TOWARDS A PROGRAMMABLE DATAPLANE

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Han Wang

May 2017

TOWARDS A PROGRAMMABLE DATAPLANE

Han Wang, Ph.D.

Cornell University 2017

Programmable network dataplanes can significantly improve the flexibility and functionality of computer networks. This dissertation investigates two building blocks of network dataplane programming for network devices: the packet processing pipeline and network device interface. In the first part of the dissertation, we show that designing packet processing pipelines on hardware can be fast and flexible (programmable). A network dataplane compiler and runtime is presented that generates a custom FPGA dataplane designed and built from a dataplane programming language called P4 (programming protocol independent packet processors). P4FPGA generates designs that can be synthesized to either Xilinx or Altera FPGAs. We have benchmarked several representative P4 programs, and the experiments show that code generated by P4FPGA runs at line-rate at all packet sizes with latencies comparable to commercial ASICs. In the second part of the dissertation, we present a programmable network interface for the network dataplane. We show that a software programmable physical layer (programmable PHY) can capture and control the timing of physical layer bits with sub-nanosecond precision greatly increasing precision in network measurements. The benefits of a programmable PHY is demonstrated with an available bandwidth estimation algorithm and a decentralized clock synchronization protocol that provides bounded precision where no two clocks differ by more than tens of nanoseconds.

BIOGRAPHICAL SKETCH

Han Wang attended the University of Auckland from 2003 to 2007 for his undergraduate studies in Computer Systems Engineering. Han spent a few months working as an embedded system software engineer before starting his PhD program in Electrical and Computer Engineering in Cornell University in 2008. He spent the first two years working with Professor Francois Guimbretiere on low power embedded system design before joining Hakim Weatherspoon's group to focus on system and networking research. In 2013 and 2014, Han spent two summers at Nicira/VMware working on Software Defined Networking (SDN) controller and dataplane projects. In 2014 and 2015, Han Wang helped building the initial prototypes that form the basis of Waltz Networks Inc with Kevin Tang and Nithin Michael. In 2016, Han Wang accepted a position at Barefoot Networks Inc to pursue what he believed as the next generation programmable switch ASICs and compiler technology.

for my family

ACKNOWLEDGEMENTS

First, I would like to express my deepest gratitude to my advisor, Hakim Weatherspoon, who has profoundly influenced my life at Cornell. I have been extremely lucky to have an advisor who put enormous trust and confidence into my ability, who offered unparalleled guidance for me to become an independent researcher, and who provided the complete freedom for me to pursue exciting ideas. I could not ask for more from an advisor.

My work wouldn't be possible without my wonderful collaborators. I would like to thank Ki Suh Lee, for his extrodinary ability in system programming and his meticulous attention to details. Without him, the SoNIC project would never be possible. Hyun Tu Dang and Robert Soule are the co-authors of the P4Paxos project. Thanks to Tu for taking the leap-of-faith to use my P4 compiler. Thanks to Robert for being a great collaborator and a great advisor. His help on my writing and presentation skills have been invaluable in the late stage of my PhD career. Jamey Hicks introduced me to Bluespec and Connectal, which has been enormously helpful for the development P4FPGA. Thanks to Jamey for his unreserved help and guidance during my visit to MIT and after I returned to Cornell.

I would like to thank Hakim Weatherspoon, Emin Gun Sirer, Rajit Manohar for serving on my thesis committee, and giving me valuable feedback on the works.

Kevin A. Tang, Nithin Michael and Ning Wu have been another source of inspiration for my PhD career. Thanks to Kevin and Nithin for giving me the opportunity to participate in their early endeavors of Waltz Networks Inc. The experience of working with an early stage start-up is an invaluable lesson.

I would like to thank Alan Shieh and Mukesh Hira for taking me as an intern at VMWare networking group in Palo Alto. Thanks to Baris Kasikci, Jeff Rasley for being wonderful friends, team mates at VMWare. I enjoyed the brainstorming, and lunch

discussions.

I am gratefully to many friends at Cornell. Songming Peng, Yuerui Lu, Han Zhang, Pu Zhang, Xi Yan, Ryan Lau, Weiwei Wang, Jiajie Yu, Jiahe Li, Qi Huang, Zhiming Shen, Weijia Song, Haoyan Geng, Erluo Li, Zhefu Jiang, Qin Jia, and too many others to mention. I cannot forget all the parties, game nights, mid-night conversations we have had. You made my life at Ithaca much more fun.

Last I would like to thank my family. I thank my parents, dad Xijin Wang, mom Qiaofeng Li, for their enduring love, support and belief, without which I would not have finished this journey. I thank my wife Fan Zhang, for her love and faith in me through the ups and downs of my PhD career. It is the optimism and happiness from her that drives me through the long journey. I dedicate this dissertation to them all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer networks are critical to our society. They are the basis of large scale distributed applications, such as search engines, e-commerce, and social networks, which have profoundly changed how we access information and interact with each other. These applications are hosted in data centers, where a large group of networked computer servers are connected by large-scale data center networks. Further, emerging new applications, such as machine learning [140], and big data processing [134] are implemented in data centers. These data center networks, and networks in general, must be flexible to cope with new system architectures, higher performance requirements and stringent application demands.

The part of the network that performs all the heavy lifting is the *network dataplane*. The network dataplane forwards and processes packets. Given the demands of continual technological development, a network dataplane needs to be fast, and needs to evolve over time. But currently, the dataplane is not directly accessible by network developers, operators or programmers. Instead, the programming to evolve the network has to go through a layer of software called the *control plane*. The control plane is typically co-located with the dataplane in the same network device (e.g. in a router). Enabling programmable access to the network dataplane is the focus of this dissertation.

This dissertation represents a step towards identifying and addressing the challenges of designing a framework for programmable dataplanes, which can significantly improve the flexibility and functionality of networks. To this end, we investigate two building blocks of the network dataplane programming for network devices: the packet processing pipeline and network device interface. We asked the research question: *how can we build a programmable dataplane to improve network measurement, enable novel*

network functionality, and accelerate development time from concept to prototype? We explore this question with two approaches: a programmable dataplane that automates generation of the network dataplane from a high-level dataplane programming language; and a programmable network interface that allows precise control and measurement of packet timing.

1.1 Network Dataplane Programming

Dataplane programming is distinct from control plane programming. The former enables programming of the entire network while the latter enables programming network devices through interfaces exposed by control plane software. In order to illustrate the value of network dataplane programming, we discuss some concrete examples that are not possible through control plane programming alone.

Rapid Prototyping of Protocols Data center networks are often under one administrative domain [4], which means a cloud provider, such as Google, Amazon, or Facebook can design their own network hardware and protocol suite to optimize for their network use cases. However, if a custom protocol cannot be supported by the network dataplane, deployment of custom protocols becomes difficult. The history of VxLAN (Virtual eXtensible LANs) protocol deployment is a good example [131]. As the name VxLAN implies, the technology is meant to provide the same services to connected Ethernet end systems that VLANs (Virtual Local Area Network) do, but in a more extensible manner. Compared to VLANs, VxLANs are extensible with regard to scale of the network, and extensible with regard to the reach of their deployment. The idea of VxLAN was proposed initially in 2011 in a IETF (Internet Engineering Task Force) draft [131], as a method to efficiently support network virtualization at data center scale. However,

an efficient implementation of VxLAN in a router (or switch) ASIC (Application Specific Integrated Circuit) was not available until 2014 [187], which inevitably affected the adoption of the new technology. A programmable dataplane allows network operators to rapidly prototype new protocols without waiting for the protocol to be fabricated into a commodity switch ASIC.

Accelerating Distributed Applications Data centers are built with hundreds of thousands of servers and network switches, with large distributed applications running on top of the servers. This is what we call the *cloud*. Increasingly, cloud providers are moving towards adding accelerators, e.g., Graphical Processing Units (GPUs) [97], Field Programmable Gate Arrays (FPGAs) [43], Tensor Processing Units (TPUs) [201] to the cloud, to improve the performance of cloud applications, such as machine learning, video encoding, or artificial intelligence [140]. It is interesting to ask if the network dataplane can accelerate distributed applications, offloading work from the servers. For example, researchers have demonstrated how the Paxos distributed consensus protocol can be added to the network by implementing a portion of the protocol in network dataplane [57]. A similar trend can be found in the financial industry, which has used dataplane based acceleration in high frequency trading [71].

Precise Network Measurement The physical Layer (PHY) of the network dataplane can be used for precise network measurement if made accessible to programmers. For example, available bandwidth, which is defined as the maximum data rate that a process can send to another process without going over the network path capacity between the two, can be estimated with algorithms based on active probing [173]. Available bandwidth estimation algorithms work by examining and detecting any changes in measured one-way delay of probe packets to infer queuing (buffering) in the network. The one-

way delay is the amount of time for a packet to travel from the sender to the receiver of the packet. An increase in one-way delay implies more buffering, queuing or congestion in the network path, and therefore less available bandwidth. Being able to precisely control and measure active probes in the dataplane enables accurate bandwidth estimation algorithms.

Clock Synchronization Servers in data center networks are loosely synchronized. Existing clock synchronization protocols provide different levels of precision, including NTP [141] and PTP [12]. Synchronization precision is the maximum difference between any two clocks [100]. NTP can provide millisecond to microsecond precision in a Local Area Network (LAN), and PTP can provide tens of nanosecond to microsecond precision in a LAN if properly configured [111, 112]. In particular, PTP employs many techniques to remove uncertainties in measured round-trip times. For example, hardware timestamping is commonly used and PTP-enabled switches are deployed to minimize network jitter. Nonetheless, it is not possible to completely remove the network jitter and non-deterministic delays. The accuracy of PTP can be as close as hundreds of nanoseconds, and can degrade to tens of microseconds depending on network conditions [112].

The physical layer of network devices can be used to implement a decentralized clock synchronization protocol. By doing so, the clock synchronization protocol eliminates most non-deterministic elements in clock synchronization protocols. Further, the protocol can send control messages in the physical layer for communicating hundreds of thousands of protocol messages without interfering with higher layer packets. Thus, it has virtually zero overhead since no load is added to layers 2 or higher at all.

1.2 Problems with Existing Network Dataplanes

Programming the network is meant to control the packet forwarding and processing behavior of network elements. A *network element* is a computer system that can forward, filter, or modify network packets. It is challenging to design network elements with existing dataplanes to handle the above examples due to the following problems:

Flexibility or Performance, Pick One. The implementation of the network dataplane is often a trade-off between flexibility and performance. A flexible network dataplane can be built from conventional servers and CPUs, e.g., Click [98], RouteBricks [62], OpenvSwitch [164]. However, the end of Moore's law [175] has made it more difficult to build software-based network dataplanes to cope with increases in network link technology. In particular, while processors no longer scale in frequency, network link technology continues to increase exponentially (1Gbps, 10Gbps, 100Gbps, 400Gbps). Alternatively, network dataplanes are often built with hardware network elements (e.g., switches and routers). These hardware network dataplanes are fast, but not flexible. Hardware dataplanes can be implemented with ASICs. As the name implies, ASICs are manufactured for a specific set of applications (in this case, a set of network protocols). Dataplanes that are built with ASICs are often referred to as *fixed-function* dataplanes. For example, the VxLAN example shows fixed-function dataplanes typically require a long, multi-year design cycle to support custom protocols.

Complex Programming Model and Vendor Centric Design. To keep up with changing requirements, computer architects have devised new families of computer chips. Network processors (NPs) are software programmable chips designed to process packets at line speed, but at 1/100th overall throughput compared to fixed function

ASICs [49]. Since the processing latency usually exceeds the packet inter-arrival time, multiple packets are typically processed concurrently. For this reason, network processors usually consist of multi-threaded multiprocessors. Multi-threading has also been used extensively in network processors to hide pipeline stalls [49]. Further, complex instruction sets and vendor-specific architectures result in complex programming models. Adopting a network processor often results in vendor lock-in to a particular provider's solutions.

FPGAs are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. While FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing, other issues endemic to this chip set make them difficult to use for packet processing. The programming languages for programming FPGAs, such as Verilog or VHDL, are low level languages. Hardware design with complex control logic manifests itself at module boundaries as ad-hoc protocols and assumptions on signaling. Without consistent and clear semantics on protocols and signalings, designs tend to accumulate layer upon layer of such assumptions, often poorly documented (if documented at all) and poorly communicated between the designer of a module and the users of the module, which makes designing with Verilog difficult and prone to errors [147].

In order to support custom protocols or accelerate application logic, as discussed in the first two examples, both NPs and FPGAs based solutions require extensive effort to rewrite the protocol or logic to vendor-specific architectures, programming models or programming languages that are used by these reconfigurable devices.

Opaque, Non-programmable Components. Part of the network dataplane is not programmable at all. For example, the physical layer is usually implemented in hardware,

and, consequently, its behavior cannot be modified through programs. The physical layer defines the means of transmitting raw bits and signals rather than logical data packets over a physical link connecting network nodes. The bit stream may be grouped into code words or symbols and converted to a physical signal that is transmitted over a hardware transmission medium. The physical layer provides a plethora of services to the upper layers of the network stack: modulation, line encoding, bit synchronization, circuit switching, forwarding error correction, and so on. Despite the rich services provided by the physical layers, they are largely ignored by the system programmers as opaque components. The situation is further exacerbated by commodity network interface cards (NICs), which do not provide nor allow an interface for users to access the PHY in any case. However, the physical layer has the most precise control over how bits are sent over the network link. The fact that operations on physical layers are agnostic to upper layers of the network stack provides a unique opportunity to implement accurate timestamping, network measurement and clock synchronization protocols.

1.3 The Rise of Programmable Network Dataplanes

Programmable network dataplanes can fundamentally change the way network elements are built and managed. They have the following distinct features from the existing dataplanes.

Balancing Flexibility and Performance Programmable dataplanes balance flexibility and performance. The key to this balancing act is a good abstraction for *packet processing*. A good abstraction must be high level enough, such that it allows software to be built on top of the abstraction that easily captures the intent of the packet processing task. However, the abstraction must also be at a low enough level, such that it can

be easily mapped to appropriate hardware for implementation.

A similar analogy can be found in Graphics Processing Units (GPUs) or Digital Signal Processing (DSP). In both cases, an abstraction exists to represent the underlying hardware. In the case of a GPU, a unit of execution is abstracted as a thread to be executed on one of the thousands of hardware cores available on a single GPU. In the case of DSP, the abstractions are key mathematical functions to transform signals, e.g., filtering functions like Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filter blocks, common math functions such as square root, cosine, sine, and Fast Fourier Transforms (FFT).

A common abstraction for packet processing is *match-action* pipeline, first proposed by OpenFlow [137]. With the match-action abstraction, a packet processor can be modeled as a sequence of match and action pipeline stages. Each pipeline stage performs a different operation on the packets that flow through. Such abstraction can be mapped to FPGAs and next generation ASICs.

Programming with a Network Dataplane Language Languages can be built on top of the appropriate abstraction to allow developers to program the network dataplane. One example of this is the dataplane programming language, P4 [155], which stands for *programming protocol-independent packet processors*. P4 provides a set of programming constructs that represent the basic building block for packet processing, such as parser, match table, action block. Programmers are already using the language to implement a variety of novel applications including network telemetry tools [95] and advanced load balancers [94]. The language is platform-independent, which means it can be mapped to all of the dataplane implementation technologies: software, FPGAs, ASICs, NPs. The language is also architecture-independent, which means it can be used

to describe different dataplane architectures, such as bump-in-the-wire [28], which is often used to implement network functions, such as network proxies, and multi-port switching, that are commonly found in switches and routers.

Programmable PHYs Programmable physical layers (PHYs) provide the opportunity to improve upon and develop new network research applications which were not previously feasible. First, as a powerful network measurement tool, a programmable PHY can generate packets at full data rate with minimal interpacket delay. It also provides fine-grain control over the interpacket delay. Second, a programmable PHY can accurately capture and timestamp incoming packets at any data rate. Further, this precise timestamping can improve the accuracy of research based on interpacket delay. For example, a programmable PHY can be used to profile network components [113].

1.4 Challenges in the Programmable Dataplane Design

In this section, we describe the challenges inherent to research supporting programmable network dataplanes and explain two research questions that this dissertation addresses.

1.4.1 Lack of Programmable Network Dataplanes: Packet Processing Pipeline

Packet processing pipelines are used by individual network elements to implement packet processing algorithms. Many features provided by network elements are packet processing tasks in one way or another: routing, firewall, tunneling, and encryption. A router processes packet headers, and redirects packets based on the header content. A

firewall filters packets based on content in a defined set of packet headers. A packet processing pipeline is *flexible* if it allows a user to specify complex packet processing tasks from simple high-level language constructs. A packet processing pipeline is *fast* if the pipeline can scale to multiple network interfaces simultaneously, such as 10 or 25 Gbps network interfaces. Unfortunately, state-of-the-art packet processing pipelines are either flexible or fast – not both.

The challenge is in designing a flexible *and* fast packet processing pipeline. Specifically, we address the research question: *How can packet processing pipelines be as flexible as software processors, while achieving high performance close to hardware packet processors?*

To address these challenges, a viable approach must enable users to specify the functionality of the pipeline, ideally through a programming language with high-level abstraction. A technique should minimize the effort to translate from a high-level specification to a high performance hardware implementation. It should also be portable to different hardware targets based on the complexity of the application and resource utilization. Unfortunately, existing systems do not satisfy these requirements in full. NetFPGA [128] offers a framework to implement high-performance, efficient packet processing pipeline in FPGA, but the users have to manually implement the desired packet processing algorithm in low-level hardware description language. OpenvSwitch [152] offers a flexible software packet processing pipeline, but the performance is limited to by the capability of host CPUs and the complexity of the processing algorithm.

A dataplane compiler and runtime enables users to generate a custom packet processing pipeline from a high-level dataplane language. As an instance of the approach, we present P4FPGA, an open-source P4-to-FPGA compiler and runtime that is designed to be flexible, efficient, and portable. The compiler has two parts: a front end that turns

the P4 code into a target-independent intermediate representation (IR), and a back end that maps the IR to the FPGA backend target. In our case, the backend target is implemented using a high level hardware description language, Bluespec [147](See Appendix B). The P4-to-FPGA compiler outputs Bluespec source code from a templated implementation, which is combined with a runtime framework (also written in Bluespec) to generate corresponding FPGA firmware. We describe our approach to compile P4 to FPGA target, embodied in P4FPGA, in more detail in Chapter 3. Then, we show a few prototypes of dataplane-accelerated applications built with the dataplane compiler in Section 3.5.

1.4.2 Lack of Programmable Network Dataplanes: Network Interface

Commodity network interfaces do not provide programmable access to the idle bits between packets. In other words, the PHY and Media Access Control (MAC) layer of the network protocol stack is opaque to system developers; they lack *programmability*. For example, an end host cannot use commodity network interfaces to *add* or *remove* arbitrary bits between packets, meaning it cannot accurately *pace* the packets, thus means the packet send rate cannot controlled [113]. Furthermore, the end host cannot precisely *count* the bits between adjacent packets using commodity network interfaces, which makes accurate timestamping difficult.

The challenge is in creating a programmable physical layer to provide access to physical layer bits while maintaining line rate performance. Specifically, we address the following question: *How to build a programmable PHY to access every bit in the physical layer while maintaining link speed line rates?*

To support a programmable physical layer, the approach must be able to access the PHY in software. Techniques should enable software to decide how many bits to insert or remove between packets. Furthermore, it is important to achieve real-time access, because the physical layer is always sending and receiving bits, usually at 10 gigabits per second and beyond. Finally, an approach must be able to scale to multiple line rate ports, such as 10 gigabit Ethernet ports to implement a network data plane for network measurement.

We explore a programmable PHY in order to grant users the flexibility to program every bit in the physical layer. As such, a user can implement an accurate network measurement application, create a covert timing channel, and even implement a distributed clock synchronization protocol. We investigate an approach to expose every bit in the physical layer of the network stack to software. As an instance of the approach, we present the design and implementation of SoNIC, a software network interface that implements the physical layer of the network protocol stack in software. We describe our methodology and prototype of SoNIC in more details in Chapter 4. Then, we present the design and implementation of a few prototypes, MinProbe, an available bandwidth estimator in Section 4.4, and DTP, a data center clock synchronization protocol in Section 4.5.

1.5 Contributions Towards a Programmable Network Dataplane

In this dissertation, we explore how to make every bit on a network dataplane programmable that contributes towards a programmable network dataplane.

First, **we explore a programmable packet processing pipeline for the network dataplane**. We show that designing packet processing pipelines on hardware can be

fast, and flexible (programmable). We built P4FPGA, a network dataplane compiler and runtime to generate a custom FPGA dataplane designed and built from the P4 language. We show that P4FPGA supports a variety of programmable dataplane implementations. For example, we created network routers, trading protocol parsers, a set of benchmark applications, and further highlight the benefit of dataplane programming with P4Paxos, an application that speeds up consensus protocol by offloading part of the protocol to the network dataplane.

Second, **we investigate a programmable physical layer for the network dataplane**. We show that a software programmable PHY can flexibly control every bit in the physical layer of the network, thus controlling inter-packet gaps. We designed and built SoNIC, a programmable PHY that runs at 10Gbps on a commodity server. SoNIC captures and controls the timing of physical layer bits with sub-nanosecond precision. With precise timestamping and pacing, we implemented an available bandwidth estimation algorithm, *MinProbe*. We further improved the resource-efficiency of the programmable network interface with a hardware-based programmable PHY. We then implemented a fourth primitive: modify, to embed information into inter-packet gaps. With the modify primitive, we implemented a clock synchronization protocol, Datacenter Time Protocol (DTP). DTP is a decentralized protocol that eliminates many non-deterministic factors from the network. As a result, it provides bounded precision to the tens of nanosecond in a data center network.

1.6 Organization

The rest of the dissertation is organized as follows: Chapter 2 describes the scope of the problem and the methodology for investigation. Chapter 3 describes the design of

a programmable dataplane and its prototype implementation, P4FPGA that provides a framework to compile a high-level network specification to a hardware implementation. Section 3.5 presents new network applications implemented with P4FPGA: P4Paxos, a market data analyzer, and a set of micro-benchmarks. Chapter 4 details the design of a programmable PHY and an implementation of the approach, SoNIC that provides precise control of every bit and thus precise packet timing control and measurement. Section 4.4 presents two novel applications enabled by a programmable PHY: available bandwidth estimation, and data center clock synchronization.

CHAPTER 2

SCOPE AND METHODOLOGY

This chapter describes the problem scope and methodology of this dissertation. First, we clarify the scope of the problems by reviewing the design and implementation of a network dataplane. Then, we describe our methodology for evaluating and validating our research contributions.

2.1 Scope: Understanding the Network Dataplane

A single network element consists of two planes: the control plane and the dataplane. The *control plane* configures the forwarding and packet processing behavior, while the *dataplane* forwards and processes packets based on configuration directives from the control plane. Network programming has several forms. First, manual configuration of network devices is a type of network programming. Network operators (e.g., network administrators) configure network devices (e.g., switches and routers) via a device-specific command line interface (CLI) to set up access control lists, routing policies and traffic load balancing. Second, software programs (e.g., network control programs written in C++ or Java) can automate the configuration process via a standard Application Programming Interface (API). Many of these programming tasks focus on the control plane and management plane tasks, which is referred to as *control plane programming*. Finally, modifying network forwarding behavior and introducing new dataplane functionality is another form of network programming. For example, the network dataplane could be programmed to offload distributed consensus protocols to the network [56]. The network dataplane could also be programmed to enable tightly synchronized time in data center networks [112]. This dissertation explores the last type of network pro-

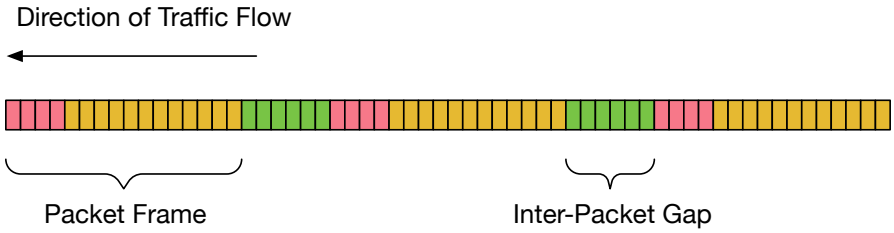


Figure 2.1: Different types of bits on dataplane, green color represents inter-packet gap, red color represents packet header, yellow color represents packet payload.

gramming – *dataplane* programming.

Dataplane programming is writing a program that can control of every single bit on a network dataplane. In particular, network dataplanes can be classified into two categories: the ability to program the bits that belong to a packet, i.e. packet headers and payload, and the ability to program bits that are between subsequent packets, i.e., the filling bits. (See Figure 2.1)

2.1.1 Dataplane Programming

Four basic primitives can be used to operate on the network dataplane. We can use these primitives to manipulate bits within packets and between packets. These four primitives are: *add*, *delete*, *count* and *modify*.

These basic primitives can be performed on packet bits, which are usually part of the dataplane programming language specification. The *add*, *delete* and *modify* primitives enable new protocols. When dealing with packets, a set of bits are operated on, instead of individual bits. The set of bits could represent a packet header or payload. For example, a tunneling protocol needs primitives to add or remove headers. A router needs primitives to update values in packet header fields, such as decreasing the TTL (Time-To-Live) fields in the IP (Internet Protocol) header. Stateful operations, such as

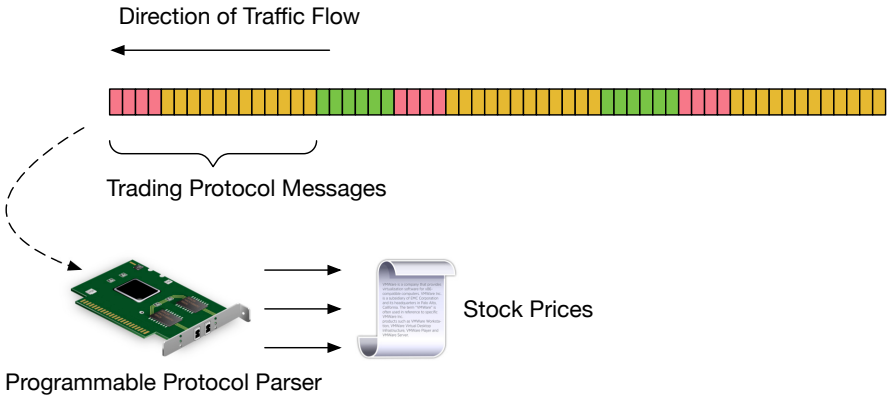


Figure 2.2: A programmable protocol parser that can parse financial trading protocol at high speed and low latency.

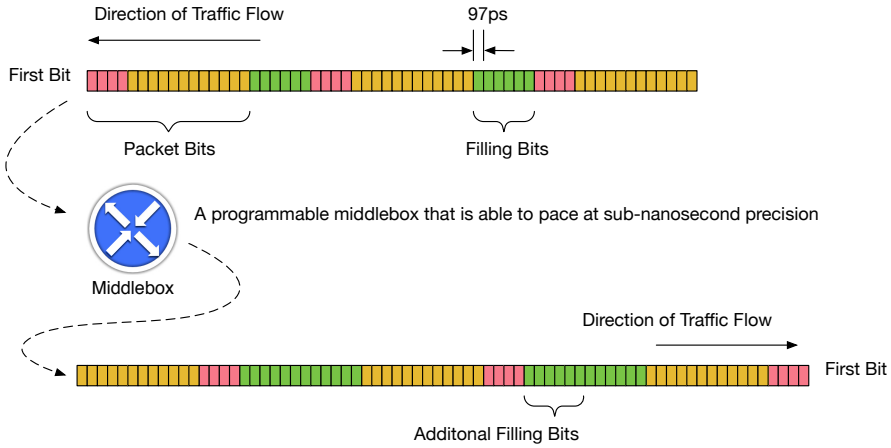


Figure 2.3: A programmable dataplane that can pace packets at sub-nanosecond precision.

count, can be used, to accumulate packet statistics, e.g., the number of bytes received and transmitted.

With the add and delete primitives, the generated packet rate can be controlled. Adding filling bits between packets will slow down the instantaneous packet rate. Removing filling bits will increase the instantaneous packet rate and creates a burst of packets. For example, a middlebox that supports adding/removing filling bit can precisely pace traffic at sub-nanosecond precision, as show in Figure 2.3. In this figure, each filling bit in 10 Gigabit Ethernet is about 97 picoseconds wide. The count primitive

enables the number of filling bits to be determined between packets. With the *count*, the time elapsed between packets can be computed and measured at sub-nanosecond precision. Furthermore, the *modify* primitive changes the values stored in the filling bits. With 'modify', a time synchronization protocol can be created with zero-overhead at the network layer.

Programmable dataplanes can be easily extended to support custom protocols. For example, Figure 2.4 shows a snippet of a description of trading protocol called Market Data Protocol (MDP) which is used by the Chicago Mercantile Exchange [190]. The implementation of MDP is complicated by the fact that the protocol header is variable length. Figure 2.4 shows the header definitions for a book refresh message. A *book* is an entity that keeps the most recent stock price. A book refresh message has a fixed header `mdp_t` that is common to all MDP protocol messages, as well as a variable length header, `refreshBook`, with one or more entries `refreshBookEntry`. A field `numEntries` in `refreshBook` dictates how many entries must be extraced by the parser.

2.1.2 Network Protocol Stack

Before we discussing about dataplane programming further, we will provide a brief overview of the network protocol stack, focusing on the aspects related to the packet bits and filling bits. A network protocol stack is an implementation of a computer networking protocol suite. An example is 10 Gigabit Ethernet protocol stack, as shown in Figure 2.5. A more detailed treatment of the network protocol stack can be found in Appendix B.

```
header_type mdp_t {
    fields {
        msgSeqNum : 32;
        sendingTime : 64;
        msgSize : 16;
        blockLength : 16;
        templateID : 16;
        schemaID : 16;
        version : 16;
    }
}

header_type event_metadata_t {
    fields {
        group_size : 16;
    }
}

header_type refreshBook {
    fields {
        transactTime : 64;
        matchEventIndicator : 16;
        blockLength: 16;
        numEntries: 16;
    }
}

header_type refreshBookEntry {
    fields {
        mdEntryPx : 64;
        mdEntrySize : 32;
        securityID : 32;
        rptReq : 32;
        numberOfOrders : 32;
        mdPriceLevel : 8;
        mdUpdateAction : 8;
        mdEntryType : 8;
        padding : 40;
    }
}
```

Figure 2.4: Header definitions for MDP.p4.

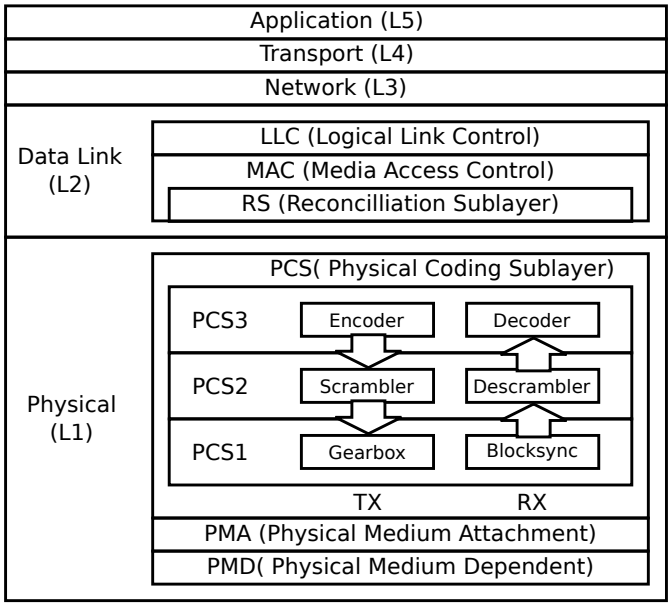


Figure 2.5: IEEE 802.3 10 Gigabit Ethernet Network stack.

Packet Bits The packet bits exist in the data link layer and above (Layer 2 to 5; L2 to L5). We discuss packet bits in the context of the most common protocol on the Internet – the Transmission Control Protocol/Internet Protocol (TCP/IP protocol stack) . TCP/IP works as follows: to send a chunk of data between applications on separate networked systems, the application layer (L5 in Figure 2.5) makes sure that the data is sent in a format that will be understandable by the recipient. Then, the transport layer (TCP) splits the application data into chunks that can fit into the maximum data packet size. It then attaches a sequence number to each packet, specifying packet order. The sequence number allows the recipient to re-assemble the data correctly at the other end. Next, the network layer (IP) attaches the IP address of the sender and the receiver to the packet, so that the network can route the messages to the destination. Finally, the link layer attaches a MAC (Media-Access-Control) address of the sender and the recipient, allowing the packets to be directed to a specific network interface on the host machine.

Filling Bits The filling bits exist in the Physical Layer (PHY) of the 10GbE protocol stack. PHY of 10 GbE consists of three sublayers: the Physical Coding Sublayer (PCS), the Physical Medium Attachment (PMA) sublayer, and the Physical Medium Dependent (PMD) sublayer. The PMA and PMD sublayers are responsible for clock recovery and (de-)serializing the bitstream. The PCS performs the blocksync and gearbox (we call this PCS1), scramble/descramble (PCS2), and encode/decode (PCS3) operations on every Ethernet frame. The filling bits (also called the idle characters) are special control characters that are used to fill any gaps between two Ethernet frames.

Network Interface The *network interface* implements the physical (PHY) and media access (MAC) layers of the network stack. On the receiving path, the network interface converts the bitstream to packets, which on the transmitting path, it does the reverse. A network interface can be a network interface card on commodity servers. It can also be part of the silicon chip (e.g., FPGA) that implements PHY and MAC.

Packet Processing Pipeline Packet processing refers to the wide variety of algorithms that are applied to a packet of data or information as it moves through the various network elements of a communications network. The *packet processing pipeline* forwards, drops and/or modifies packets. Consider a packet processing pipeline inside an Internet Procotol (IP) router. The pipeline receives an IP packet from one of its network interfaces. First it checks a routing table to determine the next hop interface to send the packet to. Next, the pipeline processes and modifies the packet by updating its Time-To-Live (TTL) field and checksum fields. Finally, the pipeline sends the packet to the next router via an outgoing network interface.

	2008	2012	2016
Model	Xeon X5492	Xeon E5-4650	Xeon E7-8890
Core Count	4	8	24
Clock Frequency	*3.4 GHz*	*2.8 GHz*	*2.2 GHz*
Process technology	45nm	32nm	14nm
L2 Cache	-	8 x 256 KB	24 x 256KB
L3 Cache	12MB	20MB	60MB
DRAM Frequency	1600MT/s	4x1600MT/s	4x1866MT/s
Release price	$1279	$3620	$7174

Table 2.1: Dell server specifications from year 2008 to 2016, Xeon CPUs in 2008 does not have per-core L2 Cache, instead it uses per-socket L3 Cache.

2.1.3 Fast and Flexible Packet Processors

Networks would benefit from fast and flexible packet processors. If a packet processor can process custom *header bits*, it would simplify the design and deployment of new network protocols. Similarly, if a packet processor can handle custom *payload bits*, critical network functions, such as packet classification and consensus protocol, can be offloaded to network dataplane. At the same time, a packet processor must be fast. For example, data center network bandwidth has been growing steadily: 10Gbps Ethernet is prevalent, 25Gbps is gaining traction, and 100 Gbps is on the horizon as of the time of writing this dissertation. Handling packet forwarding at line rate while performing complex packet processing requires significant computation power and programmability. Unfortunately, none of the existing network dataplanes can achieve both flexibility and performance at the same time.

We surveyed three dataplane implementation technologies from 2008, 2012, and 2016, to understand how different types of network dataplanes have evolved and why existing dataplanes lack flexibility or performance; that is, network dataplanes are either flexible (programmable) or performant, but not both.

