The optical networking group at UCSD has been exploring the use of circuit switching, especially optical domain switching, for datacenter networking applications for several years. Since circuit switching is a completely different paradigm from the packet switching commonly used in computer networks, it is necessary to do a significant amount of cross-layer optimization and co-design in order to build an effective system. As a result, the research in our group spans the entire stack, from high-level network architecture and protocols down to the physical layer and the design of the optical switches and transceivers. The general theme of many of the research projects is to try to use as much commercially-available hardware as possible in order to emulate a relatively realistic datacenter environment at a reasonable scale. This section will present several of those projects in the context of the lessons learned and my contributions to those projects. These lessons were the principal motivation for the research topics of my dissertation.

1.1.1 Helios

Helios [14] is some of the early work at UCSD in the domain of optical switching in datacenter networks. Helios utilized a Glimmerglass 3D-MEMS-based optical crossbar switch to interconnect programmable ToR switches. The switch used has a switching time of around 12 milliseconds. Software running in the network monitored flow-level information collected by the packet switches and determined which flows could take advantage of a direct high-bandwidth optical path, automatically reconfiguring the MEMS switch and packet switches as necessary, with a response time in the hundreds of milliseconds. This formed the “elephants and mice” era

of optical switching—find the biggest, longest-lived flows, and offload them from the electrical network and onto the optical network over timescales of seconds.

However, the real-world applications of such a strategy are somewhat limited. The vast majority of flows in datacenter networks are too short to be handled by a millisecond-scale optical switch. The faster the switch, the more traffic could be offloaded to it.

The key lesson learned from Helios is that while optical switching has merit, it must be significantly more responsive in order to be compatible with more realistic workloads.

1.1.2 Mordia

Following Helios, the goal of Mordia [15, 11, 12, 13, 43] was to build a network using an optical switch 1000 times faster, switching on the order of microseconds. The Mordia network was built using binary-MEMS-based wavelength selective switches with a switching time on the order of 12 microseconds. 24 end-hosts equipped with DWDM transceivers were connected in groups of four to a fiber optic ring containing add/drop filters and amplifiers. The add/drop filters injected light from the hosts into the ring, and the wavelength selective switches pulled it back out and routed it to the receivers. The switches were controlled by an FPGA and could be reconfigured every 100 microseconds. One lesson from the development of Mordia was that even though the optical switches could be reconfigured quickly, the end hosts could not keep up with the changing network configuration. The network stack and NICs could not reliably control packet transmission into the network on a microsecond time scale so that packets would reliably arrive at the correct destination. Reliable transmission could only be achieved if a guard interval significantly larger than the switching time was included in the schedule to account for this uncertainty. We also discovered that the standard software algorithms for computing schedules for crossbar switches have serious scaling limitations.

The key take away from Mordia is that building a very fast optical switch is not sufficient. It is imperative to use proper admission control at the end hosts that can operate reliably on the

same time scale as the reconfiguration time of the switches. Otherwise, the reconfiguration time is dominated by the guard time that must added to ensure reliable operation.

My contributions to Mordia, which were done while I was an undergraduate, are given in references [15, 11, 12, 13, 43]. For this project, I developed the variable optical attenuator (VOA) driver boards, associated firmware, and control software to balance optical power within the Mordia ring. I also developed tools for the characterization of the switching speed of the wavelength-selective switches (WSS) modules and the lock time of the DWDM transceivers. This work formed the foundation for the system-level reconfiguration time measurements presented in this dissertation.

1.1.3 REACToR

The goal of REACToR [30] was to build a control plane for Mordia. To do so, one of the main goals was to develop some form of high-precision control over packet transmission from the end hosts. This required inserting an FPGA into each link to inject priority flow control (PFC) pause frames to control the NIC transmit queues. While this methods was effective, it does not scale beyond 8 end hosts as standard PFC can only control 8 independent traffic classes. Additionally, there were complications within the network stack where packets would get stuck in NIC queues when paths were de-scheduled. The Solstice scheduling algorithm was also developed for REACToR as an improvement on existing scheduling techniques. However, the computational complexity of Solstice precluded its implementation at datacenter scale.

The lesson learned from REACToR is that hardware-based admission control at the NIC works very well, but commercial NICs do not provide sufficiently flexible and scalable control over packet transmission. Additionally, building a scalable reactive control plane including demand data collection, schedule computation, and distribution is a nontrivial problem.

My contributions to REACToR included the directing the layout of a circuit board for a 48x48 port 10G electrical crosspoint switch. I also developed and integrated the FPGA code for

controlling both the electrical crosspoint switch and the Mordia switch to test REACToR and provide FPGA-based performance measurements.

1.1.4 Optical Component Characterization

Along with the lessons learned from Mordia at the system level, we determined that we needed create tools to facilitate system-level evaluation of optical components for circuit-switched networks. These components include switches, transceivers, add/drop filters, and other components. In order to determine how these components behave in a circuit-switched environment, I developed measurement methods and tools appropriate for this application.

Some of the work in component characterization was part of a collaboration with IBM. As part of this collaboration, we received a number of high-speed silicon photonic switch chips for testing. The switches are built from Mach-Zehnder interferometers with PIN diode phase shifters. They are capable of switching in a few nanoseconds, about 1000 times faster than the MEMS-based wavelength selective switches used in Mordia. However, the devices are polarization sensitive, operate over a narrow wavelength band, and suffer from crosstalk and high coupling loss. Our group carried out BER measurements of the switch chips to evaluate the crosstalk performance [21]. My contribution to this effort was the construction of a custom FPGA-based multi-channel BER measurement tool that generated PRBS data to send through a component under test, counted the errors, and forwarded this to a control computer.

This measurement technique was then adapted to components [1] that were specifically designed to operate in the Mordia architecture and replace several discrete components with a single integrated component.

The custom FPGA-based multi-channel BER measurement tool was then used as part of a collaboration with Berkeley. As part of this collaboration, we received a packaged MEMS-based crossbar switch chip for testing in our lab. The switch chip is a 12x12 MEMS-based crossbar switch with a switching time under 1 microsecond [25, 50]. The switch is low loss and low

crosstalk and is capable of scaling to rather large port counts. However, a key lesson learned from this work was system-level cost of optical loss in the switch. In order to provide enough link margin to make BER measurements, we had to use expensive transceivers with avalanche photodiodes (APD) designed for long-range links.

1.1.5 RotorNet

Returning to system-level issues, the goal of RotorNet [37] was to build an optical switch with limited connectivity, but fast and deterministic switching between the limited number of network topologies. The development of this switch was the outcome of a study [36] that concluded that it is extremely difficult to build a MEMS-based optical switch that simultaneously provides low loss, low crosstalk, large port count, fast switching speed, and full crossbar connectivity. The selector switch project gives up full crossbar connectivity to provide a fast, low loss, low crosstalk switch that can scale to large port counts, but can only select from a limited set of fixed configurations.

The first selector switch prototype was built from a 3D MEMS chip that can switch in 100 microseconds, redirecting light from a common fiber array to one of four other fiber arrays that are looped back in fixed connection patterns. An FPGA connected a control computer to the 3D MEMS chip, permitting low latency control of the switch in synchronization with the end hosts. Custom qdisc kernel modules were used to control the flow of traffic from the end hosts, which were synchronized to each other and to the switch with PTP. The software-based admission control was able to keep up reasonably well with the relatively slow optical switch. However achieving the required timing accuracy required dedicating CPU cores to control the flow of data without being disturbed by the operating system.

Additionally, the initial design of RotorNet used a distributed algorithm to coordinate transfers between nodes. RotorNet is also an 'all-optical' network design; no parallel electrical network is necessary to handle flows separately from the optical network, as was the case in

Helios, Mordia, and REACToR.

I had several contributions to RotorNet. First, I developed an FPGA-based packet timestamping system for evaluating the accuracy of software-based transmission control techniques. Second, I developed a switch control FPGA image that received commands from a control computer over Ethernet and drive DACs on the high voltage MEMS drive board.

1.1.6 IBM Burst-Mode Link Measurements

The initial collaboration with IBM led to a series of internships where I assembled and characterized the system-level reconfiguration time of a burst-mode link through a silicon photonic switch chip. This link used two different IBM devices—a silicon photonic switch chip and a burst-mode receiver. Based on the prior work discussed in this section, I developed an FPGA design to control the components, generate test data, and count bit errors. This work included modules to control the switch chip, generate the necessary bit patterns, accumulate bit errors, and perform frame synchronization. I also assembled the test setup including both chips, the FPGA, and all of the necessary test equipment and carried out all of the measurements. This work produced two first-author papers [18, 19] and is discussed in detail in Chapter 2.

1.1.7 Corundum NIC Prototyping Platform

Building on the lessons learned in Mordia, REACToR, and RotorNet with respect to admission control techniques, I initiated work on a hardware admission control solution, which is called Corundum. Corundum provides precise hardware control over packet transmission, with a host interface and level of performance similar to that of a commercial NIC. Additionally, it provides direct access to physical layer components, enabling link-level measurements at datacenter scale. This work is has been accepted for publication at FCCM [20], and is discussed in detail in Chapter 3.

1.1.8 Summary

This background section presented a historical progression of motivations, results, and lessons learned from several system-level testbeds and methods of link-level characterization carried out by our group over several years. This section also discussed my contributions to those efforts. The topics presented in this dissertation, including the burst-mode link measurements and development of Corundum, are an outcome of the lessons learned and are a logical progression of this research.

Chapter 2

Link-level Considerations

This chapter quantifies various system-level and physical layer aspects of optically-switched links. Namely, it introduces the concept of *system-level reconfiguration time*, which includes not only the physical reconfiguration time of the switch, but also time constants associated with the receivers and system time synchronization, all of which contribute to the duration that the links are interrupted during each circuit-switch reconfiguration event.

These reconfiguration events affect all aspects of the network from the physical layer through the networking protocols. This leads to the end-to-end latency in a circuit-switched network being different than the end-to-end latency in a packet-switched network. Designing optimized optically-circuit-switched networks requires quantifying the system-level reconfiguration time with respect to other forms of latency within the network.

This chapter experimentally quantifies the system-level reconfiguration time of a burst-mode optical link through a nanosecond-scale silicon photonic switch to determine realistic estimates on how fast the physical layer can re-synchronize after each switch reconfiguration. The material is based on a recently published paper [19].

Chapter 3 follows up with a discussion of network interfaces that can provide the necessary functionality of precise packet injection to reduce the system-level reconfiguration time.

2.1 Introduction and Background

Bandwidth requirements within modern hyperscale datacenters have been increasing exponentially, approximately doubling every year [54]. Traffic within some hyperscale datacenters can exceed the backbone traffic of the Internet itself. Additionally, high-performance computing (HPC) systems also continue to scale [59], and many large-scale applications are limited by bandwidth bottlenecks in various interfaces of the system. Typically, the requirements of different workloads are not perfectly matched to a fixed set of computing hardware resources. Disaggregation of compute resources through a very high bandwidth scale-up network has significant promise to improve resource utilization. In summary, building networks capable of meeting the demands of both the hyperscale and scale-up systems is exceedingly difficult.

Modern datacenter and HPC networks are built from electrical packet switches. Electrical packet switches route packets from a source to a destination through a network in a distributed, scalable fashion by forwarding packets peer-to-peer through the network. Packet switches operate by storing packets in queues and using packet header information to make local routing decisions based on information in routing tables.

Alternative network designs based on aspects of circuit switching such as time-division multiple access (TDMA) are being considered [60]. Optical circuit switches work by making and breaking light paths through the switch. Switching in the optical domain enables rate agnostic switching, where the data rate per port is not tied to a specific line rate or modulation format. This work motivates research in datacenter networks with high-bandwidth optical circuit switches that have the potential to address some of the issues in existing datacenter networks.

The use of active optical switch technologies within a datacenter has several challenges. Due to the stateless nature of optical circuit switches, it is not possible for an optical circuit switch to store and forward packets or to make routing decisions based on information extracted from the packets themselves. Instead, data must be routed end-to-end through all of the switches

along the path. Therefore, successfully utilizing optical circuit switches in a scale-out network requires significant coordination at the link level that is not required in a standard packet-switched network. Scheduling algorithms must be used to determine switch configurations and decide which data will be sent at what time [17, 31], while synchronization techniques must be used to ensure that data successfully arrives at the correct destination. Other methods rely on variations of TDMA and do not require an explicit schedule [37, 35]. Scheduling may also be less of an issue in scale-up or disaggregated networks with fewer, newer compute nodes. In either case, the synchronization issue must be solved.

Optical circuit switches also have their own physical-layer scaling challenges. Depending on the switch technology, there are numerous trade-offs between the number of switch ports, the reconfiguration speed of the switch, and optical properties such as insertion loss, crosstalk, optical bandwidth, and noise [27]. While insertion loss limits the number of switches that can be cascaded before requiring optical amplification and the associated increase in cost, complexity, power consumption, and optical noise, multiple recent demonstrations have shown that the flip-chip integration of semiconductor optical amplifiers (SOA) on silicon photonic switches have the potential to overcome the link margin challenges [26, 7]. The speed and optical properties of a switch will also vary with the port count. Larger switches can be built as cascades of smaller switches at the expense of insertion loss and crosstalk.

Because circuit-switched networks operate by switching physical connections instead of packets, the speed at which the switches can reconfigure at the link-level is a key performance metric. Network traffic inside a datacenter consists of flows of various sizes. Faster reconfiguration of link-level circuits can route smaller blocks of data, enabling a larger fraction of the total network load to be optically circuit switched. This is illustrated in Fig. 2.1, which shows the switch throughput as a function of data transfer size for different end-to-end switching times, highlighting the importance of fast end-to-end reconfiguration when operating on packets or small flows.

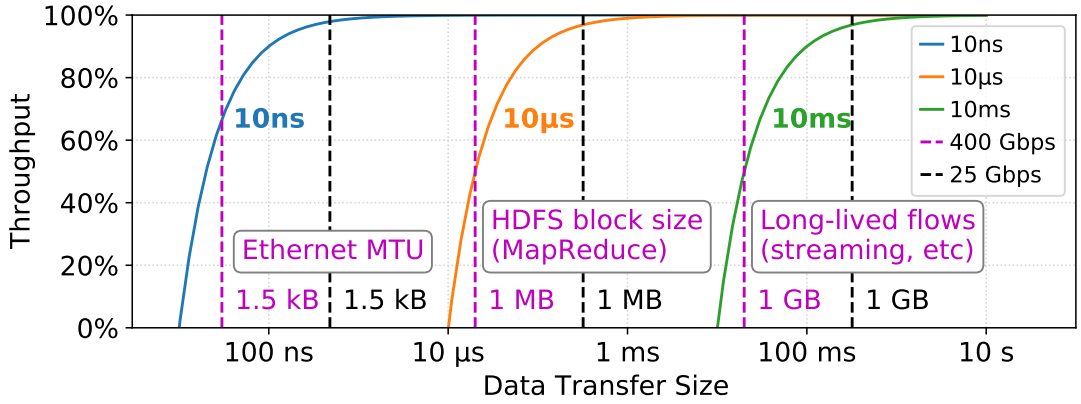


Figure 2.1: Illustration of circuit switch throughput as a function of data transfer size for different end-to-end switching times (blue: 10 ns, orange: 10 us, green: 10 ms). The vertical dashed lines related selected data sizes to link speeds (purple: 400 Gbps, black: 25 Gbps)

Faster circuit switching also requires tighter coordination between network components. In order to efficiently utilize a nanosecond-scale optical circuit switch in a network, fast-locking or burst-mode receivers are also required for link retiming and retraining. Burst-mode (BM) receivers that operate over a large range of input power levels and have locking times similar to switch reconfiguration times are critical to rapidly recover the clock and data (CDR) after an interruption in the link [49].

In this chapter, we expand and refine the preliminary results from our earlier work in [18], demonstrating a switched optical link consisting of a combination of a nanosecond silicon photonic optical switch [9] and a burst-mode optical receiver [47], coordinated by an FPGA .

2.2 Optically-Switched Architecture and Systems

A notional system-level figure of a photonic-switched network is shown in Fig. 2.2. Data from one processing node such as a network interface card (NIC) is routed to a second processing node through a series of optical circuit switches, which are coordinated by a control plane. In a conventional implementation, the control plane computes a schedule for data transmission, transfers this schedule to all the transmitters and optical switches, and synchronizes all of

the components in the network. Admission control components at end hosts report demand information to the control plane and hold data for transmission until triggered to send it by the scheduler. For maximum throughput, burst-mode data transmission is required, as will be shown in Section III. Burst-mode data transmission requires a burst-mode receiver that is designed to rapidly lock onto the incoming data stream as well as a transmitter that sends the necessary preamble signal at the correct time to facilitate the operation of the burst-mode receiver.

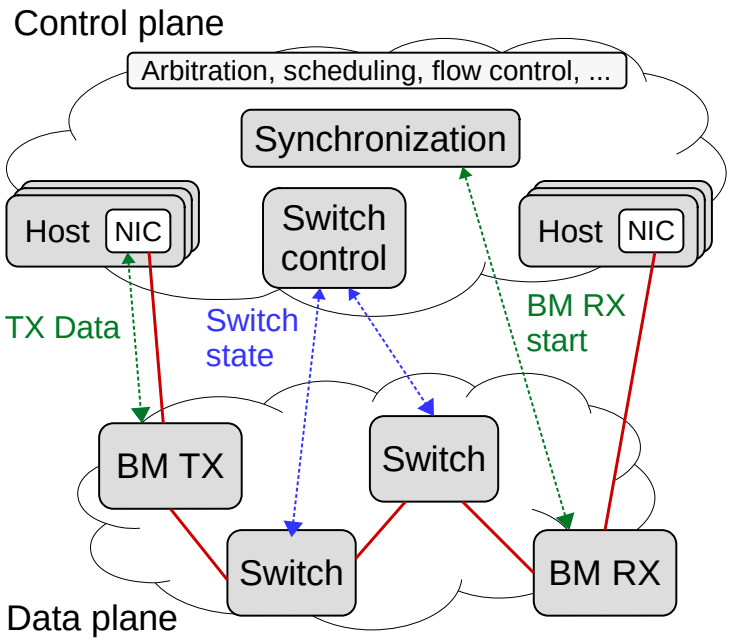


Figure 2.2: Notional schematic of future photonic switch system integration.

The scheduler in the control plane uses a scheduling algorithm to determine how to configure switches and when to transmit data. Various techniques for computing a schedule exist, but the scaling properties of these algorithms can be problematic, especially at hyperscale datacenter scales [30, 31]. Operating a global scheduler also requires collecting a set of demand information and distributing a schedule based on that information. These operations can require a significant amount of bandwidth at large scales. Minimizing the latency of these components is critical for enabling high network performance.

2.2.1 Optical Switches

There are a variety of methods for implementing routing in the optical domain [31, 35, 14, 12]. One method is to route different wavelengths with passive components such as array waveguide gratings (AWGs) and fast tunable lasers [22]. Another method is to use active switching technologies with fixed-wavelength transmitters. This chapter focuses on the latter technique.

Switches based on free-space microelectromechanical systems (MEMS) technology can scale to large port counts with low loss, but suffer from large size, slow switching speed, and high actuation voltage [36]. Semiconductor optical amplifier (SOA) based switches have been studied, but suffer from noise and nonlinearities in the amplifiers as well as polarization sensitivity [55]. Thermo-optic Mach–Zehnder (MZ) interferometers and ring resonators may be suitable, but the time constants associated with the thermo-optic effect limit switching speeds [58]. A switch design based on MEMS actuated waveguides [23] shows promise for sub microsecond switching times coupled with good optical performance and reasonable scaling properties.

Switches based on silicon photonics are capable of switching on much shorter timescales. Electro-optic ring resonators and Mach–Zehnder (MZ) interferometers can both be used to construct switches that have reconfiguration times on the order of 5 ns [9, 28].

2.2.2 Burst-Mode Receivers

During a switching event, the physical connections to the receivers are changed from one host to another host. In most deployed datacenters, these hosts do not share a common clock. Therefore, the clock frequency and phase must be recovered at the receiver. No data can successfully transit the network until the switches have completed reconfiguring and the receivers have locked on to the data. As a result, the system-level reconfiguration time is determined primarily by the sum of the switch reconfiguration time and the receiver lock time. The duty cycle

is the fraction of time in which data can be sent over the network between each reconfiguration. More reconfigurations of the switch, with slower switches, or with slower receivers can reduce the duty cycle and the overall network throughput. This is why fast-locking receivers are required when using fast optical switches.

Burst-mode receivers are designed to lock onto an input signal without a significant interruption of data. This is necessary when the links are optically multiplexed in time, either with an optical switch or by switching transmitters on and off as in passive optical networks (PON). In a PON, downstream data is transmitted using a broadcast-and-select protocol while upstream data is transmitted using a time-division multiple access (TDMA) protocol. The upstream receiver in a 10G-EPON network must be able to lock on to the preamble of each incoming frame within 400 ns, despite varying power levels and clock phase and frequency.

Burst-mode receivers consist of two main components: a variable gain transimpedance amplifier (TIA) and a burst-mode clock data recovery (CDR) circuit. The variable-gain TIA adjusts the gain and threshold to track changes in received optical power. The burst-mode CDR follows the TIA, recovers the clock signal, and detects the data bits. Both the TIA and the CDR must be able to quickly find the optimal gain, offset, and clock phase for successful recovery of the data [44].

2.2.3 System-Level Reconfiguration Time

The overall system-level reconfiguration time is the sum of the switch reconfiguration time, receiver locking time, and all necessary guard times and is illustrated in Fig. 2.3. Fast optical switches help to reduce the system-level reconfiguration time, but minimizing the overall system-level reconfiguration time requires precise synchronization between the optical switch, burst-mode transmitter, and burst-mode receiver. For efficient operation, the receiver lock time should be matched to the speed of the optical switch.

For a scale-out network, this desirable high data-rate functionality creates many challenges

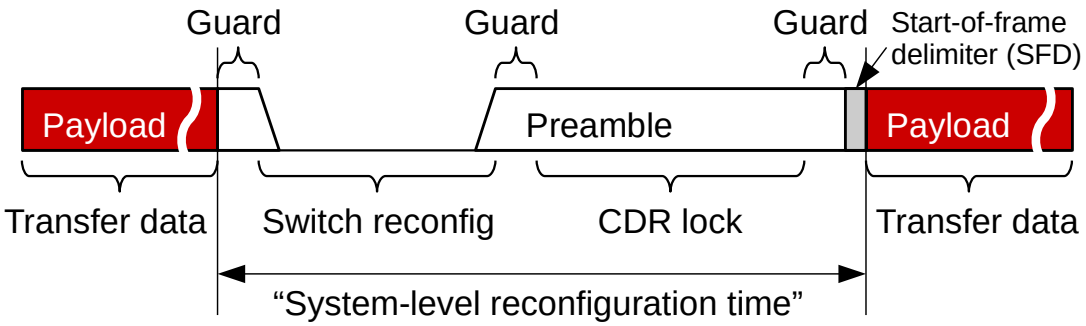


Figure 2.3: Illustration of system-level reconfiguration time, including switch reconfiguration and link training (bit- and frame-synchronization). Not included are scheduling and time-of-flight.

related to link establishment and coordination. Recent demonstrations have focused on the switch interface, while using either source-synchronous [33, 64] or non-burst-mode optical links [51]. Source-synchronous data transmission requires distributing the transmit clock to the receivers, which is complex to implement at datacenter scale. Non-burst-mode receivers recover the clock from data transmitted with an embedded clock, but are not specifically optimized for fast locking and as a result limit the achievable system-level reconfiguration time as well as the average throughput.

Here, we present detailed system-level results for a non-source-synchronous burst-mode link of an optically-switched network that can reconfigure on nanosecond time scales. The system-level timing for the link is coordinated by an FPGA.

Utilizing a burst-mode receiver enables fast switch reconfiguration in a scalable fashion. The FPGA initiates a burst from an optical transmitter, configures the photonic switch, and signals the receiver to begin a training sequence. Once, locked, the FPGA-based data plane provides error-free transmission through the switched link. The testbed demonstrates system-level reconfiguration times of 90 ns at 12.5 Gbps and 60 ns at 20 Gbps for kB-scale packets.

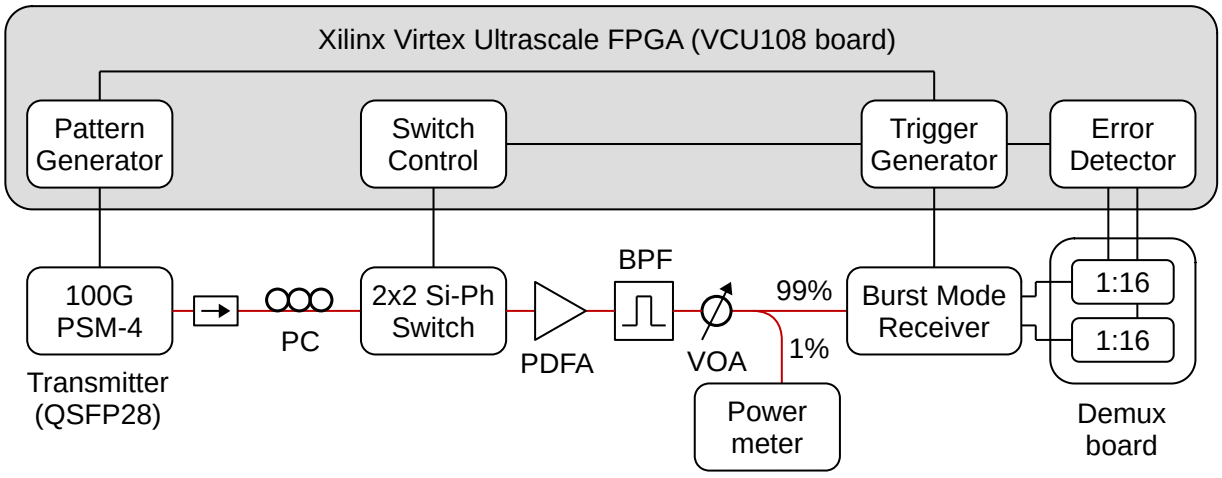


Figure 2.4: Testbed block diagram. The blocks in the shaded region are implemented in the same FPGA.

2.3 Implementation

A testbed, shown in Fig. 2.4, was developed to assess the combined performance of a silicon photonic optical switch chip and a burst-mode receiver. The testbed consists of optical components, a data plane, and an FPGA to coordinate the link. The link-level components, data pattern generator, and error detector are implemented on a Virtex UltraScale FPGA on a VCU108 development board. Data from the pattern generator is serialized with a transceiver on the FPGA and sent to a commercial 100GBASE PSM4 QSFP28 transceiver operating around 1310 nm. The output of the transceiver is coupled to the input of the silicon photonic switch chip through an isolator, polarization controller, and lensed fiber. After passing through the switch chip, the light is coupled out with another lensed fiber, amplified with an O–band praseodymium-doped fiber amplifier (PDFA), filtered with a tunable optical bandpass filter, attenuated, and coupled in to the receiver with a fiber probe. The output of the receiver is connected to the FPGA for error analysis via a pair of demultiplexer chips. The details of this part of the testbed are shown in Fig. 2.7. Optical power is measured in several places—a tap and detector in the bandpass filter module, a tap and power meter following the attenuator, and the photodiode bias source meter, which supplies the bias voltage to the high speed photodiode in the receiver and measures the

average photocurrent. Control software continuously adjusts the attenuator throughout the tests to control the photocurrent (optical power) at the receiver.

The complete test setup in the lab is shown in Fig. 2.5 and Fig. 2.6. Fig. 2.5 contains the burst-mode receiver chip, FPGA, QSFP28 transmitter, level translators, demux board, and associated components. Fig. 2.6 contains the silicon photonic switch chip, optical components, and biasing supplies for the CDR chip.