Year	**2008**	**2012**	**2016**
Series	Trident+	Trident II	Tomahawk
Model	BCM56820	BCM56850	BCM56960
Process Technology	40 nm	40 nm	28 nm
Transceivers	24 x 10Gbps	128 x 10Gbps (32 x 40Gbps)	128 x 25Gbps (32 x 100Gbps)
Forwarding Cores	1	1	4
Buffer Size	9MB	12MB	16MB
Latency	≈500us	≈500ns	≈300ns
Feature Set			

Table 2.2: Broadcom Trident-series switch ASIC specification from year 2008 to 2016

Software Packet Processors Software packet processors are flexible but not fast. Software packet processors are software programs that are written in high level programming language, such as C or C++, and executed on general purpose CPUs. A 25Gbps network interface can receive a minimum sized (64B) packet every 19.2ns. However, at this speed, even a single access to a last level cache would take longer than the arrival time of a packet. Processing packets in a software dataplane is challenging, even when using all the advanced software techniques, such as kernel bypass, receive side scaling, and data direct I/O [62]. Worse, CPU performance is unlikely to improve because of stalled frequency scaling [175]. Table 1 summarizes three examples of CPUs for building software dataplanes in the years 2008, 2012, and 2016, which compare the CPU frequency, total number of cores, fabrication process and memory bandwidth. The "Core Count" row of the table shows that the total number of CPU cores has increased from 4 to 24 in the year 2016, whereas Clock frequencies have decreased from 3.4GHz to 2.2GHz. If we use clock frequency to approximate single thread performance, the performance has not improved during the year between 2008 and 2016. As network link speeds approach 25Gbps or 100Gbps, software network dataplanes put a lot strain on server computation and memory capabilities and become impractical.

	2008	2012	2016
Family	Virtex-6	Virtex-7	Virtex Ultrascale+
Model	XCE6VHX565T	XC7VH870T	VU37P
Silicon Process	40 nm	28 nm	20 nm
SerDes	48 x 6.6Gbps (24 x 11.2Gbps)	72 x 13.1Gbps (16 x 28.05Gbps)	96 x 32.75Gbps
Logic Cells	566K	876K	2,852K
Flip-Flops	708K	1,095K	2,607K
Block RAM	32,832Kb	50,760Kb	70,900Kb
UltraRAM	-	-	270.0Mb

Table 2.3: Xilinx Virtex-series FPGA specification from year 2008 to 2016

ASIC-based Packet Processors ASIC-based packet processors are fast, but not flexible. These packet processors are often used in modern switches and routers [146], and tend to handle a limited set of protocols. For example, Table 2.2 compares three generations of Broadcom switch ASICs, the dominant device used for switch development, from year 2008 to 2016 [74]. The *Transceivers* row shows that the number of ports and per-port link speed have scaled substantially during that period. However, the feature set provided by the switch ASICs remained largely constant. Consequently, despite the performance gain, ASICs are still sub-optimal in terms of providing a programmable dataplane, because it is difficult to fulfill application requirements that are not directly supported.

FPGA-based Packet Processors Packet processors implemented in FPGAs balance hardware performance and software flexibility. For example, the NetFPGA project [128] has been used to prototype shortest path routing and congestion control algorithms [65]. The designs are expressed using a hardware description language, such as Verilog, and synthesized to FPGAs using vendor-specific place-and-route tools. (See Appendix D) Table 2.3 shows three generations of Virtex-series FPGAs from Xilinx in the years 2008,

2012 and 2016. We selected the top-of-line FPGA model for each generation. The process technology used by Xilinx in each generation is similar to the fabrication process employed by Broadcom in the same period [24]. During the period, the data rate of transceivers on Virtex FPGA has increased by 3x, (from 11.2Gbps in Virtex-6 to 32.75Gbps in Virtex UltraScale+), and the total number of transceivers has increased slightly from 72 in Virtex-6 to 96 in UltraScale+, with limited scaling due to packaging limitations. The amount of logic and memory resources on FPGA has increased by 2x and 5x, respectively. Overall, FPGAs offer competitive flexibility to software due to their reconfigurability; and they provides comparable performance to ASICs.

The challenge in FPGA-based design is that the packet processing pipelines are often hard-coded with packet processing algorithms. Implementing a new packet processing algorithm requires users to program in low-level hardware description languages, which is a difficult and error-prone process.

In this dissertation, we explore a different approach to compiling network dataplane programs written in high level language to hardware. We prototype the approach on an FPGA. The same methodology is applicable to ASICs as well, if they provide a programmable abstraction in their design [36]. In this dissertation, we propose an approach to allow users to program in a high-level declarative language. In particular, the users can describe how packets are parsed, processed and re-assembled within the pipeline using a language called P4 [38]. We demonstrate that a compiler can map the high-level description to a template implementation on FPGAs to instantiate the dataplane implementation. Further, our approach is portable and the template implementation can be synthesized to both Altera [20] and Xilinx [24] FPGAs.

In summary, there is an opportunity to leverage high level programming language to generate the network dataplane that is both high performance and flexible. We design a

system to compile network dataplane programs to FPGA devices in Chapter 3.

2.1.4 Precise Packet Timing Control with a Programmable PHY

The potential rewards of opening the inner workings of the PHY provides the ability to study networks and the network stack at a heretofore inaccessible level, by improving the precision of network measurements by orders of magnitude [67]. A 10 GbE network uses one bit per symbol. Since a 10 GbE link operates at 10.3125 Gbaud, each and every symbol length is 97 pico-seconds wide ($= 1/(10.3125 * 10^9)$). Knowing the number of bits can then translate into having a precise measure of time at the sub-nanosecond granularity. Therefore, on the receive path, we can tell the exact distance between Ethernet frames in bits by counting *every bit*. On the transmit path, we can control time gaps by controlling the number of bits (`idle` characters) between frames.

Unfortunately, current commodity network interface cards do not provide any Application Program Interfaces (API) for accessing and controlling every bit in the PHY. One way of doing these interfaces is to use physics equipment such as an oscillator for capturing signals and a laser modulator for transmitting signals as shown in BiFocals [67]. However, BiFocals is not a real time tool: It can only transmit pre-generated symbols and must perform extensive offline computation to recover symbols from captured signals by the oscillator. As a result, we take a different approach to access and control bits in the PHY: Implementing the PHY in software.

The fundamental challenge to perform the PHY functionality in software is maintaining synchronization with hardware while efficiently using system resources. Some important areas of consideration when addressing this challenge include *hardware support, real time capability, scalability and efficiency, and a usable interface*.

Hardware support The hardware must be able to transfer raw symbols from the wire to software at high speeds. This requirement can be broken down into four parts: a) Converting optical signals to digital signals (PMD), b) Clock recovery for bit detection (PMA), and c) Transferring large amounts of bits to software through a high-bandwidth interface. Additionally, d) the hardware should leave recovered bits (both control and data characters in the PHY) intact until they are transferred and consumed by the software. Commercial optical transceivers are available to convert optical signals to digital signals, but hardware that simultaneously satisfies the remaining three requirements is not common since it is difficult to handle 10.3125 Giga symbols in transit every second.

NetFPGA 10G [128] does not provide software access to the PHY. In particular, NetFPGA pushes not only layers 1-2, the physical and data link layers into hardware, but potentially layer 3 as well. Furthermore, it is not possible to easily undo this design since it uses an on-board chip to implement the PHY, which prevents direct access to the PCS sublayer. As a result, we need a new hardware platform to support software access to the PHY.

Real time Capability Both hardware and software must be able to process 10.3125 Gbps continuously. The IEEE 802.3 standard [13] requires the 10 GbE PHY to generate a continuous bitstream. However, synchronization between hardware and software and between multiple pipelined cores is non-trivial. The overheads of interrupt handlers and OS schedulers can cause a discontinuous bitstream which can subsequently incur packet loss and broken links. Moreover, it is difficult to parallelize the PCS sublayer onto multiple cores, because the (de-)scrambler relies on the state of the previous 59 bits to recover bits. This fine-grained dependency makes it hard to parallelize the PCS sublayer. The key takeaway here is that everything must be efficiently pipelined and well-optimized in order to implement the PHY in software while minimizing synchronization overheads.

Scalability and Efficiency The software must scale to process multiple 10 GbE bitstreams while efficiently utilizing resources. Intense computation is required to implement the PHY and MAC layers in software. (De-)Scrambling every bit and computing the CRC value of an Ethernet frame is especially intensive. A functional solution would require multiple duplex channels that independently perform the CRC, encode/decode, and scramble/descramble computations at 10.3125 Gbps. The building blocks for the PCS and MAC layers will therefore consume many CPU cores. In order to achieve a scalable system that can handle multiple 10 GbE bitstreams, resources such as the PCIe, memory bus, Quick Path Interconnect (QPI), cache, CPU cores, and memory must be efficiently utilized.

User Interface Users must be able to easily access and control the PHY. Many resources from software to hardware must be tightly coupled to allow realtime access to the PHY. Thus, an interface that allows fine-grained control over them is necessary. The interface must also implement an I/O channel through which users can retrieve data such as the count of bits for precise timing information.

2.2 Methodology

2.2.1 Systems

To explore the feasibility of programmable dataplanes, we have developed three prototypes: a hardware-based packet processing pipeline, a software-based programmable PHY and a hardware-based programmable PHY. These three prototypes represent different points in the design space for programmable dataplane. The software-programmable

PHY takes a software-oriented approach to favor software flexibility over hardware performance. Hence, the focus of the work is to optimize software performance, while preserving flexibility. The hardware-programmable PHY and packet processing pipeline takes the opposite approach. In both cases, the method favors hardware performance over software flexibility. Hence, the focus of the work to improve software programmability while preserving the performance gain from hardware. Below, we describe each prototypes in more details.

Hardware Packet Processing Pipeline The first prototype is a programmable packet processing pipeline and a dataplane compiler. The target throughput is 4x10Gbps line rate. The prototype implementation consists of a C++-based compiler along with a runtime system written in hardware description language. For the frontend, the P4.org's C++ compiler frontend is used to parse P4 source code and generate an intermediate representation [154]. The custom backend for FPGAs consists of 5000 lines of C++ code. The runtime is developed in a high-level hardware description language Bluespec [147]. Bluespec provides many of the higher level hardware abstractions (e.g., FIFO with back-pressure) and the language includes a rich library of components, which makes development easier. The runtime is approximately 10,000 lines of Bluespec. The Connectal Project [96] is used to implement the control plane channel, and mechanisms to replay `pcap` traces, access control registers, and program dataplane tables.

Software Programmable PHY Our second prototype of a programmable PHY targets at the 10Gbps line rate. The prototype includes an FPGA card, which performs the DMA (Direct Memory Access) transfers between network transceivers and host memory, and vice versa. The prototype also consists of a software component which implements the entire physical layer of the network protocol stack. The software is developed

and evaluated on Dell Precision T7500 workstations and Dell T710 servers. Both machines have dual socket, 2.93 GHz six core Xeon X5670 (Westmere [15]) with 12 MB of shared L3 cache and 12 GB of RAM, 6 GB on each CPU socket. This prototype is used to evaluate both SoNIC and MinProbe , discussed in Chapter 4 and 4.4.

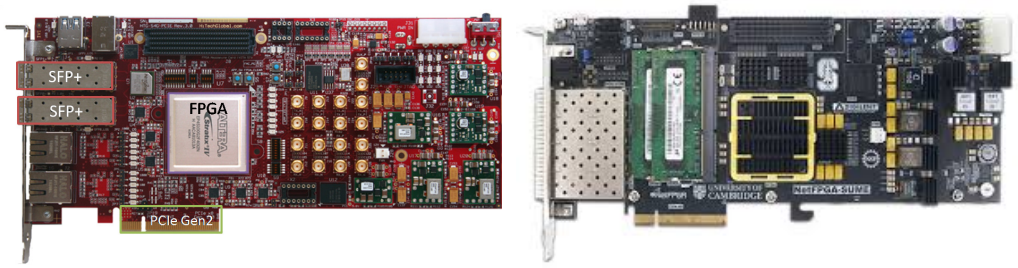
Hardware Programmable PHY The third prototype is built as a performance-enhanced version of the second prototype. In particular, the target throughput is 4x10Gbps line rate, which poses a scalability challenge for the software-based approach. As a consequence, the third prototype implements a programmable PHY in hardware. The prototype is built using an Altera DE5 board, with a Stratix V FPGA. The implementation includes the entire 10GbE physical layer designed in the Bluespec programming language [147], and also extends the physical layer to perform the *modify* operation to the value in idle characters, which is used to implement a zero-cost time synchronization protocol.

2.2.2 Evaluation

We used various types of hardware networks, and network topologies throughout this dissertation to evaluate our systems. We illustrate them in the following subsections.

National Lambda Rail

SoNIC To evaluate SoNIC, we connected the SoNIC board and the ALT10G board directly via fiber optics (Figure 2.8a). ALT10G allows us to generate random packets of any length and with the minimum inter-packet gap to SoNIC. ALT10G also provides us with detailed statistics such as the number of valid/invalid Ethernet frames, and



(a) HiTech Global FPGA board (b) NetFPGA-SUME board

Figure 2.6: FPGA development boards used for our research.

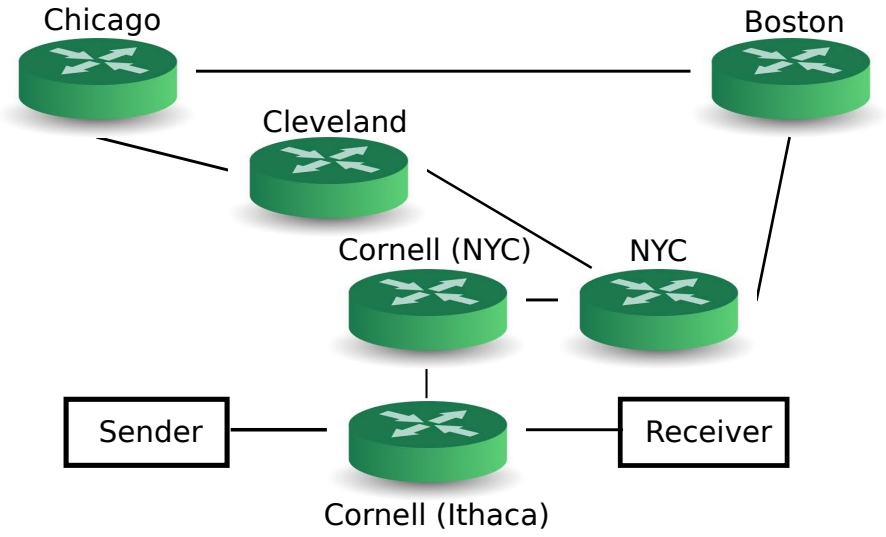
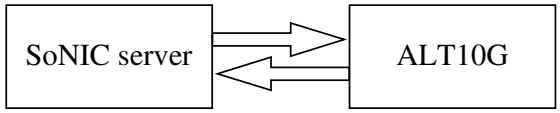


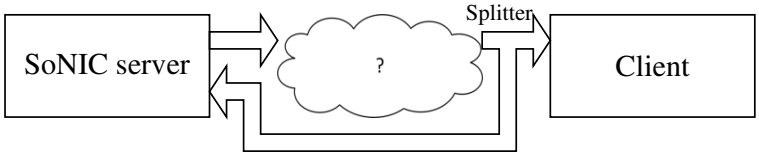
Figure 2.7: Our path on the National Lambda Rail.

frames with CRC errors. We compared these numbers from ALT10G with statistics from SoNIC to verify the correctness of SoNIC .

Further, we created a simple topology to evaluate the SoNIC: We used port 0 of the SoNIC server to generate packets to the Client server via an arbitrary network, and split the signal with a fiber optic splitter so that the same stream can be directed to both the Client and port 1 of the SoNIC server capturing packets (Figure 2.8b). We used various network topologies composed of a Cisco 4948 and IBM BNT G8264 switches for the network between the SoNIC server and the Client.



(a) Evaluation setup for SoNIC



(b) Simple Topology for evaluating SoNIC

Figure 2.8: Simple evaluation setup.

MinProbe National Lambda Rail (NLR) was a wide-area network designed for research and has significant cross traffic [148]. We set up a path from Cornell university to NLR over nine routing hops and 2500 miles one-way (Figure 2.7). All the routers in NLR are Cisco 6500 routers. The average round trip time of the path was 67.6 ms, and there was always cross traffic. In particular, many links on our path were utilized with 1~4 Gbps cross traffic during the experiment. Cross traffic was not under our control, however we received regular measurements of traffic on the external interfaces of all routers.

P4FPGA We evaluate the performance of P4FPGA generated designs against a set of representative P4 programs. Each program in our benchmark suite is compiled with the P4FPGA compiler into Bluespec source code, which is then processed by a commercial compiler from Bluespec Inc. to generate Verilog source code. Next, the Verilog source code is processed by the standard Vivado 2015.4 tool from Xilinx, which performs synthesis, placement, routing and bitstream generation. All of the above steps are automated with a `Makefile`. The compilation framework supports both the Altera tool suite, Quartus, and Xilinx tool suite, Vivado. For this evaluation, we only used Vivado. We deployed the compiled bitstream on a NetFPGA SUME platform with a

Xilinx Virtex-7 XC7V690T FPGA, with 32 high-speed serial transceivers to provide PCIe (Gen3 x8) communication and 4 SFP+ ports (10Gbps Ethernet).

P4Paxos We ran our end-to-end experiments on a testbed with four Supermicro 6018U-TRTP+ servers and a Pica8 P-3922 10G Ethernet switch, connected in the topology shown in Figure 6. The servers have dual-socket Intel Xeon E5-2603 CPUs, with a total of 12 cores running at 1.6GHz, 16GB of 1600MHz DDR4 memory and two Intel 82599 10 Gbps NICs. We installed one NetFPGA SUME [62] board in each server through a PCIe x8 slot, though NetFGPAs behave as stand-alone systems in our testbed. Two of the four SFP+ interfaces of the NetFPGA SUME board and one of the four SFP+ interfaces provided by the 10G NICs are connected to Pica8 switch with a total of 12 SFP+ copper cables. The servers were running Ubuntu 14.04 with Linux kernel version 3.19.0.

2.3 Summary

Modern networks have the challenge to provide fast and flexible (programmable) network dataplanes. First, the network dataplane needs to leverage a high level dataplane programming language, a dataplane compiler, and a packet processing pipeline to enable automatic generation of a packet processor. Second, the network interface needs to be programmed to enable software access to the filling bits or the inter-packet gaps on the network dataplane to enable precise packet pacing and timestamping. In the remainder of this dissertation, we describe how we investigate the research questions and present the system designs, implementations, and evaluations that realize our approach and validate our findings.

CHAPTER 3

TOWARDS A PROGRAMMABLE NETWORK DATAPLANE: P4FPGA AND PROGRAMMABLE PACKET PROCESSING PIPELINES

This work appears in *P4FPGA : A Rapid Prototyping Framework for P4* in SOSR 2017 with co-authors Robert Soule, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon.

To provide a flexible and programmable network dataplane, the developers must be able to specify the behavior of network dataplane in high level programming languages. Further, developers need access to platforms that can execute their designs efficiently in hardware. In this respect, Field Programmable Gate Arrays (FPGAs) are an promising target platform. As a form of reprogrammable silicon, FPGAs offer the flexibility of software and the performance of hardware. Indeed, major cloud providers, such as Microsoft and Baidu, already deploy FPGAs in their data centers to boost performance by accelerating network encryption and decryption or implement custom transport layers.

Designing a compiler using a domain specific programming language for FPGAs presents several challenges: FPGAs are typically programmed using low-level libraries that are not portable across devices. Moreover, communication between 3rd party processing elements is device-specific, adding an additional hurdle to portability; Generating an efficient implementation of a source program is difficult, since programs vary widely and architectures make different tradeoffs; Although P4 is target agnostic, it relies on a number of extern functions for critical functionality, such as checksums and encryption, complicating code generation.

In this chapter, we present P4FPGA, an open source P4-to-FPGA compiler and runtime that is designed to be flexible, efficient, and portable. To ensure that P4FPGA is

flexible enough to implement diverse network functions, the compiler allows users to incorporate arbitrary hardware modules written in the language of their choice. This approach offers a degree of flexibility that would be difficult to achieve on other targets, such as a switch ASIC. To ensure that the code generated by the compiler is efficient, P4FPGA supports both bump-in-the-wire [117] and multi-port switching architectures [78]. This design allows users to select the best design for their particular application. Finally, to ensure that programs are portable across different devices, P4FPGA provides a runtime with device-agnostic hardware abstractions. This runtime allows P4FPGA to support designs that can be synthesized to either Xilinx or Altera FPGAs.

We have evaluated our prototype implementation on a variety of representative P4 programs. Our experiments show that code generated by P4FPGA runs at line-rate throughput on all packet sizes up to MTU, and exhibits latency similar to off-the-self commodity switch ASICs. Moreover, P4FPGA is already being used by at least two research projects [56, 73] to deploy P4 programs on actual hardware.

Overall, this chapter makes the following contributions: First, we present the design and implementation of the first open-source P4-to-FPGA compiler and runtime system. Second, we evaluate the performance of the generated backend on a variety of non-trivial P4 programs and demonstrate that system performance is competitive with commercial switches—e.g., latencies are comparable to commercial cut-through switches. Last, we develop a wide variety of standard and emerging network applications using P4FPGA,

which demonstrates that the tool is broadly applicable.

3.1 Background

Before presenting the details of the design of P4FPGA, we briefly present a high-level overview of the P4 language [39], and networking processing on FPGAs.

P4 Background When describing a P4 compiler, it is important to clarify some terminology. A *target* is hardware that is capable of running a P4 program, such as FPGAs, ASICs, or CPUs. An *architecture* refers to the combination of the runtime as well as the processing pipeline specified in a P4 program. A P4 program is *target independent*, meaning the same program can be implemented on different hardware. However, a P4 program can be implemented on a target with different architectures. For example, a bump-in-the-wire architecture differs from a switching architecture in that there is no crossbar in the runtime, resulting in different performance characteristics. A P4 compiler is responsible for mapping the abstract architecture specified by the P4 program to a particular target and architecture. FPGAs are uniquely able to offer hardware-level performance for all P4 programs. In contrast, a single switch ASIC will only be able to faithfully implement a small subset of potential architectures and extern primitives. P4FPGA supports both bump-in-the-wire [117] and multi-port switching architectures [78], which are suitable for network functions and routing, respectively.

As a language, P4 allows developers to specify how packets are processed in the data plane of network forwarding elements. P4 programs are written against an *abstract architecture* that hides the actual physical implementation. In the abstract architecture, packets are first parsed, and then processed by a sequence of match-action tables. Each table matches on specified packet header fields, and then performs a sequence of actions

to modify, forward, or drop the packet. Additionally, the program collects packet metadata, such as ingress and egress port numbers, which flows through the pipeline. At the end of the pipeline, packets are reassembled for transmission in a deparser stage. Figure 3.2 illustrates a simple example with a single parser and match table pipeline. The P4 language provides syntax that mirrors this abstract model. For brevity, we do not describe the syntax in detail as the full language specification is available online [155]. We simply mention that programmers can declare packet headers, compose tables, and specify actions. Tables can be populated at runtime with flow rules via a control plane API.

The example code in Figure 3.1 shows a subset of a P4 program for counting UDP packets by destination port. Line 4 defines the layout for the UDP packet headers. Line 12 declares an instance of that header, named `udp`. Line 14 defines how to parse UDP packet headers. The `extract` keyword assigns values to the fields in the header instance. The `return` keyword returns the next parser stage, which could be `ingress`, indicating the start of the pipeline. Line 32 is the start of the flow control for the P4 program. It checks if the arriving packet is an Ethernet packet, then if it is an IPv4 packet, and finally if it is a UDP packet. If so, it passes the packet to the `table_count` table, defined on Line 28. The `table_count` table `reads` the destination port, and performs one of two possible actions: `count_c1` or `_drop` a packet. The action `count_c1` on Line 24 invokes a `count` function. The `count` function must be defined externally to the P4 program.

FPGA Background FPGAs are widely used to implement network appliances [7,53] and accelerators [29], as applications executing on FPGAs typically achieve higher throughput, lower latency, and reduced power consumption compared to implementations with general-purpose CPUs. For example, Microsoft has implemented a

```
1 // We have elided eth and ipv4 headers,
2 // and the extern declarations for brevity
3
4 header_type udp_t {
5     fields {
6         srcPort      : 16;
7         dstPort      : 16;
8         length        : 16;
9         checksum : 16;
10    }
11 }
12 header udp_t udp;
13
14 parser parse_udp {
15     extract(udp);
16     return ingress;
17 }
18
19 counter c1 {
20     type: packet;
21     numPackets : 32;
22 }
23
24 action count_c1() {
25     count(c1, 1);
26 }
27
28 table table_count {
29     reads   { udp.dstPort : exact; }
30     actions { count_c1; _drop; } }
31
32 control ingress {
33     if (valid(eth)) {
34       if (valid(ipv4)) {
35           if (valid(udp)) {
36             apply(table_count);
37           }
38       }
39     }
40 }
```

Figure 3.1: Subset of a P4 program to count UDP packets.

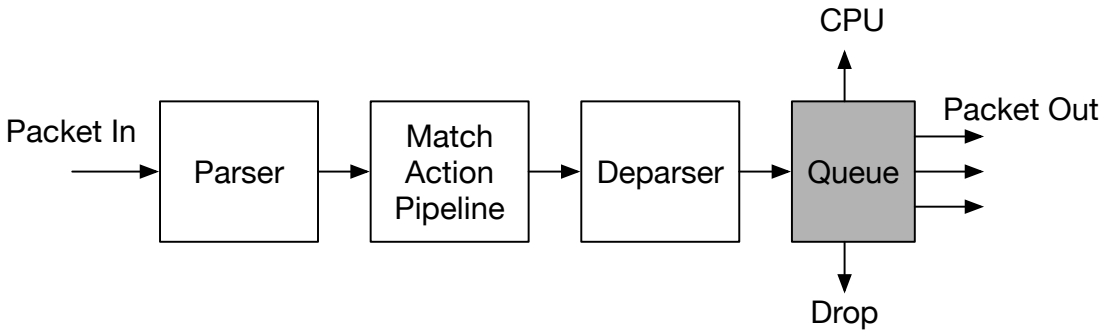


Figure 3.2: Example P4 Abstract Architecture.

lightweight UDP-based transport protocol in FPGAs to react quickly to congestion notifications and back off quickly to reduce packets that are dropped from incast patterns [172].

Development on an FPGA typically involves using a low-level hardware description languages (i.e., Verilog, HHDL) to statically specify a hardware circuit for a single application [211]. However, these languages are widely regarded as difficult to use, and consequently, there has been significant research in high-level synthesis [27, 44, 180].

P4FPGA uses one of these languages, Bluespec System Verilog [147], as both the target for compiler generation, and to implement the runtime. Bluespec is a strongly typed functional language, similar in many respects to Haskell. Users write hardware operations as guarded rules [147], where each rule consists of a set of atomic actions that all execute sequentially in one clock cycle. The language includes a large set of libraries for common hardware constructs such as registers, FIFO queues and state machines. Moreover, P4FPA uses Bluespec code from the Connectal project [96] to implement the control plane channel. Using Bluespec simplifies FPGA development by providing high-level language constructs, and is more expressive than Verilog.

P4FPGA Overview A compiler for P4 is responsible for two main tasks: generating the configuration to implement a data plane on a target platform at compile time and generating an application programming interface (API) to populate tables and other programmable elements at run time.

Figure 3.3 presents a high-level overview of the P4FPGA framework and compilation strategy. The components inside the dashed line were developed specifically for P4FPGA. The components outside the dashed line are existing open-source tools or commercial products for FPGA synthesis that are reused by P4FPGA.

P4FPGA builds on the reference P4 compiler implementation provided by the P4 organization [158]. The reference compiler parses P4 source code, and produces a standard Intermediate Representation (IR) [158]. We chose to build on the reference front end for practical reasons. It both reduces the required engineering effort, and ensures that FPGA conforms to the latest P4 syntax standards.

P4FPGA includes three main components: a code generator; a runtime system; and finally optimizers implemented as IR-to-IR transformers. The code generator produces a packet-processing pipeline inspired by the model proposed by Bosshart et al. [40] for implementing reconfigurable packet processing on switching chips. The runtime provides hardware-independent abstractions for basic functionality including memory management, transceiver management, host/control plane communication. Moreover, it specifies the layout of the packet processing pipeline (e.g. for full packet switching, or to support network function virtualization (NFV). The optimizers leveraging hardware parallelism to increase throughput and reduce latency.

The compiler produces Bluespec code as output. Bluespec is a high-level hardware description language that is synthesizable to Verilog. The Verilog code is further com-

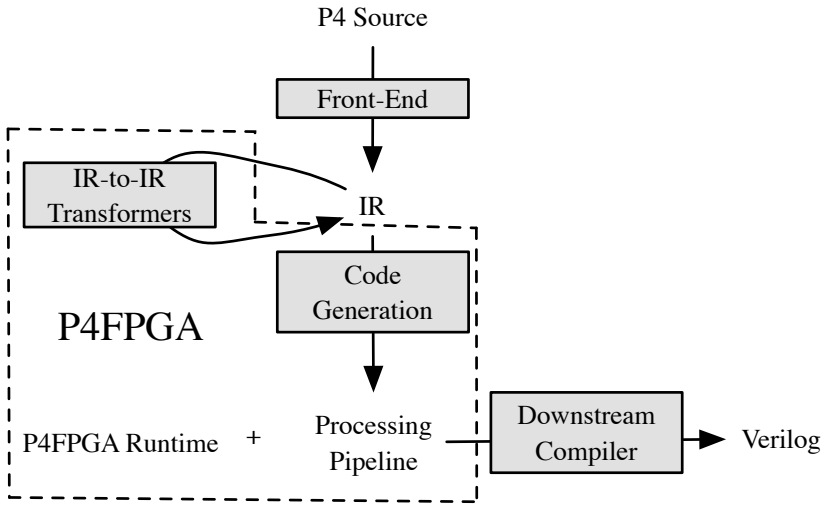


Figure 3.3: P4FPGA Framework Overview.

piled by downstream tool chains such as Xilinx's Vivado or Altera's Quartus. The output bitstream can then be used to configure an FPGA.

In the following sections, we present the code generator, runtime system, and optimizers in full detail.

3.2 Design

The core job of the P4FPGA compiler is to map logical packet-processing constructs expressed in P4 into physical packet-processing constructs expressed in a hardware description language. We organize the generated physical constructs into *basic blocks*. As in most standard compilers, a basic block is a sequence of instructions (e.g., table lookups, packet-manipulation primitives, etc.). We implement basic blocks in P4FPGA using parameterized templates. When instantiated, the templates are hardware modules that realize the logic for packet parsers, tables, actions, and deparsers.

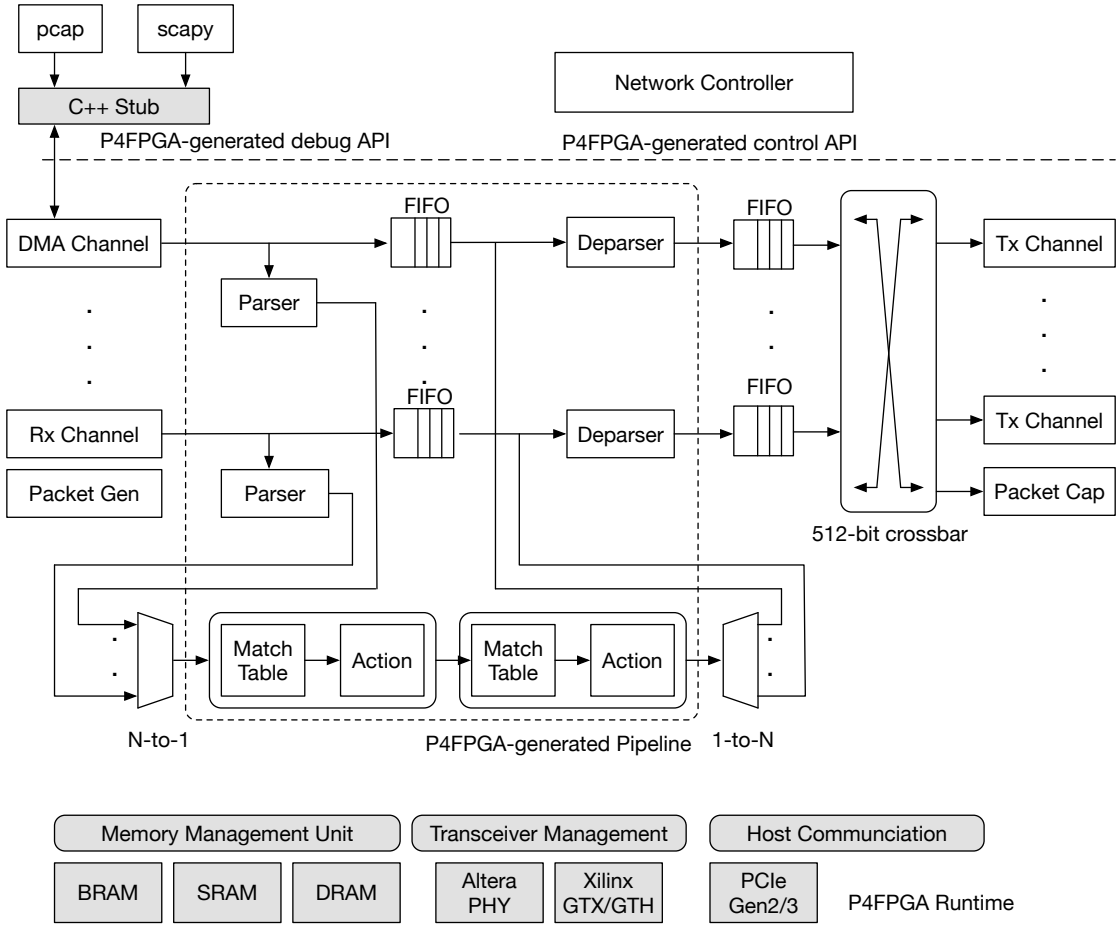


Figure 3.4: P4FPGA Runtime and Pipeline.

There are two motivations behind our use of basic blocks. First, it reduces the complexity of the compiler, since code generation simply becomes the composition of modules that implement standard interfaces. Second, the modular design enables extensibility in two ways: Programmers can easily add externally-defined functionality via a foreign-function interface (e.g., to implement a custom hash function); Programmers can modify the compiler by replacing one basic block with another that implements the same functionality (e.g., to modify the memory storage to use DRAM, SRAM, or an SSD).

The control flow constructs from the P4 source program dictate the composition of the basic blocks. We refer to this composition of blocks as the *programmable packet-*

processing pipeline. This is in contrast to the fixed-function pipeline that is realized by the P4FPGA runtime system. In other words, the programmable packet-processing pipeline is specified by the logic of a particular P4 source program, whereas the fixed-function pipeline is determined by the target platform, and is fixed for all input source programs.

3.2.1 Programmable Pipeline

The programmable packet-processing pipeline realizes the programmable logic of a P4 source program on an FPGA hardware implementation. It consists of a composition of basic blocks to parse, deparse, match, or perform an action.

Parsing Parsing is the process of identifying headers and extracting relevant fields for processing by subsequent stages of the device. Abstractly, the process of parsing can be expressed as a finite state machine (FSM) comprising a set of states and transitions. From a given state, the FSM transitions to the next state based on a given input from a header or metadata. A subset of the states identifies and extracts header fields. The FSM graphs may be acyclic, as is the case with Ethernet/IPv4 parsers, or cyclic intended to parse TCP options.

In general, there are two approaches to parsing: store-and-extract, and streaming. With store-and-extract [102], the entire packet must be buffered before parsing, resulting in higher latency. In contrast, P4FPGA adopts a streaming approach in which the packet byte stream is fed into the FSM and processed as soon as there is enough data to extract a header or execute a FSM transition.

The implementation of the parser basic block includes code that is common to all

parser instances and generated code that is customized for each specific parser. The common code includes state variables (e.g., header buffer, parse state, and offset), and a circuit that manages incoming bytes. The generated portion of a parser implements the application-specific FSM.

Deparsing As shown in Figure 3.3, there are two sources of input data to the deparser stage. One is the packet data stored in memory (§3.2.2), and one is the modified packet header processed by the programmable pipeline. The deparser reassembles the packet for transmission from these two input sources.

Like the parser, the deparser is implemented as a FSM. However, the design of deparser is more complicated, since it may add or remove headers during packet assembly.

The deparser consists of three modules: packet extender, packet merger, and packet compressor. The packet extender supports the addition of headers by inserting empty bytes at a designated offset. The packet merger writes modified packet fields, including fields added by the extender module. The packet compressor marks bytes to be removed by writing to a bit mask.

Note that the deparsing stage is responsible for modifying packets. Packet modification could be performed inline one-by-one (i.e., after every table), or all together at the end of the pipeline. P4FPGA takes the latter approach, which reduces latency. In other words, the pipeline modifies a copy of the header, and changes are merged with the actual header in the deparser stage.

Matching In P4FPGA, basic blocks for tables are implemented as FPGA hardware modules that support get/put operations via a streaming interface. P4 allows users to specify the algorithm used to match packets. Our P4FPGA prototype supports two

matching algorithms: ternary and exact-match. The ternary match uses a third-party library. We implemented two versions of exact match ourselves, one using a fully-associative content addressable memory (CAM) [26], and the other using a hash-based lookup table. Users can choose the implementation strategy by using a command line option when invoking the compiler. Note that because of the "programming with tables" abstraction that P4 provides, some programs include tables without lookup keys, whose purpose is solely to trigger an action upon every packet processed by the pipeline stage. P4FPGA handles this corner case by not allocating any table resources to this stage.

Actions P4 actions can modify a field value; remove/add a header; or modify packet metadata. Conceptually, each action operates on one packet at any given time, with all temporary variable stored in metadata on the target. P4FPGA performs inline editing to packet metadata and post-pipeline editing to packet headers. Modify actions create a copy of the updated value stored in a memory that is merged with the original packet header field in the deparser block. For actions that change a packet header length, basic blocks are created before and after the deparser block, to perform re-alignment. For example, to remove a packet header, the deparser marks the header as invalid in a bit mask. The realignment block then shifts the remaining bytes forward to cover the gap created by the removed header.

Control Flow P4 control flow constructs compose tables and actions into an acyclic graph. A naïve implementation would be to use a fixed pipeline. In such a design, the runtime would use extra metadata to implement the logic of the source program. However, because we target an FPGA, P4FPGA can map the control flow directly onto the generated hardware design. Each node corresponds to a basic block followed by a branch condition. We note that this is much more flexible than implementing con-

trol flow on an ASIC. During program execution, the parsed packet and metadata is passed along the tree structure. At each node, the runtime evaluates the conditional and passes the data to the next node along the appropriate branch, or performs a table lookup depending on the rules specified in the control plane API. P4FPGA relies on pipeline parallelism to achieve high throughput. In other words, at any given time, different nodes in the tree can process different packets in parallel.

3.2.2 Fixed-Function Runtime

The P4FPGA fixed-function runtime provides the execution environment for packet processing algorithms specified in P4. It defines an API that allows generated code to access common functionality through a set of target-agnostic abstractions. Consequently, the runtime system plays a crucial role in providing an efficient, flexible and portable environment for packet processing applications. It must provide an abstract architecture that is uniform across many different hardware platforms. It must also provide an efficient medium to transport data across processing elements. Finally, the runtime system must provide auxiliary functionalities to support control, monitoring, and debugging.

Note that P4 developers can create a variety of potential applications, ranging from packet switching to NFV style hardware offloading for packet processing. These applications have different requirements from the architecture provided by a fixed function runtime. To support these different use-cases, P4FPGA allows P4 developers to choose either of two architectures: multi-port switching or bump-in-the-wire. The multi-port switching architecture is suitable for network forwarding elements, such as switches and routers, and for experimenting with new network routing protocols. The architecture includes an output cross-bar, as shown in Figure 3.4 to arbitrate packets to the appropriate

egress port. The bump-in-the-wire architecture is suitable for network functions and network acceleration. It receives packets from a single input port, and forwards to a single output port [117].

Below, we describe the design of the major components of the P4FPGA fixed function runtime. These components, indicated as grey boxes in Figure 3.4, include memory management, transceiver management, and host communication.

Memory Management As packets arrive at the FPGA, they must be stored in memory for processing. This memory can be designed in two ways. A straight-forward approach is to use FIFO queues, which forward packets through processing elements in the order in which they are received. However, simple FIFO queues are not sufficient for implementing more advanced packet-processing features, such as quality-of-service guarantees. In particular, such features require re-ordering packets as they are processed.

Therefore, P4FPGA includes an optional memory buffer managed by a hardware memory management unit (MMU). The MMU interface defines two functions: `malloc` and `free`. The `malloc` function takes one parameter, the size of packet buffer to be allocated rounded up to 256-byte boundary, and returns a unique packet identifier (PID). The PID is similar to a pointer in C, and is used throughout the lifetime of the packet in the pipeline. Upon the completion of packet transmission, the PID (and associated memory) is returned to the MMU to be reused for future packets, via a call to `free`. Users can configure the amount of memory used for storing packets. By default, P4FPGA allocates 65,536 bytes of on-chip block RAM (BRAM) per packet buffer.

Transceiver Management P4FPGA is portable across many hardware platforms. As a result, it provides a transceiver management unit that enables it to use the media access

control (MAC) and physical (PHY) layers specific to a target platform. For instance, the P4FPGA transceiver management unit uses separate protocols depending on whether Altera or Xilinx is the target platform without requiring changes to the P4 program.

Host Communication P4FPGA integrates a host communication channel between the FPGA and host CPU. This is useful for implementing the control channel and for debugging. The host communication channel is built on top of the PCI express protocol, which is the de-facto protocol for internal communication within network devices, such as switches and routers. We provide both blocking and non-blocking remote procedure calls (RPC) between software and hardware. For example, it is possible for a host program to issue an non-blocking call to read hardware registers by registering a callback function to receive the returned value. Similarly, a controller can program match tables by issuing a function call with an encoded table entry as a parameter.

3.2.3 Control Plane API

In addition to generating code that implements the application-specific logic in the data plane, P4FPGA also generates a control plane API that exposes a set of C++ functions that allow users to insert/delete/modify match table entries and read/write stateful memory. Moreover, the generated interface includes functions to support debugging. Users can inject packets via a packet capture (PCAP) trace, or can enable/disable an on-chip packet generator and capturer.

3.2.4 Extension

One of the advantages of FPGAs over ASICs are that they are relatively more flexible and programmable. P4 offers a relatively restrictive programming interface that is targeted for network applications and is platform-agnostic by design. Therefore, it is sometimes necessary to execute additional functionality via externally defined functions. An externally defined function could be used to access a state in a register or counter or to execute custom logic, such as a checksum computation or encryption/decryption. In P4, these are declared using `extern` blocks, and the implementation of these calls are target specific.

P4FPGA allows users to implement externally defined functions in the hardware description language of their choice. However, such functions pose a challenge for efficient code generation, since they may have high latency. For example, an external function that accesses persistent states or requires complex logic may take a long time to complete. If the processing pipeline were to block while waiting for the function to return, it could significantly impact throughput. P4FPGA provides an asynchronous implementation, so that processing of other packets can continue in parallel. This is roughly analogous to multi-threading, but without the associated cost of context switching.

3.3 Implementation

3.3.1 Optimization

To ensure that the code generated by P4FPGA is efficient, we implemented a number of optimizations at both the compiler and the micro-architectural level. Based on our experience, we have identified a few principles that we followed to improve the throughput and latency of the packet processing pipeline. Below, we describe the optimizations in the context of NetFPGA SUME platform, but the same principles should apply to other platforms such as Altera DE5. For clarity, we present these principles in order of importance, not novelty.

Leverage hardware parallelism in space and time to increase throughput. FPGAs provide ample opportunities to improve system throughput by leveraging parallelism in space, e.g., by increasing the width of the datapath. The throughput of a streaming pipeline, r, is determined the datapath width, w and the clock frequency, f ($r = w \times f$). The maximum clock frequency for an FPGA is typically a few hundred MHz (a mid-end FPGA ranges from 200 to 400 Mhz). Therefore, in order to reach a throughput of 40 to 100 Gbps, it is necessary to use a datapath width in the range of 100 to 1000 bits.

On the NetFPGA SUME platform, we target an overall system throughput of 40Gbps on the 4 available 10Gbps Ethernet ports at 250 MHz. We used 128-bits for the parser datapath and 512-bits for the forwarding pipeline datapath. The theoretical throughput for the parser is 128 bits $\times$ 250 Mhz, or 32 Gbps. As a result, we replicate the parser at each port to support parsing packets at 10 Gbps.

Another important form of hardware parallelism is pipeline parallelism. We clock

the P4 programmable pipeline at 250 MHz. If we process a single packet in every clock cycle, we would be able to process 250 Mpps (million packet per second). At 10 Gbps, the maximum packet arrival rate is 14.4 Mpps for 64 byte packets. At 250 Mpps, we should be able to handle more than sixteen 10 Gbps ports simultaneously with one P4 programmable pipeline. Of course, the theoretical maximum rate does not directly translate to actual system performance. Nonetheless, we conducted extensive pipelining optimizations to ensure that all generated constructs were fully pipelined. In other words, control flow, match tables and action engines are all fully pipelined.

Transform sequential semantics to parallel semantics to reduce latency. The P4 language enforces sequential semantics among actions in the same action block, meaning that side effects of a prior action must be visible to the next. A conservative compilation strategy that respects the sequential semantics would allocate a pipeline stage for each action. Unfortunately, this strategy results in sub-optimal latency, since each stage would add one additional clock cycle to the end-to-end latency. P4FPGA optimizes latency by leveraging the fact that the hardware inherently supports parallel semantics. As a result, we opportunistically co-locate independent actions in the same pipeline stage to reduce the overall latency of an action block.

Select the right architecture for the job. Network functions can be broadly divided into two sub-categories: those that need switching and those that do not. For example, network encryption, filtering, and firewalling can be enforced on a per-port basis. This is especially true if interface speed can be increased to 50 or 100Gbps, when CPUs barely have enough cycles to keep up with data coming in from one interface. On the other hand, prototyping network forwarding elements on FPGAs requires switching capability. As mentioned in Section 3.2.2, P4FPGA allows users to select the architecture most

appropriate for their needs.

Use a resource-efficient components to implement match tables. In P4FPGA generated pipelines, match tables dominate FPGA resource consumption. This is because FPGAs lacks hardened content-addressable memory (CAM), an unfortunate reality of using FPGAs for networking processing. Although one can implement CAM using existing resources on FPGA, such as Block RAMs or LUTs, it is not efficient. High-end FPGAs have more resources on-chip to implement CAMs, but they also come at a premium price. To alleviate the situation, P4FPGA uses resource-efficient hash-based methods for table lookup. The compiler uses these more efficient implementation techniques by default. But, users may choose to use more expensive CAM implementations by specifying a compiler flag.

Eliminate dead metadata A naïve P4 parser implementation would extract full header and metadata from packets by default. This can be wasteful if the extracted headers are not used in the subsequent pipeline. P4FPGA analyzes all match and action stages, and eliminates unused header fields and metadata from the extracted packet representation.

Use non-blocking access for external modules. Stateful processing is expensive on high-performance packet-processing pipelines. Complex operations may require multiple clock cycles to finish, which can negatively affect performance if pipelining is only performed at the function level. P4FPGA implements fine-grained pipelining on stateful elements to maintain high throughput. For example, a memory read operation requires issuing a read request to memory and waiting for the corresponding response. Due to the high latency of memory, the response may only come after multiple cycles of delay.

In P4FPGA, we support split-phase read such as read request and response can happen at different clock cycle. Meanwhile, the pipeline can continue processing other packets.

3.3.2 Prototype

Our prototype P4FPGA implementation consists of a C++-based compiler along with a Bluespec-based runtime system. For the frontend, we reused P4.org's C++ compiler frontend to parse P4 source code and generate an intermediate representation [154]. We designed a custom backend for FPGAs, which consists of 5000 lines of C++ code. The runtime is developed in a high-level hardware description language Bluespec [147]. Bluespec provides many of the higher level hardware abstractions (e.g., FIFO with back-pressure) and the language includes a rich library of components, which makes development easier. The runtime is approximately 10,000 lines of Bluespec. We relied on Bluespec code from the Connectal project [96] to implement the control plane channel. We also implemented mechanisms to replay `pcap` traces, access control registers, and program dataplane tables. All code is publicly available under an open-source license.[1]

Complex FPGA-based systems often require integration with existing intellectual property (IP) components from other vendors and P4FPGA is no exception. We allow third-party IPs to be integrated with the existing P4FPGA runtime system as long as those components conform to the interfaces exposed by P4FPGA runtime. For example, we currently support IP cores such as MAC/PHY and Ternary CAM (TCAM) provided by FPGA vendors and commercial IP vendors [35].

[1] http://p4fpga.github.io

3.4 Evaluation

In this section, we explore the performance of the P4FPGA. Our evaluation is divided into two main sections. First, we evaluate the ability of P4FPGA to handle a diverse set of P4 applications. Then we use a set of microbenchmarks to evaluate the individual components of P4FPGA in isolation.

Toolchain and hardware setup We evaluate the performance of P4FPGA generated designs against a set of representative P4 programs. Each program in our benchmark suite is compiled with the P4FPGA compiler into Bluespec source code, which is then processed by a commercial compiler from Bluespec Inc. to generate Verilog source code. Next, the Verilog source code is processed by the standard Vivado 2015.4 tool from Xilinx, which performs synthesis, placement, routing and bitstream generation. All of the above steps are automated with a `Makefile`. The compilation framework supports both the Altera tool suite, Quartus, and Xilinx tool suite, Vivado. For this evaluation, we only used Vivado. We deployed the compiled bitstream on a NetFPGA SUME platform with a Xilinx Virtex-7 XC7V690T FPGA, with 32 high-speed serial transceivers to provide PCIe (Gen3 x8) communication and 4 SFP+ ports (10Gbps Ethernet).

For packet generation, we built a custom packet generator that is included as part of the P4FPGA runtime. It generates packets at a user-specified rate. We also provide a utility to program the packet generator with a packet trace supplied in the `PCAP` format or to configure/control the packet generator from userspace. Similarly, we provide a built-in packet capture tool to collect output packets and various statistics.

Name	LoC in P4	LoC in Bluespec	Framework
l2l3.p4	170	1281	33295
mdp.p4	205	1812	33295
paxos.p4	385	3306	33295

Table 3.1: Example applications compiled by P4FPGA and lines of code (LoC) in P4 and Bluespec. l2l3.p4 implements a L2/L3 router, mdp.p4 implements a variable packet length, financial trading protocol parser, paxos.p4 implements a stateful consensus protocol.

3.4.1 Case Studies

To illustrate the broad applicability of P4FPGA, we implemented three representative P4 applications as case studies. We chose these examples because (i) they represent non-trivial, substantial applications, (ii) they illustrate functionality at different layers of the network stack, and (iii) they implement diverse functionality and highlight P4's potential.

Table 3.1 shows the lines of code in P4 for each of these applications. As a point of comparison, we also report the lines of code for the generated Bluespec code. While lines of code is not an ideal metric, it does help illustrate the benefit of high-level languages like P4, which require orders-of-magnitude fewer lines of code. Below, we describe each of these applications in detail.

L2/L3 Forwarding P4 was designed around the needs of networking applications that match on packet headers and either forward out of a specific port, or drop a packet. Therefore, our first example application performs Layer 2 / Layer 3 forwarding. It uses the switching architecture, and routes on the IP destination field.

Paxos Paxos [105] is one of the most widely used protocols for solving the problem of *consensus*, i.e., getting a group of participants to reliably agree on some value used for computation. The protocol is the foundation for building many fault-tolerant distributed systems and services. While Paxos is traditionally implemented as an application-level service, recent work demonstrates that significant performance benefits can by achieved by leveraging programmable data planes to move consensus logic into network devices [56, 119]. The P4 implementation [56] defines a custom header for Paxos messages that is encapsulated inside a UDP packet. The program keeps a bounded history of Paxos packets in registers, and makes stateful routing decisions based on comparing the contents of arriving packets to stored values. Paxos uses a bump-in-the-wire architecture.

Market Data Protocol Many financial trading strategies depend on the ability to react immediately to changing market condition and place orders at high speeds and frequencies. Platforms that implement these trading algorithms would therefore benefit by offloading computations into hardware using custom packet headers and processors. As a proof-of-concept for how P4 could be used for financial applications, we implemented a commonly used protocol, the Market Data Protocol (MDP). MDP is used by the Chicago Mercantile Exchange. Essentially, MDP is a L7 load balancer. An MDP P4 implementation complicated by the fact that the protocol header is variable length. Our P4 implementation of MDP can address the header variable length and also parse input packet streaming, filter duplicated messages and extract important packet fields for additional processing.

Processing time and latency Our evaluation focuses on two metrics: processing time and latency. Table 3.2 shows the processing time for each application on a single FPGA.

App	Size	Parser	Table	Memory	Deparser
l2l3	64	2	31	21	11
	256	2	31	23	32
	512	2	31	24	66
	1024	2	31	23	130
mdp	256	15	9	23	34
	512	35	9	24	68
	1024	88	9	23	130
paxos	144	6	42	21	12

Table 3.2: Processing time breakdown, cycles @ 250MHz.

Mode	Packet Size			
	64	256	1024	1518
Cut-through	1.1us	1.2us	1.4us	1.4us
Store-and-forward	5us	4.3us	5.5us	6.1us
P4FPGA (L2/L3)	0.34us	0.42us	0.81us	1.05us

Table 3.3: Latency comparing to vendors. The numbers of cut-through and store-and-forward switches are from [193]

The numbers are in term of cycles running at 250MHz, where each cycle is 4 nanoseconds. We measured the packet-processing time of each application on small and large packets. Since the L2/L3 application only parses Ethernet and IP headers, parsing only takes 2 cycles, or 8 ns. On the contrary, the MDP application spends more time parsing because it performs variable-length header processing and inspects packet payload for market data. Match and action stages are constant time for each application. For example, L2/L3 spends 31 cycles or 128 ns in match and action stage. The time is spent on table look-up, packet field modification and packet propagation through multiple pipeline stages. The amount of time spent in match and action stage depends on the number of pipeline stages and the complexity of actions performed on a packet. Memory access accounts for time taken to access a shared memory buffer, and therefore represents a

constant overhead among all packets (+/- measurement error). The time required for the deparser, which must reassemble and transmit the packet, is proportional to the packet size.

We define latency as the time from when the first bit of packet enters the P4 pipeline (right after the RX channel in Figure 3.4) until the first bit of packet exits the pipeline (right before the TX channel 3.4). In all three cases, P4FPGA introduces less than 300ns of latency. To place the latency numbers in context, we report the performance results from two vendors in Table 3.3. As we can see, P4FPGA is able to offer very low latency comparable to commercial off-the-shelf switches and routers.

Finally, Figure 3.5 summarizes the throughput experiments for all three applications. For the L2/L3 application, we see it is able to process all packet sizes at line rate. Note that for small packets, such as 64-byte packets, the theoretical line rate is less the 10Gbps due to the minimum inter-packet gap requirement of Ethernet. The MDP application shows similar throughput to the L2/L3 application. The Paxos application, however, has slightly lower throughput compared to the other two, due to the heavy use of stateful register operations. We note, though, that the throughput for Paxos far exceeds that of software-based implementations. [56]

3.4.2 Microbenchmarks

The next part of our evaluation focuses on a set of microbenchmarks that evaluate different aspects of P4FPGA in isolation. We investigate the following questions:

- How does runtime perform?
- How does overall pipeline perform?

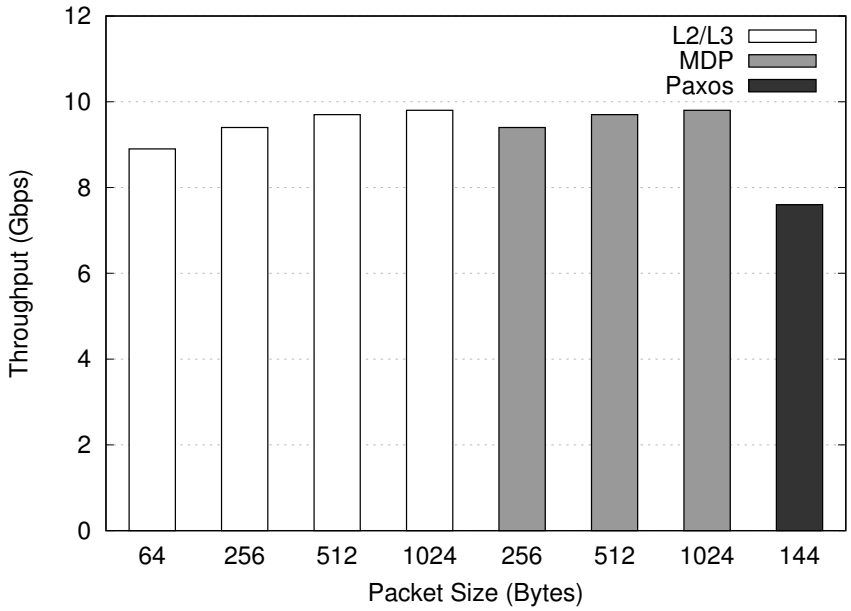


Figure 3.5: Application throughput for L2L3, MDP and Paxos.

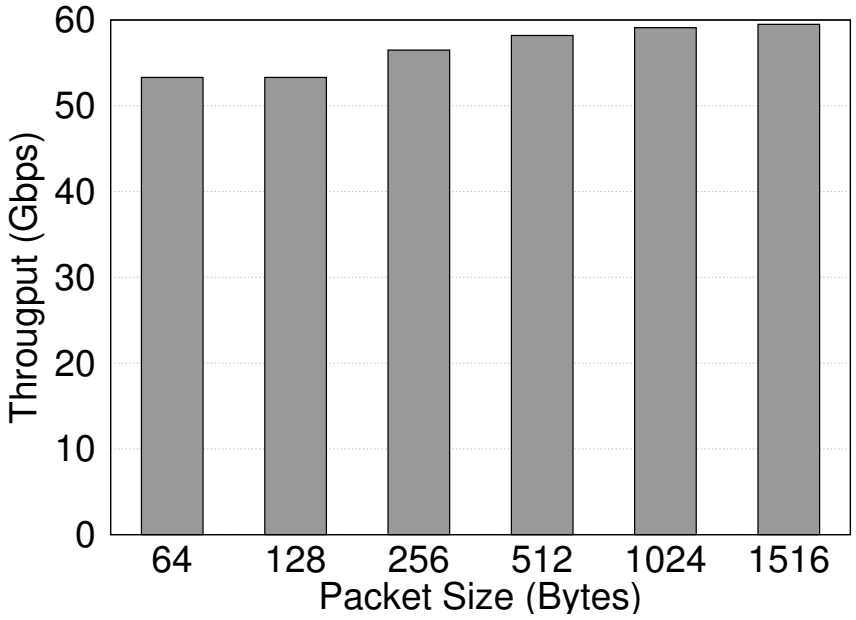
- How much of the FPGA resources are required for the pipeline and runtime?

We focus on three metrics for evaluation: throughput, latency and resource utilization on the FPGA. We present the details of these microbenchmarks below.

Fixed-Function Runtime

The fixed-function runtime system must sustain line-rate forwarding at 4 x 10Gbps to avoid being a bottleneck to overall system performance. To verify that P4FPGA is able to satisfy this requirement, we measured the raw throughput of the fixed-function runtime with an empty packet processing pipeline (no table, no action in ingress or egress pipeline).

The results are shown in Figure 3.6. In this experiment, six packet generators are configured and each generates packets at the full line rate, 10Gbps. We loaded a packet trace with packet sizes ranging from 64 to 1516 bytes and replayed the packet trace one



million times. The fixed function runtime is able to sustain over 40Gbps for all packet sizes.

Programmable pipeline

We evaluated the performance of a generated P4 pipeline with a set of micro-benchmarks that focused on each key language construct isolation: parsers, tables, and actions. As a point of comparison, we also report results for running the same experiments with the PISCES [179] software switch. PISCES extends Open vSwitch [152] with a protocol-independent design. In all cases, PISCES uses DPDK [64] to avoid the overhead of the kernel network stack. Note, to make the comparison equal, we used only two ports for PISCES.

Parser We used the packet generator to send 256-byte packets with an increasing number of 16-bit custom packet headers. We measured both latency and throughput, and the results are shown in Figures 3.7 and 3.8. As expected, we see that parsing la-

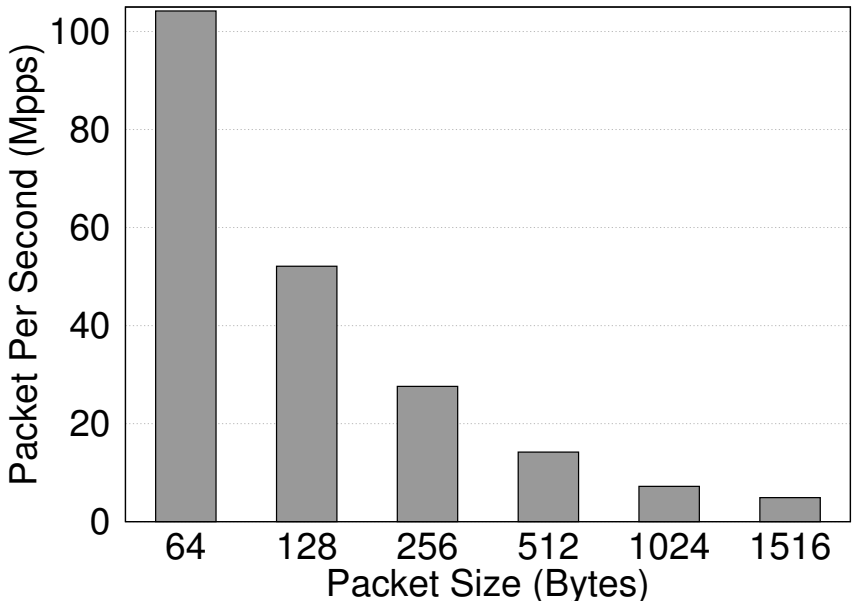


Figure 3.6: Runtime forwarding performance in gigabits per second (left) and millions packets per second (right) with a simple forwarding application on a 6-port switching architecture.

tency increases as we increase the number of extracted headers. In terms of absolute latency, P4FPGA is able to leverage the performance of FPGAs to significantly reduce throughput. P4FPGA took less than 450 ns to parse 16 headers, whereas PISCES took 6.5us. The results for throughput are similar. For both P4FPGA and PISCES, the parser throughput decreases as the number of headers increases. As expected, P4FPGA significantly outperforms PISCES in terms of absolute throughput as well as the number of headers that parse without performance degradation.

Table In this experiment, we compiled a set of synthetic programs with an increasing number of pipeline stages (1 to 32). We measured the end-to-end latency from ingress to egress. The result is shown in figure 3.9. Although the absolute latency is much better for P4FPGA, the trend shows that the processing latency increases with the number of tables. In contrast, the latency for PISCES remains constant. This is because PISCES

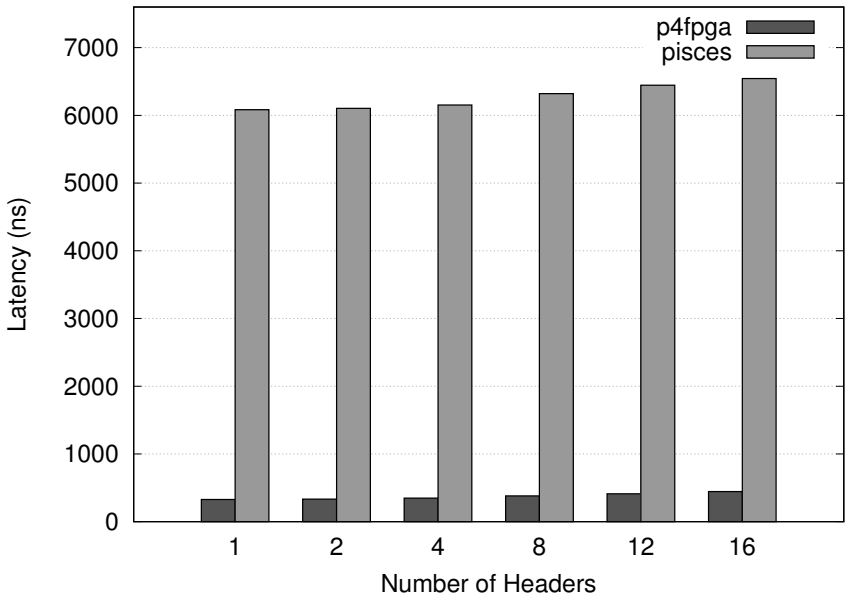


Figure 3.7: Parser Latency v.s. Number of Headers parsed

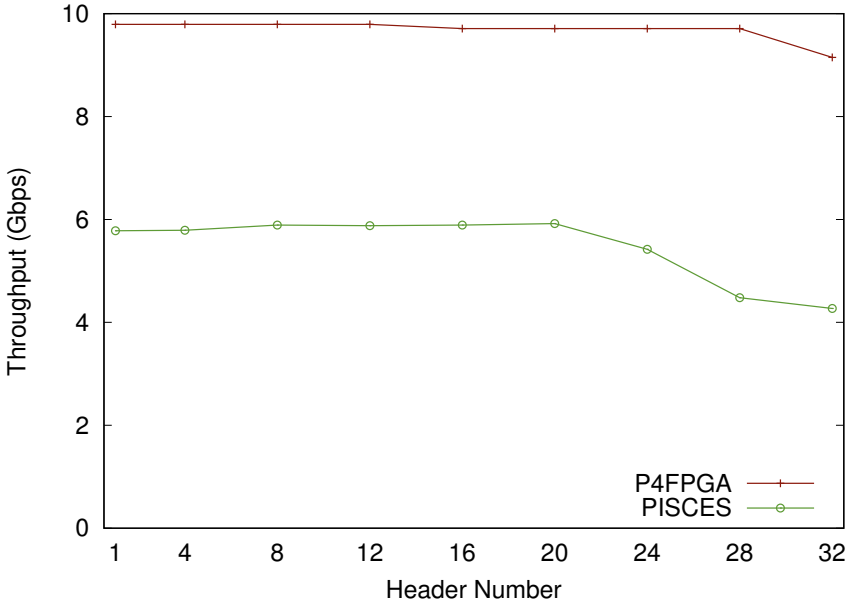


Figure 3.8: Parser Throughput v.s. Number of Headers parsed

implements an optimization that fuses multiple match-action pipeline stages into a single match-action rule. We have not yet implemented this optimization for P4FPGA.

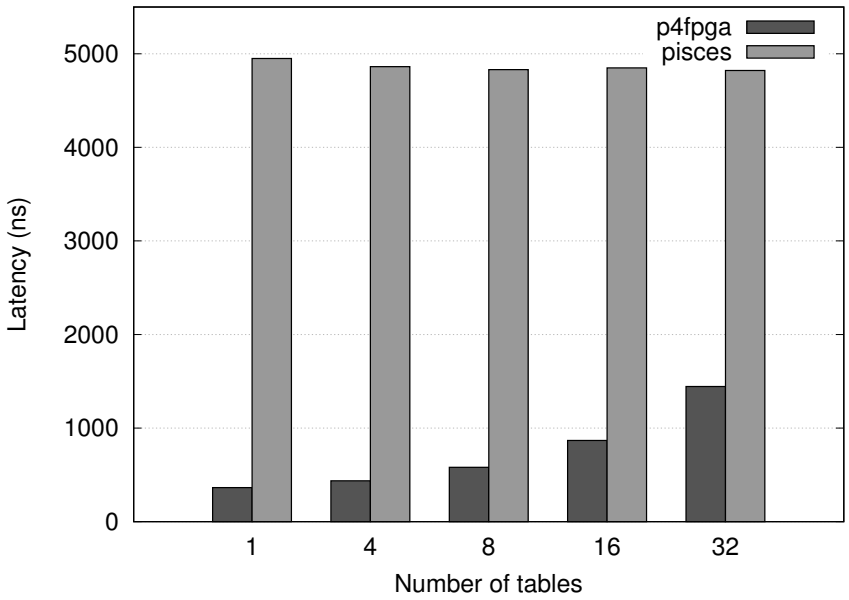


Figure 3.9: Processing latency versus number of tables

Action In this experiment, we evaluate how the action complexity can affect throughput. We vary the number of header field writes from 8 to 64. All field write operations are independent, meaning that they write to different fields in the packet header. As a result P4FPGA is able to leverage hardware parallelism to perform all write operations within the same clock cycle, as there is no dependency between any operations. Note that this faithfully implements the sequential semantics of the original P4 source program, even though all actions are performed in parallel. As shown in Figure 3.10, end-to-end packet processing latency in P4FPGA remains the same, at 364 ns. This is in contrast to PISCES, which consumes more CPU cycles to process write operations, as the operations are performed in sequence on a CPU target. [179] In other words, the absolute latency is much higher on a software target, and also increases with the number of write operations. In contrast, with P4FPGA, the latency remains low and constant independent of the number of writes in a stage.

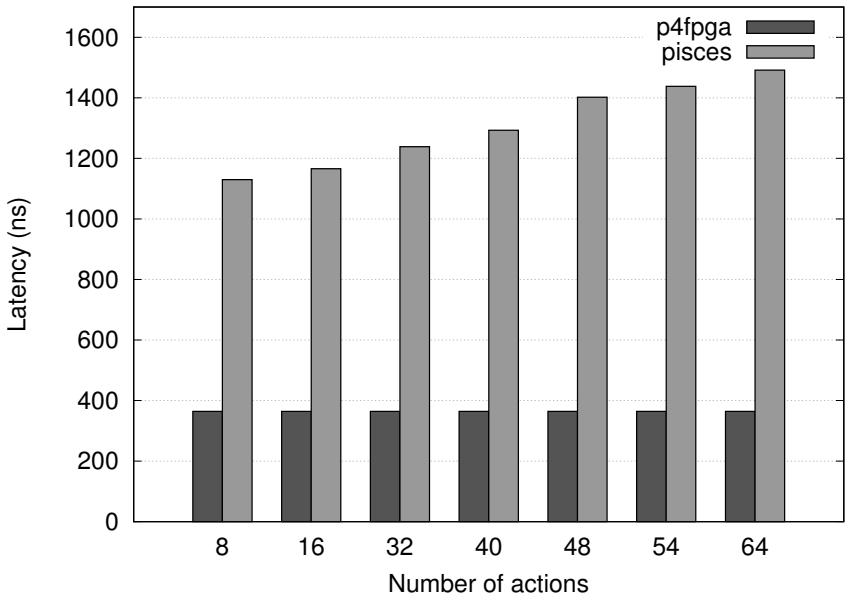


Figure 3.10: Pipeline Latency v.s. Number of Actions.

Resource Utilization

We report the resource utilization of the FPGA in two parts: the resource consumed by the fixed function runtime which is common to all P4 programs; and the resource consumed by individual P4 constructs which is variable depending on parameters specified by P4 program. We quantify resource consumption with the number and percentage of look-up tables (LUTs) and memory consumed by each block.

The runtime subsystem implements PCIe host communication, and the MAC and PHY layers of the Ethernet protocol. As shown in Table 2.1, the total resource consumption of the runtime is about 7.5% of total available LUTs and 2.3% of available memory blocks, which leaves many of the resources available to implement the actual logic of a P4 program.

Next, we profile resource consumption of major P4 constructs: match table, parser, deparser and action. Match tables are implemented with Content Addressable Mem-

	Slice LUTs	Block RAMs	MHz
PCIe	6377	9	250
10G MAC x4	8174	0	156
10G PHY x4	10422	0	644.5
Connectal	7867	25	250
Area Used	32700 (7.5%)	34 (2.3%)	-

Table 3.4: Area and frequency of fixed function runtime.