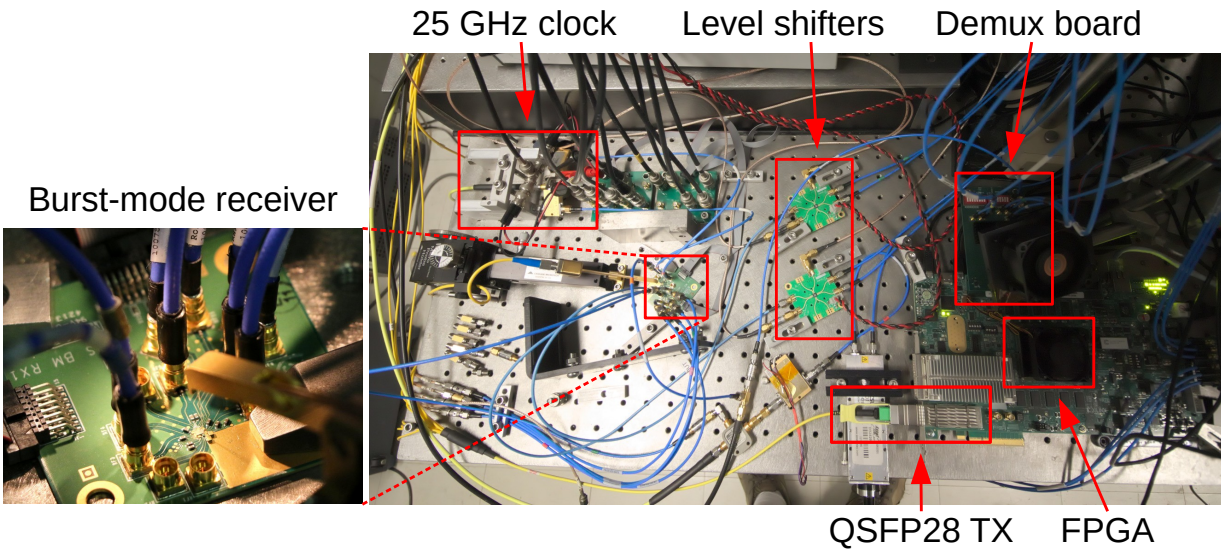


Figure 2.5: CDR side of setup, including FPGA board, QSFP28 transmitter, burst-mode receiver, and demux board

2.3.1 Optical Components

The switch used in the testbed is a 2×2 Mach–Zehnder-based silicon photonic switch [9] configured as a 1×2 switch. This switch is equipped with two electro-optic phase shifters driven in push-pull mode and two thermo-optic phase trimmers. It provides fast switching in about 5 ns, while exhibiting an insertion loss of 1 dB and an extinction ratio > 20 dB over an optical bandwidth of 12 nm in the O-band. The low crosstalk of this switch leads to a worst-case power penalty of less than 0.25 dB [8]. Electronic circuits, including digital device drivers and

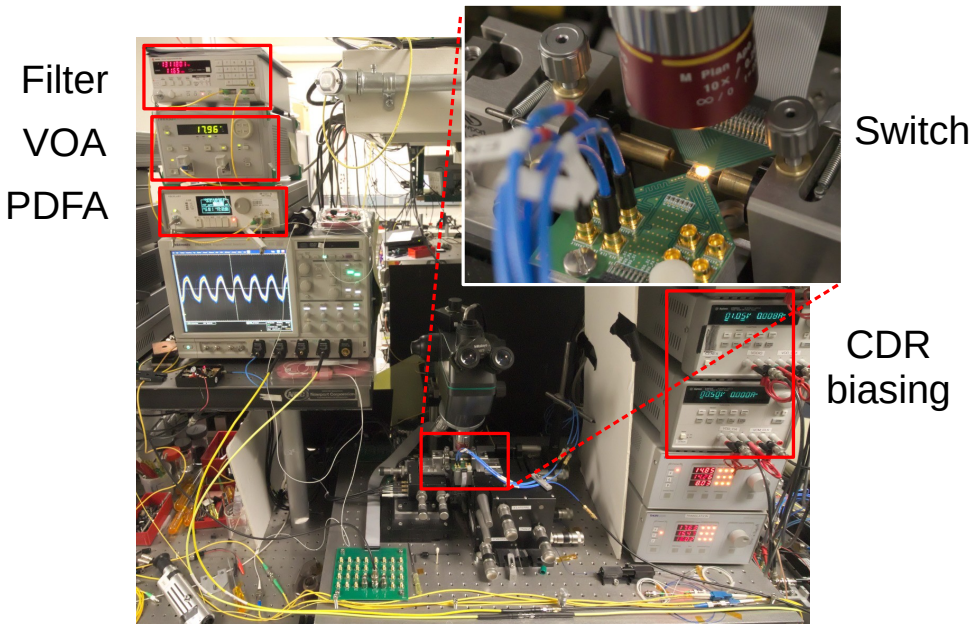


Figure 2.6: Switch side of setup, including silicon photonic switch, optical components, and biasing supplies

a serial-to-parallel interface, are integrated on the die alongside the switching elements [28]. A switch control module on the FPGA loads new configurations into the switch with a 250 MHz clock.

The burst-mode receiver used in the testbed has a measured lock time of 31 ns at 25 Gbps [47]. The receiver uses a series of steps to recover the data, including a TIA calibration to adjust the DC bias and a successive approximation search routine to determine the correct clock phase. Phase interpolators are used to generate the recovered clock. The receiver completes the input DC level calibration in 12.5 ns and locks onto the data bit edges in 18.5 ns. The receiver has an internal 1:2 demultiplexer, providing two half-rate data outputs at 12.5 Gbps as well as a half-rate clock at a nominal rate of 12.5 GHz. The 12.5 Gbps outputs are further demultiplexed by a factor of 16 in the FPGA-based data plane as described below. The trigger generator module on the FPGA provides the start signal to trigger the burst-mode locking routine in the receiver.

Due to the design of the CDR chip, the falling edge of the start signal resets the control

logic and the phase interpolators in preparation for the rising edge to initiate the locking routine, causing the link to be interrupted on the falling edge of the start signal. This means that the width of the start pulse directly affects the length of time the link is interrupted during the CDR locking operation. This is in contrast with [47], where the locking time is measured from the rising edge of the start pulse after the CDR has been reset. The shortest pulse that worked reliably was 6 ns wide and the CDR takes about 4 ns to respond to the falling edge, so the overall locking time achieved is closer to 42 ns at 20 Gbps instead of the expected 38 ns (extrapolated from 31 ns at 25 Gbps).

2.3.2 Datapath

Data is generated with a custom pattern generator, implemented on a Virtex UltraScale XCVU095 FPGA. The pattern generator is responsible for generating the preamble that the burst-mode CDR chip requires—in this case, a 2-UI square wave or 1010 data pattern—as well as a known data sequence as the payload for BER measurements. The pattern generator uses a 30-Gbps GTY transceiver on the FPGA to serialize the data for transmission to the QSFP28 module and through the link. The two data rates used in this experiment are 12.890625 Gbps and 20.625 Gbps, which are 50% and 80% of the data rate of a single lane of 100GBASE-R, respectively. For brevity, this chapter lists these rates as 12.5 Gbps and 20 Gbps. The pattern generator supports generating variable-length payloads either from an internal pattern RAM or an internal PRBS generator. When running in PRBS mode, a continuous PRBS sequence is generated and segmented into individual payloads, with each payload containing a different section of the sequence. The pattern generator continuously outputs a repeating preamble pattern, switching to the configured payload data pattern when triggered.

Following the receiver, an error detector was implemented to assess system-level performance. A block diagram of the error detector and connections to the receiver is shown in Fig. 2.7. This diagram covers the burst-mode receiver, demux, and error detector blocks from Fig. 2.4. The

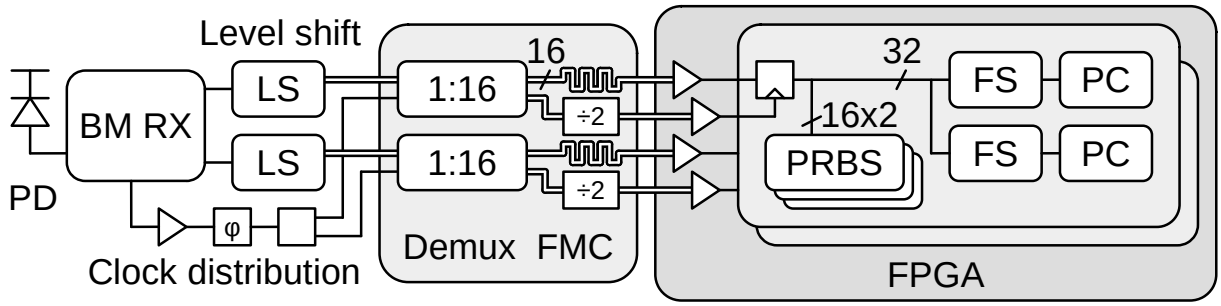


Figure 2.7: Block diagram of receiver, FPGA connections, and FPGA receive datapath logic.

burst mode receiver contains a variable gain transimpedance amplifier (TIA), 1:2 demultiplexer, phase interpolators, and the burst mode CDR logic. When running at 25 Gbps, the output of the burst-mode receiver chip is a pair of source-synchronous single-ended 12.5-Gbps outputs with a 12.5-GHz clock. The transceivers on the FPGA work at up to 30 Gbps, but can only be used for receiving serial data with an embedded clock—they do not support reception with an external full-rate recovered clock. 12.5 Gbps is also too fast for normal source-synchronous FPGA I/O, which supports a maximum of 1600 Mbps for LVDS inputs. Instead, a pair of external demultiplexer chips (Adsantec ASNT2011) on a custom FMC (FPGA mezzanine connector) board (shown in Fig. 2.8) are used to demultiplex each 12.5-Gbps output to sixteen 800-Mbps low-voltage differential signaling (LVDS) signals that are compatible with normal source-synchronous FPGA I/O. Level shifters (Adsantec ASNT3111) are used to convert between the single-ended ground-referenced outputs of the receiver and the differential Vcc-referenced inputs of the demultiplexer chips. The recovered clock from the burst-mode CDR chip is applied to the demultiplexers through an amplifier, trombone phase shifter, and wideband resistive power divider. The FMC board contains two demultiplexers, two LVDS clock dividers, and delay compensation traces to realign the data after the clock dividers. The clock dividers are required to enable operation in double data rate (DDR) mode, as 800 Mbps per pin is right on the edge of what the FPGA is capable of in single data rate (SDR) mode. The LVDS data from the demultiplexers is passed through LVDS input buffers and DDR flip flops in the FPGA I/O banks for a final demux by a

factor of 2. Once inside the FPGA, two different error detection techniques are implemented.

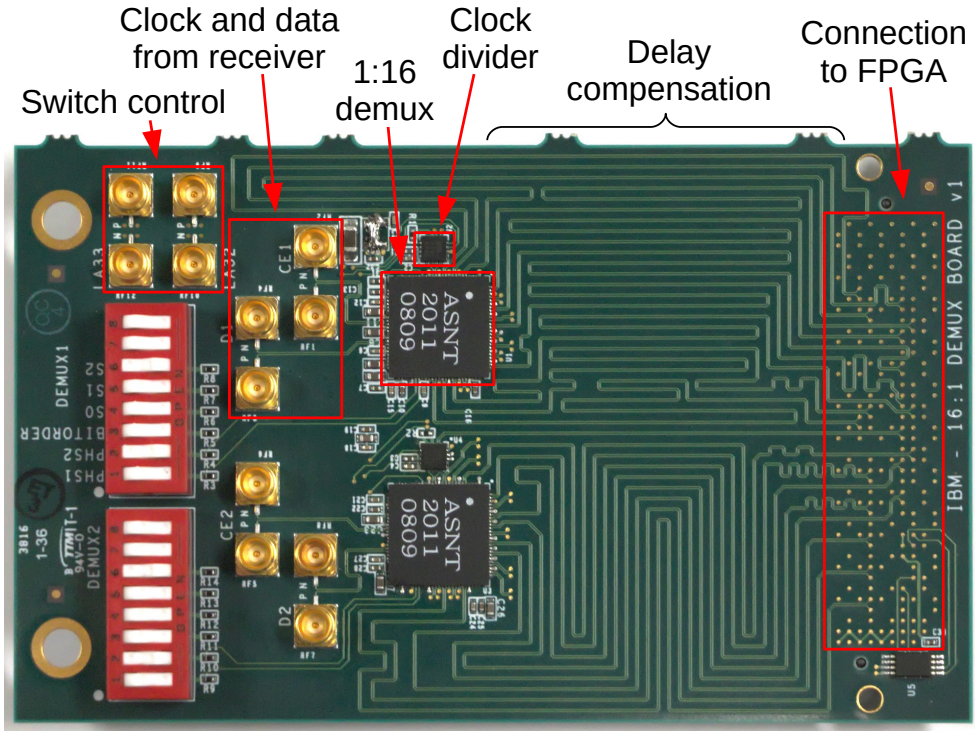


Figure 2.8: Picture of dual-channel 1:16 demux FMC board.

First, the errors on each of the 16 outputs of each demultiplexer chip are measured with 32 gated error detectors, denoted PRBS in Fig. 2.7. Each gated error detector consists of a linear feed-forward shift register with the same taps as the corresponding PRBS generator shift register followed by a gated counter. Bit errors in the data shifted into the shift register produce logical high levels at the output—one logical high per tap, per error. The gated counter accumulates the errors when the gate is open, with the gating signal provided by the trigger generator. The gated error detectors are useful for debugging purposes. Because the error detectors operate on the fully demultiplexed data, the guard times required for the detectors to synchronize with the incoming data are quite large. Our previous work [18] only used this technique for error detection. Further details are provided in Section 2.4.

In this work, an improved method of error detection was also used that is based on a set of frame-synchronized pattern checkers. Two of these pattern checkers are attached to the

outputs of each demultiplexer chip, one for each possible data alignment of the 1:2 demultiplexer in the receiver chip. It is not possible to use a single pattern checker in our testbed as the clock dividers in the two discrete demultiplexers can come out of sync with each other during the operation of the burst-mode locking routine due to runt clock pulses from stepping the phase interpolators on the CDR chip. This would not be an issue with a fully-integrated demultiplexer. A frame synchronization module, denoted FS in Fig. 2.7, detects the start of frame delimiter (SFD) between the preamble and the payload data and barrel-shifts the data into the correct alignment so the pattern checkers, denoted PC in Fig. 2.7, can check the payload data. The start-of-frame delimiter (SFD) used is the same as the standard Ethernet SFD where the last bit of the 1010 preamble is inverted. A signal from the trigger generator prepares the frame synchronization logic to lock on to the SFD after each switch reconfiguration. Using frame synchronization vastly reduces the guard times associated with the error checkers. Note that some guard time is still required, as bit errors in the preamble will cause the frame synchronizer to lock at the incorrect offset. In a network setting, the guard time can be reduced by precise injection of the packets into the network when the switch is changing its state. This is discussed in Chapter 3.

The pattern checkers are capable of comparing payload data to fixed patterns stored in SRAM or feeding payload data through an LFSR-based PRBS checker. Utilizing both the PRBS generator in the pattern generator and the PRBS checkers in the frame-synchronized pattern checkers enables testing with the entire PRBS sequence.

2.3.3 Link-Level Control

A link consisting of both an optical circuit switch and a burst-mode receiver requires precise synchronization of the datapath to ensure that the error-rate measurements are conducted when the switch is in the appropriate state. This means that the receiver is locked and valid data is present at the output of the receiver. A simplified timing diagram of the control signals to synchronize the switch chip, the burst-mode receiver, and the BER measurements as well as

selected data signals is shown in Fig. 2.9. The timing diagram shows the timing relationships between various events, but is not necessarily to scale. Further details of the specific ordering of the trigger events are provided in Section 2.4.

Software running on a control computer interfaces with all of the datapath components and test equipment. The control software sets up all of the datapath components for each test, then alternates between adjusting the optical power, accumulating errors, and reading out error counters.

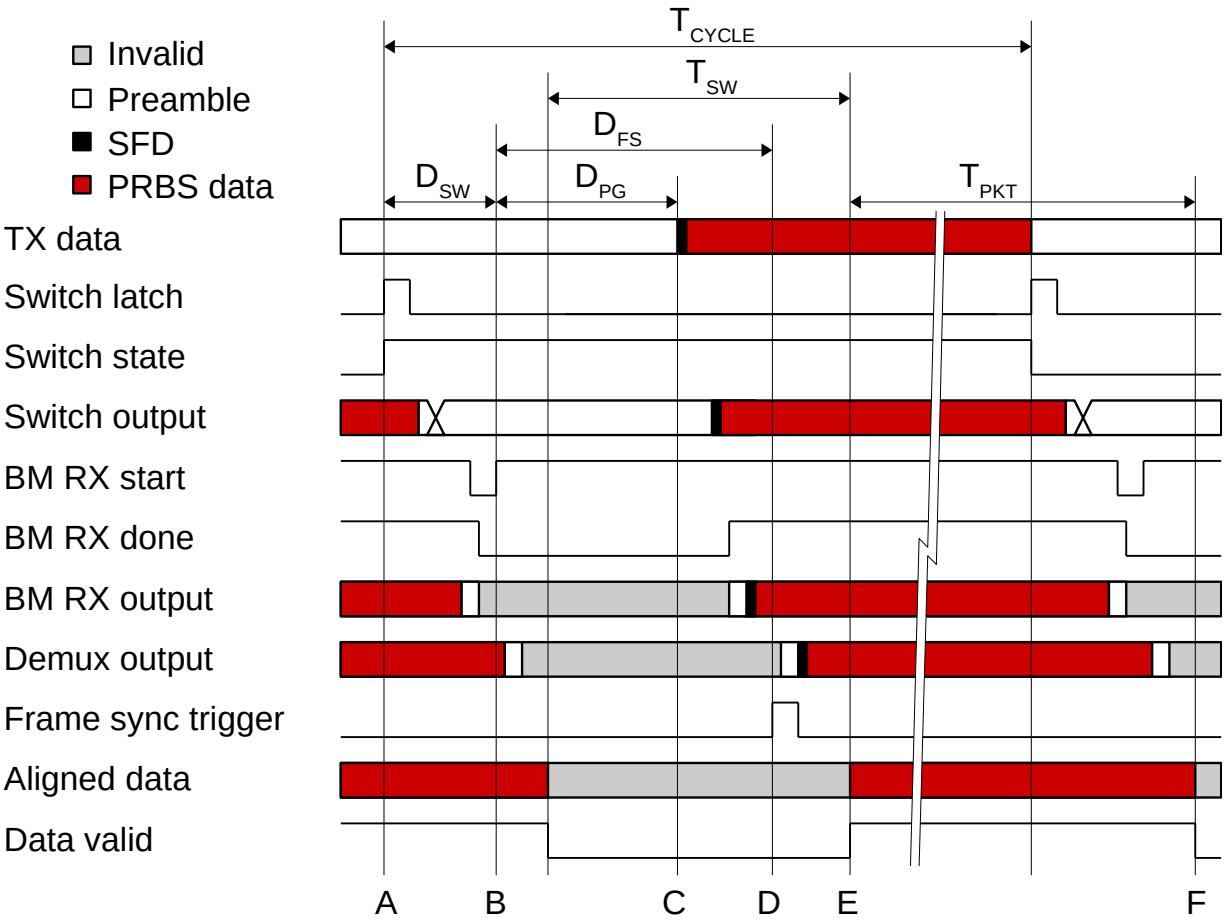


Figure 2.9: Simplified system timing diagram.

The control signals on the timing diagram are generated by a programmable trigger generator implemented within the FPGA shown in the gray box in Fig. 2.4. The trigger generator runs on a 500 MHz clock, providing a delay resolution of 2 ns. This generator triggers the

burst-mode receiver, pattern generator, switch controller, and frame-synchronized error detector. The pattern generator is responsible for generating and switching between the preamble used to lock the burst-mode receiver and the data pattern used by the error detector for evaluating link performance. In the absence of a trigger, the pattern generator continuously produces a preamble pattern as shown schematically at the top of Fig. 2.9. The preamble is a 2-unit-interval square wave (1010 data pattern). A $2^{15}-1$ bit PRBS was used as the payload because of the long cycle time of a $2^{31}-1$ sequence; there are no fundamental limitations in either the photonic switch, the burst-mode receiver, or the rest of the receive datapath when longer sequences are used [47].

Event (A) in Fig. 2.9 denotes the start of a switch cycle, when the latch signal from the switch control module on the FPGA, denoted switch latch in Fig. 2.9, commands the switch chip to latch the new configuration to the PIN diode drivers, thereby causing the switch to reconfigure. During this event, the pattern generator sends the preamble sequence to the switch during the switch transition so that the receiver will be ready to lock immediately after the switch has stabilized in the new configuration.

After a delay D_{SW}, shown at the top of Fig. 2.9, the burst-mode receiver is triggered at event (B), initiating the burst-mode locking sequence. This event occurs when the trigger generator module on the FPGA issues the start signal to the burst-mode receiver, denoted in Fig. 2.9 as BM RX start. At this point, the burst-mode receiver will start its gain adjustment and clock phase search routines. During this time interval, the preamble signal must be continuously present at the receiver so that the burst-mode locking routines in the receiver will properly lock on to the signal. When the locking routine completes, the BM RX done signal is asserted.

After a second delay D_{PG} shown as event (C) in Fig. 2.9, a trigger output from the trigger generator informs the pattern generator to stop generating the preamble sequence and switch to a start of frame delimiter (SFD) followed by the payload data, which is nominally a PRBS. The timing of the trigger event is adjusted so that the SFD and payload data arrive at the receiver after it is locked and ready to receive data.

After a third delay D_{FS}, shown as event (D) in Fig. 2.9, a trigger output from the trigger generator informs the frame synchronizer in the pattern checker module on the FPGA to start searching for the start of frame delimiter (SFD). Upon detecting the SFD, the frame synchronizer barrel-shifts the demultiplexed data into proper alignment for error detection of the entire packet payload.

Event (E) in Fig. 2.9 denotes the start of aligned payload data after the frame synchronizer. The end of the packet is denoted event (F), T_{PKT} later. The error detector accumulates errors over the entire payload.

The error measurement process is repeated with a cycle time T_{CYCLE}. For the timing parameters used in the testbed, the load signal for the next measurement cycle starts during the error measurement process due to the significant pipeline delay through the datapath on the FPGA. The duty cycle $T_{\mathrm{PKT}}/T_{\mathrm{CYCLE}}$ for several payload sizes at both data rates is shown in Table 2.1. The system-level reconfiguration time T_{SW} is defined as $T_{\mathrm{CYCLE}} - T_{\mathrm{PKT}}$.

2.4 Timing Detail and Comparison with Earlier Work

This section presents details of the experimental setup used in this work and contrasts this setup to the previous setup used in earlier work [18]. Our previous work reported a 332 ns system-level lock time using a burst-mode receiver and a nanosecond optical switch chip. While this result was novel, it resulted in a low circuit-switched duty cycle for typical packet sizes. To make this method more practical, several key changes were implemented in this experiment to improve the performance thereby making the system more practical.

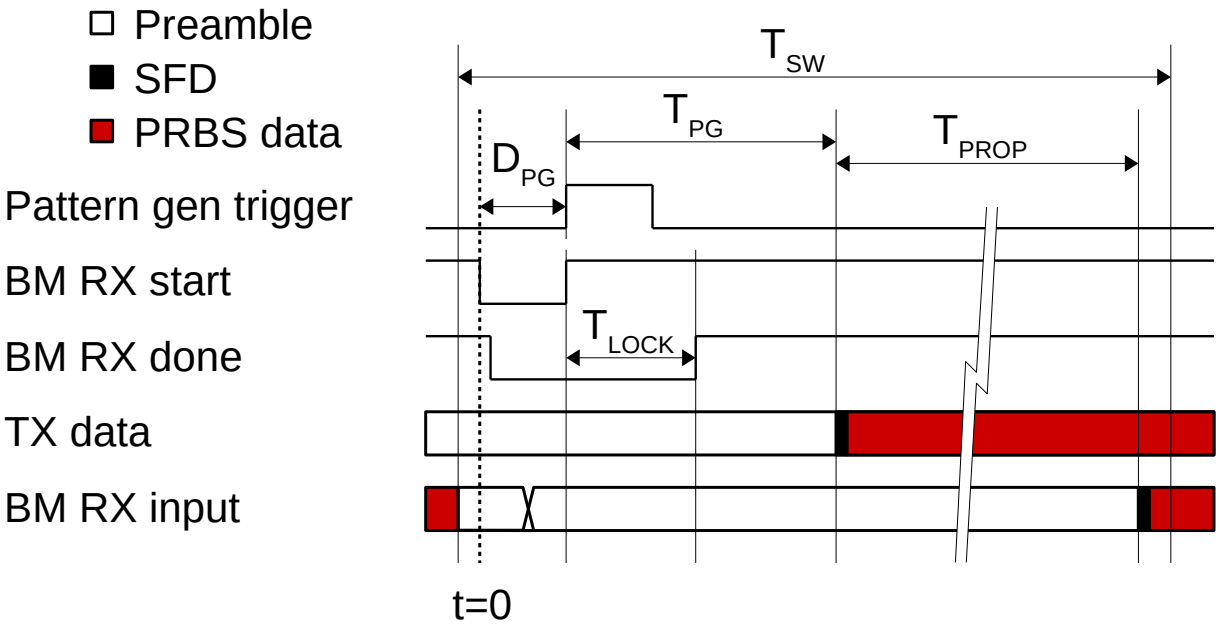
The setup used for the work presented in this chapter has the same general structure as our earlier work, but makes several key changes that lead to significant improvements in the system-level lock time. These changes were to: (1) optimize the settings of FPGA-based pattern generator, (2) reduce the width of the burst-mode receiver trigger signal pulse, (3) implement

frame synchronization, and (4) re-order of the triggering events to compensate for the large delay of the pattern generator. We consider each of these changes separately.

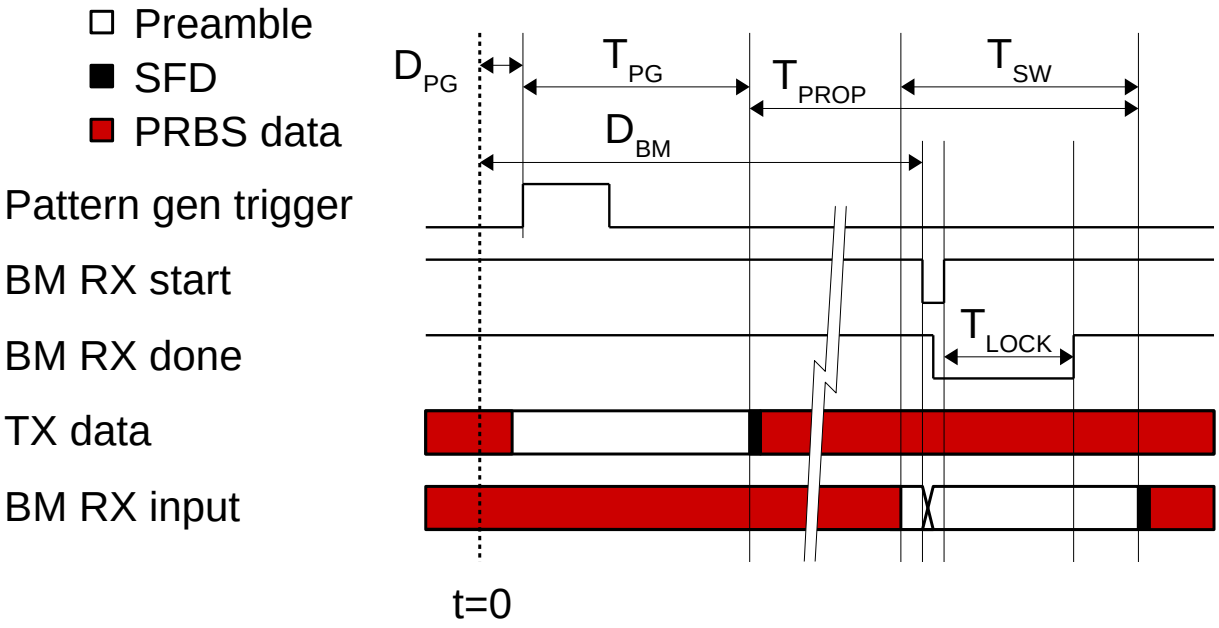
Two optimizations were made to the FPGA-based pattern generator. The first optimization was setting the pattern generator datapath width to half the previous value. The second optimization was doubling the serializer clock speed. These changes reduced the trigger uncertainty from the trigger generator from 6.2 to 3.1 ns at 20.625 Gbps. These changes also reduced the serializer latency by one cycle of the 128 bit interface, or 6.2 ns at 20.625 Gbps. The changes also reduced the latency of the rest of the pattern generator. Specifically, doubling the clock frequency cuts the latency of the rest of the pattern generator from 37.2 ns to 18.6 ns at 20.625 Gbps, saving 18.6 ns of latency. The combined changes to the FPGA-based pattern generator reduced the trigger-to-output delay by 3.1+6.2+18.6 = 27.9 ns.

Changes were also made to the burst-mode receiver trigger signal. Originally, the trigger signal was configured to be a 20 ns wide active-low pulse. However, the burst-mode receiver resets on the falling edge of a trigger pulse and initiates the locking routine on the rising edge of the pulse. This means that the width of the trigger pulse directly impacts the locking time of the receiver. Therefore, we reduced the width of the trigger pulse. For the current setup, we determined that a 6 ns pulse was the shortest pulse that would reliably trigger the receiver. This change yielded a savings of 14 ns for the lock time of the burst-mode receiver.

The use of frame synchronization also improved the system-level reconfiguration time. Our previous work used only gated BER measurements taken individually on each output of the discrete demultiplexers. Because of the lower data rate, it takes longer for the PRBS checkers to synchronize with the payload data. Specifically, when checking PRBS-15 data, each PRBS checker must see at least 15 data bits before it can lock onto the pattern. Since the data is demultiplexed by 64 in the FPGA, this requires 8 clock cycles at rate/64 or 24.8 ns at 20.625 Gbps. Frame synchronization enables handling data at the full interface width, enabling the PRBS checker to lock in a single 3.1 ns clock cycle.