Hardware	Key Size (Bits)	#Entries	% BRAM	#Flip-Flops	#LUTs
BCAM	36	1024	2.2% (32)	0.26% (2280)	0.59% (2552)
	144	1024	8.7% (128)	0.92% (7976)	2.2% (9589)
	288	1024	17.4% (256)	1.8% (15944)	4.5% (19350)
TCAM	72	2048	2.7% (40)	0.56% (4900)	2.1% (9100)
	144	2048	3.2% (48)	0.63% (5482)	2.8% (12033)
	288	2048	3.9% (58)	0.85% (7430)	4.4% (18977)
HASH	72	1024	0.7% (10.5)	0.12% (1053)	0.27% (1185)
	144	1024	0.8% (12.5)	0.16% (1440)	0.32% (1395)
	288	1024	1.1% (16.5)	0.25% (2232)	0.46% (2030)

Table 3.5: BCAM and TCAM Resource Utilization on Virtex 7 XCVX690T, which has 1470 BRAM blocks, 866400 Flip-flops and 433200 LUTs. We show resource utilization as percentage as well as actual amount of resource used.

ory (CAM) to perform key lookup, and regular memory to corresponding action for a matched key entry. Unlike ASICs, FPGAs lack native support for CAM, and as a result, we had to emulate CAM by implementing it with regular memory blocks. We evaluated three different CAM implementations on the FPGA: binary CAM for implementing exact match, ternary CAM for implementing ternary and longest prefix match, and hash-based CAM for exact match.

As shown in Table 3.5, we can implement up to a 288-bit key Binary CAM (BCAM),

Ternary CAM (TCAM), or a hash-based associative memory with minimum resource utilization. The commercial-grade TCAM implementation is more efficient than our BCAM. We suspect that the difference is due to both implementation efficiency and internal architecture of these two CAM technologies. But, the hash-based associative memory implementation is the most efficient among all three implementations. [61] If we were to use the whole FPGA for only a CAM with a 288-bit key, then a BCAM, TCAM, and hash-based associative memory can fit up to 6K, 53K, 93K entries on a Virtex-7 FPGA, respectively. To put these numbers into context, a Mellanox Spectrum ASIC allows 9K entries of 288 bit rules in a common TCAM table shared between ingress and egress pipeline.

3.5 Application: Hardware Accelerated Consensus Protocol

This work appears in *Network Hardware-Accelerated Consensus* as a technical report, with co-authors Huynh Tu Dang, Pietro Bressana, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, Robert Soule.

In this section, we described an emerging application developed using P4FPGA in more details. In particular, we describe an implementation of Paxos in the P4 language. Paxos [105] is one of the most widely used protocols for solving the problem of consensus, i.e., getting a group of participants to reliably agree on some value used for computation. Paxos is used to implement state machine replication, [104, 178], which is the basic building block for many fault-tolerant systems and services that comprise the core infrastructure of data centers such as OpenReplica, [153] Ceph, [48], and Google's Chubby. [45] Since most data center applications critically depend on these services, Paxos has a dramatic impact on the overall performance of the data center.

While Paxos is traditionally implemented as an application-level service, there are significant performance benefits to be gained by moving certain Paxos logic into network devices. Specifically, the benefits would be twofold. First, since the logic traditionally performed at servers would be executed directly in the network, consensus messages would travel fewer hops, and can be processed "on the wire," resulting in decreased latencies. Second, rather than executing server logic (including expensive message broadcast operations) in software, the same operations could be implemented in specialized hardware, improving throughput. Although Paxos is a conceptually simple protocol, there are many details that make an implementation challenging. Consequently, there has been a rich history of research papers that describe Paxos implementations, including attempts to make Paxos Simple, [106] Practical, [136] Moderately

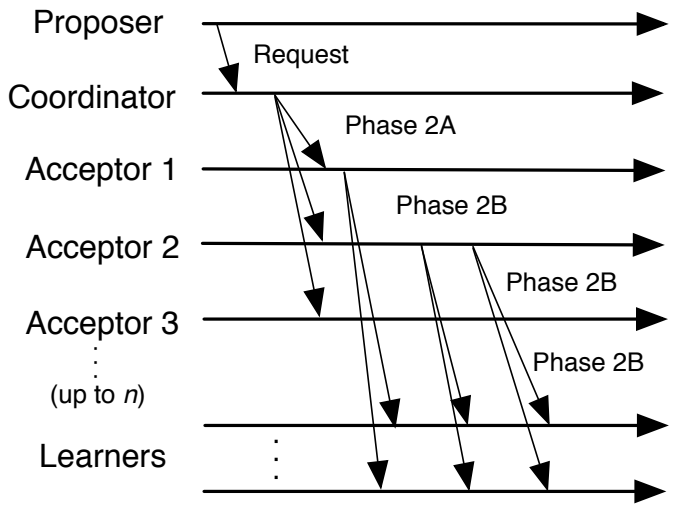


Figure 3.11: The Paxos protocol Phase 2 communication pattern.

Complex, [194] and Live. [50]

Our implementation of Paxos in the P4 is interesting beyond presenting the Paxos algorithm in a new syntax. It helps expose new practical concerns and design decisions for the algorithm that have not, to the best of our knowledge, been previously addressed. For example, a switch-based implementation cannot synthesize new messages. Instead, we have to map the Paxos logic into a "routing decision." Moreover, targeting packet headers and switch hardware imposes memory and field size constraints not present in an application library implementation.

3.5.1 Background

Paxos [105] is perhaps the most widely used consensus protocol. Participants are processes that communicate by exchanging messages. A participant may simultaneously play one or more of four roles: *proposers* propose a value; *acceptors* choose a single value; and *learners* learn what value has been chosen. One process, typically a proposer

or acceptor, acts as the *coordinator*, which ensures that the protocol terminates and establishes an ordering of messages. The coordinator is chosen via an application-specific protocol, called *leader election*, which is external to the Paxos protocol.

A Paxos *instance* is one execution of consensus. An instance begins when a proposer issues a request, and ends when learners know what value has been chosen by the acceptor. Below, we describe one instance of Paxos. However, throughout this section, references to Paxos implicitly refer to multiple instances chained together (i.e., Multi-Paxos). [50] The protocol proceeds in a sequence of rounds. Each round has two phases.

Phase 1. The coordinator selects a unique round number (*rnd*) and asks the acceptors to vote for the value in the given instance. Voting means that they will reject any requests (Phase 1 or 2) with round number less than *rnd*. Phase 1 is completed when a majority-quorum Q of acceptors confirms the promise to the coordinator. If any acceptor already accepted a value for the current instance, it will return this value to the coordinator, together with the round number received when the value was accepted (*vrnd*).

Phase 2. Figure 3.11 illustrates the communication pattern of Paxos participants during Phase 2. The coordinator selects a value according to the following rule: if no acceptor in Q returned an already accepted value, the coordinator can select any value. If however any of the acceptors returned a value in Phase 1, the coordinator is forced to execute Phase 2 with the value that has the highest round number *vrnd* associated to it. In Phase 2, the coordinator sends a message containing a round number (the same used in Phase 1.) Upon receiving such a request, an acceptors accepts it and broadcasts it to all learners, unless it has already received another message (Phase 1 or 2) with a higher round number. Acceptors update their *rnd* and *vrnd* variables with the round number in the message. When a quorum of acceptors accepts the same round number, consensus

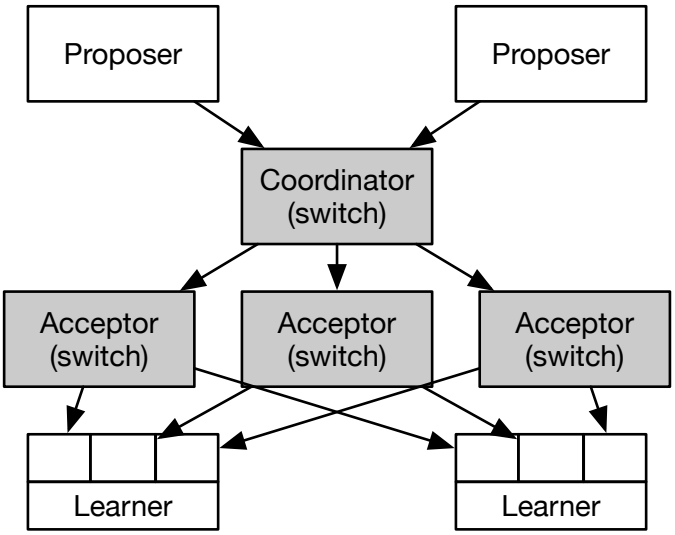


Figure 3.12: A switch-based Paxos architecture. Switch hardware is shaded grey, and commodity servers are colored white.

is reached: the value is permanently bound to the instance, and nothing will change this decision. Acceptors send a 2b message to the learners with the accepted value. When a learner recieves a majority quorum of messages, they can deliver the value.

As long as a nonfaulty coordinator is eventually selected, there is a majority quorum of nonfaulty acceptors, and at least one non-faulty proposer, every consensus instance will eventually decide on a value.

3.5.2 Design

Figure 3.12 illustrates the architecture of a switch-based Paxos, which we describe in detail below. In the figure, switch hardware is shaded grey, and commodity servers are colored white.

Overview. As with any Paxos implementation, there are four roles that participants

in the protocol play: proposers, coordinators, acceptors, and learners. However, while proposers and learners are implemented as application libraries that run on commodity servers, a switch-based Paxos differs from traditional implementations in that *coordinator* and *acceptor* logic executes on switches.

An instance of consensus is initiated when one of the proposers sends a message to the coordinator. The protocol then follows the communication pattern illustrated in Figure 3.11[2]. Although the Paxos protocol described in Section 3.5.1 has two phases, Phase 1 does not depend on any particular value, and can be run in advance for a large bounded number of instances. [105] The pre-computation needs to be re-run under two scenarios: either the Paxos instance approaches the number of pre-computed instances, or the device acting as coordinator changes. (for example due to failure)

System assumptions. It is important to note that the Paxos protocol does not guarantee or impose an ordering on consensus instances. Rather, it guarantees that for a given instance, a majority of participants agree on a value. So, for example, the i-th instance of consensus need not complete before the $(i + 1)$-th instance. The application using Paxos must detect if a given instance has not reached consensus. In such an event, the instance may be re-initiated. The protocol naturally ensures that re-executing an already agreed-upon instance cannot change the value. The process of detecting a missing instance and re-initiating consensus depends on the details of the particular application and deployment. For example, if proposers and learners are co-located, then a proposer can observe if an instance has reached consensus. If they are deployed on separate machines, then proposers would have to employ some other process (e.g., using acknowledgments and timeouts).

[2]In the figure, the initial message is called a request. This is a slight abuse of terminology, since the term request often implies a response, or a client-server architecture, neither of which is required in Paxos. However, calling it a request helps to distinguish it from other messages.

We should also point out that the illustration in Figure 3.12 only shows one coordinator. If the other participants in the Paxos protocol suspect that the switch is faulty, the coordinator functionality can be moved either to another switch or to a server that temporarily assumes the role of coordinator. The specifics of the leader-election process are application-dependent. We have elided these details from the figure to simplify the presentation of our design.

Prototype implementation. The proposer and learner code are written as Python modules, while the coordinator and acceptor code are written in P4. All messages are sent over UDP to an IP Multicast address. Using TCP is unnecessary, since we don't require reliable communication. Using IP Multicast is expedient, since it is supported by most switch hardware, and allows us to reuse readily available functionality to send messages to a group of receivers.

Paxos header. In a traditional Paxos implementation, each participant receives messages of a particular type, (e.g., Phase 1A, 2A, etc.) executes some processing logic, and then synthesizes a new message which it sends to the next participant in the protocol.

However, switches cannot craft new messages; they can only modify fields in the header of the packet that they are currently processing. Therefore, a switch-based Paxos needs to map participant logic into forwarding decisions, and each packet must contain the union of all fields in all Paxos messages.

Figure 3.13 shows the P4 specification of a common packet header for Paxos messages. To keep the header small, the semantics of some of the fields change depending on which participant sends the message. The fields are as follows: `msgtype` distinguishes the various Paxos messages (e.g., 1A, 2A, etc.); `inst` is the consensus instance number; `rnd` is either the round number computed by the proposer or the round number

for which the acceptor has cast a vote; `vrnd` is the round number in which an acceptor has cast a vote; `swid` identifies the sender of the message; and `value` contains the request from the proposer or the value for which an acceptor has cast a vote.

Given the storage limitations of the target platform, there are practical concerns that must be addressed in a switch-based Paxos that are not normally considered in a traditional implementation. First, the number of instances that can be pre-computed in Phase 1 is bound by the size of the `inst` field. If this field is too small, then consensus could only be run for a short time. On the other hand, the coordinator and acceptor code must reserve sufficient memory and make comparisons on this value, so setting the field too large could potentially impact performance. Second, it would seem natural to store the `value` in the packet payload, not the packet header. However, Paxos must maintain the history of values, and to do so in P4, the field must be parseable, and stored in a register. We are therefore forced to keep `value` in the header. Third, not all values in `value` will have the same size. This size is dependent on the application. While P4 plans to support variable length fields, the current version only supports fixed length fields. Since we have to conservatively set the value to the size of the largest value, we are storing potentially unused bytes.

We will need to run experiments on an actual hardware deployment to determine the appropriate field sizes. For now, our implementation uses reasonable default values. Constants such as message types are implemented with `#define` macros, since there is no notion of an enumerated type in P4.

Proposer. Proposers initiate an instance of consensus. The proposer logic is implemented as a library that exposes a simple API to the application. The API consists of a single method `submit`, which is used by the application to send values. The proposer component creates a switch Paxos message to send to the coordinator, and writes the

```
1 header_type paxos_t {
2     fields {
3         msgtype : 8;
4         inst    : INST_SIZE;
5         rnd     : 8;
6         vrnd    : 8;
7         swid    : 64;
8         value   : VALUE_SIZE;
9     }
10 }
11
12 parser parse_paxos {
13     extract(paxos);
14     return ingress;
15 }
```

Figure 3.13: Paxos packet header and parsers.

value into the header.

Coordinator. A coordinator brokers requests on behalf of proposers. They ensure that only one process submits a message to the protocol for a particular instance (thus ensuring that the protocol terminates), and impose an ordering of messages. When there is a single coordinator, as is the case in our prototype, a monotonically increasing sequence number can be used to order the messages. This sequence number is written to the `inst` field of the header.

A coordinator should only receive request messages, which are sent by the proposer. When messages arrive, they only contain a value. Mapping coordinator logic to stateful forwarding rules and actions, the switch must perform the following operations: write the current instance number and an initial round number into the message header; increment the instance number for the next invocation; store the value of the new instance number; and broadcast the packet to all acceptors.

Figure 3.14 shows the P4 implementation. One conceptual challenge is how to ex-

```
 1 register reg_inst {
 2     width : INST_SIZE;
 3     inst_count : 1;
 4 }
 5
 6 action handle_request() {
 7     modify_field(paxos.msgtype, PAXOS_2A);
 8     modify_field(paxos.rnd, 0);
 9     register_read(paxos.inst, reg_inst, 0);
10     add_to_field(paxos.inst, 1);
11     register_write(reg_inst, 0, paxos.inst);
12 }
13
14 table tbl_sequence {
15     reads { paxos.msgtype : exact; }
16     actions { handle_request; _nop; }
17     size : 1;
18 }
19
20 control ingress {
21     /* process other headers */
22     if (valid(paxos)) {
23         apply(tbl_sequence);
24     }
25 }
```

Figure 3.14: Coordinator code.

press the above logic as *match+action* table applications. When packets arrive in the `control` block (line 20), the P4 program checks for the existence of the Paxos header (line 22), and if so, it passes the packet to the table, `tbl_sequence` (line 23). The table performs an exact match on the `msgtype` field, and if it receives Phase 2A message, it will invoke the `handle_2a` action. The action updates the packet header fields and persistent state, relying on a register named `reg_inst` (lines 1-4) to read and store the instance number.

Acceptor. Acceptors are responsible for choosing a single value for a particular instance. For each instance of consensus, each individual acceptor must "vote" for a

value. The value can later be delivered if a majority of acceptors vote the same way. The design of a switch-based implementation is complicated by the fact that acceptors must maintain and access the history of proposals for which they have voted. This history ensures that acceptors never vote for different values in a particular instance, and allows the protocol to tolerate lost or duplicate messages.

Acceptors can receive either Phase 1A or Phase 2A messages. Phase 1A messages are used during initialization, and Phase 2A messages trigger a vote. The logic for handling both messages, when expressed as stateful routing decisions, involves: reading persistent state; modifying packet header fields; updating the persistent state; and forwarding the modified packets. The logic differs in which header fields are read and stored.

Figure 3.15 shows the P4 implementation of an acceptor. Again, the program must be expressed as a sequence of *match+action* table applications, starting at the control block (line 56). Acceptor logic relies on several registers, indexed by consensus instance, to store the history of `rounds`, `vrounds`, and `values` (lines 9-22). It also defines two actions for processing Phase 1A messages (lines 33-39) and Phase 2A messages (lines 41-47). Both actions require that the `swid` field is updated, allowing other participants to identify which acceptor produced a message.

The programming abstractions make it somewhat awkward to express the comparison between the `rnd` number in the arriving packet header, and the `rnd` number kept in storage. To do so, the arriving packet must be passed to a dedicated table `tbl_rnd`, which triggers the action `read_rnd`. The action reads the register value for the instance number of the current packet, and copies the result to the metadata construct (line 30). Finally, the number in the metadata construct can be compared to the number in the current packet header (line 60).

Learner. Learners are responsible for replicating a value for a given consensus instance. Learners receive votes from the acceptors, and "deliver" a value if a majority of votes are the same (i.e., there is a quorum). The only difference between the switch-based implementation of a learner and a traditional implementation is that the switch-based version reads the relevant information from the packet headers instead of the packet payload.

Learners only receive Phase 2B messages. When a message arrives, each learner extracts the instance number, switch ID, and value. The learner maintains a data structure that maps a pair of instance number and switch ID to a value. Each time a new value arrives, the learner checks for a majority-quorum of acceptor votes. A majority is equal to $f + 1$ where f is the number of faulty acceptors that can be tolerated.

Optimizations. Implementing Paxos in P4 requires $2f + 1$ acceptors. Considering that acceptors in our design are network switches, this could be too demanding. However, we note that one could exploit existing Paxos optimizations to spare resources. Cheap Paxos [108] builds on the fact that only a majority-quorum of acceptors is needed for progress. Thus, the set of acceptors can be divided into two classes: first-class acceptors, which would be implemented in the switches, and second-class acceptors, which would be deployed in commodity servers. In order to guarantee fast execution, we would require $f + 1$ first-class acceptors (i.e., a quorum) and f second-class acceptors. Second-class acceptors would likely fall behind, but would be useful in case a first-class acceptor fails. Another well-known optimization is to co-locate the coordinator with an acceptor, which in our case would be an acceptor in the first class. In this case, a system configured to tolerate one failure ($f = 1$) would require only two switches.

3.5.3 Discussion

The process of implementing Paxos in P4 highlights future areas of research for designers of consensus protocols. We expand the discussion of these two topics below.

Impact on P4 Language P4 provides a basic set of primitives that are sufficient for implementing Paxos. Other languages, such as POF [186] and PX, [41] offer similar abstractions. Implementing Paxos provides an interesting use case for dataplane programming languages. As a result of this experience, we developed several "big-picture" observations about the language and future directions for extensions or research.

P4 presents a paradigm of "programming with tables" to developers. This paradigm is somewhat unnatural to imperative (or functional) programmers, and it takes some time to get accustomed to the abstraction. It also, occasionally, leads to awkward ways of expressing functionality. An example was already mentioned in the description of the acceptor logic, where performing a comparison required passing the packet to a table, to trigger an action, to copy a stored value to the metadata construct. It may be convenient to allow storage accesses directly from `control` blocks.

Although P4 provides macros that allow source to be imported from other files (e.g., `#include`), the lack of a module system makes it difficult to isolate functionality, and build applications through composition, as is usually suggested as best practice for software engineering. For example, it would be nice to be able to "import" a Paxos module into an L2 learning switch. This need is especially acute in `control` blocks, where tables and control flow have to be carefully arranged. As the number of tables, or dataplane applications, grows, it seems likely that developers will make mistakes.

Attempting to access a register value from an index that exceeds the size of the array

results in a segmentation fault. Obviously, performing bounds checks for every memory access would add performance overhead to the processing of packets. However, the alternative of allowing unsafe operations that could lead to failures is equally undesirable. It may be useful in the future to provide an option to execute in a "safe mode," which would provide run-time boundary checks as a basic precaution. It would also be useful to provide a way for programs to catch and recover from errors or faults.

While P4 provides a stateful memory abstraction, (a register,) there is no explicit way of controlling the memory layout across a collection of registers and tables, and its implementation is target dependent. In our case, the tables `tbl_rnd` and `tbl_acceptor` end up realizing a pipeline that reads and writes the same shared registers. However, depending on the target, the pipeline might be mapped by the compiler to separate memory or processing areas that cannot communicate, implying that our application would not be supported in practice. It would be helpful to have "annotations" to give hints regarding tables and registers that should be co-located.

Although the standard Paxos protocol, as described in this section, does not rely on message ordering, several optimizations do. [56,107,165] One could imagine modifying the data-plane to enforce ordering constraints in switch hardware. However, there are currently no primitives in P4 that would allow a programmer to control packet ordering.

Impact on Paxos Protocol Consensus protocols typically assume that the network provides point-to-point communication, and nothing else. As a result, most consensus protocols make weak assumptions about network behavior, and therefore, incur overhead to compensate for potential message loss or re-ordering. However, advances in network hardware programmability have laid a foundation for designing new consensus protocols which leverage assumptions about network computing power and behavior in

order to optimize performance.

One potentially fruitful direction would be to take a cue from systems like Fast Paxos [107] and Speculative Paxos, [165] which take advantage of "spontaneous message ordering" to implement low-latency consensus. Informally, spontaneous message order is the property that messages sent to a set of destinations will reach these destinations in the same order with high probability. This can be achieved with a careful network configuration [165] or in local-area networks when communication is implemented with IP-multicast. [163]

By moving part of the functionality of Paxos and its variations to switches, protocol designers can explore different optimizations. A switch could improve the chances of spontaneous message ordering and thereby increase the likelihood that Fast Paxos can reach consensus within few communication steps. (i.e. low latency) Moreover, if switches can store and retrieve values, one could envision an implementation of Disk Paxos [69] that relies on stateful switches, instead of storage devices. This would require a redesign of Disk Paxos since the storage space one can expect from a switch is much smaller than traditional storage.

3.6 Summary

FPGAs offer performance that far exceeds software running on general purpose CPUs, while offering a degree of flexibility that would be difficult to achieve on other targets, such as ASICs. However, they are also notoriously difficult to program. P4FPGA lowers the barrier to entry for using this powerful hardware, giving programmers a programmable substrate for create innovative new protocols and applications.

The advent of flexible hardware and expressive dataplane programming languages will have a profound impact on networks. One possible use of this emerging technology is to move logic traditionally associated with the application layer into the network itself. In the case of Paxos, and similar consensus protocols, this change could dramatically improve the performance of data center infrastructures.

P4FPGA provides a P4-to-FPGA compiler and runtime that is flexible, portable, and efficient. It supports multiple architectures, generates code that runs on Xilinx or Altera FPGAs and runs at line-rate with latencies comparable to commercial ASICs. P4FPGA is open source and publicly available for use. Indeed, it has already been used by two research projects to evaluate P4 programs on hardware. We hope that this tool will help other users in real environments or to support systems research.

```
1 header_type paxos_metadata_t {
2     fields {
3         rnd : 8;
4     }
5 }
6
7 metadata paxos_metadata_t meta_paxos;
8
9 register swid {
10     width      : 64;
11     inst_count : 1;
12 }
13
14 register rnds {
15     width      : 8;
16     inst_count : NUM_INST;
17 }
18
19 register vrnds {
20     width      : 8;
21     inst_count : NUM_INST;
22 }
23
24 register values {
25     width      : VALUE_SIZE;
26     inst_count : NUM_INST;
27 }
28
29 action read_rnd() {
30     register_read(meta_paxos.rnd, rnds, paxos.inst);
31 }
32
33 action handle_1a() {
34     modify_field(paxos.msgtype, PAXOS_1B);
35     register_read(paxos.vrnd, vrnds, paxos.inst);
36     register_read(paxos.value, values, paxos.inst);
37     register_read(paxos.swid, switch_id, 0);
38     register_write(rnds, paxos.inst, paxos.rnd);
39 }
```

```
40
41 action handle_2a() {
42     modify_field(paxos.msgtype, PAXOS_2B);
43     register_read(paxos.swid, switch_id, 0);
44     register_write(rnds, paxos.inst, paxos.rnd);
45     register_write(vrnds, paxos.inst, paxos.rnd);
46     register_write(values, paxos.inst, paxos.value);
47 }
48
49 table tbl_rnd { actions { read_rnd; } }
50
51 table tbl_acceptor {
52     reads   { paxos.msgtype : exact; }
53     actions { handle_1a; handle_2a; _drop; }
54 }
55
56 control ingress {
57     /* process other headers */
58     if (valid(paxos)) {
59         apply(tbl_rnd);
60         if (paxos.rnd > meta_paxos.rnd) {
61             apply(tbl_acceptor);
62          } else apply(tbl_drop);
63      }
64 }
```

Figure 3.15: Acceptor code.

CHAPTER 4

TOWARDS A PROGRAMMABLE NETWORK DATAPLANE: SONIC AND PROGRAMMABLE PHYS

This work appears in *Precise Realtime Software Access and Control of Wired Networks* in NSDI 2013, with co-authors Ki Suh Lee and Hakim Weatherspoon.

Precise timestamping and pacing can offer untapped potential to network applications. For instance, precise timestamping and pacing can accurately estimate available bandwidth [122, 124, 188], characterize network traffic [86, 115, 199], and create, detect and prevent covert timing channels [46, 126, 127]. In this chapter, we demonstrate that a programmable PHY can improve the precision of timestamping and pacing significantly and, as a result, the performance of network applications. A programmable PHY allows users to control and access every single bit between any two Ethernet frames. Considering each bit in 10 GbE is about 100 picosecond wide, controlling the number of bits between Ethernet frames in the PHY means users can precisely pace packets with sub-nanosecond precision. Similarly, counting the number of bits between Ethernet frames means users can precisely timestamp packets with sub-nanosecond precision.

The physical layer is usually implemented in hardware and not easily accessible to users. As a result, it is often treated as a black box. Further, commodity network interface cards (NICs) do not provide nor allow an interface for users to access the PHY. Consequently, operating systems cannot access the PHY either. Software access to the PHY is only enabled via special tools such as BiFocals [67] which uses physics equipment, including a laser and an oscilloscope.

As a new approach for accessing the PHY from software, we present SoNIC. SoNIC provides users with unprecedented flexible realtime access to the PHY from

software. In essence, all of the functionality in the PHY that manipulate bits are implemented in software. SoNIC consists of commodity off-the-shelf multi-core processors and a field-programmable gate array (FPGA) development board with peripheral component interconnect express (PCIe) Gen 2.0 bus. High-bandwidth PCIe interfaces and powerful FPGAs can support full bidirectional data transfer for two 10 GbE ports. Further, we created and implemented optimized techniques to achieve not only high-performance packet processing, but also high-performance 10 GbE bitstream control in software. Parallelism and optimizations allow SoNIC to process multiple 10 GbE bitstreams at line-speed.

With software access to the PHY, SoNIC provides the opportunity to improve upon and develop new network research applications which were not previously feasible. First, as a powerful network measurement tool, SoNIC can generate packets at full data rate with minimal interpacket delay. It also provides fine-grain control over the interpacket delay; it can inject packets with no variance in the interpacket delay. Second, SoNIC accurately captures packets at any data rate including the maximum, while simultaneously timestamping each packet with sub-nanosecond granularity. In other words, SoNIC can capture exactly what was sent. Further, this precise timestamping can improve the accuracy of research based on interpacket delay. For example, SoNIC can be used to profile network components. It can also create timing channels that are undetectable from software application and accurately estimate available bandwidth between two end hosts.

This chapter makes the following contributions. First, we present the design and implementation of SoNIC , a new approach for accessing the entire network stack in software in realtime. Second, we designed SoNIC with commodity components such as multi-core processors and a PCIe pluggable board, and present a prototype of SoNIC .

Third, we demonstrate that SoNIC can enable flexible, precise, and realtime network research applications. SoNIC increases the flexibility of packet generation and the accuracy of packet capture. Last, we also demonstrate that network research studies based on interpacket delay such as available bandwidth estimation and clock synchronization can be significantly improved with SoNIC .

4.1 Design

The design goals of SoNIC are to provide 1) access to the PHY in software, 2) realtime capability, 3) scalability and efficiency, 4) precision, and 5) user interface. As a result, SoNIC must allow users realtime access to the PHY in software, provide an interface to applications, process incoming packets at line-speed, and be scalable. Our ultimate goal is to achieve the same flexibility and control of the entire network stack for a wired network, as a software-defined radio [189] did for a wireless network, while maintaining the same level of precision as BiFocals [67]. Access to the PHY can then enhance the accuracy of network research based on interpacket delay. In this section, we discuss the design of SoNIC and how it addresses the challenges.

4.1.1 Access to the PHY in software

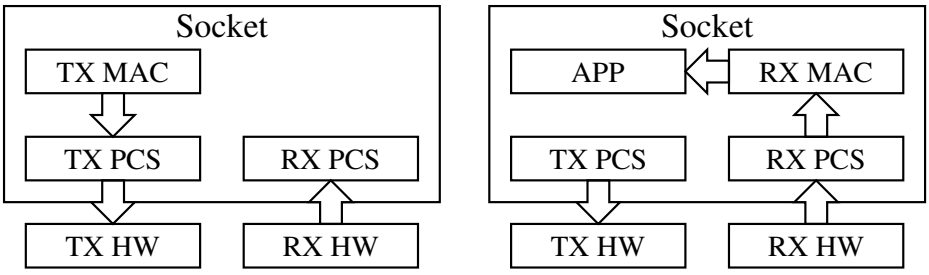
An application must be able to access the PHY in software using SoNIC . Thus, our solution must implement the bit generation and manipulation functionality of the PHY in software. The transmission and reception of bits can be handled by hardware. We carefully examined the PHY to determine an optimal partitioning of functionality between hardware and software.

The PMD and PMA sublayers of the PHY do not modify any bits or change the clock rate. They simply forward the symbol stream/bitstream to other layers. Similarly, PCS1 only converts the bit width (gearbox), or identifies the beginning of a new 64/66 bit block (blocksync). Therefore, the PMD, PMA, and PCS1 are all implemented in hardware as a forwarding module between the physical medium and SoNIC 's software component (See Figure B.1). Conversely, PCS2 (scramble/descramble) and PCS3 (encode/decode) actually manipulate bits in the bitstream and so they are implemented in SoNIC 's software component. SoNIC provides full access to the PHY in software; as a result, all of the functionality in the PHY that manipulate bits (PCS2 and PCS3) are implemented in software.

For this partitioning between hardware and software, we chose an Altera Stratix IV FPGA [3] development board from HiTechGlobal [11] as our hardware platform. The board includes a PCIe Gen 2 interface (=32 Gbps) to the host PC, and is equipped with two SFP+ (Small Form-factor Pluggable) ports (Figure 2.6a). The FPGA is equipped with 11.3 Gbps transceivers which can perform the 10 GbE PMA at line-speed. Once symbols are delivered to a transceiver on the FPGA they are converted to bits (PMA), and then transmitted to the host via PCIe by direct memory access (DMA).

4.1.2 Realtime Capability

Toachieve realtime, it is important to reduce any synchronization overheads between hardware and software, and between multiple pipelined cores. In SoNIC , the hardware does not generate interrupts when receiving or transmitting. Instead, the software decides when to initiate a DMA transaction by *polling* a value from a shared data memory structure where only the hardware writes. This approach is called *pointer polling* and



(a) Packet Generator (b) Packet Capturer

Figure 4.1: Example usages of SoNIC .

is better than interrupts because there is always data to transfer due to the nature of continuous bitstreams in 10 GbE.

In order to synchronize multiple pipelined cores, a *chasing-pointer FIFO* from Sora [189] is used which supports low-latency pipelining. The FIFO removes the need for a shared synchronization variable and instead uses a flag to indicate whether a FIFO entry is available to reduce the synchronization overheads. In our implementation, we improved the FIFO by avoiding memory operations as well. Memory allocation and page faults are expensive and must be avoided to meet the realtime capability. Therefore, each FIFO entry in SoNIC is pre-allocated during initialization. In addition, the number of entries in a FIFO is kept small so that the amount of memory required for a port can fit into the shared L3 cache.

We use the Intel Westmere processor to achieve high performance. Intel Westmere is a Non-Uniform Memory Access (NUMA) architecture that is efficient for implementing packet processing applications [63, 79, 135, 174]. It is further enhanced by a instruction `PCLMULQDQ`. `PCLMULQDQ`instruction performs carry-less multiplication and we use it to implement a fast CRC algorithm [72] that the MAC requires. Using `PCLMULQDQ`instruction makes it possible to implement a CRC engine that can process 10 GbE bits at line-speed on a single core.

4.1.3 Scalability and Efficiency

The FPGA board we use is equipped with two physical 10 GbE ports and a PCIe interface that can support up to 32 Gbps. Our design goal is to support two physical ports per board. Consequently, the number of CPU cores and the amount of memory required for one port must be bounded. Further, considering the intense computation required for the PCS and MAC, and that processors come with four to six or even eight cores per socket, our goal is to limit the number of CPU cores required per port to the number of cores available in a socket. As a result, for one port we implement four dedicated kernel threads each running on different CPU cores. We use a PCS thread and a MAC thread on both the transmit and receive paths. We call our threads: TX PCS, RX PCS, TX MAC and RX MAC. Interrupt requests (IRQ) are re-routed to unused cores so that SoNIC threads do not give up the CPU and can meet the realtime requirements.

Additionally, we use memory very efficiently: DMA buffers are preallocated and reused and data structures are kept small to fit in the shared L3 cache. Further, by utilizing memory efficiently, dedicating threads to cores, and using multi-processor QPI support, we can linearly increase the number of ports with the number of processors. QPI provides enough bandwidth to transfer data between sockets at a very fast data rate (> 100 Gbps).

A significant design issue still abounds: communication and CPU core utilization. The way we pipeline CPUs, i.e. sharing FIFOs depends on the application. In particular, we pipeline CPUs differently depending on the application to reduce the number of active CPUs; unnecessary CPUs are returned to OS. Further, we can enhance communication with a general rule of thumb: Taking advantage of the NUMA architecture and L3 cache and placing closely related threads on the same CPU socket.

Figure 4.1 illustrates examples of how to share FIFOs among CPUs. An arrow is a shared FIFO. For example, a packet generator only requires TX elements (Figure 4.1a); RX PCS simply receives and discards bitstreams, which is required to keep a link active. On the contrary, a packet capturer requires RX elements (Figure 4.1b) to receive and capture packets. TX PCS is required to establish and maintain a link to the other end by sending `/I/`s. To create a network profiling application, both the packet generator and packet capturer can run on different sockets simultaneously.

4.1.4 Precision

As discussed in Section 4.1.1, the PCS2 and PCS3 are implemented in software. Consequently, the software receives the entire raw bitstream from the hardware. While performing PCS2 and PCS3 functionalities, a PCS thread records the number of bits in between and within each Ethernet frame. This information can later be retrieved by a user application. Moreover, SoNIC allows users to precisely control the number of bits in between frames when transmitting packets, and can even change the value of any bits. For example, we use this capability to give users fine-grain control over packet generators and can even create virtually undetectable covert channels.

4.1.5 User Interface

SoNIC exposes fine-grained control over the path that a bitstream travels in software. SoNIC uses the `ioctl`system call for control, and the character device interface to transfer information when a user application needs to retrieve data. Moreover, users can assign which CPU cores or socket each thread runs on to optimize the path.

```
#include "sonic.h"

struct sonic_pkt_gen_info info = {
    .pkt_num   = 1000000000UL,
    .pkt_len   = 1518,
    .mac_src   = "00:11:22:33:44:55",
    .mac_dst   = "aa:bb:cc:dd:ee:ff",
    .ip_src    = "192.168.0.1",
    .ip_dst    = "192.168.0.2",
    .port_src  = 5000,
    .port_dst  = 5000,
    .idle         = 12, };

fd1 = open(SONIC_CONTROL_PATH, O_RDWR);
fd2 = open(SONIC_PORT1_PATH, O_RDONLY);

ioctl(fd1, SONIC_IOC_RESET)
ioctl(fd1, SONIC_IOC_SET_MODE, SONIC_PKT_GEN_CAP)
ioctl(fd1, SONIC_IOC_PORT0_INFO_SET, &info)
ioctl(fd1, SONIC_IOC_RUN, 10)

while ((ret = read(fd2, buf, 65536)) > 0) {
        // process data
 }

close(fd1);
close(fd2);
```

Figure 4.2: Packet Generator and Capturer.

To allow further flexibility, SoNIC allows additional application-specific threads, called APP threads, to be pipelined with other threads. A character device is used to communicate with these APP threads from userspace. For instance, users can implement a logging thread pipelined with receive path threads (RX PCS and/or RX MAC). Then the APP thread can deliver packet information along with precise timing information to userspace via a character device interface. There are two constraints that an APP thread must always meet: Performance and pipelining. First, whatever functionality is implemented in an APP thread, it must be able to perform it faster than 10.3125 Gbps

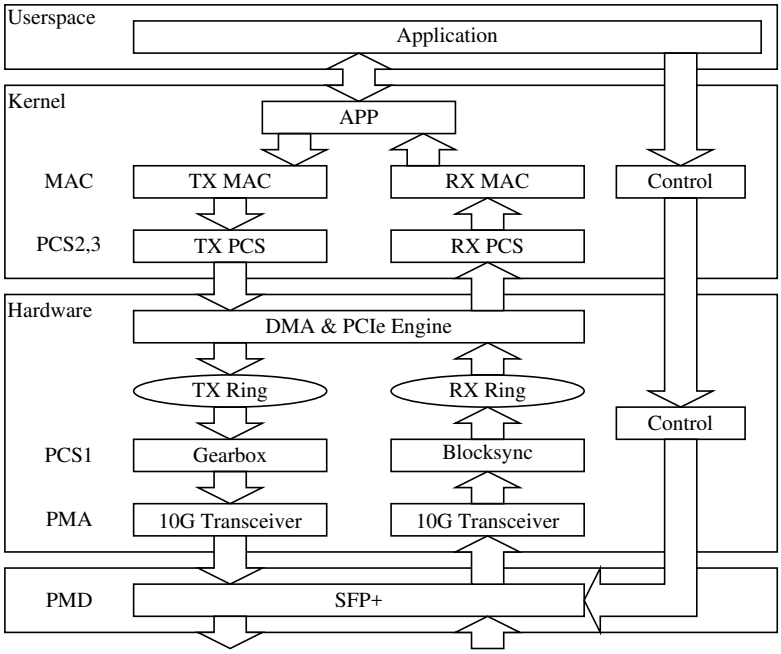


Figure 4.3: SoNIC architecture.

for any given packet stream in order to meet the realtime capability. Second, an APP thread must be properly pipelined with other threads, i.e. input/output FIFO must be properly set. Currently, SoNIC supports one APP thread per port.

Figure 4.2 illustrates the source code of an example use of SoNIC as a packet generator and capturer. After `SONIC_IOC_SET_MODE` is called (line 18), threads are pipelined as illustrated in Figure 4.1a and 4.1b. After `SONIC_IOC_RUN` command (line 20), port 0 starts generating packets given the information from `info` (line 3-12) for 10 seconds (line 20) while port 1 starts capturing packets with very precise timing information. Captured information is retrieved with `read` system calls (line 22-23) via a character device. As a packet generator, users can set the desired number of `/I/`sbetween packets (line 12). For example, twelve `/I/`characters will achieve the maximum data rate. Increasing the number of `/I/`characters will decrease the data rate.

4.1.6 Discussion

We have implemented SoNIC to achieve the design goals described above, namely, software access to the PHY, realtime capability, scalability, high precision, and an interactive user interface. Figure 4.3 shows the major components of our implementation. From top to bottom, user applications, software as a loadable Linux kernel module, hardware as a firmware in FPGA, and a SFP+ optical transceiver. Although Figure 4.3 only illustrates one physical port, there are two physical ports available in SoNIC . SoNIC software consists of about 6k lines of kernel module code, and SoNIC hardware consists of 6k lines of Verilog code excluding auto-generated source code by Altera Quartus [2] with which we developed SoNIC 's hardware modules.

The idea of accessing the PHY in software can be applied to other physical layers with different speeds. The 1 GbE and 40 GbE PHYs are similar to the 10 GbE PHY in that they run in full duplex mode, and maintain continuous bitstreams. Especially, the 40GbE PCS employees four PCS lanes that implements 64B/66B encoding as in the 10GbE PHY. Therefore, it is possible to access the PHYs of them with appropriate clock cycles and hardware supports. However, it might not be possible to implement four times faster scrambler with current CPUs.

4.2 Implementation

Performance is paramount for SoNIC to achieve its goals and allow software access to the entire network stack. In this section we discuss the software (Section 4.2.1) and hardware (Section 4.2.2) optimizations that we employ to enable SoNIC. Further, we evaluate each optimization (Sections 4.2.1 and 4.2.2) and demonstrate that they help to

enable SoNIC and network research applications (Section 4.3) with high performance.

4.2.1 Software Optimizations

MAC Thread Optimizations As stated in Section 4.1.2, we use `PCLMULQDQ`instruction which performs carry-less multiplication of two 64-bit quadwords [75] to implement the fast CRC algorithm [72]. The algorithm *folds* a large chunk of data into a smaller chunk using the `PCLMULQDQ`instruction to efficiently reduce the size of data. We adapted this algorithm and implemented it using inline assembly with optimizations for small packets.

PCS Thread Optimizations Considering there are 156 million 66-bit blocks a second, the PCS must process each block in less than 6.4 nanoseconds. Our optimized (de-)scrambler can process each block in 3.06 nanoseconds which even gives enough time to implement decode/encode and DMA transactions within a single thread.

In particular, the PCS thread needs to implement the (de-)scrambler function, $G(x) = 1 + x^{39} + x^{58}$, to ensure that a mix of 1's and 0's are always sent (DC balance). The (de-)scrambler function can be implemented with Algorithm 1, which is very computationally expensive [67] taking 320 shift and 128 xor operations (5 shift operations and 2 xors per iteration times 64 iterations). In fact, our original implementation of Algorithm 1 performed at 436 Mbps, which was not sufficient and became the bottleneck for the PCS thread. We optimized and reduced the scrambler algorithm to a *total* of 4 shift and 4 xor operations (Algorithm 2) by carefully examining how hardware implements the scrambler function [198]. Both Algorithm 1 and 2 are equivalent, but Algorithm 2 runs 50 times faster (around 21 Gbps).

Algorithm 1 Scrambler

$s \leftarrow$ state
$d \leftarrow$ data
for $i = 0 \rightarrow 63$ **do**
  $in \leftarrow (d >> i)$ & 1
  $out \leftarrow (in \oplus (s >> 38) \oplus (s >> 57))$ & 1
  $s \leftarrow (s << 1) \mid out$
  $r \leftarrow r \mid (out << i)$
  state $\leftarrow s$
end for

Algorithm 2 Parallel Scrambler

$s \leftarrow$ state
$d \leftarrow$ data
$r \leftarrow (s >> 6) \oplus (s >> 25) \oplus d$
$r \leftarrow r \oplus (r << 39) \oplus (r << 58)$
state $\leftarrow r$

Memory Optimizations We use *packing* to further improve performance. Instead of maintaining an array of data structures that each contains metadata and a pointer to the packet payload, we pack as much data as possible into a preallocated memory space: Each packet structure contains metadata, packet payload, and an offset to the next packet structure in the buffer. This packing helps to reduce the number of page faults, and allows SoNIC to process small packets faster. Further, to reap the benefits of the `PCLMULQDQ`instruction, the first byte of each packet is always 16-byte aligned.

Evaluation We evaluated the performance of the TX MAC thread when computing CRC values to assess the performance of the fast CRC algorithm and packing packets we implemented relative to batching an array of packets. For comparison, we computed the theoretical maximum throughput (Reference throughput) in packets per second (pps) for any given packet length (i.e. the pps necessary to achieve the maximum throughput of 10 Gbps less any protocol overhead).

If only one packet is packed in the buffer, packing will perform the same as batching

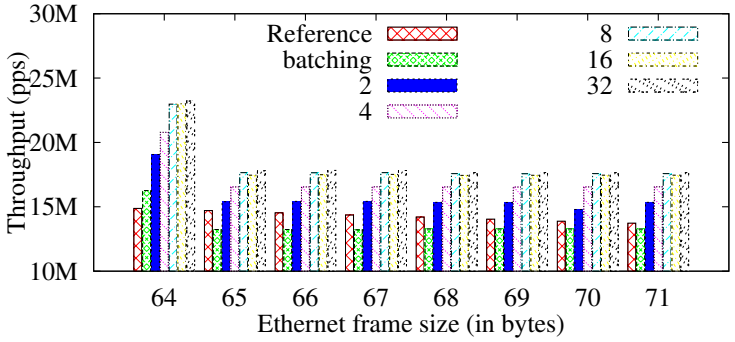


Figure 4.4: Throughput of packing

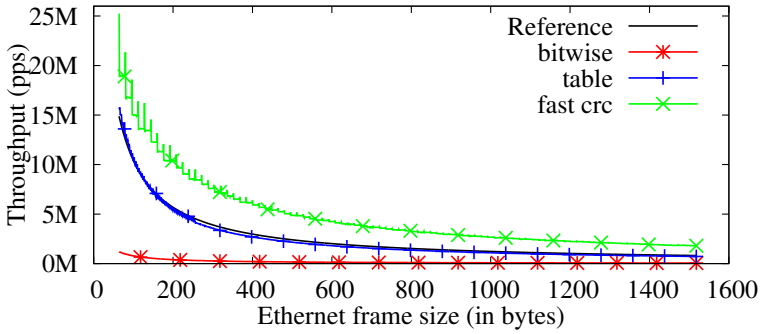


Figure 4.5: Throughput of different CRC algorithms.

since the two are essentially the same in this case. We doubled the factor of packing from 1 to 32 and assessed the performance of packing each time, i.e. we doubled the number of packets written to a single buffer. Figure 4.4 shows that packing by a factor of 2 or more always outperforms the Reference throughput and is able to achieve the max throughput for small packets while batching does not.

Next, we compared our fast CRC algorithm against two CRC algorithms that the Linux Kernel provides. One of the Linux CRC algorithms is a naive bit computation and the other is a table lookup algorithm. Figure 4.5 illustrates the results of our comparisons. The x-axis is the length of packets tested while the y-axis is the throughput. The Reference line represents the maximum possible throughput given the 10 GbE standard. Packet lengths range the spectrum of sizes allowed by 10 GbE standard from 64

bytes to 1518 bytes. For this evaluations, we allocated 16 pages *packed* with packets of the same length and computed CRC values with different algorithms for 1 second. As we can see from Figure 4.5, the throughput of the table lookup closely follows the Reference line; however, for several packet lengths, it underperforms the Reference line and is unable to achieve the maximum throughput. The fast CRC algorithm, on the other hand, outperforms the Reference line and target throughput for all packet sizes.

Lastly, we evaluated the performance of pipelining and using multiple threads on the TX and RX paths. We tested a full path of SoNIC to assess the performance as packets travel from the TX MAC to the TX PCS for transmission and up the reverse path for receiving from the RX PCS to the RX MAC and to the APP (as a logging thread). All threads perform better than the Reference target throughput. The overhead of FIFO is negligible when we compare the throughput of individual threads to the throughput when all threads are pipelined together. Moreover, when using two ports simultaneously (two full instances of receive and transmit SoNIC paths), the throughput for both ports achieve the Reference target maximum throughput.

4.2.2 Hardware Optimizations

DMA Controller Optimizations Given our desire to transfer large amounts of data (more than 20 Gbps) over the PCIe, we implemented a high performance DMA controller. There are two key factors that influenced our design of the DMA controller. First, because the incoming bitstream is a continuous 10.3125 Gbps, there must be enough buffering inside the FPGA to compensate for a transfer latency. Our implementation allocates four rings in the FPGA for two ports (Figure 4.3 shows two of the rings for one port). The maximum size of each ring is 256 KB, with the size being

limited by the amount of SRAM available on hardware.

The second key factor we needed to consider was the efficient utilization of bus bandwidth. The DMA controller operates at a data width of 128 bits. If we send a 66-bit data block over the 128-bit bus every clock cycle, we will waste 49% of the bandwidth, which was not acceptable. To achieve more efficient use of the bus, we create a data structure and separated the syncheader from the packet payload before storing a 66-bit block in the data structure. Sixteen two-bit syncheaders are concatenated together to create a 32-bit integer and stored in the `syncheaders` field of the data structure. The 64-bit packet payloads associated with these syncheaders are stored in the `payloads` field of the data structure. For example, the ith 66-bit PCS block from a DMA page consists of the two-bit sync header from `syncheaders[`$i/16$`]` and the 64-bit payload from `payloads[`i`]`. With this data structure there is a 32-bit overhead for every page, however it does not impact the overall performance.

PCI Express Engine Optimizations When SoNIC was first designed, it only supported a single port. As we scaled SoNIC to support multiple ports simultaneously, the need for multiplexing traffic among ports over the single PCIe link became a significant issue. To solve this issue, we employ a two-level arbitration scheme to provide fair arbitration among ports. A lower level arbiter is a fixed-priority arbiter that works within a single port and arbitrates between four basic Transaction Level Packet (TLP) types: Memory, I/O, configuration, and message. The TLPs are assigned with fixed priority in favor of the write transaction towards the host. The second level arbiter implements a virtual channel, where the Traffic Class (TC) field of TLP's are used as de-multiplexing keys. We implemented our own virtual channel mechanism in SoNIC instead of using the one available in the PCIe stack since virtual channel support is an optional feature for vendors to comply with. In fact, most chipsets on the market at the time of this dissertation

Configuration	Same Socket?	# pages	Throughput (RX)		# pages	Throughput (TX)		Realtime?
			Port 0	Port 1		Port 0	Port 1	
Single RX		16	25.7851					
Dual RX	Yes	16	13.9339	13.899				
	No	8	14.2215	13.134				
Single TX					16	23.7437		
Dual TX	Yes				16	14.0082	14.048	
	No				16	13.8211	13.8389	
Single RX/TX		16	21.0448		16	22.8166		
Dual RX/TX	Yes	4	10.7486	10.8011	8	10.6344	10.7171	No
		4	11.2392	11.2381	16	12.384	12.408	Yes
		8	13.9144	13.9483	8	9.1895	9.1439	Yes
		8	14.1109	14.1107	16	10.6715	10.6731	Yes
	No	4	10.5976	10.183	8	10.3703	10.1866	No
		4	10.9155	10.231	16	12.1131	11.7583	Yes
		8	13.4345	13.1123	8	8.3939	8.8432	Yes
		8	13.4781	13.3387	16	9.6137	10.952	Yes

Table 4.1: DMA throughput. The numbers are average over eight runs. The delta in measurements was within 1% or less.

do not support the virtual channel mechanism. By implementing the virtual channel support in SoNIC , we achieve better portability since we do not rely on chip vendors that enable PCI arbitration.

Evaluation

We examined the maximum throughput for DMA between SoNIC hardware and SoNIC software to evaluate our hardware optimizations. It is important that the bidirectional data rate of each port of SoNIC is greater than 10.3125 Gbps. For this evaluation, we created a DMA descriptor table with one entry, and changed the size of memory for each DMA transaction from one page (4K) to sixteen pages (64KB), doubling the number of pages each time. We evaluated the throughput of a single RX or TX transaction, dual RX or TX transactions, and full bidirectional RX and TX transactions with both one and two ports (see the rows of Table 4.1). We also measured the throughput when traffic was sent to one or two CPU sockets.

Table 4.1 shows the DMA throughput of the transactions described above. We first measured the DMA without using pointer polling (see Section 4.1.2) to obtain the maximum throughput of the DMA module. For single RX and TX transactions, the maximum throughput is close to 25 Gbps. This is less than the theoretical maximum throughput of 29.6 Gbps for the x8 PCIe interface, but closely matches the reported maximum throughput of 27.5 Gbps [1] from Altera design. Dual RX or TX transactions also resulted in throughput similar to the reference throughput of Altera design.

Next, we measured the full bidirectional DMA transactions for both ports varying the number of pages again. As shown in the bottom half of Table 4.1, we have multiple configurations that support throughput greater than 10.3125 Gbps for full bidirections. However, there are a few configurations in which the TX throughput is less than 10.3125 Gbps. That is because the TX direction requires a small fraction of RX bandwidth to fetch the DMA descriptor. If RX runs at maximum throughput, there is little room for the TX descriptor request to get through. However, as the last column on the right indicates these configurations are still able to support the realtime capability, i.e. consistently running at 10.3125 Gbps, when *pointer polling* is enabled. This is because the RX direction only needs to run at 10.3125 Gbps, less than the theoretical maximum throughput (14.8 Gbps), and thus gives more room to TX. On the other hand, two configurations where both RX and TX run faster than 10.3125 Gbps for full bidirections are not able to support the realtime capability. For the rest of the chapter, we use 8 pages for RX DMA and 16 pages for TX DMA.

4.3 Evaluation

How can SoNIC enable flexible, precise and novel network research applications? Specifically, what unique value does software access to the PHY buy? SoNIC can literally count the number of bits between and within packets, which can be used for timestamping at the sub-nanosecond granularity (again each bit is 97 ps wide, or about ~0.1 ns). At the same time, access to the PHY allows users control over the number of `idles` (`/I/`s) between packets when generating packets. This fine-grain control over the `/I/`smeans we can precisely control the data rate and the distribution of interpacket gaps. For example, the data rate of a 64B packet stream with uniform 168 `/I/`sis 3 Gbps. When this precise packet generation is combined with exact packet capture, also enabled by SoNIC , we can improve the accuracy of any research based on interpacket delays [46, 86, 115, 122, 124, 126, 127, 199, 207].

In this section, we demonstrate SoNIC's accurate packet generation capability in Section 4.3.1 and packet capture capability in Section 4.3.2, which are unique contributions and can enable unique network research in and of themselves given both the flexibility, control, and precision. Further, we demonstrate that SoNIC can precisely and flexibly characterize and profile commodity network components like routers, switches, and NICs. Section 4.3.3 discusses the profiling capability enabled by SoNIC .

4.3.1 Packet Generator

Packet generation is important for network research. It can stress test end-hosts, switches/routers, or a network itself. Moreover, packet generation can be used for replaying a trace, studying distributed denial of service (DDoS) attacks, or probing firewalls.

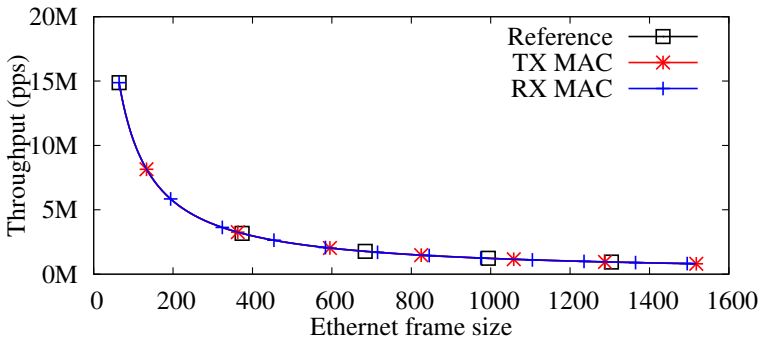


Figure 4.6: Throughput of packet generator and capturer.

In order to claim that a packet generator is accurate, packets need to be crafted with fine-grained precision (minimum deviations in IPD) at the maximum data rate. However, this fine-grained control is not usually exposed to users. Further, commodity servers equipped with a commodity NIC often does not handle small packets efficiently and require batching [63,79,135,174]. Thus, the sending capability of servers/software-routers are determined by the network interface devices. Myricom Sniffer 10G [18] provides line-rate packet injection capability, but does not provide fine-grained control of IPGs. Hardware based packet generators such as ALT10G can precisely control IPGs, but do not provide any interface for users to flexibly control them.

We evaluated SoNIC as a packet generator (Figure 2.8a and 4.1a). Figure 4.7 compares the performance of SoNIC to that of Sniffer 10G. Note, we do not include ALT10G in this evaluation since we could not control the IPG to generate packets at 9 Gbps. We used two servers with Sniffer 10G enabled devices to generate 1518B packets at 9 Gbps between them. We split the stream so that SoNIC can capture the packet stream in the middle (we describe this capture capability in the following section). As the graph shows, Sniffer 10G allows users to generate packets at desired data rate, however, it does not give the control over the IPD; that is, 85.65% packets were sent in a burst (instantaneous 9.8 Gbps and minimum IPG (14 `/I/`s)). SoNIC , on the other hand, can generate packets with uniform distribution. In particular, SoNIC generated packets with

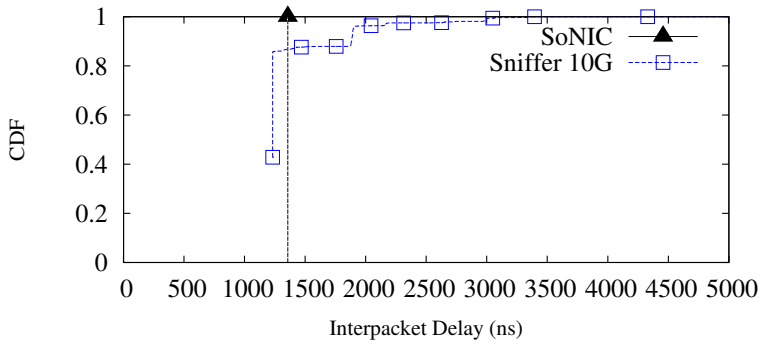


Figure 4.7: Comparison of packet generation at 9 Gbps.

no variance for the IPD (i.e. a single point on the CDF, represented as a triangle). Moreover, the maximum throughput perfectly matches the Reference throughput (Figure 4.6) while the TX PCS consistently runs at 10.3125 Gbps (which is not shown). In addition, we observed no packet loss, bit errors, or CRC errors during our experiments.

SoNIC packet generator can easily achieve the maximum data rate, and allows users to precisely control the number of `/I/`sto set the data rate of a packet stream. Moreover, with SoNIC , it is possible to inject less `/I/`sthan the standard. For example, we can achieve 9 Gbps with 64B packets by inserting only eight `/I/`sbetween packets. This capability is not possible with any other (software) platform. In addition, if the APP thread is carefully designed, users can flexibly inject a random number of `/I/`sbetween packets, or the number of `/I/`sfrom captured data. SoNIC packet generator is thus by far the most flexible and highest performing.

4.3.2 Packet Capturer

A packet capturer (a.k.a. packet sniffer, or packet analyzer) plays an important role in network research; it is the opposite side of the same coin as a packet generator. It can record and log traffic over a network which can later be analyzed to improve the perfor-

mance and security of networks. In addition, capturing packets with precise timestamping is important for High Frequency Trading [99, 176] or latency sensitive applications.

Similar to the sending capability, the receiving capability of servers and software routers is inherently limited by the network adapters they use; it has been shown that some NICs are not able to receive packets at line speed for certain packet sizes [174]. Furthermore, if batching is used, timestamping is significantly perturbed if done in kernel or userspace [67]. High-performance devices such as Myricom Sniffer10G [18, 92] provide the ability of sustained capture of 10 GbE by bypassing kernel network stack. It also provides timestamping at 500 ns resolution for captured packets. SoNIC , on the other hand, can receive packets of any length at line-speed with precise timestamping.

Putting it all together, when we use SoNIC as a packet capturer (Figure 2.8a and 4.1b), we are able to receive at the full Reference data rate (Figure 4.6). For the APP thread, we implemented a simple logging application which captures the first 48 bytes of each packet along with the number of `/I/`sand bits between packets. Because of the relatively slow speed of disk writes, we store the captured information in memory. This requires about 900MB to capture a stream of 64 byte packets for 1 second, and 50 MB for 1518 byte packets. We use ALT10G to generate packets for 1 second and compare the number of packets received by SoNIC to the number of packets generated.

SoNIC has perfect packet capture capabilities with flexible control in software. In particular, Figure 4.8 shows that given a 9 Gbps generated traffic with uniform IPD (average IPD=1357.224ns, stdev=0), SoNIC captures what was sent; this is shown as a single triangle at (1357.224, 1). All the other packet capture methods within userspace, kernel or a mixture of hardware timestamping in userspace (Sniffer 10G) failed to accurately capture what was sent. We receive similar results at lower bandwidths as well.

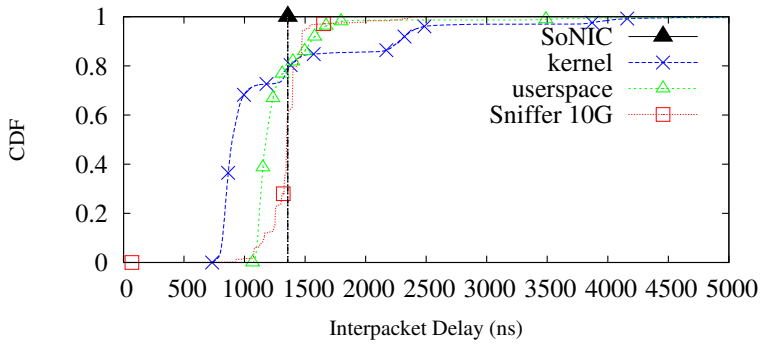


Figure 4.8: Comparison of timestamping.

4.3.3 Profiler

Interpacket delays are a common metric for network research. It can be used to estimate available bandwidth [122, 124], increase TCP throughput [207], characterize network traffic [86,115,199], and detect and prevent covert timing channels [46,126,127]. There are a lot of metrics based on IPD for these areas. We argue that SoNIC can increase the accuracy of those applications because of its precise control and capture of IPDs. In particular, when the SoNIC packet generator and capturer are combined, i.e. one port transmits packets while the other port captures, SoNIC can be a flexible platform for various studies. As an example, we demonstrate how SoNIC can be used to profile network switches.

Switches can be generally divided into two categories: store-and-forward and cut-through switches. Store-and-forward switches decode incoming packets, buffers them before making a routing decision. On the other hand, cut-through switches route incoming packets before entire packets are decoded to reduce the routing latency. We generated 1518B packets with uniform 1357.19 ns IPD (=9 Gbps) to a Cisco 4948 (store-and-forward) switch and a IBM BNT G8264 (cut-through) switch. These switches show different characteristics as shown in Figure 4.9. The x-axis is the interpacket delay; the y-axis is the cumulative distribution function. The long dashed vertical line on the left

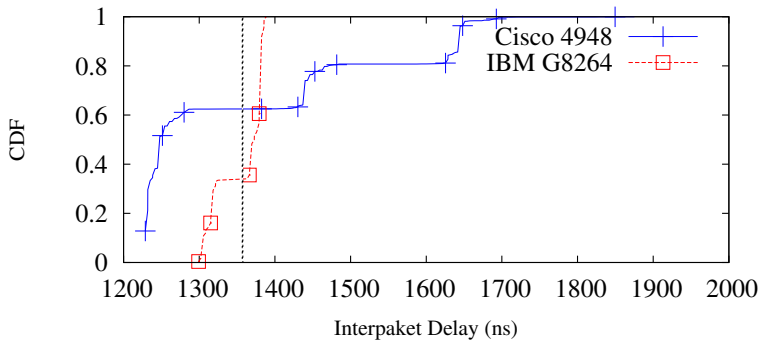


Figure 4.9: IPDs of Cisco 4948 and IBM G8264. 1518B packets at 9 Gbps.

is the original IPD injected to the packet stream.

There are several takeaways from this experiment. First, the IPD for generated packets had no variance; none. The generated IPD produced by SoNIC was *always* the same. Second, the cut-through switch introduces IPD variance (stdev=31.6413), but less than the IPD on the store-and-forward switch (stdev=161.669). Finally, the average IPD was the same for both switches since the data rate was the same: 1356.82 (cut-through) and 1356.83 (store-and-forward). This style of experiment can be used to profile and fingerprint network components as different models show different packet distributions.

4.4 Application: Measuring Available Bandwidth

This work appears in *Timing is everything: Accurate, Minimum-cost, Available Bandwidth Estimation in High-speed Wired Network* in IMC 2014, with co-authors Ki Suh Lee, Erluo Li, ChiunLin Lim, Hakim Weatherspoon, Ao Tang.

In this section, we present an application built on SoNIC to measure end-to-end available bandwidth of a network path. The concept of available bandwidth estimation stems from a simple observation: Send a train of probe packets through a net-

work path to momentarily congest the bottleneck link, then infer the available bandwidth at the receiving end from probe packet timestamps [84, 138, 173, 188]. Bandwidth estimation is important for designing high performant networked systems, improving network protocols, building distributed systems, and improving application performance [82, 182, 184].

Existing bandwidth estimation methods are intrusive, expensive, inaccurate, and does not work well with bursty cross traffic or on 10Gbps links [76, 130]. There are three significant problems with current approaches. First, such approaches are intrusive since load is being added to the network. Current measurement methods require creating explicit probe packets that consume bandwidth and CPU cycles. Second, current approaches often do not have enough fidelity to accurately generate or timestamp probe packets. By operating in userspace to maximize programming flexibility, precise control and measurement of packet timings is sacrificed. As a result, the probe packet timings are often perturbed by operating systems (OS) level activities. Without precise timestamps, accurate estimation becomes very challenging, especially in high-speed networks. Finally, Internet traffic is known to be bursty at a range of time scales [87], but previous studies have shown that existing estimation approaches perform poorly under bursty traffic conditions [197].

MinProbe performs available bandwidth estimation in real-time with minimal-cost, high-fidelity for high-speed networks (10 Gigabit Ethernet), while maintaining the flexibility of userspace control and compatibility with existing bandwidth estimation algorithms. MinProbe minimizes explicit probing costs by using application packets as probe packets whenever possible, while generating explicit packets only when necessary. MinProbe greatly increases measurement fidelity via software access to the wire and maintains wire-time timestamping of packets before they enter the host. Finally,

a userspace process controls the entire available bandwidth estimation process giving flexibility to implement new algorithms as well as reuse existing estimation algorithms. Importantly, MinProbe operates in real-time, such that it is useful as a networked systems building block.

MinProbe was designed with commodity components and achieves results that advance the state of the art. It can be built from a commodity server and an FPGA (field programmable gate array) PCIe (peripheral component interconnect express) pluggable board, SoNIC [113]. As a result, any server can be turned into a MinProbe middlebox. Indeed, we have turned several nodes and sites in the GENI (global environment for networking innovations) network into MinProbe middleboxes. It has been evaluated on a testbed that consists of multiple 10 Gigabit Ethernet (10 GbE) switches and on the wide-area Internet via the National Lambda Rail (NLR). It achieves high accuracy: Results illustrate available bandwidth estimation with errors typically no more than 0.4 Gbps in a 10 Gbps network. It estimates available bandwidth with minimal overhead: Available bandwidth estimates use existing network traffic as probes. Overall, MinProbe allows software programs and end-hosts to accurately measure the available bandwidth of high speed, 10 gigabit per second (Gbps), network paths even with bursty cross traffic, which is a unique contribution.

4.4.1 Background

Available bandwidth estimation is motivated by a simple problem: What is the maximum data rate that a user could send down a network path without going over capacity? This data rate is equal to the available bandwidth on the *tight link,* which has the minimum available bandwidth among all links on the network path. While the prob-

lem sounds simple, there are four main challenges to available bandwidth estimation—timeliness, accuracy, non-intrusiveness, and consistency. In particular, an available bandwidth estimation methodology and tool would ideally add as little overhead as possible and return timely and accurate estimation on the available bandwidth consistently across a wide range of Internet traffic conditions.

In this section, we first discuss many of the key ideas and assumptions underlying existing available bandwidth estimation methods and tools. Then, motivate the development of MinProbe by illustrating the limitations faced by these current methods and tools.

Methodology Many existing available bandwidth estimation tools take an end-to-end approach. A typical setup sends probe packets with a predefined interval along the path under measurement, and observe change in certain packet characteristics at the receiver to infer the amount of cross traffic in the network. The key idea to such inferences: When the probing rate exceeds the available bandwidth, the observed packet characteristics undergo a significant change. The turning point where the change occurs is then the estimated available bandwidth. See Figure 4.18, for example, the turning point where queuing delay variance increases is the available bandwidth estimate. Most existing available bandwidth estimation tools can be classified according to the packet characteristics and metrics used to observing a turning point.

Bandwidth estimation tools such as Spruce [188] and IGI [81] operate by observing the change in the output probing rate. The idea is that when the probing rate is below the available bandwidth, probe packets should emerge at the receiver with the same probing rate. However, once the probing rate exceeds the available bandwidth, the gap between packets will be stretched due to queuing and congestion, resulting in a lower output

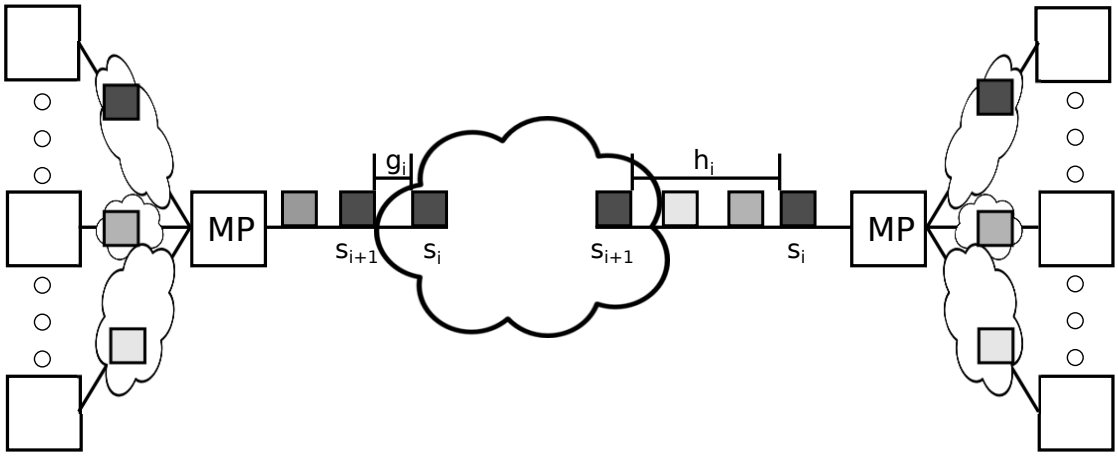


Figure 4.10: Usage of bandwidth estimation

probing rate. On the other hand, bandwidth estimation tools such as Pathload [84], PathChirp [173], TOPP [138], and Yaz [185] operate by observing the change in one-way delay (OWD), i.e. the time required to send a packet from source to destination. OWD increases when the probing rate exceeds the available bandwidth, as packets have to spend extra time in buffers.

An alternative methodology to the end-to-end approach would be to query every network element (switch/router) along a network path. A network administrator could collect the statistical counters from all related ports, via protocols such as sFlow [19]; or infer from Openflow control messages such as `PacketIn` and `FlowRemoved` messages [208]. However, obtaining an accurate, consistent, and timely reading from multiple switches in an end-to-end manner can be very difficult [9]. Further, collecting counters from switches is done at per-second granularity and requires network administrative privileges, which makes the approach less timely and useful for building distributed systems, improving network protocols, or improving application performance. As a result, we assume and compare to other end-to-end approaches for the rest of this section.