(a) Timing details our previous setup [18].)



(b) Timing details of the setup used in this experiment.

Figure 2.10: Comparison of the timing parameters used for our earlier work and the setup used for this work.

Finally, changes were made to the ordering of the triggering events to compensate for the large delay of the pattern generator. This large delay was primarily due to the FPGA serializer and propagation delay from the pattern generator to the burst-mode receiver. Specifically, we interchanged the ordering of the burst-mode start trigger and the pattern generator trigger. This re-ordering led to the burst-mode receiver being triggered *after* the pattern generator instead of before as was the case for the previous setup.

With respect to the timing diagram, the previous setup considered the burst-mode trigger signal to be the reference time (i.e. "$t = 0$") for the rest of the trigger signals, as shown in Fig. 2.10a. For this previous case, the pattern generator start signal was triggered 20 ns after the burst-mode receiver, shown as D_{PG} in Fig. 2.10a. The pattern generator latency is T_{PG}, the propagation of the pattern generator output through the optical components and back to the receiver is T_{PROP}, the lock time of the burst-mode receiver is T_{LOCK}, and the overall system-level reconfiguration time is T_{SW}. In the timing diagram, the TX data trace represents the output of the pattern generator, and the BM RX input trace represents the optical signal at the input of the receiver, after passing through the optical switch. Unlike Fig. 2.9, Fig. 2.10a shows the data at the input of the burst-mode receiver instead of the output as the focus is on the propagation of the data from the pattern generator to the receiver along with the related control signals.

In the current setup, the reference time for the rest of the trigger signals was changed to be before the pattern generator trigger signal. Therefore, in this new reference frame, the burst-mode trigger signal occurs *after* the pattern generator trigger signal as shown in Fig. 2.10b. This change in the experimental setup set the pattern generator trigger at a fixed offset of $t = 10$ ns after the reference time. The maximum delay D_{BM} for the burst-mode trigger that produced no bit errors was $t = 252$ ns after the pattern generator trigger. This delay yielded the minimum system-level lock time. Accounting for the 20 ns offset used in the previous setup gives an overall change of 20+252-10=262 ns in the relative timing of the burst-mode trigger signal and the pattern generator trigger signal between the previous setup and the setup used in this work.

Because the burst-mode trigger pulse width was reduced by 14 ns, this results in the rising edge of the burst-mode trigger signal changing to 248 ns. Adding 25 ns from the use of frame synchronization gives an overall improvement of 273 ns for the system-level locking. This value is in excellent agreement with the measured 272 ns difference between the 332 ns system-level reconfiguration time reported in our previous work [18] and the 60 ns reconfiguration time reported in this work. The system-level reconfiguration time in Fig. 2.10 is depicted as T_{SW}.

2.5 Results

The bit error ratio (BER) of the burst-mode link was tested in three measurement configurations at data rates of 12.5 Gbps and 20 Gbps. The test data included a preamble consisting of the pattern 0101 . . . and a 1024-byte PRBS15 payload, with a 730-ns overall period at 12.5 Gbps and a 460-ns overall period at 20 Gbps.

The first configuration was a back-to-back set-up without the photonic switch. With the photonic switch inserted, the cross and bar states were used to assess the performance. The same BER measurement was performed in each case, sending multiple data packets containing segments of PRBS15 over the link and re-locking the burst-mode CDR before each packet.

Fig. 2.11 shows several measured waveforms. The upper two waveforms (denoted BM RX start and BM RX done) are used to start the burst-mode locking process and notify the system when that process is completed. The next trace from the top is the input waveform to the switch. The darker part of this trace is the preamble and the lighter part is the data pattern. The slight change in the amplitude between these two parts of the waveform is attributed to high-frequency roll-off of the preamble compared with the PRBS payload that includes more low-frequency components.

The lower two traces are the waveforms at the cross and bar outputs of the switch. Fig. 2.12 shows a zoomed-in version showing the switching transition. As in Fig. 2.11, the upper two

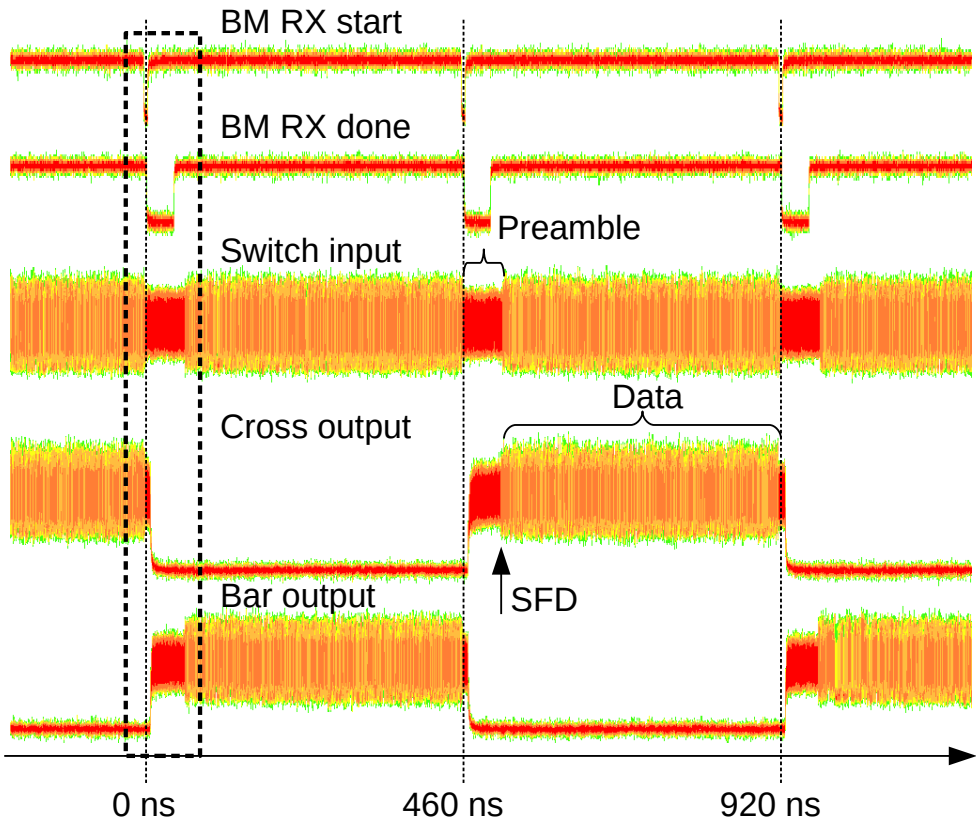


Figure 2.11: Waveforms at 20 Gbps with a 1024-Byte payload.

waveforms are the trigger signal to the burst-mode receiver and the done signal in response, the third waveform is the input to the switch, and the bottom two waveforms are the two outputs of the switch. The data patterns in the high-speed waveforms do not align between the waveforms as the waveforms were captured separately and combined based on the burst-mode start trigger. The data pattern alignment difference of up to 5 ns between the trigger signal and the pattern generator in the FPGA is due to the crossing from the trigger generator's 500-MHz clock domain to the pattern generator, running in the 201.4-MHz serializer transmit clock domain.

The bit error ratio (BER) curves for the back-to-back configuration and for the cross and bar configurations of the switch chip for each of the two data rates are shown in Fig. 2.13. The measured eye diagrams (back-to-back and through the switch) at 12.5 Gbps and 20 Gbps are shown in Fig. 2.14. The BER curves indicate minimal impact on the sensitivity at 12.5 Gbps

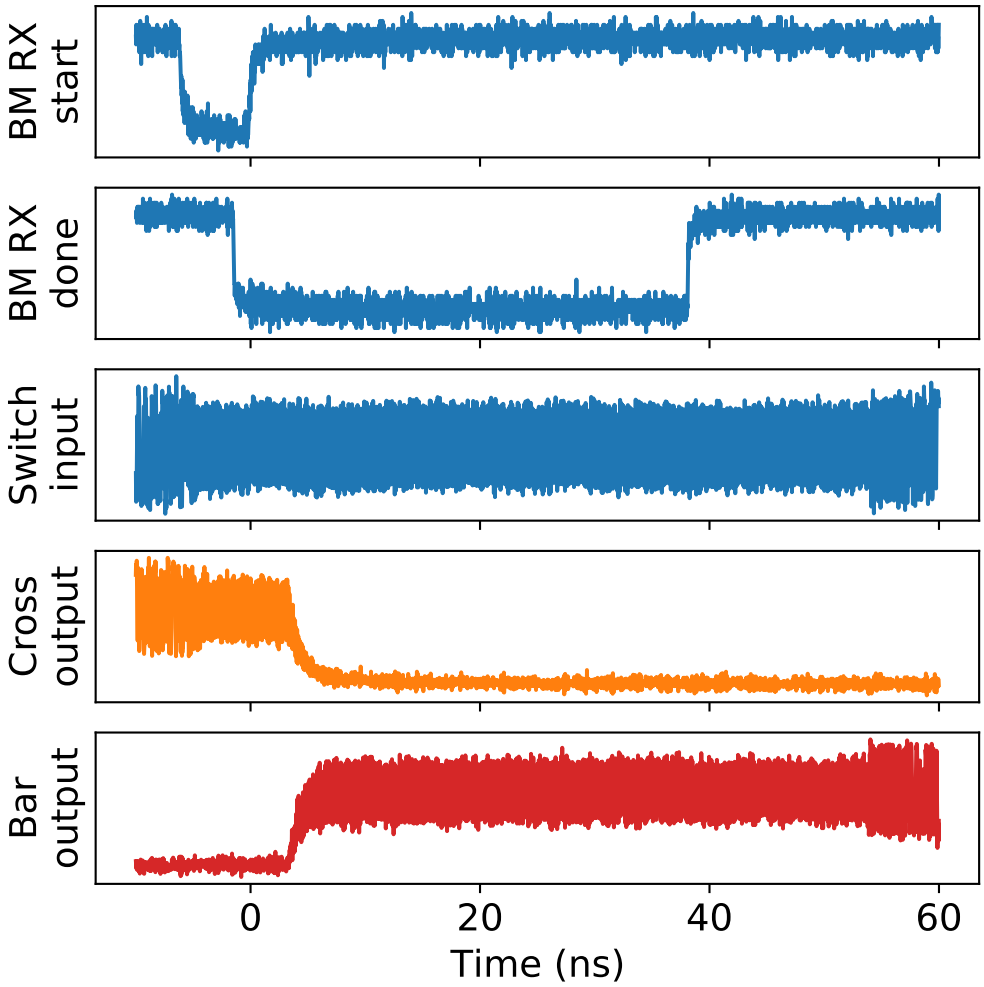


Figure 2.12: Traces inside the dashed box of Fig. 2.11.

and about a 1.5-dB penalty at 20 Gbps, both measured at a BER of 10^{-12}. Part of the penalty at 20 Gbps may be due to interference effects from the switch chip itself or the fiber coupling to the chip. The sensitivity degradation of approximately 2 dB compared to [47] is mostly attributed to our data source, the transmitter of a commercial 100GBASE PSM4 QSFP28 module driven by FPGA serializers with integrated PLLs.

Table 2.1 contains the timing parameters for the links that were evaluated. Data rates of 12.5 and 20 Gbps and payload sizes of 1024 and 2048 bytes were tested. For each configuration, the timing parameters were adjusted for zero-error performance. T_{CYCLE} is the overall cycle time,

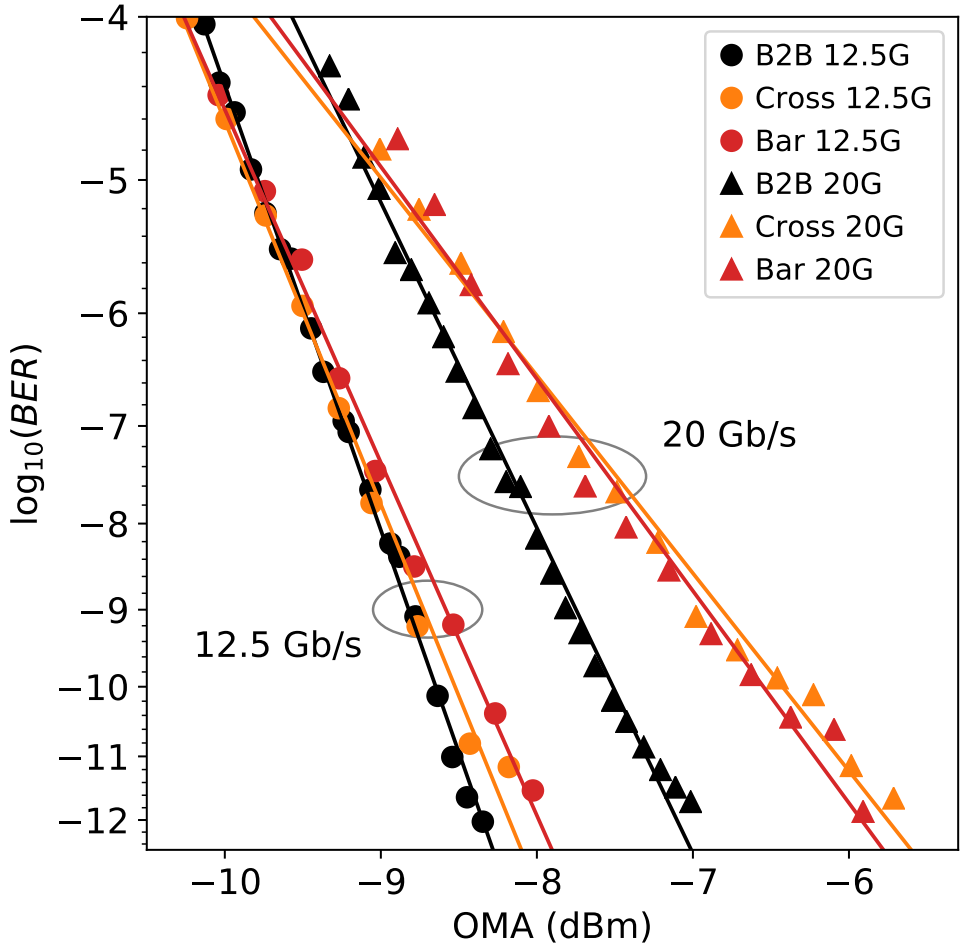


Figure 2.13: BER at 12.5 Gbps (●) and 20 Gbps (▲).

Table 2.1: Timing parameters and overall duty cycle.

Payload size (B)	2048	1024	2048	1024
Data rate (Gbps)	12.5	12.5	20	20
T_{CYCLE} (ns)	1366	730	858	460
T_{PKT} (ns)	1276.0	640.5	797.5	400.3
T_{CDR} (ns)	64.0	64.0	41.5	41.5
T_{SW} (ns)	90	90	60	60
Duty cycle (%)	93	87	93	87

the time between the start of each switch reconfiguration. T_{PKT} is the length of time required to transmit the packet, which is checked by the pattern checkers on the FPGA. T_{CDR} is the measured locking time of the CDR chip, measured from the done output falling to done output rising.

During this time, data cannot reliably transit through the CDR chip, even if the link has not been interrupted by the optical switch. T_{SW} is the system-level reconfiguration time; the time during each switch cycle in which data cannot be sent. In this case, $T_{\mathrm{SW}} = T_{\mathrm{CYCLE}} - T_{\mathrm{PKT}}$. Finally, the duty cycle is the fraction of the switch cycle that can be utilized for data transmission. In this case, $D = T_{\mathrm{PKT}}/T_{\mathrm{CYCLE}}$.

The system-level reconfiguration time is the time during a complete switching cycle that cannot be used for sending valid data. It includes not only the optical switching and burst-mode receiver lock time, but also a settling delay required before triggering the receiver, and guard time on both ends for triggering the pattern generator and frame-synchronized error detector. Our measured values of 60 and 90 ns could be improved through further FPGA optimization or an ASIC implementation.

2.6 Discussion

This work in this chapter employed a 32-nm CMOS burst-mode receiver that can lock in 31 ns at 25 Gbps [47], which contributes a large part to our measured end-to-end reconfiguration time. Faster burst-mode receivers locking in 6.8 ns at 56 Gbps have since been demonstrated in 14-nm CMOS, although that implementation did not include an automatic gain control block to enable operation over the large dynamic range that may be required in optical switching applications [40]. Our measured reconfiguration times could also be further improved through increased integration to remove the discrete demultiplexing of our lab demonstration and by using an ASIC implementation instead of an FPGA. System-wide synchronization may also be implemented through other means such as clock distribution, phase-caching [5], or in-band protocols such as WhiteRabbit [46, 53]. While standards such as IEEE 1588 (PTP) have been developed for synchronization across a large number of end hosts, very fast synchronization at the nanosecond scale has only been demonstrated in small-scale test beds.

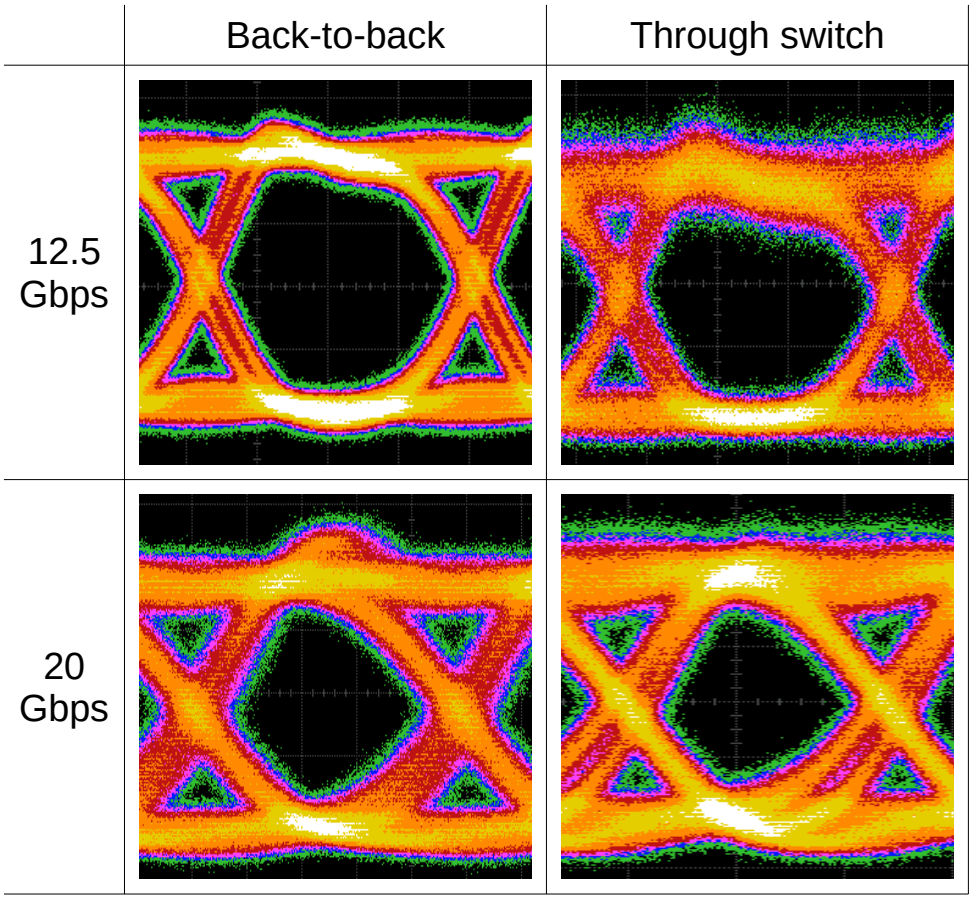


Figure 2.14: Optical eye diagrams.

While the work presented is necessary for an optical circuit-switched network, it is not sufficient. To build a complete network, additional components are required that have their own set of challenges. Any optically-switched network must have a scheduler to compute switch state and data transmission schedules. For this experiment, a simple deterministic round-robin schedule was used that did not dynamically adjust to traffic demand. In practice, this system-level issue is a significant challenge in particular for scale-out networks [35].

2.7 Conclusion

Fast optical switching has significant potential to improve throughput and power efficiency in datacenter and high-performance computing applications. However, effectively utilizing fast optical switching in a network requires precise coordination of the switches and burst-mode receivers in order to minimize overall system reconfiguration time.

Detailed system-level results for a non-source-synchronous link of an optically-switched network that can reconfigure on a nanosecond time scale have been presented. The system combines a nanosecond scale optical switch with a fast-locking burst-mode optical receiver. An FPGA-based testbed was used for coordination to demonstrate error-free performance of switched links at 12.5 Gbps and 20 Gbps with measured system-level reconfiguration times of 90 ns and 60 ns, respectively. These system-level results are a vital step in the development of scalable nanosecond optical networks.

Moreover, any circuit-switched network must synchronize the transmitters, switches, and receivers. Implementing global synchronization at datacenter scale across many separate components is complex. For this experiment, the major components were all implemented on the same FPGA, avoiding several issues that would have to be addressed in a distributed synchronization scheme.

Admission control at the transmitters is also required to ensure the correct data is sent on the wire at the correct time. Implementing admission control for nanosecond optical switches at any scale requires hardware support at the end hosts to achieve the necessary control over transmit timing.

The work presented here is a significant step towards the integration of optical circuit switching into datacenter and HPC networks.

2.8 Sources for Material Presented in This Chapter

Chapter 2, in part, reprints material as it appears in the paper titled: "A dynamically-reconfigurable burst-mode link using a nanosecond photonic switch," published in the *Journal of Lightwave Technology*, 2020, by Alex Forencich, Valerija Kamchevska, Nicolas Dupuis, Benjamin G. Lee, Christian Baks, George Papen, and Laurent Schares. The dissertation author was the primary researcher and author of this material.

Chapter 3

Corundum NIC

This chapter presents the design of Corundum, an open-source, high-performance, FPGA-based NIC for circuit-switched networks. As discussed in Chapter 2, precise injection of packets at the edge of a circuit-switched network is vital to effectively utilize a high speed optical switch. Software solutions and commercial NICs are not designed for operation in circuit-switched networks. Therefore, they do not provide the required level of timing performance. This means that high-precision injection of packets at realistic line rates requires some form of custom hardware. The motivation for the development of Corundum is the need for precise control of the injection of packets from multiple queues into a circuit-switched network under the control of a hardware-based scheduler. The development of such a network interface will enable the construction of practical sub-microsecond circuit-switched networks at scale.

Corundum is designed to provide a network interface similar in performance to a commercially-available NIC, while enabling implementation of these additional hardware features. Corundum provides the capability to control packet transmissions with microsecond precision, enabling operation with microsecond-scale optical switches with no additional software overhead. Corundum also provides direct access to physical layer components, enabling in situ physical layer link characterization methods similar to those discussed in Chapter 2.

3.1 Introduction and Overview

The network interface controller (NIC) is the gateway through which a computer interacts with the network. The NIC forms a bridge between the software stack and the network, and the functions of this bridge define the network interface. Both the functions of the network interface as well as the implementation of those functions are evolving rapidly. These changes have been driven by the dual requirements of increasing line rates and NIC features that support high-performance distributed computing and virtualization. The increasing line rates have led to many NIC functions that must be implemented in hardware instead of software. Concurrently, new network functions such as precise transmission control for multiple queues are needed to implement advanced protocols and network architectures.

To meet the need for an open development platform for new networking protocols and architectures at realistic line rates, we are developing an open-source[1] high-performance, FPGA-based NIC prototyping platform. This platform, called Corundum, is capable of operation up to at least 94.4 Gbps, is fully open source and, along with its driver, can be used across a complete network stack. The design is also portable across many different FPGAs, with additional gates available for further customization even on smaller devices. We show that Corundum's modular design and extensibility permit co-optimized hardware/software solutions to develop and test advanced networking applications in a realistic setting.

3.1.1 Motivation and Previous Work: Transmit Scheduling

The motivation for the development of Corundum can be understood by looking at how network interface features in existing NIC designs are currently partitioned between hardware and software. Hardware NIC functions fall into two main categories. The first category consists of simple offloading features that remove some per-packet processing from the CPU—such

[1] Corundum codebase: https://github.com/ucsdsysnet/corundum

as checksum/hash computation and segmentation offloading that enables the network stack to process packets in batches. The second category consists of features that must be implemented in hardware on the NIC to achieve high performance and fairness. These features include flow steering, rate limiting, load balancing, and time stamping.

Traditionally, the hardware functions of NICs are built into proprietary application-specific integrated circuits (ASICs). Coupled with economies of scale, this enables high performance at low cost. However, the extensibility of these ASICs is limited and the development cycle to add new hardware functions can be expensive and time-consuming [16]. To overcome these limitations, a variety of smart NICs and software NICs have been developed. Smart NICs provide powerful programmability on the NIC, generally by providing a number of programmable processing cores and hardware primitives. These resources can be used to offload various application, networking, and virtualization operations from the host. However, smart NICs do not necessarily scale well to high line rates, and hardware features can be limited [16].

Software NICs offer the most flexibility by implementing network functionality in software, bypassing most of the hardware offloading features. As a result, new functions can be developed and tested quickly, but with various trade-offs including consuming host CPU cycles and not necessarily supporting operation at full line rate. Additionally, because of the inherent random interrupt-driven nature of software, the development of networking applications that require precise transmission control is infeasible [57]. Despite this, many research projects [45, 48, 24, 56] have implemented novel NIC functions in software by either modifying the network stack or by using kernel-bypass frameworks such as the Data Plane Development Kit (DPDK) [6].

FPGA-based NICs combine features of ASIC-based NICs and software NICs: they are capable of running at full-line rate and delivering low latency and precision timing, while having a relatively short development cycle for new functions. High-performance, proprietary, FPGA-based NICs have also been developed. For example, Alibaba developed a fully custom

FPGA-based RDMA-only NIC that they used to run a hardware implementation of a precision congestion control protocol (HPCC) [29]. Commercial products also exist, including offerings from Exablaze [10] and Netcope [39].

Unfortunately, similar to ASIC-based NICs, commercially-available FPGA-based NICs tend to be proprietary with basic "black-box" functions that cannot be modified. The closed nature of basic NIC functionality severely limits their utility and flexibility for developing new networking applications.

Commercially-available high-performance DMA components such as the Xilinx XDMA core and QDMA cores, and the Atomic Rules Arkville DPDK acceleration core [3] do not provide fully configurable hardware to control the flow of transmit data. The Xilinx XDMA core is designed for compute offload applications and as such provides very limited queuing functionality and no simple method to control transmit scheduling. The Xilinx QDMA core and Atomic Rules Arkville DPDK acceleration core are geared towards networking applications by supporting a small number of queues and providing DPDK drivers. However, the number of queues supported is small—2K queues for the XDMA core and up to 128 queues for the Arkville core—and neither core provides a simple method for precise control over packet transmission.

Open-source projects such as NetFPGA [66] exist, but the NetFPGA project only provides a toolbox for general FPGA-based packet processing and is not specifically designed for NIC development. Moreover, the NetFPGA NIC reference design utilizes the propriety Xilinx XDMA core, which is not designed for networking applications. Replacing the Xilinx XDMA core in the reference NIC design for the NetFPGA board with Corundum results in a much more powerful and flexible prototyping platform.

FPGA based packet-processing solutions include Catapult [4], which implements network application offloading, and FlowBlaze [42], which implements reconfigurable match-action engines on FPGAs. However, these platforms leave the standard NIC functions to a separate ASIC-based NIC and operate entirely as a "bump-in-the-wire", providing no explicit control over

the NIC scheduler or queues.

Other projects use pure software implementations or partial hardware implementations. Shoal [53] describes a network architecture that performs cell routing with custom NICs and fast Layer 1 electrical crosspoint switches, with the crosspoint switches implementing a fixed schedule. Shoal was constructed in hardware, but was only evaluated with synthetic traffic with no connection to any hosts. For a scaled-up evaluation with realistic traffic and application-level benchmarking, Corundum could be used to provide a host interface for the Shoal NICs.

SENIC [45] describes scalable NIC-based rate-limiting, supporting 10s of thousands of hardware rate-limiters. A hardware implementation of a simplified scheduler was evaluated in isolation, but the system-level evaluation was carried out in software with a custom queuing discipline (qdisc) module. For a scaled-up evaluation with realistic traffic and application-level benchmarking, Corundum could be used to provide a datapath and host interface for the SENIC scheduler.

PIEO [52] describes a hardware architecture for a flexible NIC scheduler that is capable of implementing various types of rate-limiting and scheduling algorithms. The architecture was implemented and evaluated in hardware in isolation. For a scaled-up evaluation with realistic traffic and application-level benchmarking, Corundum could be used to provide a datapath and host interface for PIEO.

NDP [24] is a pull-mode transport protocol for datacenter applications. NDP is a receiver-pulled protocol, where the receivers control the rate of the senders. NDP was evaluated with DPDK software NICs and FPGA-based switches. For a scaled-up evaluation with realistic traffic and application-level benchmarking, Corundum could be used as a base for a hardware implementation of an NDP NIC, utilizing support for event-driven transmission control in Corundum.

Loom [56] describes a NIC architecture that supports fine-grained scheduling. The architecture operates on packet descriptors, classifying and enqueuing them in hierarchy of priority queues (also known as push-in, first-out or PIFOs). The architecture was evaluated as a

software NIC with BESS. For a scaled-up evaluation with realistic traffic and application-level benchmarking, Corundum could be used to provide a datapath for the Loom NIC architecture.

The development of Corundum is distinguished from all of these projects because it is completely open source and can operate with a standard host network stack at practical line rates. It provides thousands of transmit queues coupled with extensible transmit schedulers for fine-grained control of flows. This leads to a powerful and flexible open-source platform for the development of networking applications that combine both hardware and software functionalities.

3.1.2 Motivation and Previous Work: Circuit Switch Admission Control

A particular application of transmit scheduling is to enforce a time-division multiple-access (TDMA) schedule. This is a requirement in circuit-switched networks where data sent at the wrong time will not arrive at the intended destination, instead either getting lost during a switch reconfiguration or arriving at the incorrect destination.

In packet-switched networks, generally the transmit scheduler enforces fairness, ensuring that the link can be shared fairly between multiple applications, virtual machines, or other data sources, and/or limiting the data rate to prevent congestion elsewhere in the network. Since packet switches contain buffers, ensuring that the result is fair on average or the target rates are correct on average over long timescales is usually sufficient. In this case, software-based implementations such as Carousel [48] can be highly effective.

However, in circuit-switched networks, the switches operate at the level of links instead of packets. Circuit switches periodically break physical links and reconnect them in a new configuration. Any packets traversing the affected links during the reconfiguration process will be lost, and packets sent while the switch is in an incorrect configuration will arrive at the incorrect destination. Because of this, in a circuit-switched network, the precise timing of transmitted packets is paramount. Additionally, as stated in Chapter 2, the system-level reconfiguration time consists of not only the physical reconfiguration time of the switch itself, but also the locking time

of the receivers and any guard bands necessary to account for variance in time synchronization and packet transmissions. This means that the precision of the admission control method directly impacts the reconfiguration time of the whole system by determining the size of the guard bands. The precision of the admission control must be of a similar timescale to the reconfiguration time of the switch, otherwise the speed of a fast optical switch will be wasted.

Considering the guard delay in terms of bytes and packets is useful in addition to time, as the number of bytes scales with the line rate and the number of packets is dependent on both the line rate and packet size.

In software, achieving precise control over packet transmission is extremely difficult at microsecond timescales. This is due to a large number of sources of timing uncertainty in software, including operating system scheduling, interrupts, caching, branch prediction, etc. Additionally, the delay incurred through the driver and NIC can be quite variable, and with commercial NICs there is little visibility or control over what happens once a packet is handed off to the NIC for transmission.

In RotorNet [37], two different software-based admission control solutions were implemented. The first utilized a custom qdisc module in the linux kernel, and the second used a Myricom-specific "Sniffer" kernel bypass framework. The initial optical switch used for RotorNet has a switching time of 150 μs.

The qdisc solution was able to operate at 10 Gbps on an 800 μs period, divided into a 400 μs *day* for transferring data and a 400 μs *night* for reconfiguring the switch. This corresponds to a guard time of 250 μs, which is 312.5 KB or 208 1.5 KB MTU packets at 10 Gbps. Even with 250 μs of guard delay, this technique was not completely reliable at preventing packet loss—some experiment runs had around 0.1% packet loss, while a repeat with identical settings might result in no packet loss. The qdisc solution did not scale to bandwidths significantly higher than 10 Gbps, and the required guard time was heavily dependent on the NIC used.

The Myricom Sniffer solution was able to operate at 10 Gbps on a 1 ms, divided into a

650 µs *day* and a 350 µs *night*. This corresponds to a guard time of 200 µs, which corresponds to 250 KB or 167 1.5 KB MTU packets at 10 Gbps. Even with a 200 µs of guard delay, this technique was not completely reliable at preventing packet loss—some experiment runs had around 0.1% packet loss, while a repeat with identical settings might result in no packet loss. Additionally, the Myricom Sniffer solution supported only synthetic traffic generated within the userspace appliation, and is only compatible with Myricom NICs.

In an effort to scale precision admission control to higher data rates, a solution based on the Berkeley Extensible Software Switch (BESS) was investigated [32]. The BESS software NIC was capable of reliable operation at around 40 Gbps on a 50 µs period with a 5 µs guard delay, which corresponds to 62.5 KB or 42 1.5 KB MTU packets at 100 Gbps. However, this solution only worked with synthetic traffic generated within BESS, and it could not scale to higher line rates without significantly enlarging the guard delay.

Most recently, experiments with the hardware RDMA_WAIT primitive have shown some promise. On NICs that support RDMA_WAIT, it is possible for a control host to precisely trigger data transfers over the network. Since this is a NIC hardware primitive, the response latency is consistent and it operates at full line rate. The measured latency has a variance of around 13 µs. However, this technique can only be used on RDMA applications. Additionally, it requires a separate host to transmit the control packets which are subject to contention and other delays in the network as well as three queue pairs per destination.

Commercial smart NICs are another possible platform for implementing transmission control. However, many smart NICs are designed for line-rate packet processing and as such are not particularly conducive to implementing flow control across a large number of queues. In many cases, commercial smart NICs simply do not provide enough queues to provide one queue per destination or per flow. If packets are classified and enqueued on the NIC itself, then it is not possible to exert proper backpressure to the networking stack, resulting in head-of-line blocking and/or packet drops. For example, the Cavium LiquidIO smart NIC only supports 128

hardware transmit queues over the PCIe interface. It is possible to pause these queues and apply backpressure to the network stack, but the number of queues is too small to be useful.

A TDMA implementation on a Netronome NFP-3240 NIC with 16 queues was able to operate on a 100 µs period with 10 µs guard delays [65]. However, the NIC only supported operation at around 5 Gbps, and since packet classification was implemented on the NIC, it was susceptible to head-of-line blocking and/or packet loss.

Other commercial smart NICs, such as the Mellanox Innova Flex and the Solarflare Application Onload Engine, actually provide an FPGA on the NIC in conjunction with a NIC ASIC. However, these NICs are designed for application offload and as such the FPGA may not be integrated in a way that provides any control over the flow of data through the NIC. For example, in the case of the Innova Flex NIC, the only connection to the FPGA is a direct PCIe link to a PCIe switch on the NIC ASIC. As a result, the FPGA has no control over the NIC datapath at all and therefore cannot be used to enforce an admission control scheme.

In contrast, the Corundum NIC with its hardware TDMA scheduler controller provides scalable, accurate TDMA performance across thousands of hardware-managed queues. It is capable of enforcing a TDMA schedule for arbitrary traffic at full 100 Gbps line rate on a 100 µs period with a guard delay of less than 2 µs with no software overhead, which corresponds to 25.5 KB or 17 1.5 KB MTU packets at 100 Gbps. This delay is an upper bound and can be improved with further optimizations. The development of this capability enables the construction of practical sub-microsecond circuit-switched networks.

3.2 Background

The network interface controller (NIC) is the gateway that connects software running on a computer to other computers over a network. It is responsible for sending data from application software on the wire, and for receiving data from the wire to pass to application software for

processing.

In many modern computers, the NIC sits on the PCIe bus and interacts with the host CPU via memory read and write operations, which are called memory-mapped IO (MMIO) from the host to the NIC and direct memory access (DMA) from the NIC to the host.

3.2.1 NIC-to-Host Interface: PCI Express

In many modern computers and servers, the NIC is a peripheral that sits on the PCI express bus. There are some exceptions to this where the NIC is either integrated onto the same die as the CPU, including the SPARC Network Interface Unit on certain SPARC processors and many system-on-chip based devices, or connected using some other interface, such as OpenCAPI or CCIX. This dissertation will focus on NICs connected via PCI express.

PCI express is a common interface that serves to connect the processor to high bandwidth peripherals in a computer. The high-level operation of PCI express draws heavily on conventional PCI as it was intended to be a drop-in replacement from the standpoint of software—it shares the same configuration spaces, enumeration, bus number and address assignments, etc. as conventional PCI. However, unlike the shared parallel bus of conventional PCI, PCI express is a packet-switched interface built from point-to-point links in a tree topology.

The *root* of a PCI express system is the root complex, which connects the PCI express components to the host CPU and system memory. The root complex contains one or more root ports that can be connected to PCI express devices and switches. PCI express devices are connected to root ports either directly or via PCIe switches in a tree topology.

The transaction layer of PCI express is responsible for transferring data in the form of memory read and write operations. PCI express devices are allocated ranges of system address space via base address registers. PCI express switches are configured with the address range of the devices behind them so they can route requests appropriately. Reliable transmission and order preservation of transaction layer packets, or TLPs, is ensured by link-level sequence numbers and

CRCs (LCRC) while traversing links and a retransmission scheme to transparently replay lost or corrupted TLPs.

Transaction layer packets that carry PCIe traffic consist of a header and a payload. The payload size is restricted to a configurable maximum payload size parameter which can range from 128 to 4096 bytes. This parameter is available in the PCIe configuration space of each device. Most systems support a maximum payload size of 128 or 256 bytes. The maximum payload size mainly affects write requests and completions carrying data in response to read requests. The maximum length of a read request is set by a different parameter, the maximum read request size, which similarly ranges from 128 to 4096 bytes. Most systems set this to 512 bytes, but is common to see it set higher.

The protocol overhead of PCIe TLPs includes 12 or 16 bytes at the transaction layer for the TLP header plus an additional 8 bytes at the link layer for the LCRC, sequence number, and framing.

There are two main ways of transferring data over the PCIe bus. First, when the CPU initiates memory read and write operations against devices, this is referred to as memory-mapped IO (MMIO). Second, when a device other than the CPU initiates a memory read or write operation, this is referred to as direct memory access (DMA). When the target is another device instead of system memory, this is called peer-to-peer DMA.

Transferring large amounts of data over PCI express traditionally relies on the devices themselves to manage the transfers through DMA instead of the CPU. There are several reasons for this. The central reason is that the CPU can only operate on very small chunks of data at once—perhaps 8 bytes at a time for a 64 bit CPU, and possibly up to the size of a cache line with the aid of prefetching and write combining. Because of this, it requires a large number of CPU operations to read the data out of main memory into CPU registers, then write it out again to the PCIe device, resulting in high CPU utilization. Additionally, each one of those writes will likely traverse the bus as a single write operation, incurring at least 20 bytes additional overhead in TLP

headers and link layer overhead, resulting in very poor utilization of the PCIe link. When DMA is used, a device can directly request a large block of data with a single PCIe read request, which then comes back as one or more completions that can be far larger than the CPU word or cache line size, resulting in significantly reduced CPU load as well as significantly improved utilization of the PCIe interface. Additionally, the device can initiate the read or write on its own terms, when it is ready for the data, reducing buffering requirements. This is especially important for inbound network traffic that can arrive at any time, even while the CPU is busy.

PCIe also supports the use of interrupts through in-band messages. PCIe supports 3 types of interrupts: legacy, message-signaled interrupts (MSI), and MSI-X. Legacy interrupts use PCIe messages and emulate the shared interrupt lines of PCI. MSI and MSI-X use memory writes to addresses associated with the system interrupt controller to trigger interrupts.

3.2.2 Network Traffic Over PCIe

Transferring network traffic to and from the NIC over PCIe requires a careful design. The goal is to maximize throughput over the PCIe bus and to minimize per-packet CPU overhead. To that end, it is necessary to use direct memory access (DMA) to perform most of the actual data transfers and minimize the use of MMIO.

From the NIC end, sending and receiving packets over PCIe simply requires reading them out of memory via DMA read requests or writing them into memory via DMA write requests. In the transmit direction, each packet would be read with a series of one or more read requests, ideally the maximum possible size (max read request size), extracting the data from the completions (max payload size), and sending the packet on the wire. In the receive direction, each packet would be written out with a series of one or more write requests, again ideally of the maximum possible size (max payload size). This requires the minimum possible number of PCIe TLPs, and hence the minimum possible PCIe overhead.

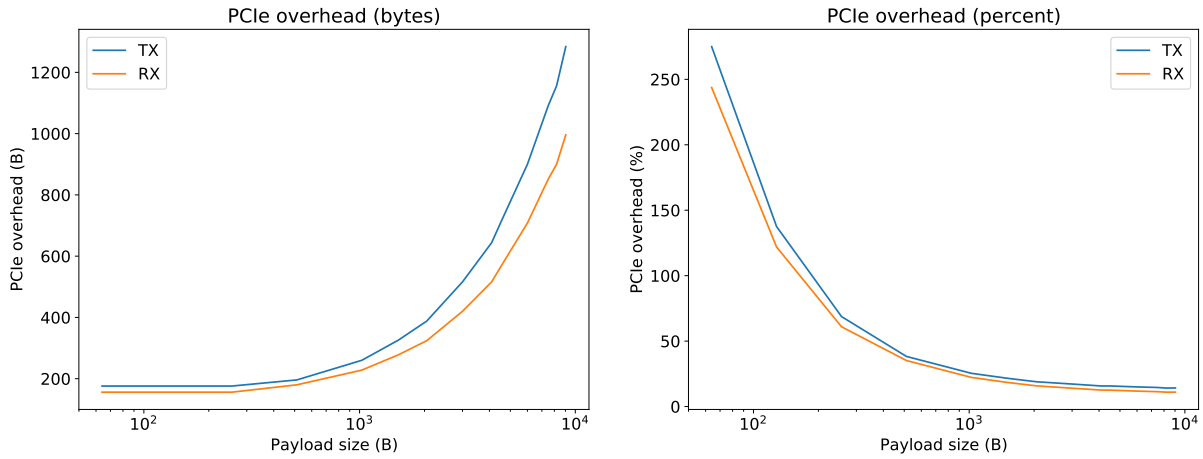
However, this is only part of the story—the NIC has to know where transmit packets are

located in memory, and where it can write received packets. It also has to be able to communicate when it is finished processing each packet. This is the function of the driver. The device driver is the software counterpart to the NIC; it is responsible for connecting the NIC to the operating system and ultimately to applications that require access to the network.

In the case of outgoing packets, the driver must take packets from the operating system and then inform the NIC of the corresponding memory address and length of each packet. Once the NIC has sent the packets on the wire, the driver must free the associated resources (memory, etc.) so they can be reused. In the case of incoming packets, the driver must allocate space, inform the NIC of the corresponding memory addresses, and then hand off the received packets to the operating system once the NIC is finished receiving each packet. Coordinating all of this requires some form of communication channel between the driver and the NIC.

Context switching on the software side is another factor to take into consideration. Drivers are event-driven software, so the driver sits around idle until something happens, then is called in to action, either by the operating system/application software or by an interrupt. Once running, it makes sense to try to do as much work as possible—instead of processing one single packet at a time, the driver should be able to process as many packets as possible, amortizing the context switching overhead across many packets. It is also important to minimize the number of interrupts that the NIC generates as handling interrupts in the operating system is expensive. To that end, queues are a very common sight in network stacks. Packets from application software can be enqueued by the operating system, and then the driver can dequeue and process several packets in one operation.

Queues also form an efficient means of communication between the driver and the NIC. Namely, descriptors describing the memory address and size of packets to be sent can be written in to a DMA accessible buffer, then the NIC can read the descriptors out and process the packets at its leisure. When finished with a packet, the NIC can write a completion record into a different DMA accessible buffer than the driver can read out and process. The driver notifies the NIC of



(a) Overhead in bytes

(b) Overhead as percentage of payload

Figure 3.1: NIC PCIe bandwidth overhead

enqueued packets via MMIO, and the NIC notifies the driver of completions via interrupts. This communication technique is quite efficient and quite common in commercial NICs.

The overheads of the TLP headers and descriptor and completion data affect the rate at which network packets can be sent over the PCIe bus. Fig. 3.1 depicts the overhead required for transmitting and receiving packets over PCI express using descriptor and completion queues for communication with the driver, assuming a max TLP size of 256 bytes and a max read request size of 512 bytes. The overhead is quite significant for larger packet sizes; around 1 KB for 9 KB MTU frames. However, the more important figure is the overhead as a fraction of the packet size. In this case, the overhead for 64 byte minimum size frames is 156 bytes for RX or 176 bytes for TX, or around 250%. This overhead can significantly limit throughput for small packets. There are two common methods to reduce PCIe overheads. The first is to batch descriptor reads and completion writes to reduce the TLP header overhead. The second method is to inline small packets or packet headers directly in the descriptor queue, avoiding the overhead of a separate DMA read for the payload data.

3.2.3 OS-Driver Interface

The network device driver interfaces with the operating system through some set of APIs. These APIs enable the operating system to hand packets off to the driver for transmission as well as enable the driver to hand received packets off to the operating system for processing. There will be additional APIs for configuration. This dissertation will focus on Linux, but other operating systems will have a similar interface.

In a Linux network device driver, the main interface from the operating system is via a `net_device` object. The driver allocates, configures, and registers one `net_device` object for each network interface. The `net_device` structure has a field called `netdev_ops`. The driver initializes this field to point to a `net_device_ops` structure that contains a number of function pointers. These function pointers implement various pieces of driver functionality that the operating system will call on when needed. Most of these are for configuration and other ancillary functions, but the most important one is `ndo_start_xmit`. This function is called with a reference to the `sk_buff` for the packet that the kernel wants to transmit.

In the receive direction, the driver allocates `sk_buff` structures from the kernel with `netdev_alloc_skb_ip_align` and passes the associated memory to the NIC by writing descriptors in the receive queues. The NIC writes out the receive packet data and hands them back to the driver, which in turn hands the packets to the kernel with `napi_gro_receive`. It is also possible to allocate memory pages with `dev_alloc_pages`, hand those off to the NIC to fill with receive data, then attach the pages to an `sk_buff` from `napi_get_frags`, and then hand it off to `napi_gro_frags`.

3.2.4 Offloading

For operation at high data rates, minimizing CPU cycles per packet is paramount. Zero copy transmit and receive operations are one component of this—copying data using the CPU

is expensive, so keeping it in the same location in memory and having the NIC read it out from that location significantly reduces CPU overhead. The same goes for receive operations: if receive buffers are allocated properly, they can be directly handed off to the operating system for processing.

Getting the data organized is only one piece of the puzzle. Many protocols require some form of checksumming for data integrity. All Ethernet frames have a frame check sequence (FCS) attached that consists of a cyclic redundancy check (CRC) that's computed over the entire payload. Similarly, TCP and UDP headers contain a ones complement checksum of the packet header and payload data. Computing and checking these checksums is also an expensive operation if done in software.

If these operations are performed in hardware, this results in a significant savings in CPU overhead. Most NICs offload the Ethernet FCS computation and verification; this is a standard function of most Ethernet MACs. On may high performance NICs, this is actually a mandatory offload; the hardware does not support bypassing the FCS computation and verification.

Since TCP and UDP are extremely common protocols, offloading TCP and UDP checksums to the NIC is also a common feature. In this case, there is a trade-off between complexity and flexibility. Some NICs are capable of performing a full checksum offload in hardware, interpreting the header fields, summing the correct portion of the packet, and then appropriately inserting or verifying the checksum. This is complicated to implement in hardware, and baking the functionality into hardware means that it cannot be adapted to new protocols. A more flexible approach is to leave interpreting the packet headers to software, and have the hardware only deal with computing checksums over the payload data, under the direction of software. In this way, the hardware implementation on the NIC is simple and can be reused for many protocols.

Therefore, the standard transmit checksum offload involves computing the header checksum in the network stack, inserting it into the checksum field in the packet header, then handing the packet off to the NIC to complete the checksum. The NIC computes a ones-complement

checksum from a network-stack-supplied start offset to the end of the packet, then inserts the checksum into the packet header at the offset provided by the network stack. In the receive direction, the NIC simply computes the ones complement checksum of the entire Ethernet frame payload and returns this in the completion record. The network stack must then adjust the checksum by subtracting out packet header fields that should not be part of the checksum.

3.3 Implementation

Corundum has several unique architectural features. First, hardware queue states are stored efficiently in FPGA block RAM, enabling support for thousands of individually-controllable queues. These queues are associated with interfaces, and each interface can have multiple ports, each with its own independent transmit scheduler. This enables extremely fine-grained control over packet transmission. The scheduler module is designed to be modified or swapped out completely to implement different transmit scheduling schemes, including experimental schedulers. Coupled with PTP time synchronization, this enables time-based scheduling, including high precision TDMA.

The design of Corundum is modular and highly parametrized. Many configuration and structural options can be set at synthesis time by Verilog parameters, including interface and port counts, queue counts, memory sizes, scheduler type, etc. These design parameters are exposed in configuration registers that the driver reads to determine the NIC configuration, enabling the same driver to support many different boards and configurations without modification[2].

The current design supports PCIe DMA components for the Xilinx UltraScale PCIe hard IP core interface. Support for the PCIe TLP interface commonly used in other FPGAs is not implemented, and is future work. This support should enable operation on a much larger set of FPGAs.

[2]Corundum codebase: https://github.com/ucsdsysnet/corundum

The footprint of Corundum is rather small, leaving ample space available for additional logic, even on relatively small FPGAs. For example, the Corundum design for the ExaNIC X10 [10], a dual port 10G design with a PCIe gen 3 x8 interface and 512 bit internal datapath, consumes less than a quarter of the logic resources available on the second smallest Kintex UltraScale FPGA (KU035). Table 3.1 lists the resources for several target platforms.

The rest of this section describes the implementation of Corundum on an FPGA. First, a high-level overview of the main functional blocks is presented. Then, details of several of the unique architectural features and functional blocks are discussed.

3.3.1 High-Level Overview

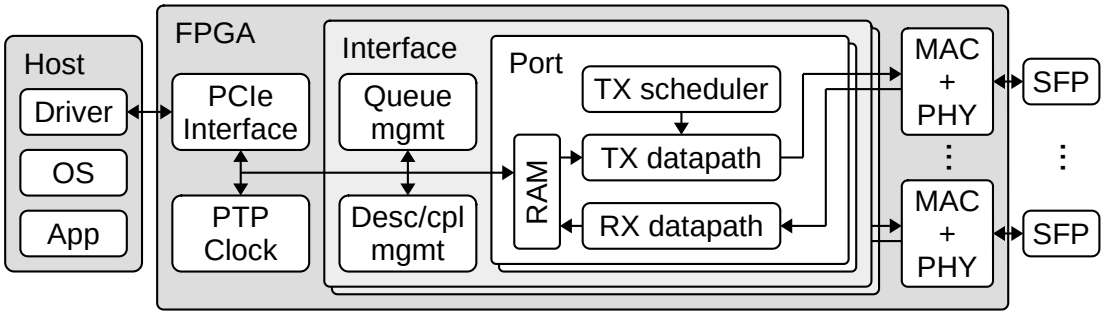


Figure 3.2: Simplified block diagram of the Corundum NIC.

A simplified block diagram of the Corundum NIC is shown in Fig. 3.2. At a high level, the NIC consists of 3 main nested modules. The top-level module primarily contains support and interfacing components. These components include the PCI express hard IP core and DMA interface, the PTP hardware clock, and Ethernet interface components including MACs, PHYs, and associated serializers. The top-level module also includes one or more `interface` module instances. The interface module corresponds to an operating-system-level network interface (e.g. eth0). Each network interface module contains the queue management logic which maintains the queue state for all of the NIC queues—transmit, transmit completion, receive, receive completion, and event—as well as descriptor and completion handling logic. The network interface module

also contains one or more `port` module instances. The port module contains the transmit and receive datapaths, transmit and receive engines, transmit scheduler, and a scratchpad RAM for temporarily storing incoming and outgoing packets during DMA operations.

For each network interface, the transmit scheduler in the port module decides which queues to send from. The transmit scheduler generates commands for the transmit engine, which coordinates operations on the transmit datapath. The scheduler module is a flexible functional block that can be modified to support arbitrary schedules, which may be event driven. The default implementation of the scheduler is simple round robin. All ports that use the same interface module share the same set of transmit queues and appear as a single, unified interface to the operating system. This enables flows to be migrated between ports or load-balanced across multiple ports by changing only the transmit scheduler settings without affecting the rest of the network stack. This dynamic, scheduler-defined mapping of queues to ports is a unique feature of Corundum that can enable research into new protocols and network architectures, including parallel networks such as P-FatTree [38] and optically-switched networks such as RotorNet [37] and Opera [35].

In the receive direction, incoming packets pass through a flow hash module to determine the target receive queue and generate a command for the receive engine, which coordinates operations on the receive datapath. Since all ports in the same interface module share the same set of receive queues, incoming flows on different ports are merged together into the same set of queues. It is also possible to add customized modules to the NIC to perform processing on incoming data before it traverses the PCIe bus.

A more detailed block diagram of Corundum is shown in Fig. 3.3. This diagram includes key functional blocks and interconnections inside the NIC.

The components on the NIC are interconnected with several different interfaces including AXI lite, AXI stream, and a custom segmented memory interface for DMA operations, which will be discussed later. AXI lite is used for the control path from the driver to the NIC. It is

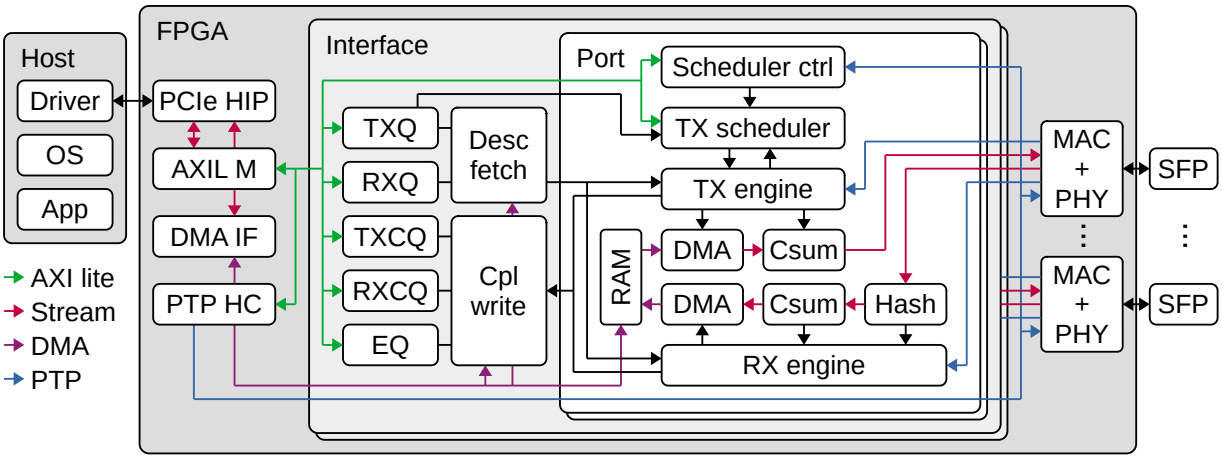


Figure 3.3: Block diagram of the Corundum NIC. PCIe HIP = PCIe hard IP core; AXIL M = AXI lite master; DMA IF = DMA interface; PTP HC = PTP hardware clock; TXQ = transmit queue manager; TXCQ = transmit completion queue manager; RXQ = receive queue manager; RXCQ = receive completion queue manager; EQ = event queue manager; MAC + PHY = Ethernet media access controller (MAC) and physical interface layer (PHY).

used to initialize and configure the NIC components and to control the queue pointers during transmit and receive operations. AXI stream interfaces are used for transferring packetized data within the NIC, including both PCIe transmission layer packets (TLPs) and Ethernet frames. The segmented memory interface serves to connect the PCIe DMA interface to the NIC datapath and to the descriptor and completion handling logic.

The majority of the NIC logic runs in the PCIe user clock domain, which is nominally 250 MHz for all of the current design variants. Asynchronous FIFOs are used to interface with the MACs, which run in the serializer transmit and receive clock domains as appropriate—156.25 MHz for 10G, 390.625 MHz for 25G, and 322.266 MHz for 100G.

The following sections describe key functional blocks within the NIC.

3.3.2 Pipelined Queue Management

Communication of packet data between the Corundum NIC and the driver is mediated via descriptor and completion queues. Descriptor queues form the host-to-NIC communications channel, carrying information about where individual packets are stored in system memory.

Completion queues form the NIC-to-host communications channel, carrying information about completed operations and associated metadata. The descriptor and completion queues are implemented as ring buffers that reside in DMA-accessible system memory, while the NIC hardware maintains the necessary queue state information. This state information consists of a pointer to the DMA address of the ring buffer, the size of the ring buffer, the producer and consumer pointers, and a reference to the associated completion queue. The required state for each queue fits into 128 bits.

The queue management logic for the Corundum NIC must be able to efficiently store and manage the state for thousands of queues. This means that the queue state must be completely stored in block RAM (BRAM) or ultra RAM (URAM) on the FPGA. Since a 128 bit RAM is required and URAM blocks are 72x4096, storing the state for 4096 queues requires only 2 URAM instances. Utilizing URAM instances enables scaling the queue management logic to handle at least 32,768 queues per interface.

In order to support high throughput, the NIC must be able to process multiple descriptors in parallel. Therefore, the queue management logic must track multiple in-progress operations, reporting updated queue pointers to the driver as the operations are completed. The state required to track in-process operations is much smaller than the state required to describe the queue state itself. Therefore the in-process operation state is stored in flip-flops and distributed RAM.