Limitations While useful, the end-to-end approach for available bandwidth estimation has inherent issues that limit its applicability, especially in high-capacity links. We outline a few of these challenges.

- **High-fidelity measurements require high-fidelity instruments.** Generating packet trains in userspace typically require a CPU-intensive busy-wait loop, which is prone to operating system level noise, such as OS scheduling, interrupt coalescing, and batching. As a result, generated interpacket gaps of a packet train rarely match the interpacket gaps specified by the userspace program. Moreover, timestamping in user or kernel space adds considerable error to the available bandwidth estimation. Even timestamping in hardware adds noise when timestamping packet (albeit, less noise than kernel space, but still enough to make available bandwidth estimation unusable in many situations [114]). The takeaway: Without high-fidelity capabilities, available bandwidth estimation lacks accuracy.
- **Non-negligible overhead.** Bandwidth estimation tools add traffic to the network path under measurement. This may adversely affect application traffic and measurement accuracy [160]. The amount of probe traffic is proportional to the rate of sampling and the number of concurrent measurement sessions. As a result, the effect of probe packets on cross traffic exacerbates with traffic increase.
- **Traffic burstiness impacts measurement results.** Bandwidth estimation results are sensitive to traffic burstiness. However, existing tools typically ignore the burstiness of cross traffic in favor of simple estimation algorithms. Limitations of batching and instrumentation further mask burstiness to some degree to userspace tools. As a result, existing tools may not be able to reflect true available bandwidth with bursty traffic.

In this section, we introduce a new tool that can be applied to existing available

bandwidth estimation algorithms and tools while overcoming the limitations discussed above. Indeed, we demonstrate that we can accurately estimate available bandwidth in high-speed networks with minimal overhead with existing algorithms. In the next section, we discuss the details of the design and implementation of our methodology.

4.4.2 Design

MinProbe measures the available bandwidth with high-fidelity, minimal-cost and in userspace; thus, enabling cheaper (virtually no overhead) and more accurate available bandwidth estimation. It achieves these goals through three main features:

First, MinProbe is a middlebox architecture that uses application network traffic as probe traffic to eliminate the need for explicit probe packets. When an application packet arrives at MinProbe , MinProbe decides whether the packet should be used as a probe; if so, MinProbe modulates the timings between application packets chosen as probe packets before forwarding them to their destination. A programmable flow table accessible from userspace controls the selection of an application packet as a probe packet. Thus, by using application traffic implicitly as available bandwidth probes, we are able to remove all the traditional costs and overheads. A similar idea was proposed in MGRP [160], which has the same goal of lower overhead, but MGRP lacks high-fidelity measurement capability.

Second, MinProbe is a high-fidelity network measurement substrate that is capable of modulating and capturing traffic timings with sub-nanosecond precision. The high-precision is achieved by enabling software access to the physical layer of the network protocol stack. When MinProbe modulates probe packets (application packets), it adds and removes minute spacings between packets through direct access to the physical

layer.

Finally, MinProbe is accessible from userspace. From userspace, users can control sub-nanosecond modulations between packets to generate probes and obtain sub-nanosecond timings between received packets to estimate available bandwidth. Further, all probes can be generated and timing measurements can be received in real-time from userspace.

We envision MinProbe to be deployed in an architecture where a separate control plane manages a rich set of middlebox functionalities with event trigger support(e.g [33]). In particular, we believe the separation of measurement hardware and production hardware enables high-fidelity measurement in high-speed networks (and in real-time) that is difficult to achieve otherwise.

Precise Probe Control MinProbe offers enhanced network measurement capabilities: High-fidelity packet pacing to generate probe packets and high-fidelity packet timestamping to measure received packet times. We achieve high-fidelity capabilities via direct access to the physical layer from software and in real-time. To see how MinProbe works, we describe how MinProbe takes advantage of the software access to the physical layer to achieve high-fidelity network measurement.

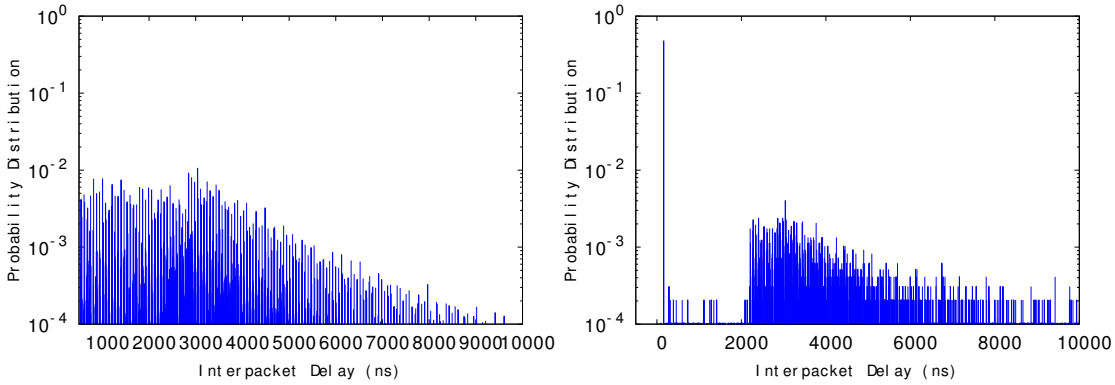
High-Fidelity Measurement The key insight and capability of how MinProbe achieves high-fidelity is from its direct access to the `/I/` characters in the physical layer. In particular, MinProbe can measure (count) and generate (insert or remove) an exact number of `/I/` characters between each subsequent probe packet to measure the relative time elapsed between packets or generate a desired interpacket gap. Further, if two subsequent probe packets are separated by packets from a different flow (*cross*

traffic), the gap between probe packets will include /I/characters as well as data characters of the cross traffic packets, but the measurement will still be exact (i.e. /I/and data characters represent time with sub-nanosecond precision). This level of access from software is unprecedented. Traditionally, however, an end host may timestamp packets in userspace, kernel, or network interface hardware which all add significant noise. None of these methods provide enough precision for high-fidelity network measurements; consequently, many existing bandwidth estimation tools report significant estimation error (large variation and low accuracy) [114].

In a similar fashion to measuring the space between packets, MinProbe generates probe packets through high-fidelity pacing, by inserting an exact spacing between packets. By accessing the physical layer of 10 GbE in software in real-time, MinProbe can insert or remove /I/characters from application traffic as needed. In particular, two variables of the probe traffic are of interest: The gap between packets and the overall rate of packet trains. Users can program MinProbe via command line calls or API calls to specify the number of /I/characters (i.e. the number of 100s of pico-seconds) to be maintained between subsequent probe packets, allowing userspace programs to perform high-fidelity pacing.

Direct access to /I/characters from software and in real-time differentiates MinProbe from other measurement tools. Specifically, MinProbe is able to characterize and generate an *exact* spacing (interpacket gap) between probe packets.

Explicit probes are not necessary for MinProbe . Similar to MGRP [160], MinProbe can use application traffic as probe packets. It has a programmable flow table that performs flow matching on pass-through traffic. Users can insert entries into the flow table to specify which flows are probes. Traffic that have a match in the flow table are modulated before they are forwarded. Other flows that do not have a match are



(a) original traffic (b) modulated traffic

Figure 4.11: Comparison of traffic pattern before and after passed through middlebox

simply forwarded without any change in timing. With MinProbe , we are able to perform line-rate forwarding at 10 Gbps, even for the minimum size packets. Moreover, MinProbe has the ability to generate explicit probe packets, especially when there is insufficient application traffic. Dummy probing packets are padded with zero bytes and transmitted at pre-defined intervals as programmed by the users.

Figure 4.11 illustrates MinProbe 's ability to perform high-fidelity measurement and pacing as a middlebox with only application packets as probes.

Figure 4.11a shows the distribution of incoming application traffic to a MinProbe middlebox. The x-axis is the distance in time between the first byte of subsequent packets (or *interpacket delay*), and y-axis is the frequency of occurrences. We generated two flows of application traffic at 1 Gbps, and a MinProbe middlebox was configured to only modulate one of the flows. As can be seen in Figure 4.11b, after MinProbe , the packets of the probing flow have minimal spacing (interpacket delay) between subsequent packets, and exhibit a much higher probe rate: The peak just to the right of 0 ns interpacket delay was the modulated probing traffic. Even though the overall application traffic rate was 1 Gbps, we were able to increase the instantaneous probe rate up to 10

Gbps for short periods of time by creating short packet trains with minimal interpacket delay. The packets of the other flows, on the other hand, were forwarded as-is, with no changes in timing.

How would MinProbe impact the application traffic performance? Obviously, if any traffic were paced, then MinProbe would change the pacing. But, for general TCP traffic, the minute modulation that MinProbe causes does not affect the rate or TCP throughput. Similar results have been demonstrated in prior art MGRP [160]. One problem may occur when a TCP cwnd (congestion window) is small, e.g. when cwnd = 1. In this case, the application does not create enough traffic and dummy probe needs to be created.

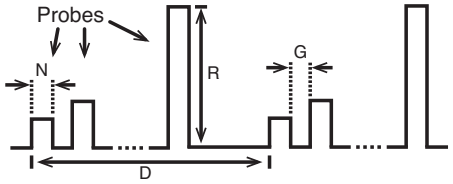


Figure 4.12: Generalized probe train model.

Generalized Probe Model Figure 4.12 shows a generalized probe train model we developed to emulate a number of existing bandwidth estimation algorithms and used for the rest of the section. The horizontal dimension is time. The vertical dimension is the corresponding (instantaneous) probe rate. Each pulse (we call it a *train*) contains multiple probe packets sent at a particular rate, as depicted by the height. In particular, the parameter N represents the number of packets in each train and parameter R represents the (instantaneous) probe rate of the train (i.e. we are able to change the interpacket gap between N packets to match a target probe rate R). Packet sizes are considered in computing the probe rate. For a mixed-size packet train, MinProbe is able to adjust the space between all adjacent packets within the train to achieve the desired probe rate R. The gaps between successive probe trains are specified with parameter G (gap). Finally,

Algorithm	N	R	G	D
Pathload	20	[0.1:0.1:9.6] Gbps	Variable	Variable
Pathchirp	1	[0.1:0.1:9.6] Gbps	Exponential decrease	Variable
Spruce	2	500Mbps	1.2ns	48us
IGI	60	[0.1:0.1:9.6] Gbps	30s	30s

Table 4.2: Parameter setting for existing algorithms. G is the gap between packet trains. R is the rate of probe. N is the number of probe packets in each sub-train. D is the gap between each sub-train.

each measurement *sample* consists of a set of probe trains with increasing probe rate. The distance in time between each measurement sample is identified with parameter D (distance). With these four parameters, we can emulate most probe traffic used in prior work, as shown in Table 4.2. For example, to emulate Spruce probes, we can set the parameters (N, R, G, D) to the values (2, 500Mbps, 1.2ns, 48us), which would generate a pair of probe packets every 48 us with minimal inter-packet gap (12 `/I/`characters or 1.2ns) at 500Mbps (5% of the capacity of a 10Gbps link), as intended by the original Spruce Algorithm [188]. Similarly, to reproduce the IGI experiment results [81], we set the parameters (N, R, G, D) to the values (60, [0.1:0.1:9.6]Gbps, 30s, 30s) to generate probe trains of 60 packets every 30 seconds with rate ranging from 100Mbps to 9.6Gbps at 100Mbps increments. In summary, most types of existing probe trains can be emulated with the generalized model we developed for MinProbe , where different points in the parameterization space can represent different points in the entire design space covering prior art and possibly new algorithms.

In the following sections, we use a parameterization that is similar to Pathload, which estimates available bandwidth based on the increasing one-way delay (OWD) in probe trains. If the rate of a probe train is larger than the available bandwidth of the bottleneck link, the probe train will induce congestion in the network. As a result, the last packet of the probe train will experience longer queuing delays compared to the first packet of the same probe train. The difference between the OWD of the last packet and the first

packet of the same probe train can be used to compute the increasing OWD in the probe train. On the other hand, if a probe train does not induce congestion, then there will be no change in the OWD between the first and last packet. Thus, if an available bandwidth estimation algorithm sends probe trains at different rates R, it will estimate the available bandwidth to be the lowest probe train rate R where the OWD (queuing delay) increases.

A bandwidth estimation algorithm based on Pathload needs to measure the change (increase) in OWD between the first and last packet in a probe train. Since we can measure the gaps between subsequent packets, we show that the interpacket gap information is sufficient to compute the increase in the OWD of a probe packet train.

Proof. Consider sending a train of n packets with packet sizes s_1, s_2, ..., s_n and interpacket gaps g_1, g_2, ..., g_{n-1} from host A to host B through a network path. The received packets at host B experiences one-way delays of q_1, q_2, ..., q_n through the network and now have interpacket gaps h_1, h_2, ..., h_{n-1}. Assume no packet losses due to network congestion, and no packet reordering in the network. Then we want to show that the difference in one-way delay of the first and last packets is equal to the total increase in interpacket gaps, i.e. $q_n - q_1 = \sum h_i - \sum g_i$.

Initially, the length of the packet train measured from the first bit of the first packet to the last bit of the nth packet is given

$$l^A = \sum_{i=1}^{n-1} g_i + \sum_{j=1}^{n} s_j \tag{4.1}$$

Similarly at the receiving end

$$l^B = \sum_{i=1}^{n-1} h_i + \sum_{j=1}^{n} s_j \tag{4.2}$$

Additionally, the difference in one-way delay tells us that

$$l^B = l^A + (q_n - q_1) \tag{4.3}$$

Substitute the relationship for l^A and l^B and we can see that the difference in one-way delay is equivalent to the difference of interpacket gap. □

4.4.3 Implementation

We built a prototype of MinProbe on the programmable network interface, SoNIC , platform with two 10 GbE ports [113]. The detailed design of the SoNIC platform has been described in prior work [113]. Here, we briefly recount the main features of SoNIC . SoNIC consists of two components: a software stack that runs on commodity multi-core processors, and a hardware PCIe pluggable board. The software implements all of the functionality in the 10GbE physical layer that manipulates bits to enable access to `/I/` in software. The hardware performs line-speed data transfer between the 10 GbE transceiver and the host. To enable the high-fidelity network measurements required by MinProbe , we extended SoNIC 's capabilities with three new features:

Packet Filtering and Forwarding: We extended SoNIC to support packet forwarding and filtering at line rate. Packet forwarding preserves the timing characteristics of the pass-through traffic exactly: Data is copied from an incoming port to an outgoing port, including the exact number of `/I/` characters in between each packet. To filter for probe packets, we use an in-kernel flow table that matches on the 5-tuples of a packet IP header. The packets that have a match are temporarily buffered in a queue, interpacket gap modulated, and sent once enough packets have been buffered for a single packet train (a measurement sample).

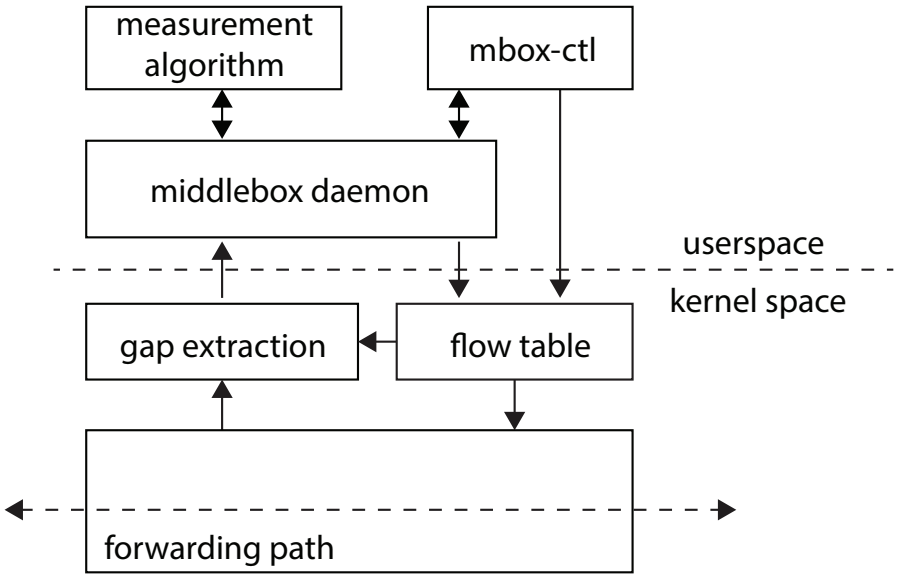


Figure 4.13: MinProbe Architecture.

Packet Gap Extraction and Manipulation: MinProbe has two operational modes: a modulation mode for sending probes and an extraction mode for receiving probes. In the modulation mode, we extended SoNIC with additional packet queues for temporarily buffering of probe packets, and modified the amount of `/I/`characters in front of the buffered packet to change the packet spacing. If `/I/`characters are removed from the probe packet trains to increase the probe rate, we compensate for the loss of `/I/`characters at the end of the probe packet train to conform to the 10 GbE standard, and vice versa. In the extraction mode, the number of `/I/`characters are extracted from the matched probe packet trains and sent to the userspace programs via kernel upcalls.

Application Programming Interface: The last feature we added was the userspace accessibility. Users can insert/remove flow entries, dynamically adjust probe packet gaps, and retrieve probe packet gap counts from MinProbe . It is required to implement a flexible user and kernel space communication channel for passing measurement data and control messages between user and kernel space. We used the *netlink* [16] protocol as the underlying mechanism and implemented the exchange of information using

Function	Description	Example
set_mode	set middlebox mode	set_mode
set_gap	set gap between probes	set_gap(120,120,120)
flow_add	add flow used as probes	flow_add(srcip,dstip)

Table 4.3: Application Programming Interface.

netlink messages. For example, we encapsulate the interpacket gaps in a special netlink message, and transmitted it from kernel to userspace, and vice versa.

4.4.4 Evaluation

MinProbe can accurately estimate the available bandwidth in high-speed networks with minimal overhead. To demonstrate its capabilities and accuracy, we evaluated MinProbe over three different physical networks: Two topologies in our controlled environment and the National Lambda Rail (NLR). We used a controlled environment to study the sensitivity of estimation accuracy to various network conditions. We also evaluated MinProbe on NLR to study its performance in wide area networks. To highlight the accuracy MinProbe is capable of, Figure 4.17 in Section 4.4.4 shows that MinProbe can accurately estimate the available bandwidth within 1% of the actual available bandwidth in a 10Gbps network with topology shown in Figure 4.15a.

Experimental setup Our evaluation consists of two parts: Controlled environment experiments and wide area network (WAN) experiments. For the WAN experiments, we evaluate MinProbe over the National Lambda Rail (NLR). NLR is a transcontinental production 10Gbps Ethernet network shared by research universities and laboratories with no restriction on usage or bandwidth. As a result, traffic in the NLR is typically

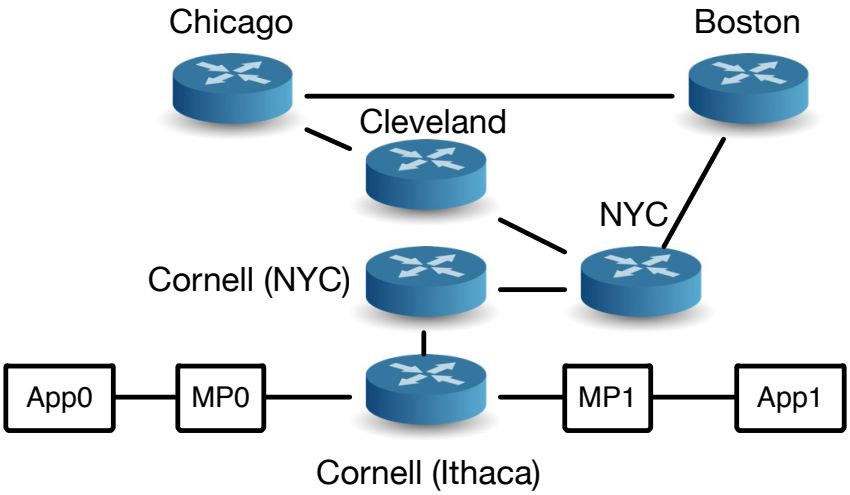
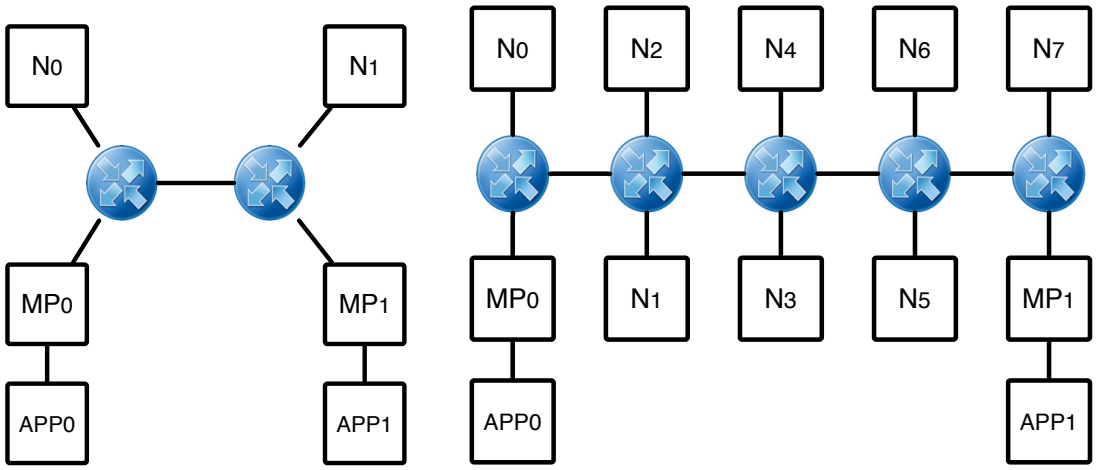


Figure 4.14: National Lambda Rail Experiment.

bursty, yet persistent, and similar to the Internet. We provisioned a dedicated virtual route that spans nine routing hops over 2500 miles, as shown in Figure 4.14. Since the underlying physical network is also shared with other provisioned routes, we observed persistent cross traffic between 1 Gbps to 3.5 Gbps.

For the controlled environment experiments, we set up two different network topologies in a 10 Gbps network testbed: Dumb-bell (DB) and parking-lot (PL), as illustrated in Figure 4.15 and described in [202]. We used one MinProbe middlebox (MP_0) to modulate the application traffic generated from a server directly connected to MP_0 . We used the other MinProbe middlebox (MP_1) as the receiver of the modulated probes to capture the timing information from the application traffic. A network of one or more hops separated the two middleboxes, where one link was a bottleneck link (i.e. a tight link with the least available bandwidth). Our experiments attempted to measure and estimate the available bandwidth of this bottleneck (tight) link. The testbeds were built with commercial 10 GbE switches, represented by circles in Figure 4.15. MinProbe middleboxes and hosts that generated cross traffic were separately connected to the switches



(a) Dumb-bell (b) Parking lot

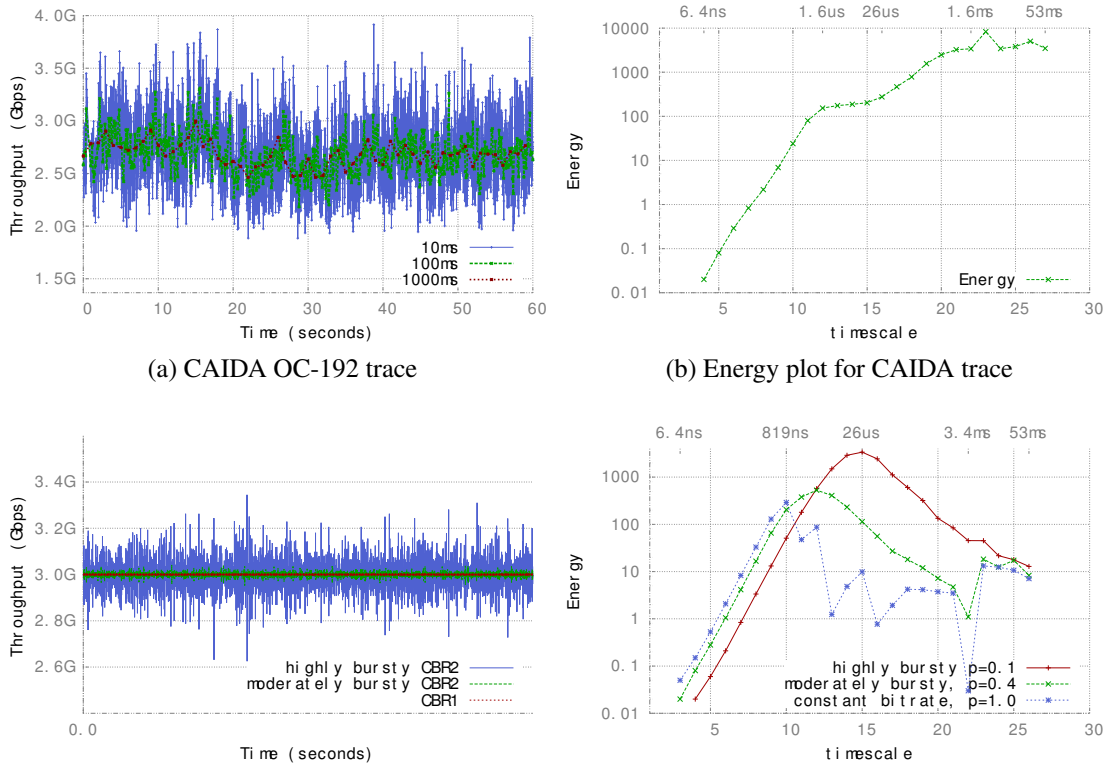
Figure 4.15: Controlled Experiment Topologies.

using 10 GbE links, illustrated with the solid line. In the DB topology, the link between the two switches is the bottleneck (tight) link. In the PL topology, the bottleneck (tight) link depends on the pattern of the cross traffic.

MP_0 and MP_1 were Dell PowerEdge T710 servers. Each server had two Intel Xeon Westmere [15] X5670 2.93GHz processors, with six cores in each CPU and a total of 24 GB RAM. The Westmere architecture of the processor is well-known for its capability of processing packets in a multi-threaded environment [62, 79, 135]. The switches used in the environment consisted of an IBM G8264 RackSwitch and a Dell Force10 switch, with network divided into separate logical areas using VLANs.

We used three different classes of cross traffic for our controlled experiments. Figure 4.16 illustrates the time series characteristics of all three traffic classes. The different classes of cross traffic are also described below.

Constant bit rate cross traffic (CBR1): CBR1 cross traffic consists of fixed-sized and uniformly distributed packets. In particular, CBR1 traffic is a stationary traffic with



(a) CAIDA OC-192 trace

(b) Energy plot for CAIDA trace

(c) Synthesized traffic

(d) Energy plot for synthesized traffic

Figure 4.16: The time series and wavelet energy plots for cross traffic used in controlled experiments. Figure 4.16a shows a time series of a CAIDA trace in three different time scales: 10ms, 100ms and 1s. Coarser time scale means longer averaging period, hence less burstiness. Figure 4.16b shows the corresponding wavelet energy plot for the trace in Figure 4.16a. Figure 4.16c shows three different traces with different traffic burstiness of the same time scale. Figure 4.16d shows the corresponding wavelet energy plot, with higher energy indicating more burstiness.

constant average data rate over long time scale (milliseconds), and interpacket gaps are uniform (i.e. variance is zero).

Stationary, but bursty cross traffic (CBR2): The second class of cross traffic was generated from stationary but bursty distributions, CBR2. CBR2 is created by varying both the packet size and packet chain length distributions with two parameters $\mathcal{D}_{size}$ and $\mathcal{D}_{len}$. Both parameters range from 0.0 (zero variance) to 1.0 (high variance). We draw the packet size from a log-normal distribution which has been demonstrated to closely resemble the distribution of Internet traffic in [120]. The mean of the log-normal

distribution is fixed, while the variance is controlled by the parameter $\mathcal{D}_{size}$ to be $\mathcal{D}_{size} \cdot \mathcal{V}_{size}$, where $\mathcal{V}_{size}$ is a fixed high variance value. A packet chain is a series of consecutive packets with minimal interpacket gap in between, and from [67], we know its length takes on a geometric distribution. The parameter of the geometric distribution is taken to be $1 - \mathcal{D}_{len}$. Thus, CBR1 is a special case of CBR2 where both $\mathcal{D}_{size}$ and $\mathcal{D}_{len}$ equals 0.0.

CAIDA: In the third scheme, we generated cross traffic from a real internet trace. In particular, we used the CAIDA OC-192 [21] trace which was recorded using a DAG card with nano-second scale timestamps.

While there is not one commonly accepted definition for traffic burstiness in the literature, burstiness generally refers to the statistical variability of the traffic. In other words, high variability in traffic data rates implies very bursty traffic. Variability, and by extension, traffic burstiness could be captured by wavelet-based energy plot [196]. Specifically, the energy of the traffic represents the level of burstiness at a particular timescale (e.g. at the microsecond timescale). The higher the energy, the burstier is the traffic. Figure 4.16b and 4.16d illustrate this wavelet-based energy plot when applied to the byte arrivals for CAIDA Internet traces and synthetically generated traffic, respectively. The x-axis represents different time scale, ranging from nano-second to second in log scale. The y-axis represents the abstracted energy level at a particular time scale. We were mostly interested in the micro-second timescale, which correspond to x-axis values around 15. Notice that the synthetic traffic behave very much like CAIDA internet traces for energy levels for these values.

For most experiments, we used 792 bytes as the application traffic packet size, which according to [21], is close to the average packet size observed in the Internet. We adjusted the traffic data rate by varying interpacket gaps (IPG). In particular, we insert a

Packet size [Bytes]	Data Rate [Gbps]	Packet Rate [pps]	IPD [ns]	IPG [/I/]
792	1	156250	6400	7200
792	3	467814	2137	1872
792	6	935628	1068	536
792	8	1250000	800	200

Table 4.4: IPD and IPG of uniformly spaced packet streams.

specific number of `/I/`'s between packets to generate a specific data rate.

We controlled the exact data rate of CBR1 and CBR2 since we control the exact number of `/I/`'s inserted between packets. However, for the CAIDA traces, we selected portions of the trace recording times to obtain different average data rates. In addition, when we replay a trace, we only need to preserve the timing information of the trace, not the actual payload. We used SoNIC to replay the CAIDA trace. As described in [113], SoNIC can regenerate the precise timing characteristics of a trace with no deviation from the original.

Baseline Estimation How does available bandwidth estimation perform using MinProbe in a base case: A simple topology with a couple to several routing hops (Figure 4.15.a and b) and uniform cross traffic (uniform packet size and interpacket gaps)? We illustrate the result with the following experiments.

In the first experimental network setup, we use a dumb-bell topology with two routing hops and a single bottleneck link (Figure 4.15a). Cross traffic is generated with a constant bit rate that has uniform packet sizes and interpacket gaps (CBR1). Node N_0 sends cross traffic to node N_1 according to the CBR1 uniform distribution.

The second experimental setup is similar to the first except that we use a parking-lot topology (Figure 4.15b). Further, cross traffic is generated with the CBR1 uniform

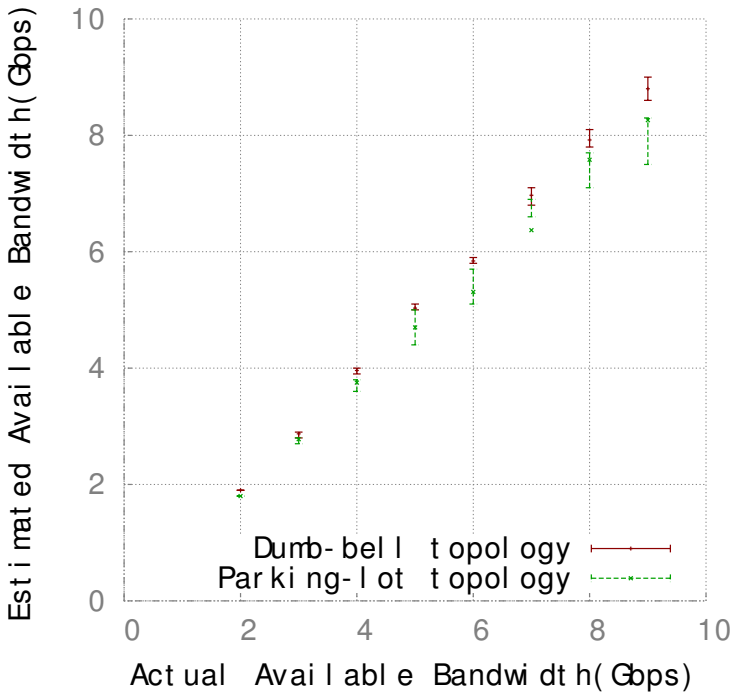
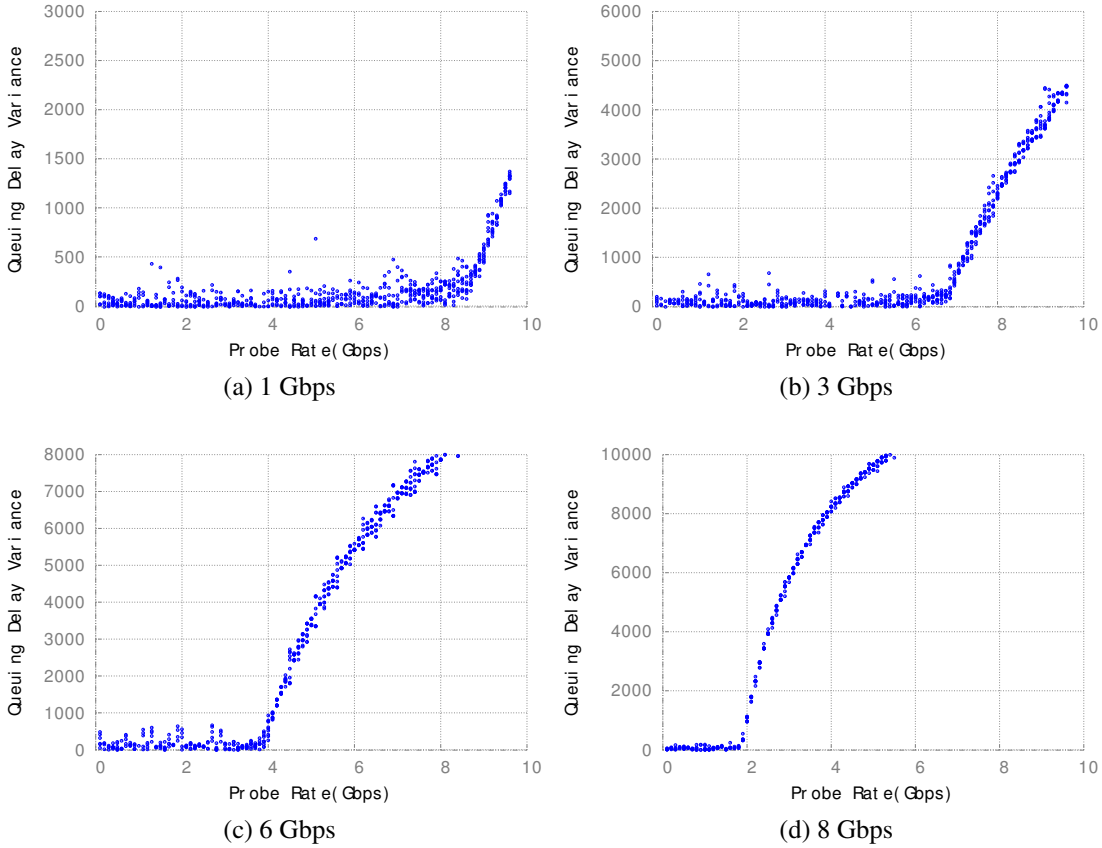


Figure 4.17: Available bandwidth estimation in a dumb-bell and parking-lot topology under CBR traffic. Both cross traffic and probe traffic share one bottleneck with the capacity of 10Gbps. The x-axis represents the actual available bandwidth of the bottleneck link. The y-axis represents the estimation by MinProbe . This evaluation demonstrates MinProbe 's ability to accurately measure the available bandwidth and achieve the estimation with minimal probing overhead.

distribution between neighboring nodes: N_{even} to N_{odd} (e.g. N_0 to N_1, N_2 to N_3, etc). Thus, we can control the cross traffic separately on each link.