The NIC design uses two queue manager modules: `queue_manager` is used to manage host-to-NIC descriptor queues, while `cpl_queue_manager` is used to manage NIC-to-host completion queues. The modules are similar except for a few minor differences in terms of pointer handling, fill handling, and doorbell/event generation. Because of the similarities, this section will discuss only the operation of the `queue_manager` module.

The BRAM or URAM array used to store the queue state information requires several cycles of latency for each read operation, so the `queue_manager` is built with a pipelined architecture to facilitate multiple concurrent operations. The pipeline supports four different operations:

register read, register write, dequeue/enqueue request, and dequeue/enqueue commit. Register-access operations over an AXI lite interface enable the driver to initialize the queue state and provide pointers to the allocated host memory as well as access the producer and consumer pointers during normal operation.

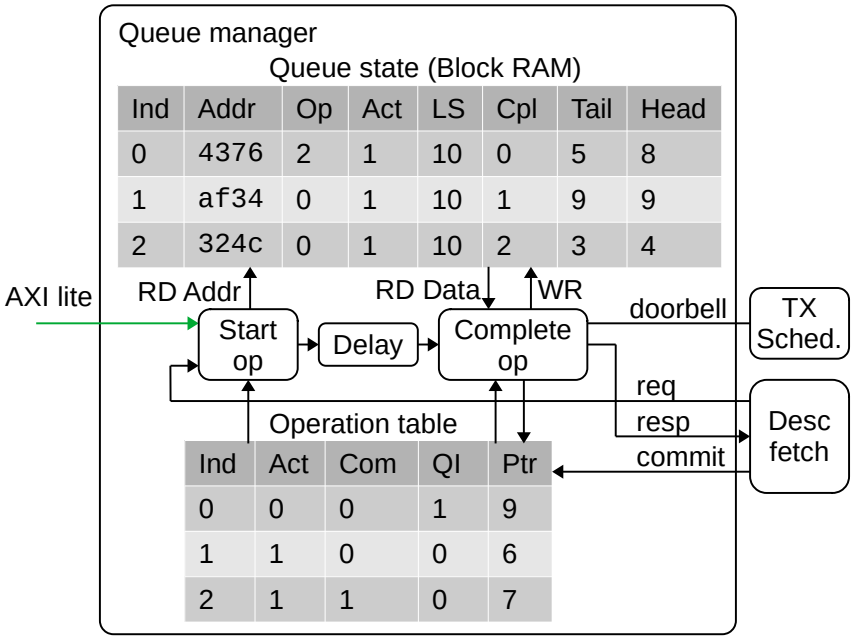


Figure 3.4: Block diagram of the queue manager module, showing the queue state RAM and operation table. Ind = index, Addr = DMA address, Op = index in operation table, Act = active, LS = log base 2 of queue size, Cpl = completion queue index, Tail = tail or consumer pointer, Head = head or producer pointer, Com = committed; QI = queue index; Ptr = new queue pointer

A block diagram of the queue manager module is shown in Fig. 3.4. The BRAM or URAM array used to store the queue state information requires several cycles of latency for each read operation, so the `queue_manager` is built with a pipelined architecture to facilitate multiple concurrent operations. The pipeline supports four different operations: register read, register write, dequeue/enqueue request, and dequeue/enqueue commit. Register-access operations over an AXI lite interface enable the driver to initialize the queue state and provide pointers to the allocated host memory as well as access the producer and consumer pointers during normal operation.

Each queue has three pointers associated with it, as shown in Fig. 3.5—the producer

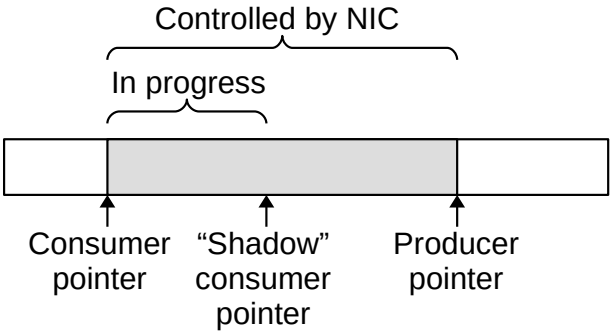


Figure 3.5: Queue pointers on software ring buffers.

pointer, the host-facing consumer pointer, and the shadow consumer pointer. The driver has control over the producer pointer and can read the host-facing consumer pointer. Entries between the consumer pointer and the producer pointer are under the control of the NIC and must not be modified by the driver. The driver enqueues a descriptor by writing it into the ring buffer at the index indicated by the producer pointer, issuing a memory barrier, then incrementing the producer pointer in the queue manager. The NIC dequeues descriptors by reading them out of the descriptor ring via DMA and incrementing the consumer pointer. The host-facing consumer pointer must not be incremented until the descriptor read operation completes, so the queue manager maintains an internal shadow consumer pointer to keep track of read operations that have started in addition to the host-facing pointer that is updated as the read operations are completed.

The dequeue request operation on the queue manager pipeline initiates a dequeue operation on a queue. If the target queue is disabled or empty, the operation is rejected with an *empty* or *error* status. Otherwise, the shadow consumer pointer is incremented and the physical address of the queue element is returned, along with the queue element index and an operation tag. Operations on any combination of queues can be initiated until the operation table is full. The dequeue request input is stalled when the table is full. As the read operations complete, the dequeue operations are committed to free the operation table entry and update the host-facing consumer pointer. Operations can be committed in any order, simply setting the commit flag in the operation table, but the operation table entries will be freed and host-facing consumer pointer

will be updated in-order to ensure descriptors being processed are not modified by the driver.

The operation table tracks in-process queue operations that have yet to be committed. Entries in the table consist of an active flag, a commit flag, the queue index, and the index of the next element in the queue. The queue state also contains a pointer to the most recent entry for that queue in the operation table. During an enqueue operation, the operation table is checked to see if there are any outstanding operations on that queue. If so, the consumer pointer for the most recent operation is incremented and stored in the new operation table entry. Otherwise, the current consumer pointer is incremented. When a dequeue commit request is received, the commit bit is set for the corresponding entry. The entries are then committed in-order, updating the host-facing consumer pointer with the pointer from the operation table and clearing the active bit in the operation table entry.

Both the queue manager and completion queue manager modules generate notifications during enqueue operations. In a queue manager, when the driver updates a producer pointer on an enabled queue, the module issues a doorbell event that is passed to the transmit schedulers for the associated ports. Similarly, completion queue managers generate events on hardware enqueue operations, which are passed to the event subsystem and ultimately generate interrupts. To reduce the number of events and interrupts, completion queues also have an *armed* status. An armed completion queue will generate a single event, disarming itself in the process. The driver must re-arm the queue after handling the event.

3.3.3 Transmit Scheduler

The default transmit scheduler used in the Corundum NIC is a simple round-robin scheduler implemented in the `tx_scheduler_rr` module. The scheduler sends commands to the transmit engine to initiate transmit operations out of the NIC transmit queues. The round-robin scheduler contains basic queue state for all queues, a FIFO to store currently-active queues and enforce the round-robin schedule, and an operation table to track in-process transmit operations.

Similar to the queue management logic, the round-robin transmit scheduler also stores queue state information in BRAM or URAM on the FPGA so that it can scale to support a large number of queues. The transmit scheduler also uses a processing pipeline to hide the memory access latency.

The transmit scheduler module has four main interfaces: an AXI lite register interface and three streaming interfaces. The AXI lite interface permits the driver to change scheduler parameters and enable/disable queues. The first streaming interface provides doorbell events from the queue management logic when the driver enqueues packets for transmission. The second streaming interface carries transmit commands generated by the scheduler to the transmit engine. Each command consists of a queue index to transmit from, along with a tag for tracking in-process operations. The final streaming interface returns transmit operation status information back to the scheduler. The status information informs the scheduler of the length of the transmitted packet, or if the transmit operation failed due to an empty or disabled queue.

The transmit scheduler module can be extended or replaced to implement arbitrary scheduling algorithms. This enables Corundum to be used as a platform to evaluate experimental scheduling algorithms, including those proposed in SENIC [45], Carousel [48], PIEO [52], and Loom [56]. It is also possible to provide additional inputs to the transmit scheduler module, including feedback from the receive path, which can be used to implement new protocols and congestion control techniques such as NDP [24] and HPCC [29]. Connecting the scheduler to the PTP hardware clock can be used to support TDMA, which can be used to implement RotorNet [37], Opera [35], and other circuit-switched architectures.

The structure of the transmit scheduler logic is similar to the queue management logic in that it stores queue state in BRAM or URAM and uses a processing pipeline. However there are a number of significant differences. First, the scheduler logic is designed so that the scheduler does not stall when a queue is empty and a subsequent dequeue operation fails. Second, the scheduler contains a FIFO to enforce the round-robin schedule. The use of this FIFO requires an explicit

reset routine to make the internal state (namely the scheduled flag bits) consistent after a reset. Third, the scheduler also contains logic to track the active state of each queue based on incoming doorbell requests and dequeue failures.

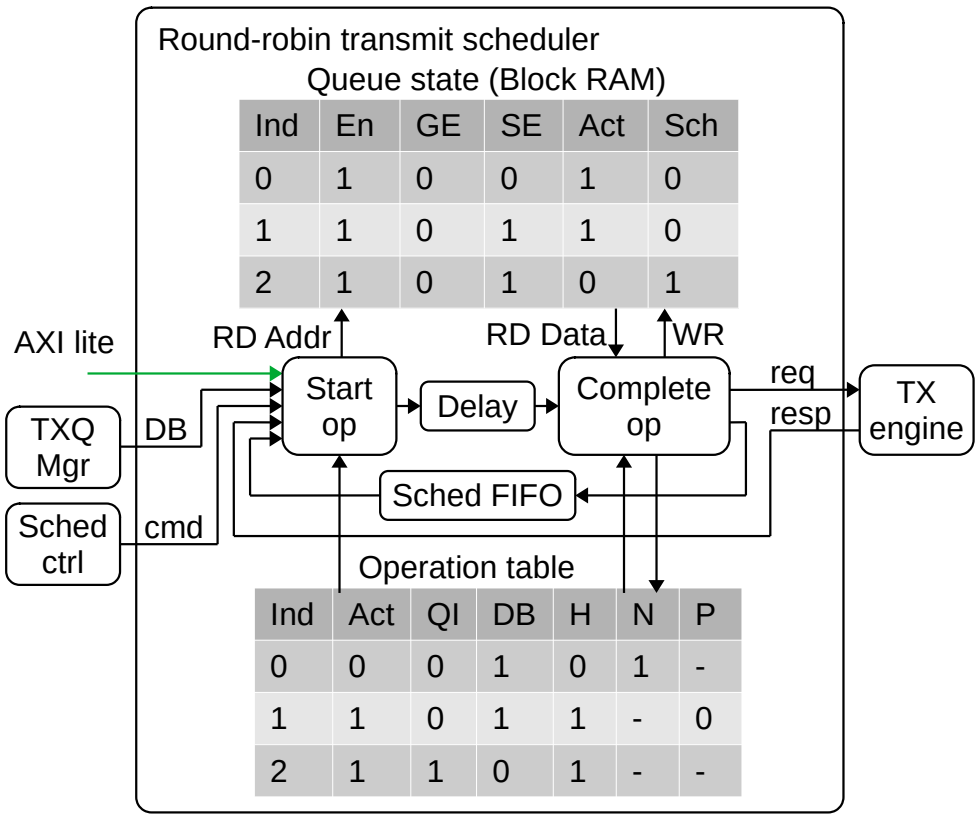


Figure 3.6: Block diagram of the transmit scheduler module, showing queue state RAM and operation table. Ind = index, En = queue enable, GE = global enable, SE = schedule enable, Act = active, Sch = scheduled, QI = queue index, DB = doorbell, H = head, N = next, P = previous

A block diagram of the transmit scheduler module is shown in Fig. 3.6. The transmit scheduler is built around a *scheduled queue* FIFO. This FIFO stores the indices of the currently-scheduled queues. An active queue is one that is presumed to have at least one packet available for transmission, an enabled queue is one that has been enabled for transmission, and a scheduled queue is one that has an entry in the scheduler FIFO. A queue will be scheduled (marked as scheduled and inserted into the FIFO) if it is both active and enabled. A queue will be descheduled when it reaches the front of the schedule FIFO, but is not enabled or not active. Queue enable

states are controlled via three different enable bits per queue: queue enable, global enable, and schedule enable. The queue enable and global enable bits are writable via AXI lite, while the schedule enable bit is controlled from the scheduler control module via an internal interface. A queue is enabled when the queue enable bit and either the global enable or schedule enable bits are set. Queues become active when doorbell events are received, and queues become inactive when a transmit request fails due to an empty queue.

Tracking the queue active states must be done carefully for several reasons. First, the driver can update the producer pointer after enqueuing more than one packet, so the number of generated doorbell events does not necessarily correspond to the number of packets that were enqueued. Second, because the queues are shared among all ports on the same interface, multiple ports can attempt to send packets from the same queue, and the port transmit schedulers have no visibility into what the other schedulers are doing. Therefore, the most reliable method for determining that a queue is empty is to try sending from it, and flagging the failure. Note that the cost of an error is much higher when the queue is active than when the queue is empty. Attempting to send from an empty queue costs a few clock cycles and temporarily occupies a few slots in corresponding operation tables. However, assuming a queue is empty when it is not will result in packets getting stuck in the queue. Fixing this stuck queue will not occur until the OS sends another packet on that queue and triggers another doorbell. Therefore, it is imperative to properly track doorbell events during transmit operations, as it is possible for a doorbell event to arrive after a dequeue attempt has failed, but before the failed transmit status arrives at the transmit scheduler module.

The pipeline in the transmit scheduler supports seven different operations: initialize, register read, register write, handle doorbell, transmit complete, scheduler control, and transmit request. The initialize operation is used to ensure the scheduler state is consistent after a reset. Register access operations over an AXI lite interface enable the driver to read all of the per-queue state and set the queue enable and global enable bits. The pipeline also handles incoming doorbell

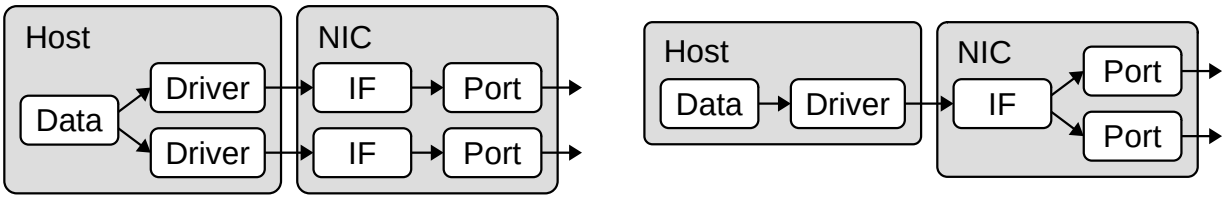
requests from the transmit queue manager module as well as queue enable/disable requests from the scheduler control module. Finally, the transmit request and transmit complete operations are used to generate transmit requests and handle the necessary queue state updates when the transmit operations complete.

Queues can become scheduled based on a register write that enables an active queue, a doorbell that activates an enabled queue, a scheduler operation that enables an active queue, and a transmit completion on an enabled queue that is either successful or has the doorbell bit set in the operation table. Queues can only be descheduled when the queue index advances to the front of the scheduler FIFO. If this occurs when the queue is both active and enabled, then the queue can be rescheduled and a transmit request generated. When the transmit operation completes, the transmit status response will be temporarily stored in a small FIFO and then processed by the pipeline to update the corresponding operation table entry and, if necessary, reschedule the queue.

The operation table tracks in-process transmit operations. Entries in the table consist of an active flag, the queue index, a doorbell flag, a head flag, a next pointer, and a previous pointer. The next and previous pointers form a linked list, enabling entries to be removed in any order while preserving the doorbell flag in the table. This prevents doorbells from getting 'lost' and the queue being mistakenly marked as inactive. A separate linked list is formed for each queue with active transmit operations. The operation table is implemented in such a way that it fits in distributed RAM.

3.3.4 Ports and Interfaces

A unique architectural feature of Corundum is the split between the port and the network interface so that multiple ports can be associated with the same interface. Most current NICs support a single port per interface, as shown in Fig. 3.7a. When the network stack enqueues a packet for transmission on a network interface, the packets are injected into the network via the network port associated with that interface. However, in Corundum, multiple ports can be



(a) Traditional NIC, port assignment made in software

(b) Corundum NIC, port assignment made in hardware

Figure 3.7: NIC port and interface architecture comparison

associated with each interface, so the decision over which port a packet will injected into the network can be made by hardware at the time of dequeue, as shown in Fig. 3.7b.

All ports associated with the same network interface module share the same set of transmit queues and appear as a single, unified interface to the operating system. This enables flows to be migrated between ports or load-balanced across multiple ports by changing only the transmit scheduler settings without affecting the rest of the network stack. The dynamic, scheduler-defined mapping of queues to ports enables research into new protocols and network architectures, including parallel networks such as P-FatTree [38] and optically-switched networks such as RotorNet [37] and Opera [35].

3.3.5 Datapath and Transmit and Receive engines

Corundum uses both memory-mapped and streaming interfaces in the datapath. AXI stream is used to transfer Ethernet packet data between the port DMA modules, Ethernet MACs, and the checksum and hash computation modules. AXI stream is also used to connect the PCIe hard IP core to the PCIe AXI lite master and PCIe DMA interface modules. A custom, segmented memory interface is used to connect the PCIe DMA interface module, port DMA modules, and descriptor and completion handling logic to internal scratchpad RAM.

The width of the AXI stream interfaces is determined by the required bandwidth. The core datapath logic, except the Ethernet MACs, runs entirely in the 250 MHz PCIe user clock domain. Therefore, the AXI stream interfaces to the PCIe hard IP core must match the hard core

interface width—256 bits for PCIe gen 3 x8 and 512 bits for PCIe gen 3 x16. On the Ethernet side, the interface width matches the MAC interface width, unless the 250 MHz clock is too slow to provide sufficient bandwidth. For 10G Ethernet, the MAC interface is 64 bits at 156.25 MHz, which can be connected to the 250 MHz clock domain at the same width. For 25G Ethernet, the MAC interface is 64 bits at 390.625 MHz, necessitating a conversion to 128 bits to provide sufficient bandwidth at 250 MHz. For 100G Ethernet, the MAC interface is 512 bits at 322.266 MHz, which is connected to the 250 MHz clock domain at 512 bits because it only needs to run at around 195 MHz to provide 100 Gbps.

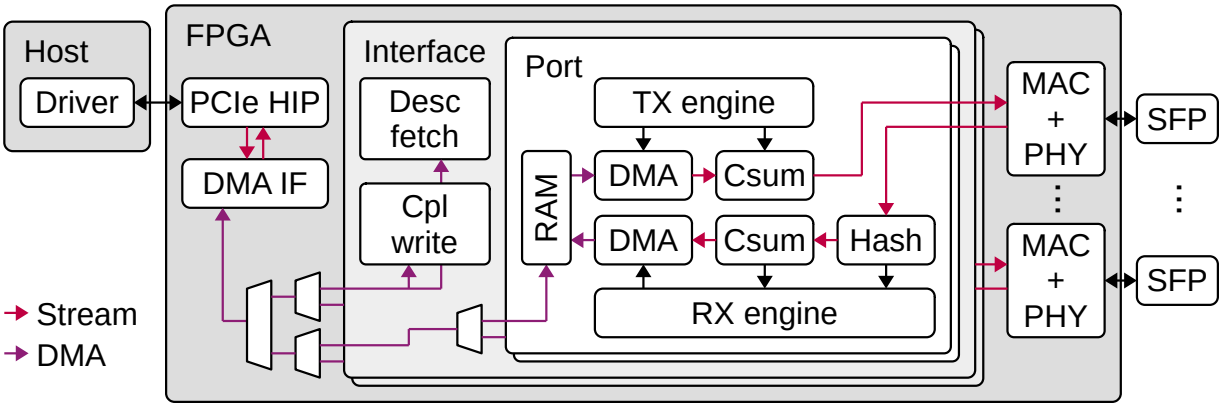


Figure 3.8: Simplified version of Fig. 3.3 showing the NIC datapath.

A block diagram of the NIC datapath is shown in Fig. 3.8, which is a simplified version of Fig. 3.3. The PCIe hard IP core (PCIe HIP) connects the NIC to the host. Two AXI stream interfaces connect the PCIe DMA interface module to the PCIe hard IP core—one for read and write requests, one for read data. The PCIe DMA interface module is then connected to the descriptor fetch module, completion write module, port scratchpad RAM modules, and the RX and TX engines via a set of DMA interface multiplexers. In the direction towards the DMA interface, the multiplexers combine DMA transfer commands from multiple sources. In the opposite direction, they route transfer status responses. They also manage the segmented memory interfaces for both reads and writes. The top-level multiplexer combines descriptor traffic with packet data traffic, giving the descriptor traffic higher priority. Next, a pair of multiplexers

combine traffic from multiple interface modules. Finally, an additional multiplexer inside each interface module combines packet data traffic from multiple port instances.

The transmit and receive engines are responsible for coordinating many of the operations necessary for transmitting and receiving packets. The transmit and receive engines can handle multiple in-progress packets for high throughput. As shown in Fig. 3.3, the transmit and receive engines are connected to several modules in the transmit and receive data path, including the port DMA modules and hash and checksum offload modules, as well as the descriptor and completion handling logic and the timestamping interfaces of the Ethernet MACs.

The transmit engine is responsible for coordinating packet transmit operations. The transmit engine handles transmit requests for specific queues from the transmit scheduler. For each request, the transmit engine will allocate an entry in its operation table and issue a request to the descriptor read module. Once the descriptor is read, the transmit engine will extract the packet data pointer and length as well as the transmit checksum command from the descriptor. The transmit engine will then issue a read request to the PCIe DMA engine to read the packet data into internal RAM. When the read completes, the transmit engine will issue a read request to the port DMA engine to read the packet data for transmission. The packet will then pass through the transmit checksum module, MAC, and PHY. Once the packet is sent, the transmit engine will receive the PTP timestamp from the MAC, build a completion record, and pass it to the completion write module.

Similar to the transmit engine, the receive engine is responsible for coordinating packet receive operations. Incoming packets pass through the PHY and MAC and arrive at the receive hashing module. This module extracts the IP addresses and TCP/UDP ports to compute a standard Toeplitz flow hash, which is used to generate the receive queue index in the receive request for the receive engine. For each request, the receive engine will allocate an entry in its operation table and issue a write request to the port DMA engine to write the packet data into internal RAM. The receive engine will also store the PTP timestamp from the MAC as well as the computed IP

checksum and flow hash. Once the packet data is written, the receive engine will issue a request to the descriptor read module. Once the descriptor is read, the receive engine will extract the packet data pointer and the packet length and issue a write request to the PCIe DMA engine to write the packet data out into the buffer in host memory. When the write completes, the receive engine will build a completion record and pass it to the completion write module.

The descriptor read and completion write modules are similar in operation to the transmit and receive engines. These modules handle descriptor/completion read/write requests from the transmit and receive engines, issue enqueue/dequeue requests to the queue managers to obtain the queue element addresses in host memory, and then issue requests to the PCIe DMA interface to transfer the data. The completion write module is also responsible for handling events from the transmit and receive completion queues by enqueuing them in the proper event queue and writing out the event record.

3.3.6 PCI Express

For optimal performance, the Corundum NIC uses a fully custom PCI express DMA engine. This is necessary due to the tight integration required between the PCIe DMA engine and the rest of the NIC components.

The PCI express interface consists of several pieces. The Xilinx UltraScale FPGA provides a PCI express hard IP core that handles the PCI express link layer and physical layer components, presenting a streaming transaction layer interface to the user logic. This interface is connected to a set of PCI express modules to bridge the PCI express transaction layer to the design. There are two main components involved in this: the PCIe AXI lite master, and the PCIe DMA interface module.

The PCIe AXI lite master module terminates memory read and write operations against a PCI express BAR to an AXI lite interface. The AXI lite master module provides access via memory-mapped IO (MMIO) to internal register space to the driver for configuration and

management, including access to the queue pointers in the queue management logic.

The PCIe DMA interface itself consists of two parts: the write side (device-to-host) and the read side (host-to-device). The PCIe DMA write engine receive commands from user logic that contain the transfer internal address, PCIe address, and length. The transfers are divided into individual PCIe write request operations based on PCIe TLP size and address alignment rules. For each TLP, read requests are issued on the internal segmented interface to fetch the write data. As the read data arrives, TLP headers are attached and the data is barrel-shifted into proper alignment. Since the segmented interface is twice the width of the AXI stream interface to the PCIe hard IP core, no extra cycles are required for data realignment. The write DMA engine tracks the write request TLPs through the PCIe hard IP core with transmit sequence numbers, only reporting the write operation as complete once it has been transmitted over the PCIe link. When all TLPs have been sent for a given operation, a completion status record is returned to user logic. The core also tracks the number of available transmit flow control credits to ensure that outgoing requests are not blocked by the PCIe hard IP core.

The PCIe read DMA interface is a bit more complex. It receives read requests from user logic that contain the internal address, PCIe address, and length. The transfers are divided into individual PCIe read request operations based on PCIe TLP size and address alignment rules. PCIe tags are then allocated, and read request TLPs are issued. The destination internal address for each issued read request is stored in an intermediate table, indexed by PCIe tag. As completion TLPs arrive, the internal address is fetched from the table, the data is barrel-shifted to the proper alignment, and write requests are issued on the internal segmented interface. Again, since the segmented interface is twice the width of the AXI stream interface to the PCIe hard IP core, no extra cycles are required for data realignment. After all writes for a given DMA transfer are completed, a completion status record is returned to user logic. The core also tracks transmit sequence numbers to ensure the PCIe hard IP core transmit buffer does not overflow, which would result in blocking outgoing requests.

3.3.7 Segmented Memory Interface

For high performance, Corundum internally uses a custom segmented memory interface. The interface is split into segments of maximum 128 bits, and the overall width is double that of the AXI stream interface from the PCIe hard IP core. For example, a design that uses PCIe gen 3 x16 with a 512-bit AXI stream interface from the PCIe hard core would use a 1024-bit segmented interface, split into 8 segments of 128 bits each. This interface provides an improved "impedance match" over using a single AXI interface, enabling higher PCIe link utilization by eliminating issues with backpressure and alignment. Namely, the interface guarantees that the DMA interface can perform a full-width, unaligned read or write on every clock cycle.

Each segment operates similar to AXI lite, except with three interfaces instead of five: one channel provides the write address and data, one channel provides the read address, and one channel provides the read data. Unlike AXI, bursts and reordering are not supported, simplifying the interface logic. Interconnect components (multiplexers) are responsible for preserving the ordering of operations, even when accessing multiple RAMs. The segments operate completely independently of each other with separate flow control connections and separate instances of interconnect ordering logic. Also, operations are routed based on a separate select signal and not by address decoding, eliminating the need to assign addresses and enabling the use of parametrizable interconnect components that appropriately route operations with minimal configuration.

Byte addresses are mapped onto segmented interface addresses with the lowest-order address bits determining the byte lane in a segment, the next bits selecting the segment, and the highest-order bits determining the word address for that segment. For example, in a 1024-bit segmented interface, split into 8 segments of 128 bits, the lowest 4 address bits would determine the byte lane in a segment, the next 3 bits would determine the segment, and the rest would drive the address bus for that segment.

3.3.8 Ethernet Interfaces

For the 10G and 25G design variants, Corundum uses an open source 10G/25G Ethernet MAC and PHY that supports PTP timestamping. The open source MACs are more compatible with the Python simulation framework. Additionally, using open-source MACs and PHYs also permits optimization for operation in a circuit-switched environment, including the removal of features that can cause unnecessary disruptions to the transmit side when the receive side is down and changes to the PHY layer components to improve frame synchronization performance.

The 100G design variants currently rely on the Xilinx UltraScale+ hard 100G CMAC cores with support for RS-FEC [62].

3.3.9 Checksum Offloading

Checksum offloading moves the task of computing a ones complement checksum over the packet payload data from the CPU to the NIC, resulting less CPU load per packet sent or received over the network. Corundum currently implements basic transmit and receive IP checksum offloading.

Receive IP checksum offloading hardware on the NIC computes a ones complement sum over the entire Ethernet payload of every received frame, which is included in the completion record. The networking stack can then offset this checksum by subtracting out the checksum of any header fields that are not part of the IP layer checksum, which requires fewer operations than summing the complete payload, resulting in a reduction in per-packet CPU overhead. This checksum offload technique has the additional advantage of being protocol agnostic; it can be applied to any protocol that uses ones-complement checksums without changes to the hardware.

Transmit IP checksum offloading hardware on the NIC computes a ones complement sum over a portion of the packet, inserting the result into the packet at the specified location. The network stack is responsible for computing the pseudo-header checksum over the appropriate

fields, inserting the partial checksum in the packet header, and then informing the NIC of the location of both the checksum field and the offset to the start of the payload data.

3.3.10 Device Driver

The Corundum NIC is connected to the Linux kernel networking stack with a kernel module. The module is responsible for initializing the NIC, registering kernel interfaces, allocating DMA-accessible buffers for descriptor and completion queues, handling device interrupts, and passing network traffic between the kernel and the NIC. The kernel module also provides an interface for userspace software to access NIC registers, permitting configuration of NIC components including port schedulers, visibility into internal NIC state for debugging and characterization, and access to other hardware components including optical module configuration interfaces for monitoring and FPGA configuration flash for firmware updates.

The NIC uses register space to expose parameters including the number of interfaces, number of ports, number of queues, number of schedulers, maximum transport unit (MTU) size, and presence of PTP timestamping and offload support. The driver reads these registers during initialization so it can configure itself and register kernel interfaces to match the NIC design configuration. This auto-detection capability means that the driver and NIC are loosely coupled; the driver generally does not need to be modified with respect to the core datapath when used across different FPGA boards, different Corundum design variants, and different parameter settings.

3.3.11 Simulation Framework

An extensive open-source, Python-based simulation framework is included to evaluate the complete design. The framework is built using the Python library MyHDL and includes simulation models of the PCI express system infrastructure, PCI express hard IP core, NIC driver,

and Ethernet interfaces. The simulation framework facilitates debugging the complete NIC design by providing visibility into the state of the entire system.

The core of the PCIe simulation framework consists of about 4,500 lines of Python and includes transaction-layer models of PCIe infrastructure components including root complex, functions, endpoints, and switches as well as high-level functionality including configuration space, capabilities, bus enumeration, root complex memory allocation, interrupts, and other functions. The framework supports both memory-mapped and IO space operations between any combination of devices, including host-to-device memory-mapped IO (MMIO), device-to-host direct memory access (DMA), and device-to-device peer-to-peer DMA. Additional modules, consisting of another 4,000 lines of Python, provide models of the FPGA PCIe hard IP cores, exchanging transaction-layer traffic with the simulated PCIe infrastructure and driving signals that can be connected to a cosimulated Verilog design.

Simulating Corundum requires a few lines of code to instantiate and connect all of the components. Listing 3.1 shows an abbreviated testbench to send and receive packets of various sizes using the simulation framework, with the Verilog design cosimulated in Icarus Verilog. The testbench instantiates simulation models for the Ethernet interface endpoints, PCIe root complex, and driver, and connects these to the cosimulated design. Then, it initializes the PCIe infrastructure, initializes the driver model, and sends, receives, and verifies several test packets of various lengths.

3.4 Results

The 100G variant of the Corundum NIC was evaluated on an Alpha Data ADM-PCIE-9V3 board, installed in a Dell R540 server (dual Xeon 6138), connected to a Mellanox ConnectX-5 NIC in an identical server with a QSFP28 direct attach copper cable. Two more Mellanox ConnectX-5 NICs installed in the same machines were also evaluated for comparison. Eight

```
from myhdl import *
import pcie, pcie_us, pcie_usp, axis_ep
# signals
clk_250mhz = Signal(bool(0))
# etc.
# sources and sinks
qsfp_0_source = axis_ep.AXIStreamSource()
qsfp_0_source_logic = qsfp_0_source.create_logic(
    qsfp_0_rx_clk,
    # etc.
)
# etc.
# set up PCIe infrastructure
rc = pcie.RootComplex()
# create driver instance
driver = mqnic.Driver(rc)
# create PCIe hard IP core instance
dev = pcie_usp.UltrascalePlusPCIe()
rc.make_port().connect(dev)
pcie_logic = dev.create_logic(
    m_axis_cq_tdata=s_axis_cq_tdata,
    # etc.
)
# connect to Verilog design
dut = Cosimulation(
    "vvp -m myhdl testbench.vvp -lxt2",
    clk_250mhz=user_clk,
    # etc.
)

@instance
def check():
    # initialization
    yield rc.enumerate()
    yield from driver.init_dev(dev.functions[0].get_id())
    yield from driver.interfaces[0].open()
    # test packets of various lengths
    for k in range(64, 1515):
        data = bytearray([x%256 for x in range(k)])
        # send and receive a packet
        yield from driver.interfaces[0].start_xmit(data, 0)
        yield qsfp_0_sink.wait()
        pkt = qsfp_0_sink.recv()
        assert pkt.data == data
        qsfp_0_source.send(pkt)
        yield driver.interfaces[0].wait()
        pkt = driver.interfaces[0].recv()
        assert pkt.data == data
```

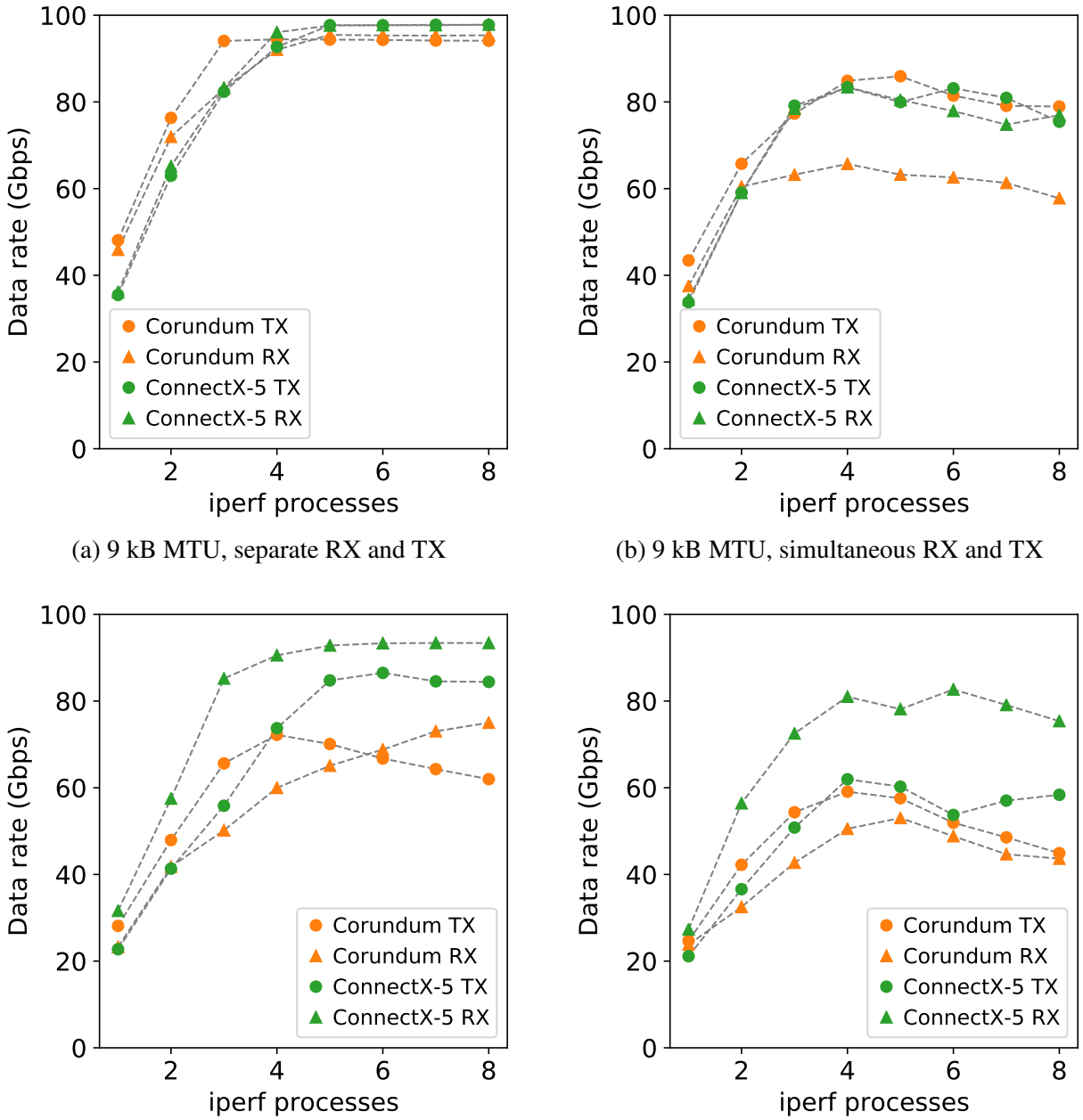
Listing 3.1: Abbreviated NIC testbench. Includes setting up PCIe, Ethernet, and driver models, initialization, and sending and receiving test packets. Most signals removed for brevity.

instances of iperf3 were used to saturate the link, and both NICs were configured with an MTU of 9000 bytes.

For the current implementation, the NIC is able to separately achieve 95.5 Gbps RX and 94.4 Gbps TX (Fig. 3.9a). Under the same conditions, the Mellanox ConnectX-5 NIC runs at 97.8 Gbps for both RX and TX. When running additional instances of iperf to fill the link in both directions simultaneously, the performance degrades to 65.7 Gpbs RX and 85.9 Gbps TX (Fig. 3.9b). For the existing testbed, the performance of the Mellanox NIC also degraded to 83.4 Gbps for both RX and TX. The degradation of both Corundum and the ConnectX-5 in full-duplex mode suggests that the software driver may be a significant contributor to the reduction in performance. Specifically, the current version of driver only supports the Linux kernel networking stack. A driver that supports a kernel-bypass framework such as DPDK should improve the performance for full-duplex mode and is an objective of future work.

Figures 3.9c and 3.9d compare the performance for an MTU of 1500 bytes. For this case, Corundum can separately achieve 75.0 Gbps RX and 72.2 Gbps TX (Fig. 3.9c) and simultaneously achieve 53.0 Gbps RX and 57.6 Gbps TX (Fig. 3.9d). The performance difference for Corundum between TX and RX seen in Fig. 3.9c as the number of iperf instances increases is caused by a limitation on the number of in-process transmit packets coupled with PCIe round-trip delay. Increasing the number of in-process transmit operations supported in hardware should increase the throughput and is planned future work. For comparison, under the same conditions, the Mellanox ConnectX-5 NIC can separately achieve 93.4 Gbps for both RX and TX and simultaneously achieve 70.6 Gbps RX and 72.1 Gbps TX.

To test the performance of PTP timestamping, two Corundum NICs in 10G mode were connected to an Arista 40G packet switch operating as a PTP boundary clock. The NICs were configured to output a fixed frequency signal derived from PTP time, which was captured by an oscilloscope. When Corundum is implemented with PTP timestamping enabled, the hardware clocks can be synchronized with linuxptp to better than 50 ns. The time synchronization



(a) 9 kB MTU, separate RX and TX

(b) 9 kB MTU, simultaneous RX and TX

(c) 1.5 kB MTU, separate RX and TX

(d) 1.5 kB MTU, simultaneous RX and TX

Figure 3.9: NIC TCP throughput measurements

performance is unchanged when the link is saturated.

Table 3.1: Resource utilization

Board	FPGA	PCIe	IF	TXQ	MTU	MAC	Speed	LUT	FF	BRAM	URAM
ADM-PCIE-9V3	XCVU3P	3 x16	2x1	8K	2 K	10 G	10 G	69.5 K (18%)	73.1 K (9%)	252 (33%)	20 (6%)
ADM-PCIE-9V3	XCVU3P	3 x16	2x1	8K	2 K	25 G	25 G	66.9 K (17%)	73.1 K (9%)	239 (33%)	20 (6%)
ADM-PCIE-9V3	XCVU3P	3 x16	2x1	8K	16 K	100 G	94.4 G	61.8 K (16%)	75.6 K (10%)	331 (33%)	20 (6%)
ExaNIC X10	XCKU035	3 x8	2x1	1K	2 K	10 G	10 G	41.0 K (20%)	47.1 K (12%)	149 (28%)	-
ExaNIC X25	XCKU3P	3 x8	2x1	8K	2 K	10 G	10 G	43.9 K (27%)	51.8 K (16%)	132 (37%)	20 (42%)
ExaNIC X25	XCKU3P	3 x8	2x1	8K	2 K	25 G	25 G	41.5 K (26%)	52.5 K (16%)	127 (35%)	20 (42%)
NetFPGA SUME	XC7V690T	3 x8	2x1	512	2 K	10 G	10 G	43.0 K (10%)	50.8 K (6%)	133 (9%)	-
VCU108	XCVU095	3 x8	1x1	2K	2 K	10 G	10 G	28.4 K (5%)	26.7 K (2%)	107 (6%)	-
VCU118	XCVU9P	3 x16	2x1	8K	2 K	10 G	10 G	70.1 K (6%)	73.9 K (3%)	252 (12%)	20 (2%)
VCU118	XCVU9P	3 x16	2x1	8K	16 K	100 G	94.4 G	62.5 K (5%)	78.3 K (3%)	331 (15%)	20 (2%)
VCU1525	XCVU9P	3 x16	2x1	8K	2 K	10 G	10 G	69.6 K (6%)	73.1 K (3%)	252 (12%)	20 (2%)
VCU1525	XCVU9P	3 x16	2x1	8K	16 K	100 G	94.4 G	62.4 K (5%)	77.6 K (3%)	331 (15%)	20 (2%)

Resource utilization of several variants of the Corundum design on several FPGA boards is shown in Table 3.1. The footprint of Corundum is rather small, leaving ample space available for additional logic, even on relatively small FPGAs. For example, the Corundum design for the ExaNIC X10, a dual port 10G design with a PCIe gen 3 x8 interface and 512 bit internal datapath, consumes less than a quarter of the logic resources available on the second smallest Kintex UltraScale FPGA (KU035).

3.5 Case-Study: Time-Division Multiple Access (TDMA)

Precise network admission control is an vital networking functionality at high line rates. Corundum provides thousands of transmit queues that can be used to separate and control transmit data on a fine time scale synchronized across multiple end hosts. This functionality provides a unique toolbox that can be used develop new and powerful NIC functions. Determining what network functions to implement and the impact these functions have on network performance is an active research area [45] [48] [24] [52].

To demonstrate how Corundum can be used for precision transmission control, we implemented a simple form of TDMA with a fixed schedule. The schedule can be synchronized across multiple hosts via IEEE 1588 PTP. Basic TDMA support in Corundum is designed to

be minimally intrusive on the overall architecture. TDMA is implemented by enabling and disabling queues in the transmit scheduler according to PTP time, under the control of the `tx_scheduler_ctrl_tdma` module. Queue enable and disable commands are generated in the TDMA scheduler control module and sent to the transmit scheduler at the beginning and end of each timeslot. The TDMA scheduler operates under the assumption that the timeslots are sufficiently long so that the TDMA scheduler control module can prepare for the next timeslot during the current timeslot. In addition, a relatively small number of queues must be active during each timeslot so the skew between the first and last queue enabled or disabled is small.

Timing signals for the schedule are generated from PTP time by the `tdma_scheduler` module. This module provides several signals: single cycle pulses at the start of the schedule and the start and end of each timeslot, along with the index of the current timeslot. The timing signals for the TDMA schedule are defined by the schedule start time, schedule period, timeslot period, and timeslot active period. The TDMA scheduler module computes the start and stop times for each timeslot in each iteration of the schedule and generates the appropriate timing signals via threshold comparison with the current PTP time provided by the PTP hardware clock (PHC) on the NIC.

3.5.1 TDMA Scheduler Control Module

The TDMA scheduler control module is responsible for generating the queue enable and disable commands, based on the timing information from the TDMA scheduler module.

A block diagram of the TDMA scheduler control module is shown in Fig. 3.10. The TDMA scheduler control module consists of per-queue, per-timeslot enable bits and three FIFOs. Two FIFOs, `start_queue_ts_0` and `start_queue_ts_n`, store the active queue indicies for timeslot index 0 and index *N*, the third, `stop_queue` stores the active queue indicies for the current timeslot. The enable bits are exposed to the driver via a register interface over AXI lite. During each timeslot, the TDMA scheduler control module resets `start_queue_ts_0` and

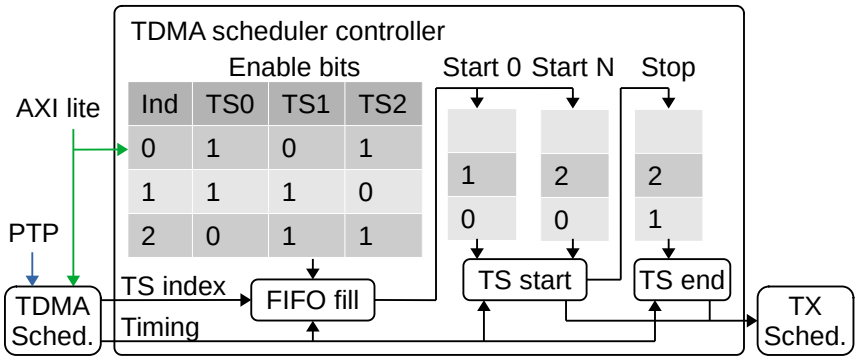


Figure 3.10: Block diagram of the TDMA scheduler control module.

`start_queue_ts_n` and then iterates over all of the enable bits to populate the FIFOs with the appropriate queue enable commands for timeslot 0 and the presumed next timeslot, given by incrementing the current timeslot index. At the start of the next timeslot, the module will check the timeslot index and generate queue enable commands from the appropriate FIFO. While generating queue enable commands, the module also fills `stop_queue` with the same queue indicies. After generating all of the enable commands, the module will prepare for the next timeslot. At the end of the timeslot, the module generates queue disable commands from `stop_queue`. The timeslot must be long enough for these operations to complete.

The TDMA scheduler control module runs in the 250 MHz PCIe user clock domain. As a result, it takes 4 ns per queue to iterate over each transmit queue to prepare for the next timeslot (about 32.8 us total for 8,192 transmit queues). Similarly, it takes 4 ns to generate each enable or disable command to send to the transmit scheduler module.

3.5.2 TDMA Performance

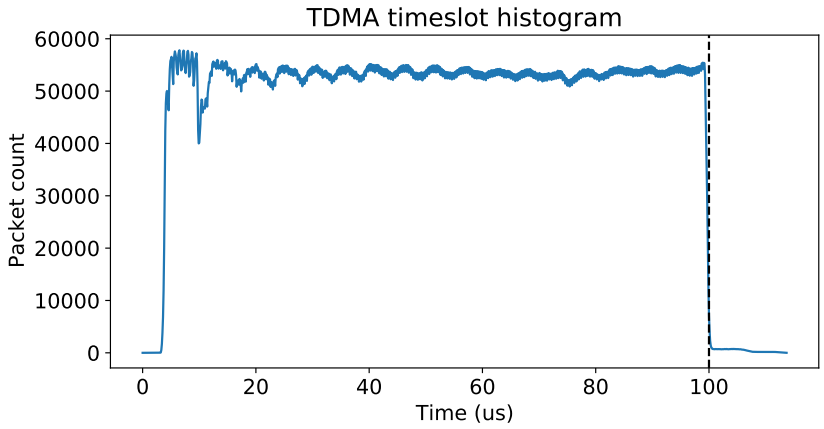
The 100G TDMA variant of the Corundum NIC with 256 transmit queues was evaluated on an Alpha Data ADM-PCIE-9V3 board, installed in a Dell R540 server (dual Xeon 6138), connected to a Mellanox ConnectX-5 NIC. Eight instances of iperf3 were used to saturate the link, and both NICs were configured with an MTU of 9 kB. With TDMA disabled, the NIC runs

at 94.0 Gbps. The TDMA scheduler was configured run a schedule with period 200 µs containing two timeslots of 100 µs, enabling all transmit queues in the first timeslot and disabling them in the second. A timeslot active period of 90 µs was used to account for an 8 µs interval for the 11 packets in the transmit datapath (11 $\times$ 0.72 µs per packet) at 100 Gbps plus 1 µs to disable all 256 queues. Under those conditions, Corundum could control the data leaving the NIC with a precision of two packet lengths or 1.4 µs. A histogram of the achieved transmit timing is shown in Fig. 3.11a, with a detail view of the start and end of the timeslot shown in Fig. 3.11b and Fig. 3.11c. There is an approximately 13 µs tail that consists of around 0.1% of the transmitted packets; it should be possible to minimize this tail with additional work on the transmit datapath components.

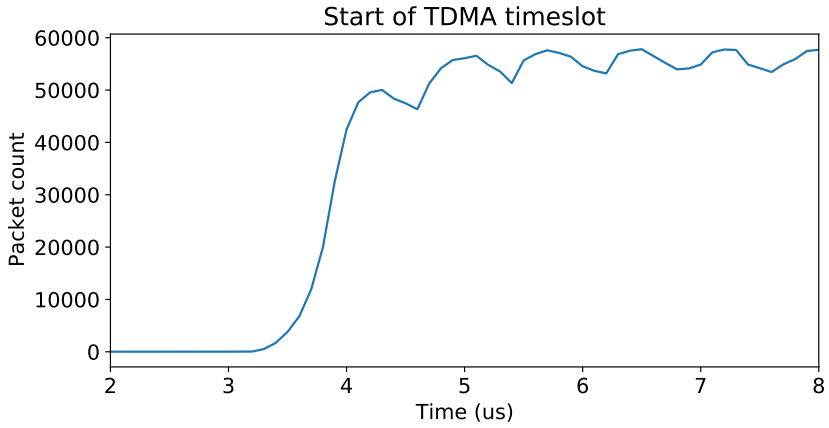
An additional test was run at 10 Gbps line rate with an MTU of 1500 bytes using a schedule with a period of 200 µs. This period was partitioned into two timeslots of 100 µs. A timeslot active period of 60 µs was used to account for a 38 µs interval for the 32 packets in the transmit datapath (32 $\times$ 1.2 µs per packet) at 10 Gbps plus 1 µs to disable all 256 queues. Under those conditions, Corundum could control the data leaving the NIC with a precision of two packet lengths or 2.4 µs. A histogram of the achieved transmit timing is shown in Fig. 3.12a, with a detail view of the start and end of the timeslot shown in Fig. 3.12b and Fig. 3.12c.

3.6 Case-Study: In Situ Physical Layer Characterization

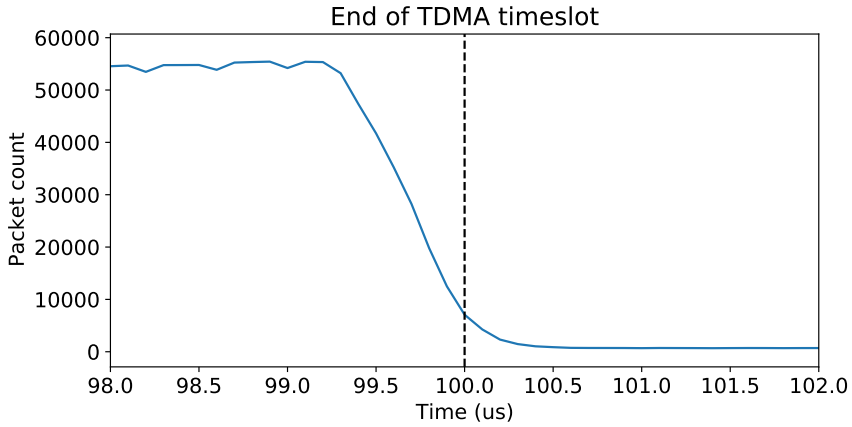
The NIC is an ideal vantage point for performing physical layer measurements of links in a datacenter environment. This is particularly useful when optical switching is involved as it can provide detailed insight into the operation of all of the links through the switch in all possible switch configurations, without requiring physical disruption of the network. Corundum provides an ideal platform for this type of measurement as it provides direct access to the serializers as well as a high-resolution time reference that can be synchronized across all of the NICs in the



(a) Histogram of full time slot

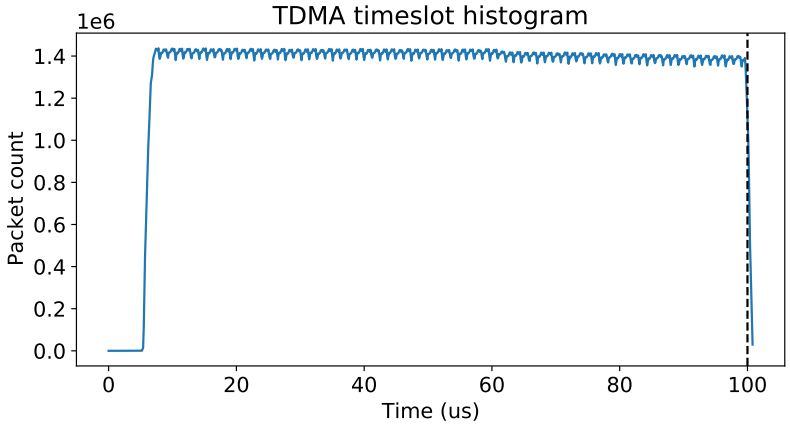


(b) Detail at start of timeslot

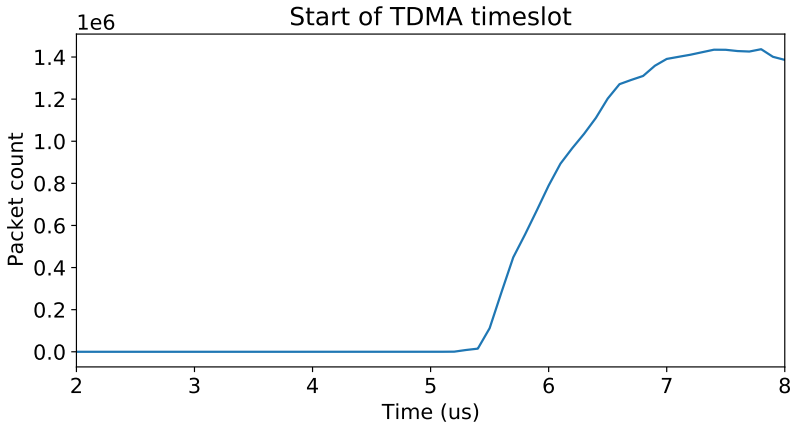


(c) Detail at end of timeslot

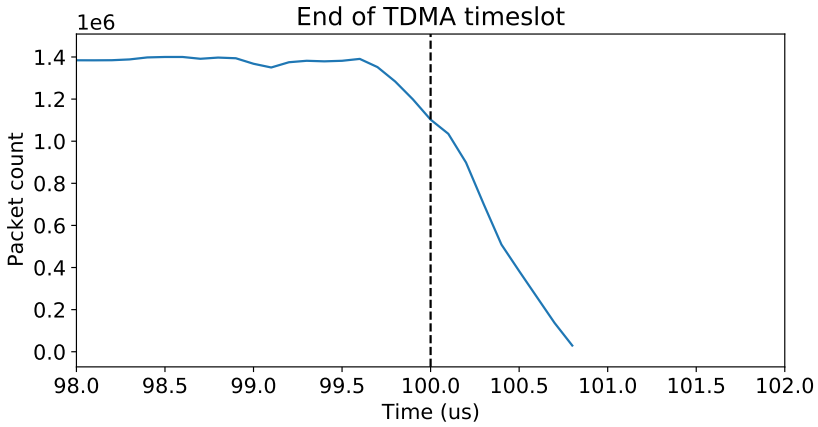
Figure 3.11: TDMA timing histogram with 9 KB packets at 100 Gbps



(a) Histogram of full time slot



(b) Detail at start of timeslot



(c) Detail at end of timeslot

Figure 3.12: TDMA timing histogram with 1.5 KB packets at 10 Gbps

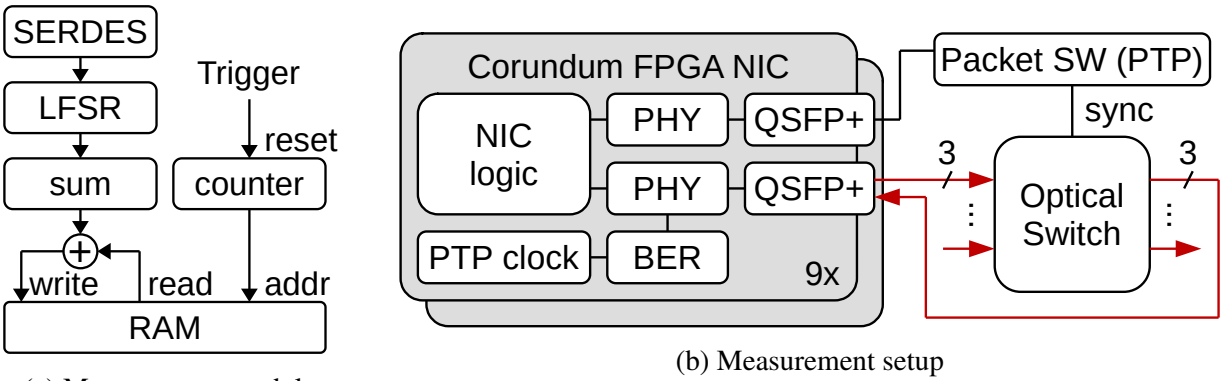
network.