In all experiments, we varied the data rate of the CBR1 cross traffic from 1 Gbps to 8 Gbps with an increment of 1 Gbps each time. We configured MP_0 to modulate application traffic from APP_0 destined to APP_1 to create probe packet samples. Note that there is no overhead since we are not introducing any new packets. Further, we configured MP_1 to capture the timing information of the probe packet samples (i.e. application traffic destined to APP_1 from source APP_0). The probe packets modulated by MP_0 were parameterized using the model introduced in Section 4.4.2. We used the parameters (N, R, G, D) = (20, [0.1 : 0.1 : 9.6] Gbps, 10 us, 4 ms) where MP_0 sends a probe packet



(a) 1 Gbps (b) 3 Gbps

(c) 6 Gbps (d) 8 Gbps

Figure 4.18: Scatter-plot showing the queuing delay variance of probe packets versus the probe rate. The cross traffic rate are constant at 1Gbps, 3Gbps, 6Gbps and 8Gbps. We used probe with N=20, R=[0.1:0.1:9.6]Gbps, G=10us, D=4ms.

sample every 4 ms enabling us to collect 250 samples per second. Each probe sample consists of 96 constant bit rate (CBR) trains with a 10 us gap between trains. Each train runs at an increasing data rate ranging from 0.1 Gbps to 9.6 Gbps, with an increment of 0.1 Gbps. Recall that we are able to precisely control the data rate of trains by controlling the interpacket gap between probe packets within the train (e.g. See Table 4.4). We assume node APP_0 generates enough application traffic to create probe packet samples.

Figure 4.17 shows the result. It shows the actual available bandwidth on the x-axis and estimated available bandwidth on the y-axis for the the dumb-bell and parking-lot topologies. We estimated the available bandwidth with the mean of 10 measurement

samples. The estimations in the dumb-bell topology were within 0.1 Gbps of the actual available bandwidth. The estimations in parking-lot topology tended to under-estimate the available bandwidth, due to the multiple bottleneck links in the network path. Higher available bandwidth scenarios were more difficult to measure as accurately in a multiple-hop network path. Alternatively, the maximum value of a measurement could be used to estimate available bandwidth if the estimation was always overly conservative; this would actually increase the estimation accuracy.

Figure 4.18 shows the raw probe sample data that was used to estimate the available bandwidth of different cross traffic rates in the dumb-bell topology. The x-axis is the rate (R) at which the probe trains were sent. The y-axis is the variance of the queuing delay, which was computed from the one-way delay (OWD) experienced by the packets within the same probe train. We kept the cross traffic rate constant at 1Gbps, 3Gbps, 6Gbps and 8Gbps. The available bandwidth was estimated to be the turning point where the delay variance shows an significant trend of increasing.

Impact of Application Traffic Next, we study whether MinProbe is impacted by characteristics of application traffic. In particular, we focus on two characteristics that matter the most for available bandwidth estimation: the length of a probe train and the distribution of probe packet sizes. The length of a probe train affects the delay experienced by application traffic (See the description of probe train length in Section 4.4.2 and equation 4.1). The longer the probe train, the longer the application traffic has to be buffered in the middlebox, resulting in longer delays the application traffic experiences. Typically, a probe train consists of around 100 or more packets [84, 173]. A probe train of two packets is essentially similar to the packet pair scheme described by Spruce [188].

Creating probe packet trains from variable sized packets is difficult because packets with different sizes suffer different delays through a network. Because MinProbe does not have control over the application traffic that is used as packet probes, it is important to understand how the accuracy of MinProbe changes with different distributions of probe packet sizes. Another consideration is that many estimation algorithms prefer large probe packets to small packets [84, 173].

We evaluate the effect of the length of a probe packet train (i.e. the number of packets in a train) to the accuracy of available bandwidth estimation in Section 4.4.4 and the effect of the distributions of probe packet sizes in Section 4.4.4 and 4.4.4. For these experiments, we always used CBR1 as cross traffic.

Impact of Varying Probe Train Length In order to understand how the number of packets in a probe packet train affects the accuracy of MinProbe , we varied the number of packets in each train while using the same parameters as the baseline experiment (Section 4.4.4). In particular, we used (N, [0.1 : 0.1 : 9.6] Gbps, 10 us, 4 ms) while increasing N from 5 to 100. Table 4.5 illustrates the estimated available bandwidth when different train lengths were used.

The actual available bandwidth is shown in the row marked "Actual" and the estimated available bandwidth is in the rows below "Length" in Table 4.5.

As shown in the table, increasing the number of probe packets per flow yields diminishing returns. A probe train with five packets is not as accurate as the baseline experiment with 20 packets regardless of the actual available bandwidth. However, trains with larger than 20 packets result in similar estimation accuracy.

The take-away: Increasing the number of packets in a probe packet train does not

	Available Bandwidth [Gbps]							
	Dumb-bell				Parking-lot			
Actual	1.9	3.9	6.9	8.9	1.9	3.9	6.9	8.9
Length	Estimated Available Bandwidth [Gbps]							
5	2.57	5.57	8.54	9.5	1.99	4.41	6.57	8.59
20	2.07	3.96	6.97	8.8	1.80	3.80	6.90	8.30
40	1.9	3.87	6.94	8.68	1.80	3.86	6.70	8.50
60	1.85	3.79	6.79	8.70	1.80	3.80	6.76	8.56
80	1.86	3.79	6.90	8.70	1.80	3.75	6.78	8.44
100	1.83	3.96	6.79	8.55	1.80	3.70	6.56	8.02

Table 4.5: Estimation with different probe train length.

	Available Bandwidth [Gbps]							
	Dumb-bell				Parking-lot			
	1.9	3.9	6.9	8.9	1.9	3.9	6.9	8.9
Size [B]	Estimated Bandwidth [Gbps]							
64	9.47	9.50	9.50	9.50	9.50	9.50	9.50	9.50
512	2.06	4.51	7.53	8.9	1.85	3.76	6.64	8.09
792	2.07	3.96	6.97	8.8	1.80	3.80	6.90	8.30
1024	1.90	3.97	7.01	8.83	1.80	3.75	6.72	8.54
1518	1.81	3.88	6.91	8.84	1.80	3.81	6.83	8.48

Table 4.6: Estimation results with different probe packet size.

necessarily result in more accurate estimation. The minimum number of packets to obtain accurate estimation results was about 20 packets. More packets in a probe train elongate the delay of applications traffic due to buffering, but do not improve estimation accuracy.

What happens when there is not enough packets between a source and destination pair? This highlights the fundamental trade-off between application latency and probing overhead. MinProbe has a timeout mechanism to deal with this corner case. If the oldest intercepted packet has exceeded a pre-defined timeout value, MinProbe will generate dummy probe packets to fulfill the need for probe packets and send out the probe

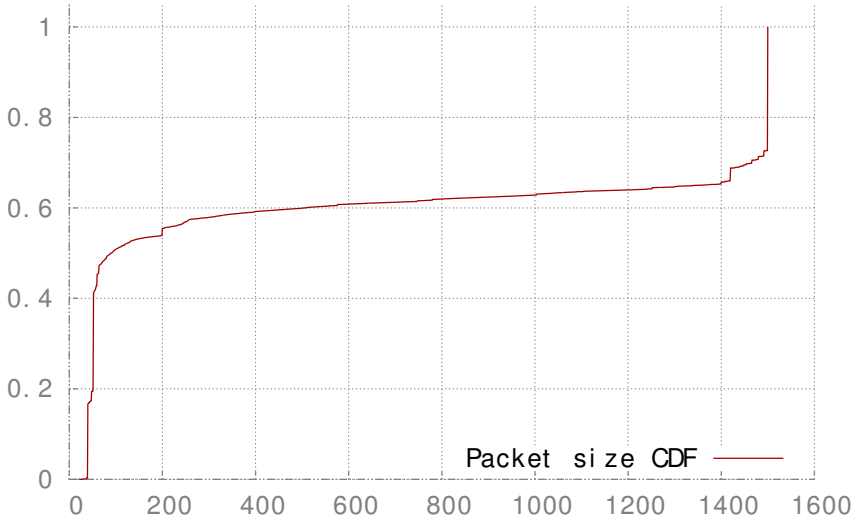


Figure 4.19: The distribution of probe packet sizes from the CAIDA trace.

train. Since MinProbe requires much shorter probe trains compared to existing tools, opportunistically inserting additional dummy packets add minimal overhead.

Impact of Varying Probe Size Next, we evaluated MinProbe with different packet sizes for probes. All the parameters were the same as in the baseline experiment except that we varied the size of packets for probes from 64 bytes to 1518 bytes (the minimum to maximum sized packets allowed by Ethernet). In particular, we used (20, [0.1 : 0.1 : 9.6] Gbps, 10 us, 4 ms) while increasing probe packet sizes from 64 to 1518 bytes. As can be seen from Table 4.6, larger packet sizes (more than 512 bytes) perform better than smaller packet sizes in general. Note that we used 792 byte packets in our baseline experiment, which resulted in similar available bandwidth estimate accuracy as 1518 byte probe packets. On the other hand, smaller probe packets, such as 64 byte packets resulted in poor accuracy. This result and observation is consistent with the previous results from literature such as [173]. The take-away is that probe trains with medium to large packet sizes perform better than trains of smaller packet sizes.

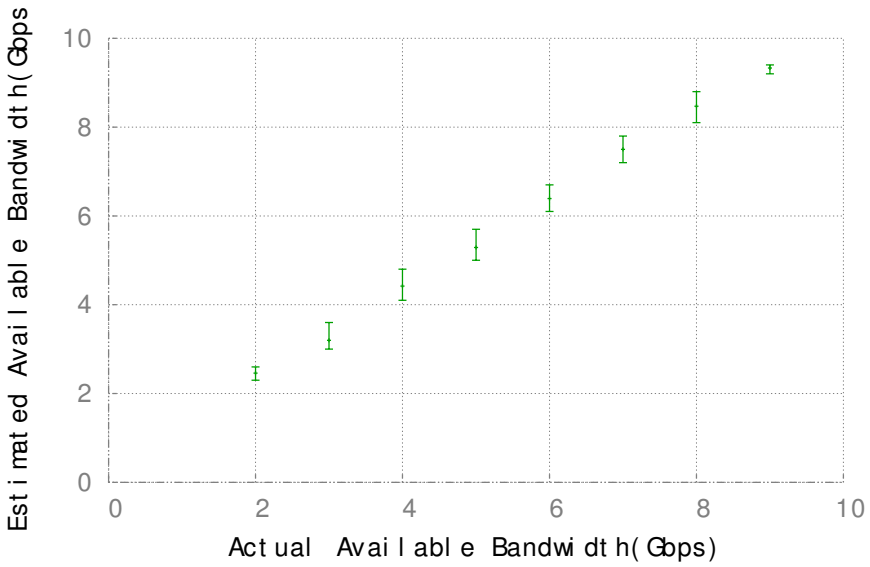


Figure 4.20: Estimation with probe packets drawn from the CAIDA trace.

Mixed Probe Packet Sizes In a real network, the distribution of packet sizes is usually mixed with large and small packets. Figure 4.19 shows the distribution of packet sizes in a recorded Internet trace. We repeat the previous experiment using the baseline setting, except that in this case, we use APP_0 to replay the Internet trace. We let MP_0 modulate the probe train in presence of mixed-sized probe traffic. For each rate $R \in [0.1, 9.6]$, MP_0 ensures that the interpacket gaps between packets of the same flow are uniform. Figure 4.20 shows the result for CBR1 cross traffic. As can be seen, even though a probe train has mixed packet sizes, the estimation result is still accurate. The reason is that small packets only take a small fraction of total probe train length with respect to large packets. The estimation accuracy is largely determined by the large packets, thus remains accurate.

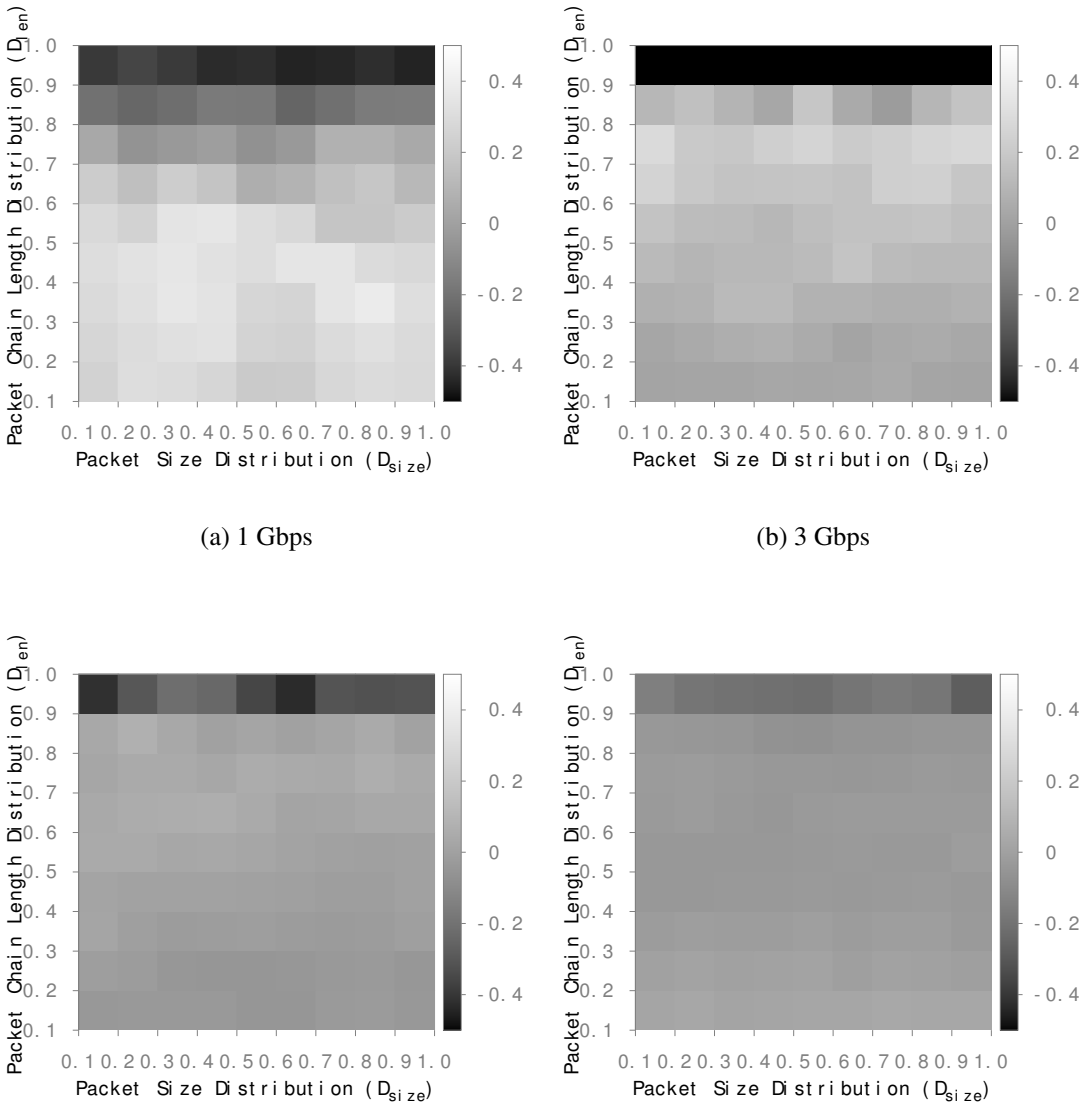
Impact of Cross Traffic Characteristics So far, we have evaluated MinProbe under the assumption of constant bit-rate cross traffic. In this section, we no longer make this assumption and instead examine MinProbe under bursty cross traffic. In particular, we evaluate MinProbe using modeled synthetic bursty traffic (CBR2) and realistic Internet

traces.

Cross Traffic Burstiness Synthesized Traffic: We first show the results of synthesized bursty cross traffic (CBR2). We generated CBR2 with rate $\in [1, 3, 6, 8]$ Gbps. We set $\mathcal{V}_{size}$ to be a fixed large variance value, e.g. 8×10^4 $bytes^2$, as specified in [120]. Next, we varied the distribution $\mathcal{D}_{size}$ of packet sizes in CBR2 from 0 to 1 with 0.1 step size. 0 means uniform distribution, 1 means distribution with large variance. Similarly, we varied the distribution $\mathcal{D}_{len}$ of packet chain length in CBR2 from between 0 to 1 with 0.1 step size. Note that even though CBR2 is bursty in terms of the variation of packet distribution, the long term average data rate of CBR2 remains constant.

Figure 4.21 shows the estimation error (in Gbps) over a range of cross traffic burstiness for four different cross traffic data rates. In each figure, we plot the difference between the actual available bandwidth and the estimated available bandwidth for the cross traffic burstiness identified by the $(\mathcal{D}_{size}, \mathcal{D}_{len})$ parameters. Traffic close to the bottom left corner was more uniformly distributed in both packet sizes and train lengths. Traffic on the top right corner was burstier. Dark gray colors mean under estimation, and light gray colors mean over estimation. As shown in the figure, the estimation error by MinProbe is typically within 0.4 Gbps of the true value except when the link utilization is low, or the cross traffic is bursty.

Real Internet Trace: Next, we evaluate MinProbe with traces extracted from CAIDA anonymized OC192 dataset [21], and replayed by SoNIC as a traffic generator. Figure 4.22a shows an example of the raw data captured by MinProbe . We observed that real the traffic traces were burstier than the synthesized traffic in the previous section. To compensate, we used standard exponential moving average (EMA) method to smooth out the data in Figure 4.22a. For each data points in Figure 4.22a, the value



(a) 1 Gbps

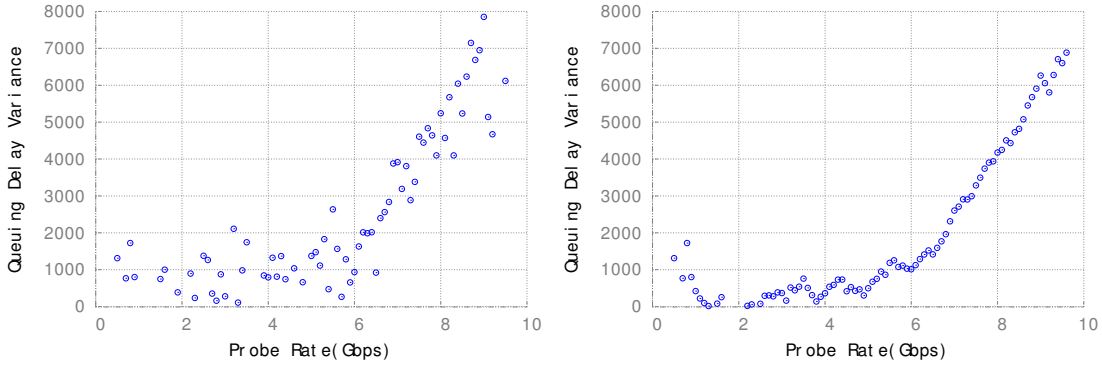
(b) 3 Gbps

(c) 6 Gbps

(d) 8 Gbps

Figure 4.21: Bandwidth estimation accuracy with different cross traffic burstiness. On y-axis, we turn the knob from no clustering to batching. On x-axis, we turn the knob on cross traffic packet size distribution from uniform distribution to log-normal distribution. We plot the graph for different cross traffic rate: 1Gbps, 3Gbps, 6Gbps and 8Gbps.

was replaced by the weighted average of the five data points preceding the current data point. Figure 4.22b shows the result. Similar to the highly bursty synthesized cross traffic, we achieved a similarly accurate estimation result for the real traffic trace via



(a) raw measurement (b) after moving average

Figure 4.22: Bandwidth estimation of CAIDA trace, the figure on the left is the raw data trace, the figure on the right is the moving average data.

using the EMA. The estimation error was typically within 0.4Gbps of the true available bandwidth.

In summary, while bursty cross traffic resulted in noisier measurement data than CBR, we were able to compensate for the noise by performing additional statistical processing in the bandwidth estimation. Specifically, we found using the EMA smoothed the measurement data and improved the estimation accuracy. As a result, we observed that cross traffic burstiness had limited impact on MinProbe .

MinProbe In The Wild In this section, we demonstrate that MinProbe works in a wide area Internet network, the National Lambda Rail (NLR). Figure 4.14 illustrates the experimental network topology. We setup a dedicated route on the NLR with both ends terminating at Cornell University. Node APP_0 generated application traffic, which was modulated by MP_0 using parameter (N, R, G, D) = (20, [0.1 : 0.1 : 9.6] Gbps, 10 us, 4 ms). The traffic was routed through the path shown in Figure 4.14 across over 2500 miles. Since it was difficult to obtain accurate readings of cross traffic at each hop, we relied on the router port statistics to obtain a 30-second average of cross traffic. We observed that the link between Cleveland and NYC experienced the most cross traffic

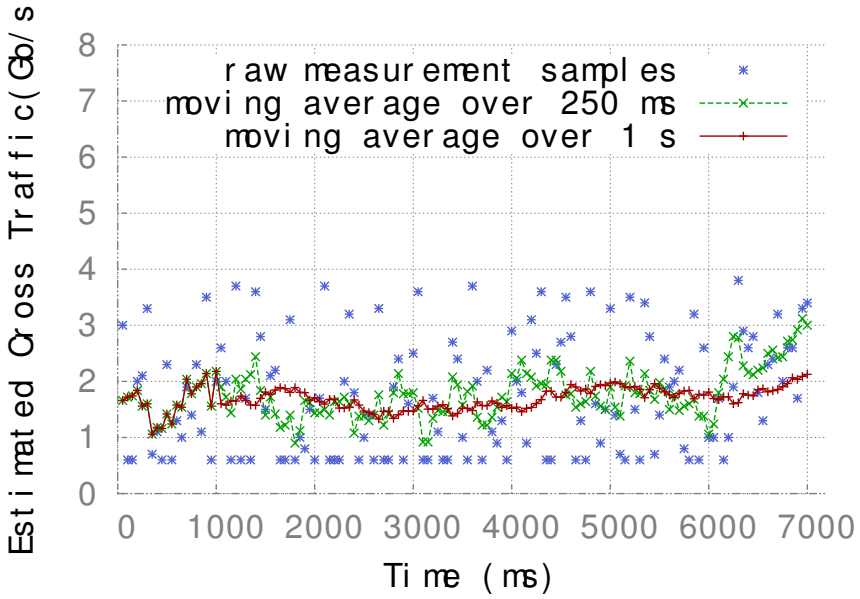


Figure 4.23: Measurement result in NLR.

compared to other links, and was therefore the bottleneck (tight) link of the path. The amount of cross traffic was 1.87 Gbps on average, with a maximum of 3.58 Gbps during the times of our experiments.

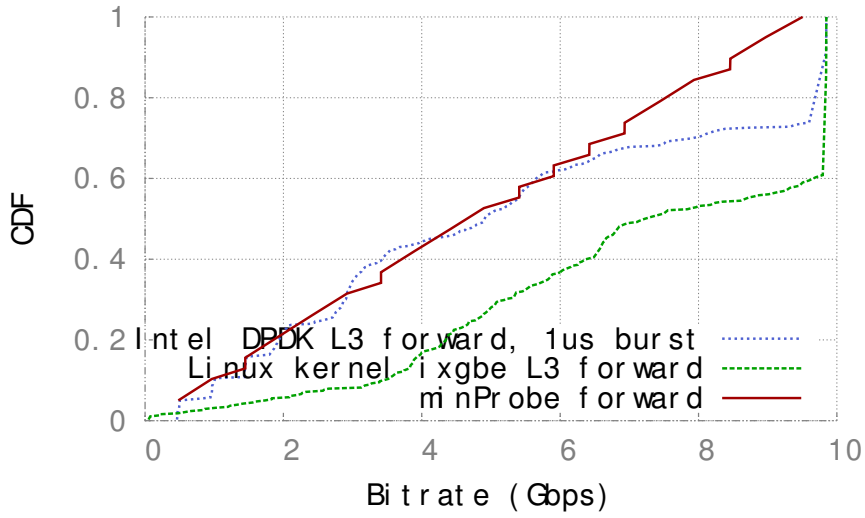
Figure 4.23 shows the result of MinProbe in the wild. We highlight two important takeaways: First, the average of our estimation is close to 2 Gbps, which was consistent with the router port statistics collected every 30 seconds. Second, we observed that the readings were very bursty, which means the actual cross traffic on NLR exhibited a good deal of burstiness. In summary, MinProbe performed well in the wild, in the wide area. The available bandwidth estimation by MinProbe agreed with the link utilization computed from switch port statistical counters: The mean of the MinProbe estimation was 1.7Gbps, which was close to the 30-second average 1.8Gbps computed from switch port statistical counters. Moreover, MinProbe estimated available bandwidth at higher sample rate and with finer resolution.

Many cloud datacenters and federated testbeds enforce rate limits on their tenants. For example, the tenants are required to specify the amount of bandwidth to be re-

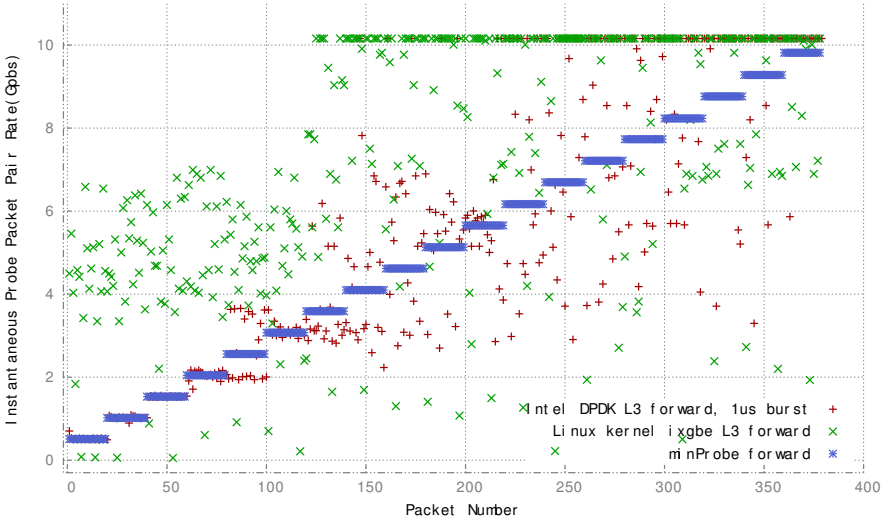
served at the time of provisioning. However, many bandwidth estimation algorithms need to send probe traffic at high data rates to temporarily saturate the bottleneck link. As a result, rate limiting may affect the applicability of the algorithms. We tested the MinProbe middlebox inside a federated testbed, ExoGENI, with a virtual network provisioned at 1Gbps on a 10Gbps physical network. We used MinProbe to send probe packet trains with different lengths at line rate to find the maximum length of packet train that could be sent before throttled by rate limiters. We found that, in the provisioned 1Gbps virtual network, if the probe train was less than 1200 packets, then there was no observable packet loss. As has been shown in Section 4.4.4, MinProbe only uses probe trains with less than 100 probe packets. As a result, and interestingly, rate limiters do not impact the applicability of MinProbe in federate testbed environments.

Software Router and Other Hardware Finally, we evaluate whether other middleboxes, such as software routers, can be used to perform high-fidelity network measurements required to accurately estimate available bandwidth without any overhead. In particular, we evaluated two software routers, one based on the Linux kernel network datapath, and another based on the Intel DPDK driver [64]. We configured the software routers to perform L2 forwarding from one port to another. Again, we used the baseline setup, with the exception that software routers were placed between MinProbe and the switch. The network path started at the application. After MinProbe modulated the application traffic to create packet probe trains, the probe trains were passed through a software router before being sent to a 10 GbE switches.

The experiment was generated from probe packet trains with data rates ranging according to (N, R, G, D) = (20, [0.5 : 0.5 : 9.5] Gbps, 10 us, 4 ms). We show two figures from the this experiment to illustrate the result. First, Figure 4.24a shows the CDF of the measured instantaneous data rate of probe packet pairs passing through a software



(a)



(b)

Figure 4.24: Software Routers do not exhibit the same fidelity as MinProbe .

router (i.e. we measured the interpacket gap of probe packet pairs after passing through a software router). We configured each software router to forward each packet as soon as it received the packet to minimize any batching. Since we generated an equal number of probe packets (20) for a probe train and increased the probe packet train data rate by 0.5 Gbps, the ideal CDF curve should increase along a diagonal frp, 0.5 Gbps to 9.5 Gbps at 0.5 Gbps increments (the red line). However, as seen in the figure, over 25% of the probe packet pairs in the Intel DPDK-based software router were batched with

minimum interpacket gap: Batching can be seen via the large increase in the CDF near the highest data rate close to 10 Gbps. Worse, the vanilla kernel datapath (with ixgbe kernel driver) batched over 40% of packets. These results were consistent with the existing research findings that show that network interface cards (NICs) generate bursty traffic at sub-100 microsecond timescale. [93]

Second, Figure 4.24b highlights the same result but slightly differently. The x-axis is the packet number and the y-axis is the measured instantaneous probe packet pair rate (in Gbps). The figure shows that MinProbe was able to modulate interpacket gaps to maintain the required probe data rate (all 20 packets in a probe train exhibited the target data rate), while the software routers were not able to control the probe rate at all. To summarize, due to batching, software routers are not suitable for performing the required high-fidelity available bandwidth estimation on high-speed networks.

Next, we investigated the question if prior work such as MGRP [160] could perform on high-speed networks? To answer this, we setup MGRP to perform the same baseline estimation experiment in Figure 4.17. Unfortunately, the result was consistent with our findings above and in Figures 4.24a and 4.24b. In particular, we found that MGRP could not report valid results when it was used in high-speed networks (10Gbps Ethernet). The accuracy of MGRP was limited by its capability to precisely timestamp the received probe packets and its ability to control the probe packet gaps. Both of these capabilities were performed at kernel level in MGRP and were prone to operating system level noise. The accuracy of MGRP was better than Pathload [84], it eliminated the overhead of extra probing traffic, but the fundamental accuracy was still constrained by the measurement platform.

Finally, we studied the sensitivity of bandwidth estimation with respect to network hardware. In particular, we experimented with three different 10 GbE switches: IBM

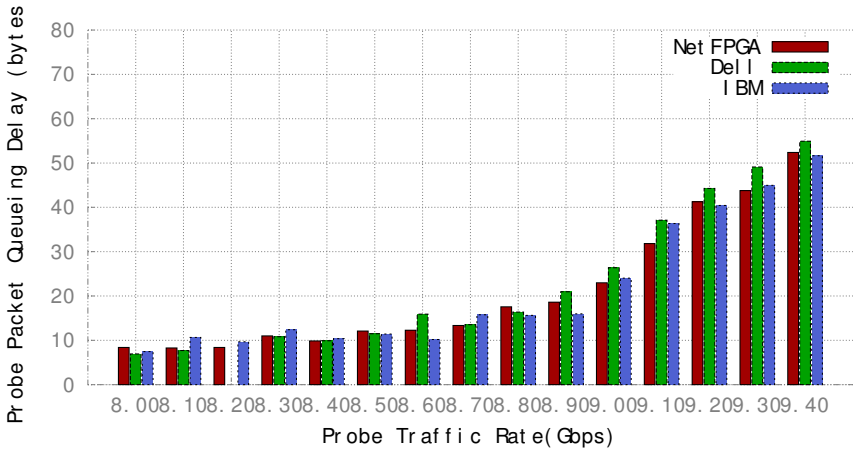


Figure 4.25: Estimating Available Bandwidth on different switches.

G8264T, Dell S4810, and NetFPGA-10G reference switch design [128]. We used the baseline setup for all experiments and configured all three switches to perform L2 forwarding. Figure 4.25 illustrates an example of captured queuing delay after each hardware. As can be seen from the figure, the three switches can be used nearly interchangeably since there was no significant difference in queuing delay. As a result, we have found that the underlying hardware itself may not contribute to the available bandwidth estimation. Instead, as earlier results in the section indicate, the fidelity of the available bandwidth estimation depend more on the ability to control generated interpacket gaps and measure received interpacket gaps.

4.5 Application: Precise Clock Synchronization

This work appears in *Globally Synchronized Time via Datacenter Networks* in SIGCOMM 2016, with co-authors Ki Suh Lee, Vishal Shrivastav, and Hakim Weatherspoon.

In this section, we describe an application that uses programmable PHY to improve the precision and efficiency of clock synchronization by running a protocol in the *phys-*

ical layer of the network protocol stack. We leverage the observation that two machines physically connected by an Ethernet link are already synchronized: Synchronization is required to reliably transmit and receive bitstreams. The question, then, is how to use the bit-level synchronization of the physical layer to synchronize clocks of distributed systems in a datacenter, and how to scale the number of synchronized machines from two to a large number of machines in a datacenter? We first state the problem of clock synchronization, why it is hard to achieve better precision and scalability with current approaches, and how synchronizing clocks in the physical layer can improve upon the state-of-the-art.

4.5.1 Background

Terminology A *clock c* of a process p[1] is a function that returns a local clock counter given a real time t, i.e. $c_p(t)$ = local clock counter. Note that a clock is a discrete function that returns an integer, which we call clock *counter* throughout the paper. A clock changes its counter at every clock *cycle* (or *tick*). If clocks c_i for all i are synchronized, they will satisfy

$$\forall i, j, t \; |c_i(t) - c_j(t)| \leq \epsilon \tag{4.4}$$

where ϵ is the level of *precision* to which clocks are synchronized. *Accuracy* refers to how close clock counters are to true time [177].

Each clock is driven by a quartz oscillator, which oscillates at a given frequency. *Oscillators* with the same nominal frequency may run at different rates due to frequency

[1]We will use the term *process* to denote not only a process running on a processor but also any system entities that can access a clock, e.g. a network interface card.

variations caused by external factors such as temperature. As a result, clocks that have been previously synchronized will have clock counters that differ more and more as time progresses. The difference between two clock counters is called the *offset*, which tends to increase over time, if not resynchronized. Therefore, the goal of clock synchronization is to periodically adjust offsets between clocks (offset synchronization) and/or frequencies of clocks so that they remain close to each other [177].

If a process attempts to synchronize its clock to true time by accessing an external clock source such as an atomic clock, or a satellite, it is called *external synchronization*. If a process attempts to synchronize with another (peer) process with or without regard to true time, it is called *internal synchronization*. Thus, externally synchronized clocks are also internally synchronized, but not vice versa [54]. In many cases, monotonically increasing and internally synchronized clocks are sufficient. For example, measuring one-way delay and processing time or ordering global events do not need true time. As a result, in this paper, we focus on how to achieve internal synchronization: We achieve clock synchronization of all clocks in a datacenter with high precision; however, their clock counters are not synchronized to an external source.

Clock Synchronization Regardless of whether the goal is to achieve internal or external synchronization, the common mechanism of synchronizing two clocks is similar across different algorithms and protocols: A process *reads* a different process's current clock counter and computes an offset, adjusting its own clock frequency or clock counter by the offset.