In an optically-switched network, the received optical power is impacted by variability in both the transmitter launch power and the attenuation through different paths in the optical switch fabric. In general, the power level at the receiver is a function of the network configuration, which can change rapidly over time. This can produce a change in the locking characteristics of the receiver and a corresponding time-varying bit error ratio (BER) that can adversely affect network performance. Accurately characterizing the receiver locking characteristics and the time-varying BER for switched optical links as function of the network configuration is a critical but challenging measurement problem.

Bit errors can be resolved in time by binning them in temporal bins. This technique can be applied to any optical network by setting the switches to reconfigure according to a sequential, repetitive schedule during the measurement phase. It is particularly well-suited to characterizing optical networks that are designed to reconfigure according to a fixed schedule [34]. Bit errors can be temporally resolved using a method analogous to a sampling oscilloscope, with errors accumulated in temporal bins based on the relative time with respect to an accurate trigger event. This is an extension of BER measurements of switched links using gated error detectors [18].

The BER measurement design uses a PRBS generator and error detector, either standalone, built into the FPGA serializers, or built into Ethernet PHY modules, coupled with a BER measurement module that accumulates bit errors. A block diagram of the BER measurement components is shown in Fig. 3.13a.

The PRBS generator module generates a standard PRBS of length of $2^{31} - 1$. The serdes on the FPGA deserializes the hard-detected output from the optical transceiver and feeds the values into a linear-feedback shift register (LFSR)-based PRBS checker. This module implements a feed-forward version of the LFSR used to generate the PRBS. The output of the LFSR is one logic-high value per bit error, per LFSR tap. The output of the LFSR is summed in small batches, then passed to the BER measurement module where it is accumulated into a small RAM. A set of



(a) Measurement module

(b) Measurement setup

Figure 3.13: Link-level measurement block diagrams

counters generate the bin index, which determines the RAM address. The trigger signal resets the counters such that following the trigger, any bit errors that occur are accumulated in sequential bins. The bin counts are read out of the RAM by control software.

The temporal resolution of the accumulated errors in each bin is limited by the accuracy of the trigger signal. Practically, there is a trade-off between bin width and overall measurement time, although it is possible to trade time resolution for BER accuracy by summing adjacent bins during post-processing.

Using this capability, we characterized the time-resolved BER of an optical network. For this case, highly-concurrent synchronized BER measurements can be obtained across multiple network nodes using multiple synchronized instances of the basic measurement shown in Fig. 3.13a. A block diagram is shown in Fig. 3.13b. For this measurement, we integrated the BER measurement functionality into Corundum. This provides the capability for in situ network BER measurements. We achieve precise synchronization of the multiple FPGA-based NICs and the optical switch in the network using the IEEE 1588 Precision Time Protocol (PTP). Our implementation of this protocol yields sub-100 ns precision, which is sufficient for BER measurements with a temporal bin size larger than 1 μs.

The number of instances of the time-resolved BER measurement implemented on each NIC depends on the network configuration and number of uplinks per node. As an example, a

nine-server testbed with three 10 Gb/s uplinks per server was connected to an optical switch that supported 54 periodic network configurations (3 configurations repeated 18 times) for each uplink [34]. The BER measurement module on the NIC can concurrently acquire 32 time bins for each of the 54 configurations. The switch reconfigures every 222 μs, so capturing 128 bins per configuration in four offset measurements results in a resolution of 1.7 μs. For this measurement, each NIC collected $3 \times 54 \times 32 = 5184$ concurrent measurements. Across all of the hosts, a total of $5184 \times 9 = 46,656$ concurrent measurements can be collected. Fig. 3.15 shows heat maps collected from all 27 receivers, while Fig. 3.14 shows the heat maps for one of three Rx channels on four hosts with the y axis denoting the network configuration. There is a large variation in the time-resolved receiver locking characteristics caused by the combination of the variable transmit power and path-dependent loss through the optical network. This kind of automated diagnostic analysis is essential for identifying and correcting link-level problems in multi-node optically switched networks.

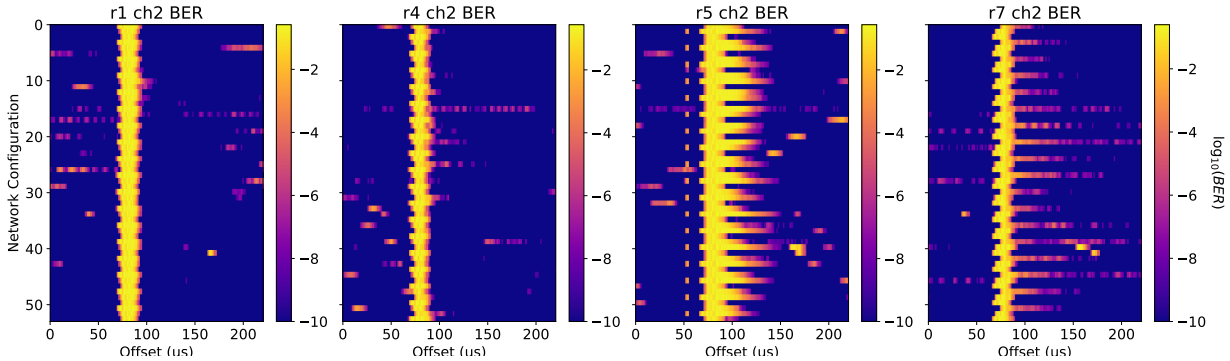


Figure 3.14: Selection of heat maps from four receivers

3.7 Conclusion

In this chapter, we presented Corundum, an open-source, high-performance, FPGA-based NIC, capable of operation at a line rate of at least 94.4 Gbps. This measured performance is sufficient to develop and test new networking applications at realistic line rates. Because

Corundum is open source, it provides a powerful prototyping platform for network research and development, including NIC-based schedulers such as SENIC [45], Carousel [48], PIEO [52], and Loom [56], new protocols and congestion control techniques such as NDP [24] and HPCC [29], and new network architectures, such as P-FatTree [38], RotorNet [37], and Opera [35]. Optimizing the design to improve performance for smaller packet sizes as well as demonstrate new networking protocols based on precise packet transmission are objectives of ongoing work.

3.8 Sources for Material Presented in This Chapter

Chapter 3, in part, reprints material that has been submitted for publication in a paper titled: "Corundum: An Open-Source 100-Gbps NIC," submitted to the FCCM conference, by Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. The dissertation author was the primary researcher and author of this material. The dissertation author was the primary researcher and author of this material.

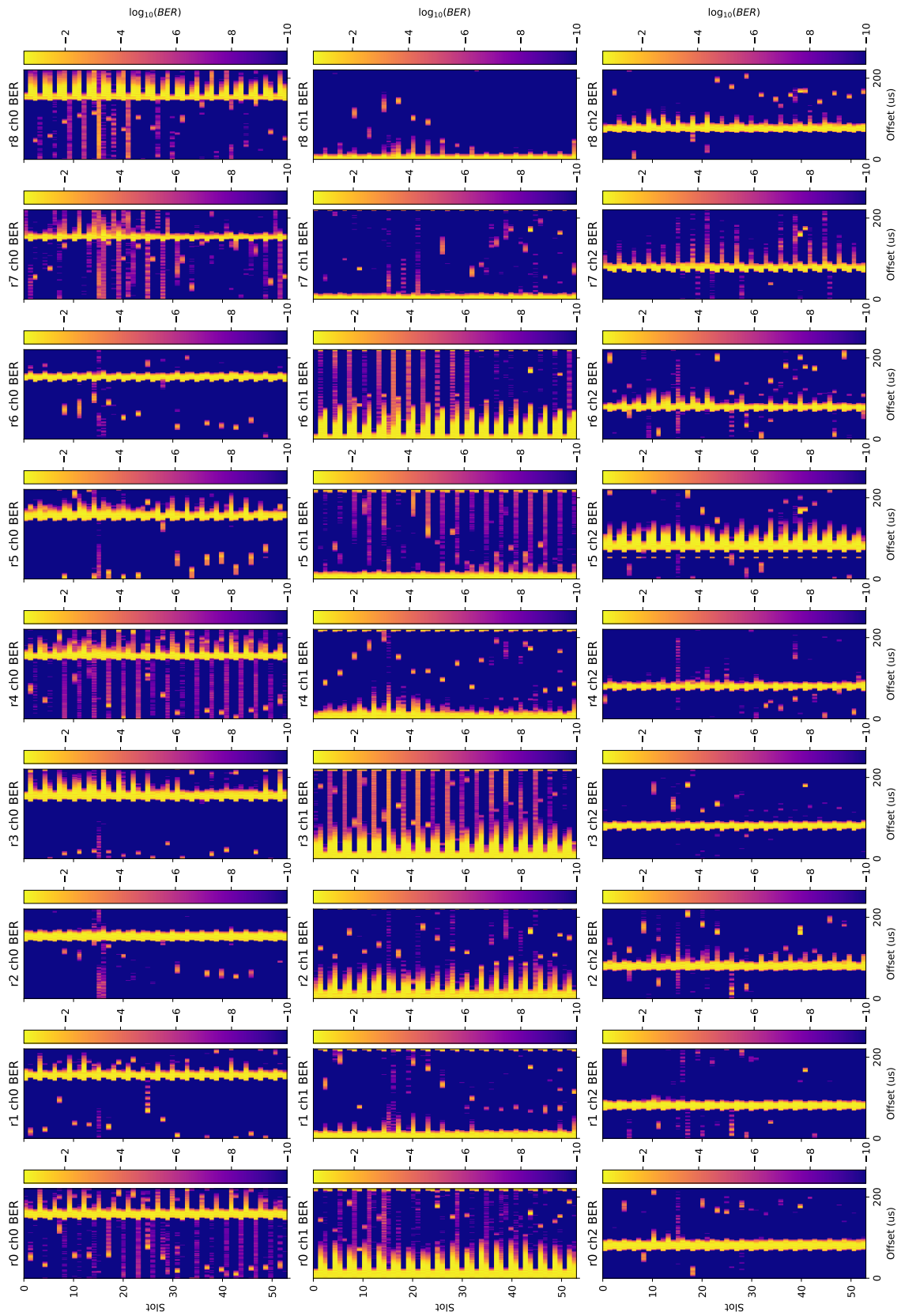


Figure 3.15: BER heat maps from all 27 receivers in all 9 hosts

Chapter 4

Future Research Directions

The work presented in this dissertation validates the hypothesis that developing a NIC capable of precision admission control and characterizing its performance can lead to practical sub-microsecond circuit-switched networks at scale.

Two essential contributions were presented. The first contribution was quantifying the system-level reconfiguration time of an optically circuit-switched link. This work was presented in Chapter 2 and included a description of novel link-level measurements on an optically switched link, including bit error rate (BER) characterization of a 25 Gbps link utilizing a burst-mode receiver, switched by a nanosecond-scale silicon photonic switch. Several key parameters that affect system-level performance were quantified including reconfiguration time, receiver lock time, and guard intervals.

The second contribution of this dissertation is the development of a network interface controller (NIC) that can precisely control the injection of packets into one or more optically-switched networks. This work was presented in Chapter 3, and describes and quantifies the performance of an FPGA-based NIC called Corundum. This NIC was designed to precisely control the injection of packets from multiple queues into a circuit-switched network under the control of a hardware-based scheduler in an otherwise standard datacenter environment. The

platform provides the flexibility to implement high-precision time synchronization as well as perform in situ link-level characterization including BER measurements similar to those described in Chapter 2.

The ability to quantify the low-level physical layer characteristics of an optically-switched link and provide precision transmission control over thousands of queues all within a common hardware platform are necessary fundamental functionalities for the development of practical sub-microsecond circuit-switched networks at scale.

In a larger context, while this dissertation establishes a necessary framework for circuit-switched networks by the development of precision hardware admission control, this work, by itself, is not sufficient to construct a practical sub-microsecond circuit-switched network. Here, in this concluding chapter, we briefly discuss future work that can provide the sufficient conditions for practical sub-microsecond circuit-switched networks.

The sufficient conditions are based on the type of circuit-switched network. Other work within our group [38] has demonstrated the advantages of parallel networks irrespective of whether they are packet-switched or circuit switched. Accordingly, future work based on this dissertation is focused on determining the sufficient conditions for the development of a parallel circuit-switched network, which may not necessarily be optical.

In this context, Corundum provides unique architectural features to support parallel networks. Specifically, all ports that are part of the same interface on the Corundum NIC share the same set of transmit queues and appear as a single, unified interface to the operating system. This enables hardware schedulers on the NIC to determine which traffic is sent out of each uplink port. Using this functionality, flows can be migrated between ports by changing only the transmit scheduler settings without affecting the rest of the network stack. This dynamic, scheduler-defined mapping of queues to ports was implemented in Corundum to enable research into new parallel network protocols and network architectures. Specifically, based on the work presented in this dissertation, future work includes using Corundum to demonstrate the viability

of the Opera networking protocol [35]. This demonstration will go a long way to establishing the sufficient conditions for a practical parallel circuit-switched network. In this larger context, the development of event-driven scheduler modules that can precisely pace packet injection may also have applications within conventional packet-based networks. Additional future work includes the investigation of network interface architectures that can scale to millions of queues to provide per-destination control at datacenter scale.

Using Corundum to demonstrate Opera is only part of the story. At the physical layer, there are many competing optical circuit-switched technologies at the sub-microsecond scale and all of these technologies require precision admission control. In this context, future work includes investigating techniques to extend these admission control features to operate on shorter timescales and support faster optical switches.

A complementary future direction is to investigate new physical-layer protocol features that are specifically designed for operation with fast optical switches. Current network line protocols are not designed for fast switching and as a result can require hundreds of microseconds or more to recover from an interruption. The development of these methods, which explicitly incorporate the burst-mode nature of an optically circuit-switched link, can increase the link bandwidth and reduce the system-level reconfiguration time. Additionally, these methods can open the door for in-band synchronization that utilizes the optical switch itself as a means for synchronizing network components.

The goal of all of this future research is to determine an appropriate and practical partitioning between hardware and software for parallel network interfaces for circuit-switched networks. The existing hardware/software partitioning of traditional, ASIC-based NICs has been a significant impediment to the development of an efficient interface with high-speed optical switches. This interface is essential to construct an efficient, scalable, optically-switched network.

Realizing a network interface specifically designed for optical switching applications in reconfigurable hardware will facilitate research across the entire stack towards practical, high-

performance optical switching at scale. The practical functionalities developed through this research can then be integrated into future networking components including software stacks, NICs, and switches and used to improve the efficiency of the next generation of datacenter networks.

Appendix A

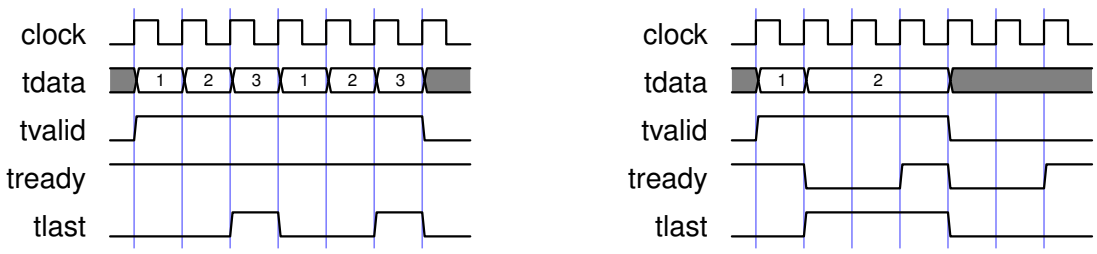
ARM AMBA AXI

The ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) is an interface protocol specification [2] for high-performance, on-chip interfaces. This appendix contains some high-level notes about AXI, for complete details refer to the specification.

There are three main flavors of AXI: AXI, AXI lite, and AXI stream. AXI stream supports streaming data transfers, optionally framed, with optional sideband signals for transaction identification and routing. AXI and AXI lite are memory-mapped interfaces that consist of five AXI stream-like interfaces in parallel, one each for read address, read data, write address, write data, and write response. AXI is a burst-oriented protocol that can efficiently transfer large blocks of data. AXI lite is a simplified version of AXI that only supports single word memory operations.

A.1 AXI Stream

The five channels of AXI all use the same handshaking scheme as AXI stream—a `valid` signal sent along with the data and sideband signals, and a `ready` signal flowing in the opposite direction. The *source* or *master* side generates the data, sideband signals, and the `valid` signal, and the *sink* or *slave* device receives the data, sideband, and `valid` signal and generates the `ready`



(a) Back-to-back transfers

(b) Pause after each transfer

Figure A.1: AXI stream handshake timing examples

signal. Data is transferred from the source to the sink when both the `ready` and `valid` signals are high. Therefore, the source can present anything on the data and sideband outputs while `valid` is low, but once it has placed valid data on its outputs and asserted `valid`, it must hold the outputs in that state with valid high until `ready` is asserted from the sink. If `ready` is high on the same cycle that `valid` is asserted, then the source can either deassert `valid` or present fresh data on the next clock edge. The sink device generates backpressure by deasserting the `ready` signal when it is not ready to receive data. It is permitted for the sink to use the the data and sideband signals while valid is high and ready is low, notifying the source by asserting `ready` when the slave is finished with that data.

A.2 Notes on Timing Closure

Inserting registers into an AXI stream interface to break timing paths can be done in one of several ways. If the `ready` path does not need to be broken, then the ready signal can be passed through combinatorially and the rest of the signals can be registered, loading new values when the registers are empty. This results in full throughput and no stall cycles with 1 register per signal.

If the `ready` path must be broken, then there are two options for inserting registers. One method is to use one set of registers for all signals and alternate between storing a new value and shifting out the stored value. This results in half throughput by inserting 1 stall cycle for every transfer, but uses 1 register per signal. Another method is to use a skid buffer.

A skid buffer has an extra set of registers in it to handle the case where the sink deasserts its ready signal in the middle of a back-to-back transfer from the source. Since the ready signal is generated by a flip flop, the ready signal to the source will be asserted for one extra cycle and so the corresponding data must be stored in the register slice until it can be transferred through to the sink. This method results in full throughput with no stall cycles, but requires two registers and one 2 to 1 mux per signal.

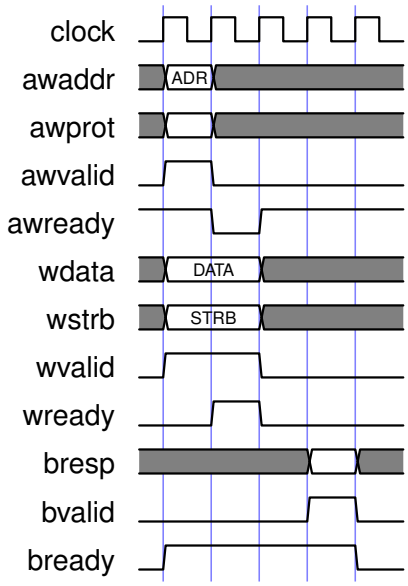
It is also possible to break timing with a FIFO, but generally FIFOs will only be inserted where they are specifically needed as they use more logic resources than a flip-flop based register slice.

One interesting component to be aware of on Xilinx FPGAs is the SRL primitive. These primitives can have very favorable input delays on some parts. As a result, building registers or even shallow FIFOs by inferring SRLs instead of flip-flops can provide timing advantages.

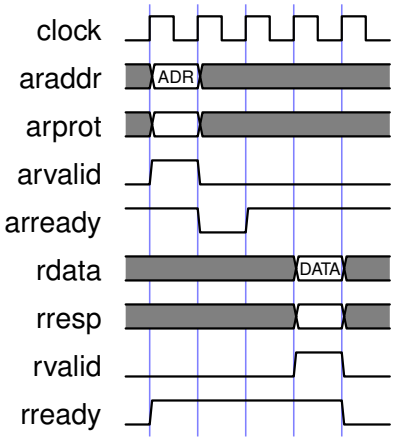
A.3 AXI and AXI Lite

AXI and AXI lite are memory-mapped interfaces that consist of five streaming interfaces—write address (AW), write data (W), write response (B), read address (AR), and read data (R). AXI is a burst-oriented protocol that can efficiently transfer large blocks of data. AXI lite is a simplified subset of AXI that is missing most of the sideband signals and only supports single word reads and writes and is not tolerant of any reordering.

AXI supports multiple burst types to enable efficient transport of large blocks of data. The supported burst types are fixed, increment, and wrap. Fixed bursts are used to repeatedly access a single address for filling or emptying a FIFO. Increment bursts start at a specified address and then increment to higher addresses. Wrap bursts are used for unaligned cache line transfers where an operation can start in the middle of a cache line and then wrap around to the beginning of the cache line.



(a) AXI lite write



(b) AXI lite read

Figure A.2: AXI lite transfers

AXI also supports locked operations (`a*lock`), secure operations (`a*prot`), cache control (`a*cache`), and quality of service (`a*qos`).

Figure A.2 depicts AXI lite read and write operations. See the full specification [2] for a complete description of the protocol.

A.4 Notes on Byte Packing and Rearranging

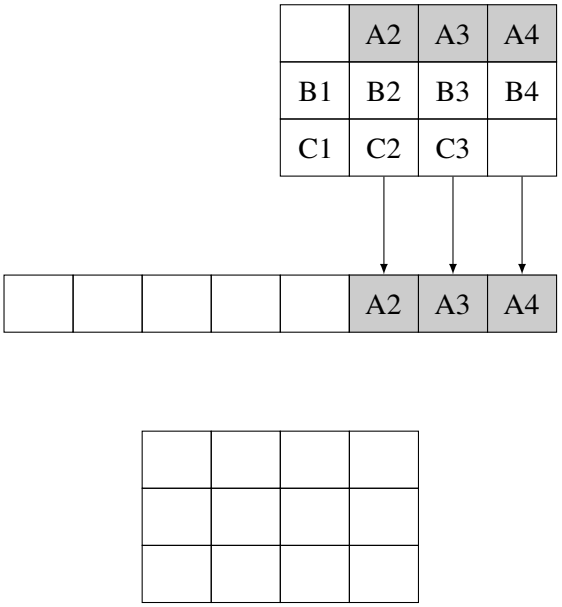
Aligned transfers across parallel interfaces are relatively simple to implement. If all operations are byte-aligned and the interfaces involved are a single byte wide, inserting, removing, truncating, copying, etc. are relatively trivial to implement. However, unaligned transfers across wide parallel interfaces can be significantly more complex to implement correctly. This is especially obvious when building high-bandwidth DMA and data mover components that move data between memory-mapped and streaming interfaces. The AXI DMA, PCIe AXI DMA, and AXI CDMA engines all need to contend with this. The PCIe DMA engine needs to deal with this even for aligned operations due to how the PCIe TLPs are formatted.

Unaligned operations my require extra cycles at the start and/or end of the transfer, depending on the source and destination alignment as well as the transfer length. Consider a transfer of length k on a parallel interface that can transfer N words per cycle, with input offset o_i and output offset o_o. For memory-mapped AXI, the offsets o_i and o_o can be computed from the address as $o_x = ADDR_x \bmod N$. The number of cycles required for a transfer is $c_x = \lceil (o_x + k)/N \rceil$.

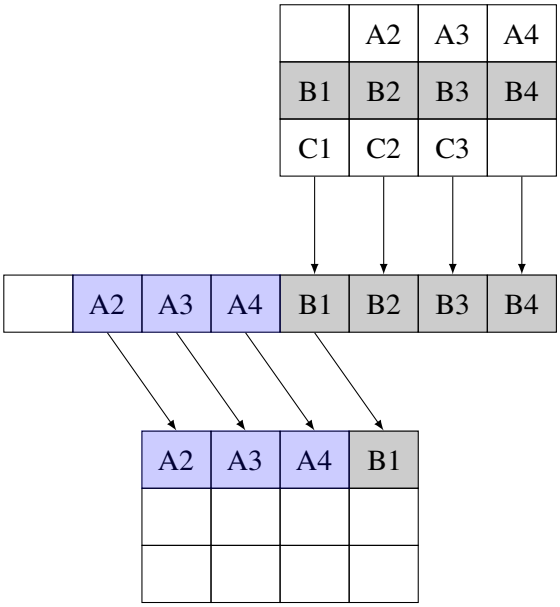
An empty cycle will be generated at the start of a transfer if the first output cycle requires bytes from the first two input cycles. This occurs when $c_i > 1$ and $o_i > o_o$. In this case, the entire first input cycle will be stored and output will not start until the second cycle.

An extra cycle will be generated at the end of a transfer if the last cycle spills over after shifting. This occurs either when $c_o > c_i$, or when $c_o = c_i$ and an empty cycle was generated at the start of the operation (i.e. when $c_i > 1$ and $o_i > o_o$). In this case, an extra output cycle is required to transfer the remainder of the last input cycle.

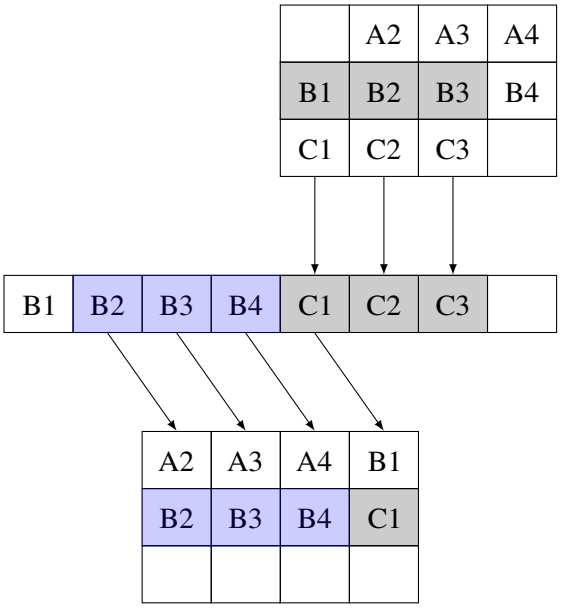
Fig. A.3 depicts an example transfer, with $k = 10$, $N = 4$, $o_i = 1$, and $o_o = 0$, depicting both an empty cycle at the start (as $c_i = \lceil (1 + 10)/4 \rceil = 3 > 1$ and $o_i > o_o$) as well as an extra cycle at the end (as $c_o = c_i$, $c_i > 1$, and $o_i > o_o$).



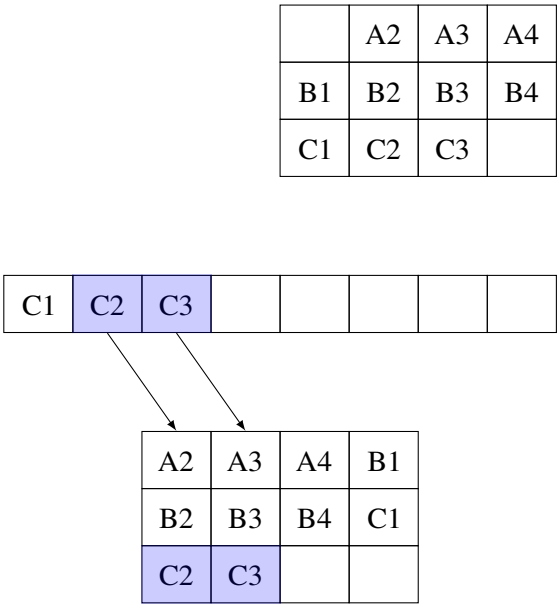
(a) Cycle 1: Fill buffer, no transfer



(b) Cycle 2: unaligned transfer



(c) Cycle 3: unaligned transfer



(d) Cycle 4: empty buffer

Figure A.3: Byte re-packing on a streaming interface

Appendix B

Segmented Memory Interface

For the highest possible performance, Corundum internally uses a custom segmented memory interface. The interface is split into segments of maximum 128 bits, and the overall width is double that of the AXI stream interface from the PCIe hard IP core. This interface provides an improved "impedance match" to the PCIe DMA engine over using a single AXI interface, enabling higher interface utilization by eliminating issues with backpressure and alignment.

Each segment operates similar to AXI lite, except with three interfaces instead of five—one channel provides the write address and data, one channel provides the read address, and one provides the read data. Bursts are not supported, and interconnect logic is responsible for preserving the ordering, even when accessing multiple RAMs. The segments operate completely independently of each other, with separate handshaking connections and separate instances of interconnect ordering logic. Also, operations are routed based on a separate select signal and not by address decoding, greatly simplifying address assignment and enabling the use of parametrizable interconnect components that appropriately route operations with minimal configuration.

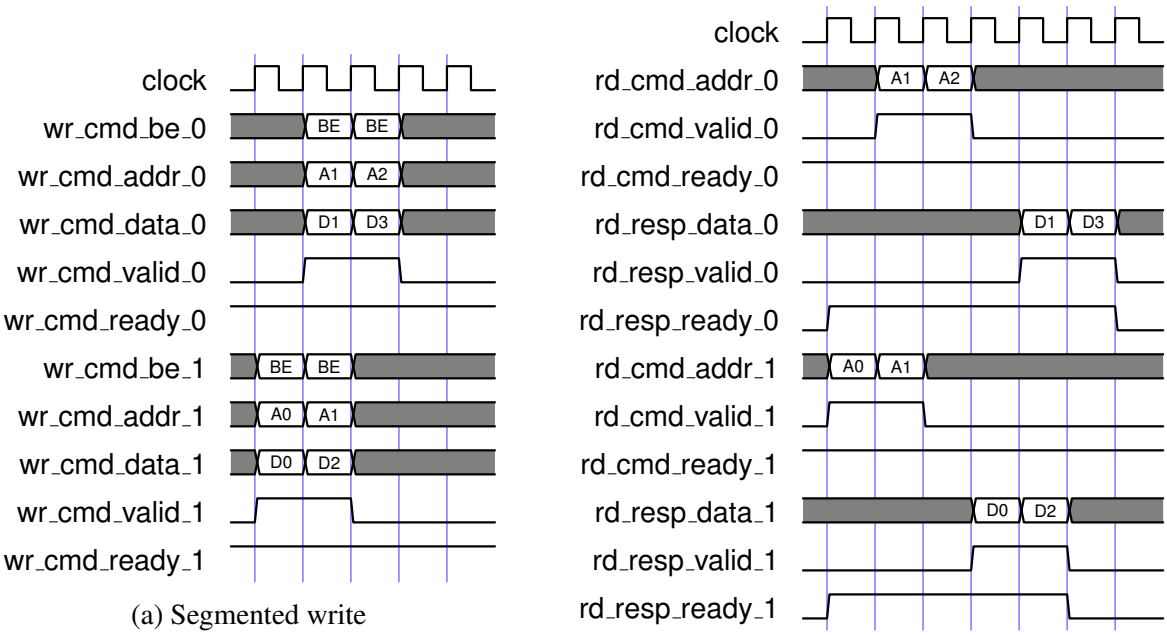
Byte addresses are mapped onto segmented interface addresses with the lowest-order address bits determining the byte lane in a segment, the next bits determining which segment,

and the highest-order bits determining the word address for that segment. For example, a 512 bit PCIe interface would use a 1024 bit segmented interface, split into 8 segments of 128 bits, where the lowest 4 address bits would determine the byte lane in a segment, the next 3 bits would determine the segment, and the rest would drive the address bus for that segment. Operations on blocks of contiguous bytes can extend across multiple segments, up to the full width of the interface, depending on the alignment.

One of the main advantages of using a double-width segmented interface is that it supports arbitrary barrel-shifts to realign data without requiring any additional clock cycles and the resulting backpressure and reduction in throughput. For a segmented interface with k segments, it is possible to operate on contiguous blocks of up to $(k-1)/k$ of the total interface width in each cycle for the worst-case alignment. With a double width interface, only two segments are required to support fully unaligned operations, but fixing the segment width at 128 bits permits efficient operation with PCIe TLP straddling.

Example reads and writes of four adjacent words on a two-segment interface, starting on the second segment, is depicted in Figure B.1.

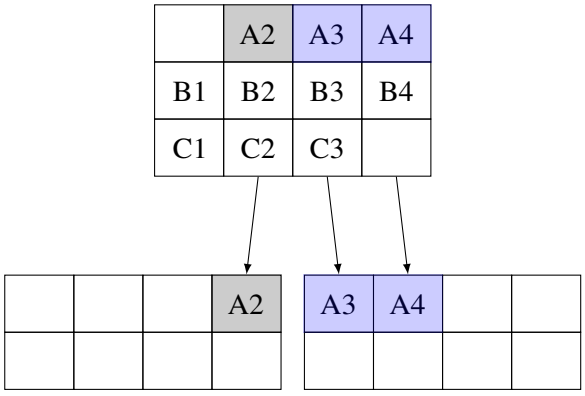
Figure B.2 depicts an example unaligned memory write on a segmented interface. The first and third cycles use the same address for both segments, while the second cycle wraps around, with the two segments accessing adjacent addresses in the same clock cycle. On an interface of the same width, one extra cycle would be required to handle the realignment. On a non-segmented double-width interface, the second cycle that wraps around would have to be split across two cycles. In this case, it would be possible to merge both of those operations with the adjacent operations, but this would not be possible if it was the first or the last cycle in the operation.



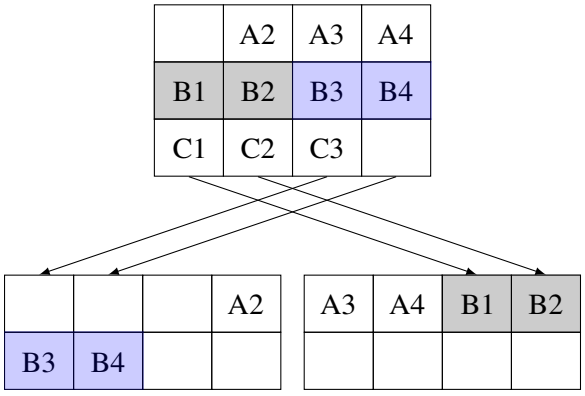
(a) Segmented write

(b) Segmented read

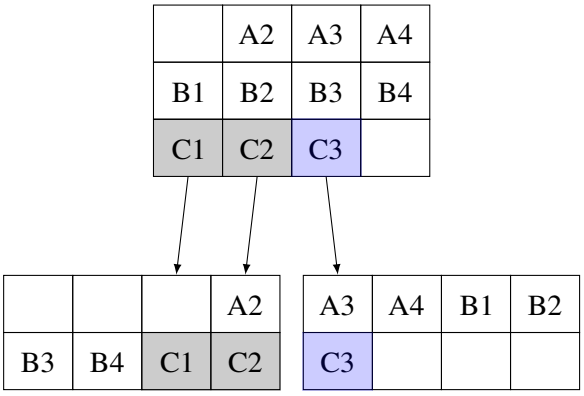
Figure B.1: Transfers on segmented interface



(a) Cycle 1: unaligned write across two segments, same address



(b) Cycle 2: wrapped unaligned write across two segments, adjacent addresses



(c) Cycle 3: unaligned write across two segments, same address

Figure B.2: Byte re-packing on a segmented interface

Appendix C

PCI Express

PCI express is a packet-based protocol enabling high performance memory-mapped access and message passing over parallel high-speed serial links. This appendix contains some high-level notes about PCI express and the Xilinx PCIe UltraScale and UltraScale+ hard IP cores, for complete details refer to the PCIe specification [41] and the relevant Xilinx documentation.

C.1 General Notes

C.1.1 Configuration

There are two types of configuration packets, type 0 and type 1. Type 1 packets are meant to be routed through switches, while type 0 packets are peer-to-peer. Endpoints that are not switches ignore type 1 configuration packets. Switches convert type 1 packets that are targeted to devices directly connected to their downstream ports into type 0 packets.

Endpoint bus and device numbers are never explicitly configured on the device through configuration space registers. The device is expected to capture them from received type 0 configuration packets. The bus and device numbers of a given device are determined by the configuration of the switches.

C.1.2 TLP Size

PCI express has three main parameters that control transaction-layer packet (TLP) sizes: maximum request size, maximum payload size, and read completion boundary. Maximum request size determines the maximum allowable read request size. This parameter ranges from 128 bytes to 4096 bytes in powers of two. On many systems, this is set to 512 bytes, but it can range up to 4096 bytes. Maximum payload size determines the maximum allowable packet payload for write requests and completions. This parameter ranges from 128 bytes to 4096 bytes in powers of two. On many systems, this is set to 128 or 256 bytes. The author is not aware of any commodity server hardware supporting TLP payload sizes larger than 256 bytes. Read completion boundary informs devices of the root complex's read completion boundary setting, either 64 or 128 bytes. All endpoints must use a read completion boundary of 128 bytes, but the root complex can optionally support a read completion boundary of 64 bytes. The read completion boundary setting is meant only to inform endpoints of the root complex configuration and can be used to enable optimizations to better utilize the 64 byte read completion boundary when it is enabled.

C.1.3 Malformed TLPs and Error Reporting

TLPs are considered *malformed* when certain rules are violated. When a malformed TLP is detected at a receiver, it is discarded and logged as an uncorrectable (fatal) error. See [41] section 2.3 and table 6-5.

If a request triggers an unsupported request or a completer abort on a posted transaction, the completer should report an uncorrectable error. If a request triggers an unsupported request or a completer abort on a non-posted transaction, the completer should return a completion with the appropriate status and report a correctable error (advisory non-fatal error). See [41] section 6.2.3.2.4.1.

C.2 Xilinx UltraScale PCIe Core

Xilinx Virtex 7, UltraScale, UltraScale+ FPGAs contain PCI express hard IP cores. However, these cores are a little bit different than what traditionally appears on FPGAs. The UltraScale PCIe hard IP core does things a little differently.

First, it does not pass TLP headers through to the user application directly, it reformats them. This provides a few advantages. First, the user logic doesn't need to know the captured device and bus information, the hard IP core captures this internally and inserts it appropriately into generated packets, simplifying the necessary interconnections. Second, the header size is fixed so user logic doesn't have to deal with multiple payload start offsets on the same interface.

The hard IP core also internally bifurcates incoming requests and messages from incoming completions and provides these TLPs on separate interfaces. The core also internally multiplexes outgoing requests and completions. This can simplify user logic somewhat as the request handling/completion generation logic (MMIO/DMA target) is usually going to be quite distinct from the request generation/completion handling logic (DMA initiator/bus master). As a result, the core has four main streaming interfaces instead of two: the requester request (RQ) interface, for outgoing DMA read and write requests; the requester completion (RC) interface, for incoming completions in response to requests on the RQ interface; the completer request (CQ) interface, for incoming MMIO/DMA read and write requests; and the completer completion (CC) interface, for outgoing completions in response to requests on the CQ interface. The core also has additional logic to handle certain messages, allowing the request handler user logic to handle only memory read and write requests to the device's BARs.

C.2.1 General Notes

Several signals on the hard core must be properly initialized for the link to operate. These are `pcie_cq_np_req`, `cfg_config_space_enable`, and `cfg_link_training_enable`. The

first parameter, `cfg_link_training_enable` must be high for the link to come up. Then, `cfg_config_space_enable` must be high for the device to be enumerated by BIOS and/or the operating system. `pcie_cq_np_req` must be properly handled or tied high so that non-posed requests (read requests) will be delivered to the completion request interface. If `pcie_cq_np_req` is tied low, reads against BARs on the device will time out. See [61] table 2-12 and table 2-18.

C.2.2 Utilization of Wide Interfaces

Since PCIe operations are relatively small and the interpacket gap is also small, transferring PCIe TLPs over a wide bus can result in low utilization.

The maximum TLP payload size rarely set above 128 or 256 bytes on modern servers. The overhead per frame is 12 or 16 bytes for the header, depending on the address size, plus 4 bytes per TLP prefix, plus 8 bytes for link layer overhead, totaling to 20 to 28 bytes for typical TLPs. The overhead is a serious problem for wide interfaces as it can significantly limit achievable throughput over wide interfaces. For an interface like AXI stream or Avalon, all of the TLPs must start in byte lane 0. On a 256 bit bus, 32 bytes are transferred on every clock cycle, so the data portion of a 256 byte TLP requires 4 clock cycles to transfer. However, the header requires 12 or 16 bytes on top of the payload, leaving 16 or 20 empty byte lanes. Since the link layer overhead is 8 bytes, that leaves 8 to 12 equivalent byte times on the table for every TLP transferred. For a 512 bit bus, this problem is even worse. With 64 bytes transferred in each clock cycle, that leaves 48 or 52 empty byte lanes after each TLP, or 40 or 44 bytes after removing link layer overhead. This gap is depicted in Figure C.1a. For 256 byte TLPs on a 512 bit bus, this reduces the achievable payload bandwidth to 80% (100.8 Gbps) of the raw link bandwidth vs. 89% (112.0 Gbps) with straddling enabled or 91.4% (115.2 Gbps) considering only TLP header and link layer overheads.

The solution to this is to modify the interface to support starting TLPs in the gap after a previous TLP. This results in increased throughput over the bus, at the cost of significant interface headaches. This is depicted in Figure C.1b. The Xilinx PCIe hard cores support *TLP straddling*

H0	H1	H2	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	D10	D11	D12
D13	D14	D15					
H0	H1	H2	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	D10	D11	D12
D13	D14	D15					
H0	H1	H2	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	D10	D11	D12
D13	D14	D15					

(a) TLP straddling disabled

H0	H1	H2	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	D10	D11	D12
D13	D14	D15		H0	H1	H2	D0
D1	D2	D3	D4	D5	D6	D7	D8
D9	D10	D11	D12	D13	D14	D15	
H0	H1	H2	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	D10	D11	D12
D13	D14	D15					

(b) TLP straddling enabled

Figure C.1: TLP packing

on some of their interfaces, which modifies the AXI interface to handle transferring (parts of) multiple TLPs per clock cycle. In 256 bit mode, straddling is only supported for completion TLPs on the requester completion (RC) interface. With TLP straddling enabled, two TLP start positions are supported at byte lane 0 and 16. In 512 bit mode, the underlying core is simply run at 500 MHz and a soft 'shim' is used to convert from 256 to 512 bits [63]. As a result, straddling is supported on all interfaces, with the RC interface supporting starting TLPs on 16 or 32 byte intervals (2 or 4 TLP start positions per cycle) and the other interfaces supporting only 32 byte intervals (2 TLP start positions per cycle).

C.2.3 Flow Control

PCI express uses credit-based flow control to manage buffer space. Each virtual channel on a given PCIe link maintains an independent set of credits. There are six types of credits: posted header, posted data, non-posted header, non-posted data, completion header, and completion data. Each TLP generally requires one data credit per 4 dwords (16 aligned bytes) of payload and 1 header credit. TLPs may only be transmitted over a link if sufficient credit is available. The link

parter releases credits as buffer space is freed by periodically sending updateFC data-link-layer packets (DLLP).

The Xilinx PCIe hard IP core on Virtex 7, UltraScale, and UltraScale+ exposes the current flow control credit counts on its `cfg_fc_*` ports. If the core cannot accept a TLP on the RQ interface, it will block and assert backpressure by deasserting `s_axis_rq_tready`. To prevent head of line blocking between read and write request TLPs, it is advisable to check the appropriate credit counts before sending requests. However, there are a few additional considerations with the Xilinx PCIe hard IP core.

First, the core does not perform any buffering for posted write requests. If a posted write it sent to the core while insufficient header and data credits are available, the core will block the RQ interface until credits are freed, preventing read request TLPs from being sent. Therefore, checking `cfg_fc_ph` and `cfg_fc_pd` (with `cfg_fc_sel` set to `3'b100` to select available transmit credits) is imperative before generating every write request TLP.

Second, the core reserves buffer space to store completion packets, and non-posted requests will not be released from the core if there is insufficient buffer space to store the corresponding completions. Inferring from ILA captures, it appears that the completion buffer on the UltraScale+ hard IP core is around 16 KB in size, supporting up to 32 outstanding read requests for 512 bytes (common max read request size setting) at any given time. Additionally, there is a FIFO inside the hard IP core for storing these non-posted requests. The FIFO appears to be able to store around 24 non-posted read request TLPs. Once the buffer is full, additional requests will cause the RQ interface on the core to block, preventing write requests from being sent to the core.

The core does provide credit counts that are intended to reflect the internal buffer status—`pcie_tfc_nph_av` and `pcie_tfc_npd_av`—but these do not appear to be implemented correctly as they can get stuck at 0 when the link is idle. Therefore, a different method is required to track the non-posted transmit FIFO occupancy.

C.2.4 Transmit Sequence Numbers

The Xilinx PCIe hard IP core on Virtex 7, UltraScale, and UltraScale+ supports transmit sequence numbers to track TLPs through the transmit pipeline. This is the only way to know when a specific TLP has actually been sent over the PCIe link. The main use case is to ensure that the view from host software is consistent. Namely, the host must always have an accurate view of which DMA read and write operations have been completed, specifically reads on internal device registers should not imply that a write operation is complete when the write request TLP has not yet left the PCIe IP core. Transmit sequence numbers enable a tag to be associated with each transmit TLP that is returned from the core via the `pcie_rq_seq_num`/`pcie_rq_seq_num_valid` ports.

Using transmit sequence numbers also appears to be the only reliable way to measure the occupancy of the core non-posted transmit FIFO. The `pcie_tfc_nph_av` and `pcie_tfc_npd_av` ports on the core that are intended to indicate the occupancy of this FIFO often return nonsensical values. However, monitoring the number of outstanding transmit sequence numbers provides the same information, and is relatively simple to implement in a DMA engine that already has to track sequence numbers to report when all requests associated with each operation have been transmitted from the core. Therefore, it is imperative to track transmit sequence numbers and limit the number of read requests in the transmit pipeline to prevent head-of-line blocking of write requests on the core RQ interface.

Bibliography

[1] R. Aguinaldo, A. Forencich, C. DeRose, A. Lentine, D. C. Trotter, Y. Fainman, G. Porter, G. Papen, and S. Mookherjea. Wideband silicon-photonic thermo-optic switch in a wavelength-division multiplexed ring network. *Opt. Express*, 22(7):8205–8218, Apr 2014.

[2] ARM. *AMBA AXI and ACE Protocol Specification*, Feb 2013.

[3] Atomic rules. http://www.atomicrules.com/.

[4] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, and et al. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[5] K. Clark, H. Ballani, P. Bayvel, D. Cletheroe, T. Gerard, I. Haller, K. Jozwik, K. Shi, B. Thomsen, P. Watts, H. Williams, G. Zervas, P. Costa, and Z. Liu. Sub-nanosecond clock and data recovery in an optically-switched data centre network. In *2018 European Conference on Optical Communication (ECOC)*, pages 1–3, Sep. 2018.

[6] Data plane development kit. https://www.dpdk.org/.

[7] N. Dupuis, F. Doany, R. A. Budd, L. Schares, C. W. Baks, D. M. Kuchta, T. Hirokawa, and B. G. Lee. A nonblocking 4×4 Mach-Zehnder switch with integrated gain and nanosecond-scale reconfiguration time. In *Optical Fiber Communication Conference (OFC) 2019*, page W1E.2. Optical Society of America, 2019.

[8] N. Dupuis and B. G. Lee. Impact of topology on the scalability of Mach-Zehnder-based multistage silicon photonic switch networks. *Journal of Lightwave Technology*, 36(3):763–772, Feb 2018.

[9] N. Dupuis, B. G. Lee, A. V. Rylyakov, D. M. Kuchta, C. W. Baks, J. S. Orcutt, D. M. Gill, W. M. J. Green, and C. L. Schow. Design and fabrication of low-insertion-loss and low-crosstalk broadband 2×2 Mach–Zehnder silicon photonic switches. *Journal of Lightwave Technology*, 33(17):3597–3606, Sep 2015.

[10] Exablaze. https://exablaze.com/.

[11] N. Farrington, A. Forencich, G. Porter, P. Chen-Sun, J. E. Ford, Y. Fainman, G. C. Papen, and A. Vahdat. A multiport microsecond optical circuit switch for data center networking. *IEEE Photonics Technology Letters*, 25(16):1589–1592, Aug 2013.

[12] N. Farrington, A. Forencich, P.-C. Sun, S. Fainman, J. Ford, A. Vahdat, G. Porter, and G. C. Papen. A 10 us hybrid optical-circuit/electrical-packet network for datacenters. In *Optical Fiber Communication Conference*, pages OW3H–3. Optical Society of America, 2013.

[13] N. Farrington, G. Porter, A. Forencich, J. Ford, Y. Fainman, A. Vahdat, and G. Papen. Optical/electrical hybrid switching for datacenter communications. In *2013 18th OptoElectronics and Communications Conference held jointly with 2013 International Conference on Photonics in Switching (OECC/PS)*, pages 1–2, June 2013.

[14] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, Aug. 2010.

[15] N. Farrington, G. Porter, P.-C. Sun, A. Forencich, J. Ford, Y. Fainman, G. Papen, and A. Vahdat. A demonstration of ultra-low-latency data center optical circuit switching. *SIGCOMM Comput. Commun. Rev.*, 42(4):95–96, Aug. 2012.

[16] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.

[17] K.-T. Foerster, M. Ghobadi, and S. Schmid. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, pages 89–96, New York, NY, USA, 2018. ACM.

[18] A. Forencich, V. Kamchevska, N. Dupuis, B. G. Lee, C. Baks, G. Papen, and L. Schares. System-level demonstration of a dynamically reconfigured burst-mode link using a nanosecond si-photonic switch. In *Optical Fiber Communication Conference*, page Th1G.4. Optical Society of America, 2018.

[19] A. Forencich, V. Kamchevska, N. Dupuis, B. G. Lee, C. Baks, G. Papen, and L. Schares. A dynamically-reconfigurable burst-mode link using a nanosecond photonic switch. *Journal of Lightwave Technology*, 38(6):1330–1340, March 2020.

[20] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 20)*, Fayetteville, AR, May 2020. IEEE.

[21] H. R. Grant, A. Forencich, G. Papen, N. Dupuis, L. Schares, R. A. Budd, and B. G. Lee. Bit error rate measurements of a 4 $\times$ 4 si-photonic switch using synchronous and asynchronous data. In *2016 IEEE Optical Interconnects Conference (OI)*, pages 32–33, May 2016.

[22] J. Gripp et al. Demonstration of a 1.2 Tb/s optical packet switch fabric. In *27th European Conference on Optical Communication (ECOC)*, volume 1, 2001.

[23] S. Han, T. J. Seok, N. Quack, B.-W. Yoo, and M. C. Wu. Large-scale silicon photonic switches with movable directional couplers. *Optica*, 2(4):370–375, Apr 2015.

[24] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[25] H. Y. Hwang, J. S. Lee, T. J. Seok, A. Forencich, H. R. Grant, D. Knutson, N. Quack, S. Han, R. S. Muller, G. C. Papen, M. C. Wu, and P. O'Brien. Flip chip packaging of digital silicon photonics MEMS switch for cloud computing and data centre. *IEEE Photonics Journal*, 9(3):1–10, June 2017.

[26] R. Konoike, K. Suzuki, T. Inoue, T. Matsumoto, T. Kurahashi, A. Uetake, K. Takabayashi, S. Akiyama, S. Sekiguchi, S. Namiki, H. Kawashima, and K. Ikeda. SOA-integrated silicon photonics switch and its lossless multistage transmission of high-capacity WDM signals. *J. Lightwave Technol.*, 37(1):123–130, Jan 2019.

[27] B. G. Lee, N. Dupuis, P. Pepeljugoski, L. Schares, R. Budd, J. R. Bickford, and C. L. Schow. Silicon photonic switch fabrics in computer communications systems. *Journal of Lightwave Technology*, 33(4):768–777, Feb 2015.

[28] B. G. Lee, A. V. Rylyakov, W. M. J. Green, S. Assefa, C. W. Baks, R. Rimolo-Donadio, D. M. Kuchta, M. H. Khater, T. Barwicz, C. Reinholm, E. Kiewra, S. M. Shank, C. L. Schow, and Y. A. Vlasov. Monolithic silicon integration of scaled photonic switch fabrics, CMOS logic, and device driver circuits. *Journal of Lightwave Technology*, 32(4):743–751, Feb 2014.

[29] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and et al. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[30] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter. Circuit switching under the radar with REACToR. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 1–15, Berkeley, CA, USA, 2014. USENIX Association.

[31] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, and A. C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 41:1–41:13, 2015.

[32] R. McGuinness and G. Porter. Evaluating the performance of software NICs for 100-Gb/s datacenter traffic control. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 74–88, New York, NY, USA, 2018. Association for Computing Machinery.

[33] H. Mehrvar, Y. Wang, X. Yang, M. Kiaei, H. Ma, J. Cao, D. Geng, D. Goodwill, and eric Bernier. Scalable photonic packet switch test-bed for datacenters. In *Optical Fiber Communication Conference*, page W3J.4. Optical Society of America, 2016.

[34] W. M. Mellette. A practical approach to optical switching in data centers. In *Optical Fiber Communication Conference (OFC) 2019*, page M2C.3. Optical Society of America, 2019.

[35] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1–18, Santa Clara, CA, Feb. 2020. USENIX Association.

[36] W. M. Mellette and J. E. Ford. Scaling limits of MEMS beam-steering switches for data center networks. *J. Lightwave Technol.*, 33(15):3308–3318, Aug 2015.

[37] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. RotorNet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 267–280, New York, NY, USA, 2017. ACM.

[38] W. M. Mellette, A. C. Snoeren, and G. Porter. P-FatTree: A multi-channel datacenter network topology. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 78–84, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Netcope technologies. https://www.netcope.com/en.

[40] I. Ozkaya, A. Cevrero, P. A. Francese, C. Menolfi, M. Braendli, T. Morf, D. Kuchta, L. Kull, M. Kossel, D. Luu, M. Meghelli, Y. Leblebici, and T. Toifl. A 56gb/s burst-mode NRZ optical receiver with 6.8ns power-on and CDR-Lock time for adaptive optical links in 14nm FinFET CMOS. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 266–268, Feb 2018.

[41] PCI-SIG. *PCI Express Base Specification Revision 3.0*, Nov 2010.

[42] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, Feb. 2019. USENIX Association.

[43] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 447–458, New York, NY, USA, 2013. Association for Computing Machinery.

[44] X. Z. Qiu, X. Yin, J. Verbrugghe, B. Moeneclaey, A. Vyncke, C. V. Praet, G. Torfs, J. Bauwelinck, and J. Vandewege. Fast synchronization 3R burst-mode receivers for passive optical networks. *Journal of Lightwave Technology*, 32(4):644–659, Feb 2014.

[45] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 475–488, Seattle, WA, 2014. USENIX Association.

[46] M. Rizzi, M. Lipiński, T. Wlostowski, J. Serrano, G. Daniluk, P. Ferrari, and S. Rinaldi. White Rabbit clock characteristics. In *2016 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, pages 1–6, Sep. 2016.

[47] A. Rylyakov, J. E. Proesel, S. Rylov, B. G. Lee, J. F. Bulzacchelli, A. Ardey, B. Parker, M. Beakes, C. W. Baks, C. L. Schow, and M. Meghelli. A 25 gb/s burst-mode receiver for low latency photonic switch networks. *IEEE Journal of Solid-State Circuits*, 50(12):3120–3132, Dec 2015.

[48] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 404–417, New York, NY, USA, 2017. Association for Computing Machinery.

[49] L. Schares, B. G. Lee, F. Checconi, R. Budd, A. Rylyakov, N. Dupuis, F. Petrini, C. L. Schow, P. Fuentes, O. Mattes, and C. Minkenberg. A throughput-optimized optical network for data-intensive computing. *IEEE Micro*, 34(5):52–63, Sept 2014.

[50] T. J. Seok, H. Y. Hwang, J. S. Lee, A. Forencich, H. R. Grant, D. Knutson, N. Quack, S. Han, R. S. Muller, L. Carroll, G. C. Papen, P. O'Brien, and M. C. Wu. 12×12 packaged digital silicon photonic MEMS switches. In *2016 IEEE Photonics Conference (IPC)*, pages 629–630, Oct 2016.

[51] Y. Shen, P. Samadi, Z. Zhu, A. Gazman, E. Anderson, D. Calhoun, M. Hattink, and K. Bergman. Software-defined networking control plane for seamless integration of silicon

photonics in datacom networks. In *2017 European Conference on Optical Communications (ECOC)*, pages 1–3, 2017.

[52] V. Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 367–379, New York, NY, USA, 2019. Association for Computing Machinery.

[53] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 255–270, Boston, MA, Feb. 2019. USENIX Association.

[54] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. SIGCOMM '15.

[55] R. Stabile, A. Albores-Mejia, and K. A. Williams. Monolithic active-passive 16×16 optoelectronic switch. *Optics Letters*, 37(22):4666–4668, Nov 2012.

[56] B. Stephens, A. Akella, and M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, Feb. 2019. USENIX Association.

[57] B. Stephens, A. Akella, and M. M. Swift. Your programmable NIC should be a programmable switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 36–42, New York, NY, USA, 2018. Association for Computing Machinery.

[58] K. Suzuki, K. Tanizawa, T. Matsukawa, G. Cong, S.-H. Kim, S. Suda, M. Ohno, T. Chiba, H. Tadokoro, M. Yanagihara, Y. Igarashi, M. Masahara, S. Namiki, and H. Kawashima. Ultra-compact 8×8 strictly-non-blocking si-wire piloss switch. *Opt. Express*, 22(4):3887–3894, Feb 2014.

[59] Top 500 supercomputers. http://top500.org.

[60] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for datacenter ethernet. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[61] Xilinx. *UltraScale Devices Gen3 Integrated Block for PCI Express v4.4*, Apr 2018.

[62] Xilinx. *UltraScale+ Devices Integrated 100G Ethernet Subsystem v2.6*, May 2019.

[63] Xilinx. *UltraScale+ Devices Integrated Block for PCI Express v1.3*, Jun 2019.

[64] Y. Xiong, F. G. de Magalhães, G. Nicolescu, F. Hessel, and O. Liboiron-Ladouceur. Co-design of a low-latency centralized controller for silicon photonic multistage mzi-based switches. In *2017 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3, March 2017.

[65] S. Zhang. Fine-grained end-host traffic control. Master's thesis, UC San Diego, 2015.

[66] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sep. 2014.