In more detail, a process p sends a time request message with its current *local* clock counter (t_a in Figure 4.26) to a process q (q reads p's clock). Then, process q responds with a time response message with its local clock counter and p's original clock counter

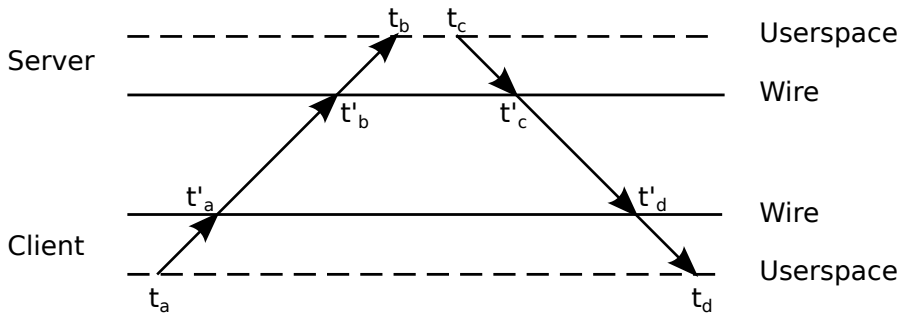


Figure 4.26: Common approach to measure offset and RTT.

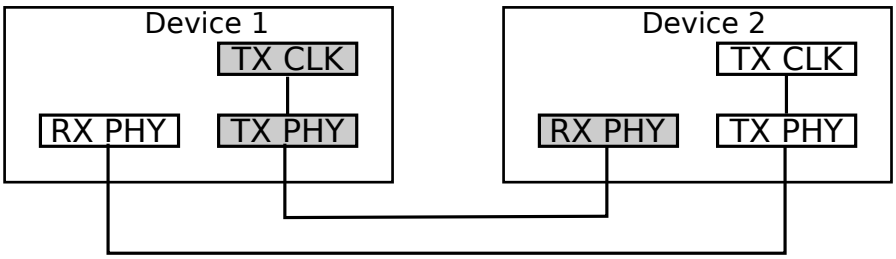


Figure 4.27: Clock domains of two peers. The same color represents the the same clock domain.

(p reads q's clock). Next, process p computes the offset between its local clock counter and the *remote clock* counter (q) and round trip time (RTT) of the messages upon receiving the response at time t_d. Finally, p adjusts its clock counter or the rate of its clock to remain close to q's clock.

In order to improve precision, q can respond with two clock counters to remove the internal delay of processing the time request message: One upon receiving the time request (t_b), and the other before sending the time response (t_c). See Figure 4.26. For example, in NTP, the process p computes RTT δ and offset θ, as follows [141]:

$$\delta = (t_d - t_a) - (t_c - t_b)$$
$$\theta = \frac{(t_b + t_c)}{2} - \frac{(t_a + t_d)}{2}$$

Then, p applies these values to adjust its local clock.

Datacenter Time Protocol (DTP): Why the PHY? Our goal is to achieve nanosecond-level precision as in GPS, with scalability in a datacenter network, and without *any* network overhead. We achieve this goal by running a decentralized protocol in the physical layer (PHY).

DTP exploits the fact that two *peers*[2] are already synchronized in the PHY in order to transmit and receive bitstreams reliably and robustly. In particular, the receive path (RX) of a peer physical layer recovers the clock from the physical medium signal generated by the transmit path (TX) of the sending peer's PHY. As a result, although there are two physical clocks in two network devices, they are virtually in the same circuit (Figure 4.27; What each rectangle means is explained in Section B).

Further, a commodity switch often uses one clock oscillator to feed the sole switching chip in a switch [5], i.e. all TX paths of a switch use the same clock source. Given a switch and N network devices connected to it, there are $N + 1$ physical oscillators to synchronize, and all of them are virtually in the same circuit.

As delay errors from network jitter and a software network stack can be minimized by running the protocol in the lowest level of a system [177], the PHY is the best place to reduce those sources of errors. In particular, we give three reasons why clock synchronization should be performed in the PHY.

First, the PHY allows accurate timestamping at sub-nanosecond scale, which can provide enough fidelity for nanosecond-level precision. Timestamping [67, 113] in the PHY achieves high precision by counting the number of bits between and within packets. Timestamping in the PHY relies on the clock oscillator that generates bits in the PHY, and, as a result, it is possible to read and embed clock counters with a deterministic number of clock cycles in the PHY.

[2]two peers are two physically connected ports via a cable.

Second, a software network stack is not involved in the protocol. As the physical layer is the lowest layer of a network protocol stack, there is always a deterministic delay between timestamping a packet and transmitting it. In addition, it is always possible to avoid buffering in a network device because protocol messages can always be transmitted when there is no other packet to send.

Lastly, there is little to no variation in delay between two peers in the PHY. The only element in the middle of two physically communicating devices is a wire that connects them. As a result, when there is no packet in transit, the delay in the PHY measured between two physically connected devices will be the time to transmit bits over the wire (propagation delay, which is always constant with our assumptions in Section 4.5.2), a few clock cycles required to process bits in the PHY (which can be deterministic), and a clock domain crossing (CDC) which can add additional random delay. A CDC is necessary for passing data between two clock domains, namely between the TX and RX paths. Synchronization FIFOs are commonly used for a CDC. In a synchronization FIFO, a signal from one clock domain goes through multiple flip-flops in order to avoid metastability from the other clock domain. As a result, one random delay could be added until the signal is stable to read.

Operating a clock synchronization protocol in the physical layer not only provides the benefits of zero to little delay errors, but also zero overhead to a network: There is no need for injection of packets to implement a clock synchronization protocol. A network interface continuously generates either Ethernet frames or special characters (Idle characters) to maintain a link connection to its peer. We can exploit those special characters in the physical layer to transmit messages (We will discuss this in detail in Section 4.5.3). The Ethernet standard [13] requires at least twelve idle characters (`/I/`) between any two Ethernet frames regardless of link speed to allow the receiving MAC

layer to prepare for the next packet. As a result, if we use these idle characters to deliver protocol messages (and revert them back to idle characters), no additional packets will be required. Further, we can send protocol messages between every Ethernet frame without degrading the bandwidth of Ethernet and for different Ethernet speeds.

4.5.2 Design

In this section, we present the Datacenter Time Protocol (DTP): Assumptions, protocol, and analysis. The design goals for the protocol are the following:

- Internal synchronization with nanosecond precision.
- No network overhead: No packets are required for the synchronization protocol.

Assumptions We assume, in a 10 Gigabit Ethernet (10 GbE) network, all network devices are driven by oscillators that run at slightly different rates due to oscillator skew, but operate within a range defined by the IEEE 802.3 standard. The standard requires that the clock frequency f_p be in the range of $[f - 0.0001f, f + 0.0001f]$[3] where f is 156.25 MHz in 10 GbE (See Section B).

We assume that there are no "two-faced" clocks [109] or Byzantine failures which can report different clock counters to different peers.

We further assume that the length of Ethernet cables is bounded and, thus, network propagation delay is bounded. The propagation delay of optic fiber is about 5 nanoseconds per meter ($2/3 \times$ the speed of light, which is 3.3 nanoseconds per meter in a vacuum) [91]. In particular, we assume the longest optic fiber inside a datacenter is 1000

[3]This is ±100 parts per million (ppm).

Algorithm 3 DTP inside a network port.

STATE:
gc : global counter, from Algorithm 4
$lc \leftarrow 0$: local counter, increments at every clock tick
$d \leftarrow 0$: measured one-way delay to **peer** p
TRANSITION:
T0: After the link is established with p
$lc \leftarrow gc$
Send (*Init*, lc)
T1: After receiving (*Init*, c) from p
Send (*Init-Ack*, c)
T2: After receiving (*Init-Ack*, c) from p
$d \leftarrow (lc - c - \alpha)/2$
T3: After a timeout
Send (*Beacon*, gc)
T4: After receiving (*Beacon*, c) from p
$lc \leftarrow \max(lc, c + d)$

meters, and as a result the maximum propagation delay is at most 5 us. Most cables inside a datacenter are 1 to 10 meters as they are typically used to connect rack servers to a Top-of-Rack (ToR) switch; 5 to 50 nanoseconds would be the more common delay.

Protocol In DTP, every network port (of a network interface or a switch) has a local counter in the physical layer that increments at every clock tick. DTP operates via protocol messages between peer network ports: A network port sends a DTP message timestamped with its current *local* counter to its peer and adjusts its local clock upon receiving a *remote* counter value from its peer. We show that given the bounded delay and frequent resynchronizations, local counters of two peers can be precisely synchronized in Section 4.5.2.

Since DTP operates and maintains local counters in the physical layer, switches play an important role in scaling up the number of network devices synchronized by the protocol. As a result, synchronizing across all the network ports of a switch (or a network device with a multi-port network interface) requires an extra step: DTP needs to synchronize the local counters of all local ports. Specifically, DTP maintains a *global*

counter that increments every clock tick, but also always picks the *maximum* counter value between it and all of the local counters.

DTP follows Algorithm 3 to synchronize the local counters between two peers. The protocol runs in two phases: `INIT` and `BEACON` phases.

INIT phase The purpose of the `INIT` phase is to measure the one-way delay between two peers. The phase begins when two ports are physically connected and start communicating, i.e. when the link between them is established. Each peer measures the one-way delay by measuring the time between sending an `INIT` message and receiving an associated `INIT-ACK` message, i.e. measure RTT, then divide the measured RTT by two (T0, T1, and T2 in Algorithm 3).

As the delay measurement is processed in the physical layer, the RTT consists of a few clock cycles to send / receive the message, the propagation delays of the wire, and the clock domain crossing (CDC) delays between the receive and transmit paths. Given the clock frequency assumption, and the length of the wire, the only non-deterministic part is the CDC. We analyze how they affect the accuracy of the measured delay in Section 4.5.2. Note that α in Transition 2 in Algorithm 3 is there to control the non-deterministic variance added by the CDC (See Section 4.5.2).

BEACON phase During the `BEACON` phase, two ports periodically exchange their local counters for resynchronization (T3 and T4 in Algorithm 3). Due to oscillator skew, the offset between two local counters will increase over time. A port adjusts its local counter by selecting the maximum of the local and remote counters upon receiving a `BEACON` message from its peer. Since `BEACON` messages are exchanged frequently, hundreds of thousands of times a second (every few microseconds), the offset can be kept to a minimum.

Algorithm 4 DTP inside a network device / switch.

STATE:
gc: global counter
$\{lc_i\}$: local counters
TRANSITION:
T5: at every clock tick
$gc \leftarrow \max(gc + 1, \{lc_i\})$

Scalability and multi hops Switches and multi-port network interfaces have two to ninety-six ports in a single device that need to be synchronized within the device[4]. As a result, DTP always picks the maximum of all local counters $\{lc_i\}$ as the value for a global counter gc (T5 in Algorithm 4). Then, each port transmits the global counter gc in a `BEACON` message (T3 in Algorithm 3).

Choosing the maximum allows any counter to increase monotonically at the same rate and allows DTP to scale: The maximum counter value propagates to all network devices via `BEACON` messages, and frequent `BEACON` messages keep global counters closely synchronized (Section 4.5.2).

Network dynamics When a device is turned on, the local and global counters of a network device are set to zero. The global counter starts incrementing when one of the local counters starts incrementing (i.e. a peer is connected), and continuously increments as long as one of the local counters is incrementing. However, the global counter is set to zero when all ports become inactive. Thus, the local and global counters of a newly joining device are always less than those of other network devices in a DTP network. We use a special `BEACON_JOIN` message in order to make large adjustments to a local counter. This message is communicated after `INIT_ACK` message in order for peers to agree on the maximum counter value between two local counters. When a network device with multiple ports receives a `BEACON_JOIN` message from one of its ports, it

[4]Local counters of a multi-port device will not always be the same because remote clocks run at different rates. As a result, a multi-port device must synchronize local counters.

adjusts its global clock and propagates BEACON_JOIN messages with its new global counter to other ports. Similarly, if a network is partitioned and later restored, two subnets will have different global counters. When the link between them is re-established, BEACON_JOIN messages allow the two subnets to agree on the same (maximum) clock counter.

Handling failures There are mainly two types of failures that need to be handled appropriately: Bit errors and faulty devices. IEEE 802.3 standard supports a Bit Error Rate (BER) objective of 10^{-12} [13], which means one bit error could happen every 100 seconds in 10 GbE. However, it is possible that a corrupted bit coincides with a DTP message and could result in a big difference between local and remote counters. As a result, DTP ignores messages that contain remote counters off by more than eight (See Section 4.5.2), or bit errors not in the three least significant bits (LSB). Further, in order to prevent bit errors in LSBs, each message could include a parity bit that is computed using three LSBs. As BEACON messages are communicated very frequently, ignoring messages with bit errors does not affect the precision.

Similarly, if one node makes too many *jumps* (i.e. adjusting local counters upon receiving BEACON messages) in a short period of time, it assumes the connected peer is faulty. Given the latency, the interval of BEACON messages, and maximum oscillator skew between two peers, one can estimate the maximum offset between two clocks and the maximum number of jumps. If a port receives a remote counter outside the estimated offset too often, it considers the peer to be faulty and stops synchronizing with the faulty device.

Analysis As discussed in Section 4.5.1, the precision of clock synchronization is determined by oscillator skew, interval between resynchronizations, and errors in reading

remote clocks [54, 77, 101]. In this section, we analyze DTP to understand its precision in regards to the above factors. In particular, we analyze the bounds on precision (clock offsets) and show the following:

- Bound of two tick errors due to measuring the one-way delay (OWD) during the `INIT` phase.
- Bound of two tick errors due to the `BEACON` interval. The offset of two synchronized peers can be up to two clock ticks if the interval of `BEACON` messages is less than 5000 ticks.
- As a result, the offset of two peers is bound by four clock ticks or $4T$ where T is 6.4ns. In 10 GbE the offset of two peers is bound by 25.6ns.
- Multi hop precision. As each link can add up to four tick errors, the precision is bounded by $4TD$ where 4 is the bound for the clock offset between directly connected peers, T is the clock period and D is the longest distance in terms of the number of hops.

For simplicity, we use two peers p and q, and use T_p (f_p) and T_q (f_q) to denote the period (frequency) of p and q's oscillator. We assume for analysis p's oscillator runs faster than q's oscillator, i.e. $T_p < T_q$ (or $f_p > f_q$).

Two tick errors due to OWD. In DTP, the one-way delay (OWD) between two peers, measured during the `INIT` phase, is assumed to be stable, constant, and symmetric in both directions. In practice, however, the delay can be measured differently depending on *when* it is measured due to oscillator skew and *how* the synchronization FIFO between the receive and transmit paths interact. Further, the OWD of one path (from p to q) and that of the other (from q to p) might not be symmetric due to the same reasons.We show that DTP still works with very good precision despite any errors introduced by measuring the OWD.

Suppose p sends an `INIT` message to q at time t, and the delay between p and q is d clock cycles. Given the assumption that the length of cables is bounded, and that oscillator skew is bounded, the delay is d cycles for both directions. The message arrives at q at $t+T_p d$ (i.e. the elapsed time is $T_p d$). Since the message can arrive in the middle of a clock cycle of q's clock, it can wait up to T_q before q processes it. Further, passing data from the receipt path to the transmit path requires a synchronization FIFO between two clock domains, which can add one more cycle randomly, i.e. the message could spend an additional T_q before it is received. Then, the `INIT-ACK` message from q takes $T_q d$ time to arrive at p, and it could wait up to $2T_p$ before p processes it. As a result, it takes up to a total of $T_p d + 2T_q + T_q d + 2T_p$ time to receive the `INIT-ACK` message after sending an `INIT` message. Thus, the measured OWD, d_p, at p is,

$$d_p \leq \lfloor \frac{T_p d + 2T_q + T_q d + 2T_p}{T_p} \rfloor / 2 = d + 2$$

In other words, d_p could be one of d, $d+1$, or $d+2$ clock cycles depending on when it is measured. As q's clock is slower than p, the clock counter of q cannot be larger than p. However, if the measured OWD, d_p, is larger than the actual OWD, d, then p will think q is faster and adjust its offset more frequently than necessary (See Transition $T4$ in Algorithm 3). This, in consequence, causes the global counter of the network to go faster than necessary. As a result, α in T2 of Algorithm 3 is introduced.

$\alpha = 3$ allows d_p to always be less than d. In particular, d_p will be $d-1$ or d; however, d_q will be $d-2$ or $d-1$. Fortunately, a measured delay of $d-2$ at q does not make the global counter go faster, but it can increase the offset between p and q to be two clock ticks most of the time, which will result in q adjusting its counter by one only when the actual offset is two.

Two tick errors due to the `BEACON` interval. The `BEACON` interval, period of resyn-

chronization, plays a significant role in bounding the precision. We show that a BEACON interval of less than 5000 clock ticks can bound the clock offset to two ticks between peers.

Let $C_p(X)$ be a clock that returns a real time t at which $c_p(t)$ changes to X. Note that the clock is a discrete function. Then, $c_p(t) = X$ means, the value of the clock is stably X at least after $t - T_p$, i.e. $t - T_p < C_p(X) \leq t$.

Suppose p and q are synchronized at time t_1, i.e. $c_p(t_1) = c_q(t_1) = X$. Also suppose $c_p(t_2) = X + \Delta P$, and $c_q(t_2) = X + \Delta Q$ at time t_2, where ΔP is the difference between two counter values of clock p at time t_1 and t_2. Then,

$$t_2 - T_p < C_p(X + \Delta P) = C_p(X) + \Delta P T_p \leq t_2$$
$$t_2 - T_q < C_q(X + \Delta Q) = C_q(X) + \Delta Q T_q \leq t_2$$

Then, the offset between two clocks at t_2 is,

$$\Delta t(f_p - f_q) - 2 < \Delta P - \Delta Q < \Delta t(f_p - f_q) + 2$$

where $\Delta t = t_2 - t_1$.

Since the maximum frequency of a NIC clock oscillator is $1.0001f$, and the minimum frequency is $0.9999f$, $\Delta t(f_p - f_q)$ is always smaller than 1 if Δt is less than 32 us. As a result, $\Delta P - \Delta Q$ can be always less than or equal to 2, if the interval of resynchronization (Δt) is less than 32 us ($\approx$ 5000 ticks). Considering the maximum latency of the cable is less than 5 us ($\approx$ 800 ticks), a beacon interval less than 25 us ($\approx$ 4000 ticks) is sufficient for any two peers to synchronize with 12.8 ns (= 2 ticks) precision.

Multi hop Precision. Note that DTP always picks the maximum clock counter of all nodes as the global counter. All clocks will always be synchronized to the fastest clock

in the network, and the global counter always increases monotonically. Then, the maximum offset between any two clocks in a network is between the fastest and the slowest. As discussed above, any link between them can add at most two offset errors from the measured delay and two offset errors from `BEACON` interval. Therefore, the maximum offset within a DTP-enabled network is bounded by $4TD$ where D is the longest distance between any two nodes in a network in terms of number of hops, and T is the period of the clock as defined in the IEEE 802.3 standard ($\approx 6.4ns$).

4.5.3 Implementation

DTP-enabled PHY The control logic of DTP in a network port consists of Algorithm 3 from Section 4.5.2 and a local counter. The local counter is a 106-bit integer (2 $\times$ 53 bits) that increments at every clock tick (6.4 ns = 1/156.25 MHz), or is adjusted based on received `BEACON` messages. Note that the same oscillator drives all modules in the PCS sublayer on the transmit path and the control logic that increments the local counter. i.e. they are in the same clock domain. As a result, the DTP sublayer can easily insert the local clock counter into a protocol message with no delay.

The DTP-enabled PHY is illustrated in Figure 4.28. Figure 4.28 is exactly the same as the PCS from the standard, except that Figure 4.28 has DTP control, TX DTP, and RX DTP sublayers shaded in gray. Specifically, on the transmit path, the TX DTP sublayer inserts protocol messages, while, on the receive path, the RX DTP sublayer processes incoming protocol messages and forwards them to the control logic through a synchronization FIFO. After the RX DTP sublayer receives and uses a DTP protocol message from the Control block (`/E/`), it replaces the DTP message with idle characters (`/I/`s, all 0's) as required by the standard such that higher network layers do not know about the

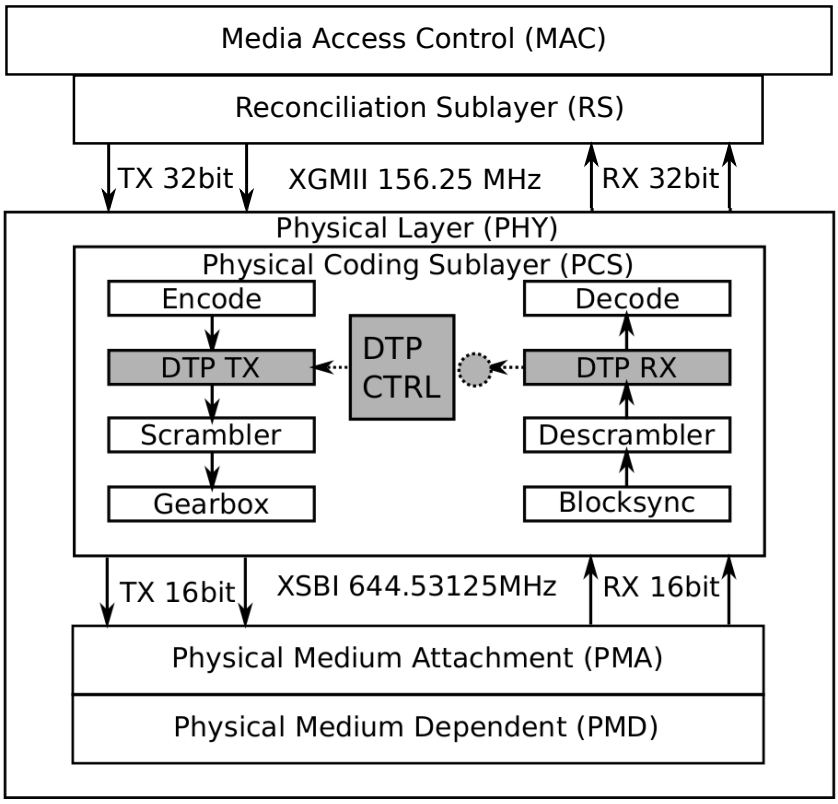


Figure 4.28: Low layers of a 10 GbE network stack. Grayed rectangles are DTP sublayers, and the circle represents a synchronization FIFO.

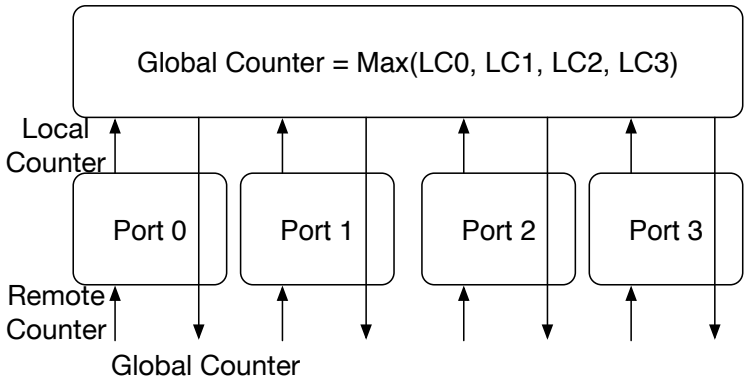


Figure 4.29: DTP enabled four-port device.

existence of the DTP sublayer. Lastly, when an Ethernet frame is being processed in the PCS sublayer in general, DTP simply forwards blocks of the Ethernet frame unaltered between the PCS sublayers.

DTP-enabled network device A DTP-enabled device (Figure 4.29) can be implemented with additional logic on top of the DTP-enabled ports. The logic maintains the 106-bit global counter as shown in Algorithm 4, which computes the maximum of the local counters of all ports in the device. The computation can be optimized with a tree-structured circuit to reduce latency, and can be performed in a deterministic number of cycles. When a switch port tries to send a `BEACON` message, it inserts the global counter into the message, instead of the local counter. Consequently, all switch ports are synchronized to the same global counter value.

Protocol messages DTP uses `/I/`sin the `/E/` control block to deliver protocol messages. There are eight seven-bit `/I/`sin an `/E/` control block, and, as a result, 56 bits total are available for a DTP protocol message per `/E/` control block. Modifying control blocks to deliver DTP messages does not affect the physics of a network interface since the bits are scrambled to maintain DC balance before sending on the wire (See the scrambler/descrambler in Figure B.1). Moreover, using `/E/` blocks do not affect higher layers since DTP replaces `/E/` blocks with required `/I/`s(zeros) upon processing them.

A DTP message consists of a three-bit message type, and a 53-bit payload. There are five different message types in DTP: `INIT`, `INIT-ACK`, `BEACON`, `BEACON-JOIN`, and `BEACON-MSB`. As a result, three bits are sufficient to encode all possible message types. The payload of a DTP message contains the local (global) counter of the sender. Since the local counter is a 106-bit integer and there are only 53 bits available in the payload, each DTP message carries the 53 least significant bits of the counter. In 10 GbE, a clock counter increments at every 6.4 ns (=1/156.25MHz), and it takes about 667 days to overflow 53 bits. DTP occasionally transmits the 53 most significant bits in a `BEACON-MSB` message in order to prevent overflow.

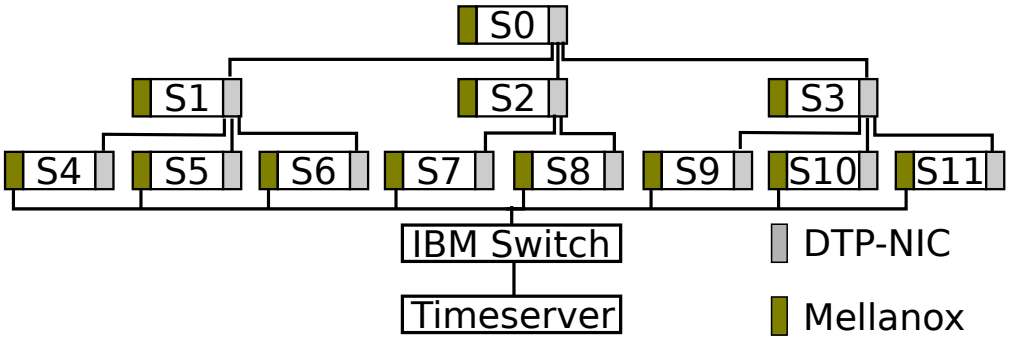


Figure 4.30: Evaluation Setup.

As mentioned in Section B, it is always possible to transmit one protocol message after/before an Ethernet frame is transmitted. This means that when the link is fully saturated with Ethernet frames DTP can send a `BEACON` message every 200 clock cycles ($\approx$ 1280 ns) for MTU-sized (1522B) frames[5] and 1200 clock cycles ($\approx$ 7680 ns) at worst for jumbo-sized (≈9kB) frames. The PHY requires about 191 66-bit blocks and 1,129 66-bit blocks to transmit a MTU-sized or jumbo-sized frame, respectively. This is more than sufficient to precisely synchronize clocks as analyzed in Section 4.5.2 and evaluated in Section 4.5.4. Further, DTP communicates frequently when there are no Ethernet frames, e.g every 200 clock cycles, or 1280 ns: The PHY continuously sends `/E/` when there are no Ethernet frames to send.

4.5.4 Evaluation

In this section, we attempt to answer following questions:

- *Precision:* In Section 4.5.2, we showed that the precision of DTP is bounded by $4TD$ where D is the longest distance between any two nodes in terms of number of hops. In this section, we demonstrate and measure that precision is indeed within the $4TD$ bound via a prototype and deployed system.

[5]It includes 8-byte preambles, an Ethernet header, 1500-byte payload and a checksum value.

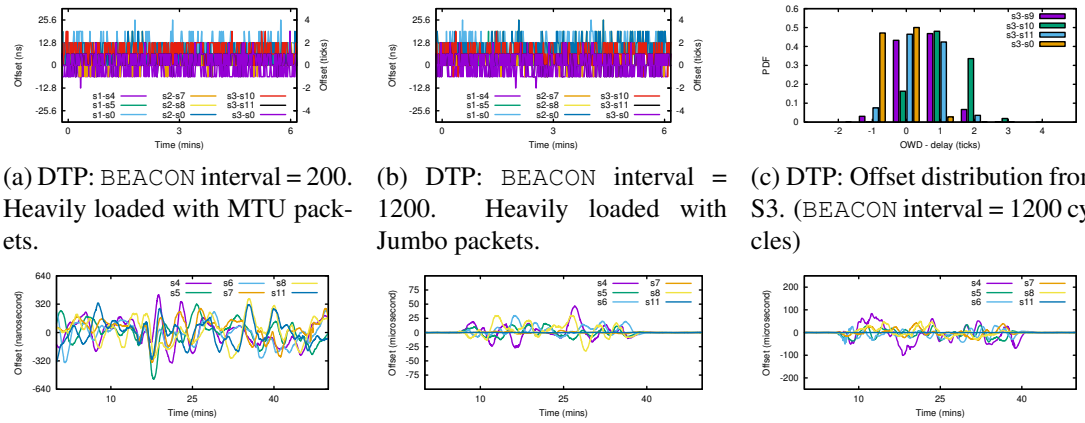
- *Scalability:* We demonstrate that DTP scales as the number of hops of a network increases.

Further, we measured the precision of accessing DTP from software and compared DTP against PTP.

Evaluation Setup For the DTP prototype and deployment, we used programmable NICs plugged into commodity servers: We used DE5-Net boards from Terasaic [191]. A DE5-Net board is an FPGA development board with an Altera Stratix V [116] and four Small Form-factor Pluggable (SFP+) modules. We implemented the DTP sublayer and the 10 GbE PHY using the Bluespec language [147] and Connectal framework [96]. More detailed description of hardware implementation can be found in Appendix D.3. We deployed DE5-Net boards on a cluster of twelve Dell R720 servers. Each server was equipped with two Xeon E5-2690 processors and 96 GB of memory. All servers were in the same rack in a datacenter. The temperature of the datacenter was stable and cool.

We created a DTP network as shown in Figure 4.30: A tree topology with the height of two, i.e. the maximum number of hops between any two leaf servers was four. DE5-Net boards of the root node, $S0$, and intermediate nodes, $S1 \sim S3$, were configured as DTP switches, and those of the leaves ($S4 \sim S11$) were configured as DTP NICs. We used 10-meter Cisco copper twinax cables to a DE5-Net board's SFP+ modules. The measured one-way delay (OWD) between any two DTP devices was 43 to 45 cycles ($\approx$ 280 ns).

We also created a PTP network with the same servers as shown in Figure 4.30 (PTP used Mellanox NICs). Each Mellanox NIC was a Mellanox ConnectX-3 MCX312A 10G NIC. The Mellanox NICs supported hardware timestamping for incoming and outgoing packets which was crucial for achieving high precision in PTP. A VelaSync time



(a) DTP: `BEACON` interval = 200. Heavily loaded with MTU packets.

(b) DTP: `BEACON` interval = 1200. Heavily loaded with Jumbo packets.

(c) DTP: Offset distribution from S3. (`BEACON` interval = 1200 cycles)

(d) PTP: Idle network

(e) PTP: Medium loaded

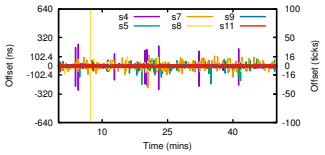
(f) PTP: Heavily loaded

Figure 4.31: Precision of DTP and PTP. A *tick* is 6.4 nanoseconds.

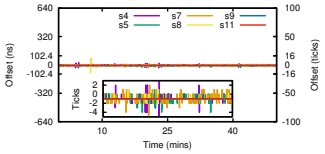
server from Spectracom was deployed as a PTP grand-master clock. An IBM G8264 cut-through switch was used to connect the servers including the time server. As a result, the number of hops between any two servers in the PTP network was always two. Cut-through switches are known to work well in PTP networks [209]. We deployed a commercial PTP solution (Timekeeper [22]) in order to achieve the best precision in 10 Gigabit Ethernet. Note that the IBM switch was configured as a transparent clock.

The time server multicasted PTP timing information every second, i.e. the synchronization rate was once per second, which was the recommended sync rate by the provider. Note that each `sync` message was followed by `Follow_Up` and `Announce` messages. Further, we enabled PTP `UNICAST` capability, which allowed the server to send unicast `sync` messages to individual PTP clients once per second in addition to multicast `sync` messages. In our configuration, a client sent two `Delay_Req` messages per 1.5 seconds.

Methodology Measuring offsets at nanosecond scale is a very challenging problem. One approach is to let hardware generate pulse per second (PPS) signals and compare them using an oscilloscope. Another approach, which we use, is to measure the preci-



(a) Before smoothing: Raw offset_{sw}



(b) After smoothing: Window size = 10

Figure 4.32: Precision of DTP daemon.

sion directly in the PHY. Since we are mainly interested in the clock counters of network devices, we developed a *logging* mechanism in the PHY.

Each leaf node generates and sends a 106-bit log message twice per second to its peer, a DTP switch. DTP switches also generate log messages between each other twice per second. A log message contains a 53-bit estimate of the DTP counter generated by the DTP daemon, t_0, which is then timestamped in the DTP layer with the lower 53-bits of the global counter (or the local counter if it is a NIC). The 53-bit timestamp, t_1, is appended to the original message generated by the DTP daemon, and, as a result, a 106-bit message is generated by the sender. Upon arriving at an intermediate DTP switch, the log message is timestamped again, t_2, in the DTP layer with the receiver's global counter. Then, the original 53-bit log message (t_0) and two timestamps (t_1 from the sender and t_2 from the receiver) are delivered to a DTP daemon running on the receiver. By computing $\text{offset}_{hw} = t_2 - t_1 - \text{OWD}$ where OWD is the one-way delay measured in the `INIT` phase, we can estimate the precision between two peers. Similarly, by computing $\text{offset}_{sw} = t_1 - t_0$, we can estimate the precision of a DTP daemon. Note that offset_{hw} includes the non-deterministic variance from the synchronization FIFO and offset_{sw} includes the non-deterministic variance from the PCIe bus. We can accurately

approximate both the offset_{hw} and offset_{sw} with this method.

For PTP, the Timekeeper provides a tool that reports measured offsets between the time server and all PTP clients. Note that our Mellanox NICs have PTP hardware clocks (PHC). For a fair comparison against DTP that synchronizes clocks of NICs, we use the precision numbers measured from a PHC. Also, note that a Mellanox NIC timestamps PTP packets in the NIC for both incoming and outgoing packets.

The PTP network was mostly idle except when we introduced network congestion. Since PTP uses UDP datagrams for time synchronization, the precision of PTP can vary relying on network workloads. As a result, we introduced network workloads between servers using `iperf` [192]. Each server occasionally generated MTU-sized UDP packets destined for other servers so that PTP messages could be dropped or arbitrarily delayed.

To measure how DTP responds to varying network conditions, we used the same `heavy load` that we used for PTP and also changed the `BEACON` interval during experiments from 200 to 1200 cycles, which changed the Ethernet frame size from 1.5kB to 9kB. Recall that when a link is fully saturated with MTU-sized (Jumbo) packets, the minimum `BEACON` interval possible is 200 (1200) cycles.

Results Figure 4.31 and 4.32 show the results: We measured precision of DTP in Figure 4.31a-c, PTP in Figure 4.31d-f, and the DTP daemon in Figure 4.32. For all results, we continuously synchronized clocks and measured the precision (clock offsets) over at least a two-day period in Figure 4.31 and at least a few-hour period in Figure 4.32.

Figures 4.31a-b demonstrate that the clock offsets between any two directly connected nodes in DTP never differed by more than four clock ticks; i.e. offsets never

differed by more than 25.6 nanoseconds ($4TD = 4 \times 6.4 \times 1 = 25.6$): Figures 4.31a and b show three minutes out of a two-day measurement period and Figure 4.31c shows the distribution of the measured offsets with node S3 for the entire two-day period. The network was always under `heavy load` and we varied the Ethernet frame size by varying the `BEACON` interval between 200 cycles in Figure 4.31a and 1200 cycles in Figure 4.31b. DTP performed similarly under `idle` and `medium load`. Since we measured all pairs of nodes and no offset was ever greater than four, the results support that precision was bounded by $4TD$ for nodes D hops away from each other. Figure 4.32 shows the precision of accessing a DTP counter via a DTP daemon: Figure 4.32a shows the raw offset_{sw} and Figure 4.32b shows the offset_{sw} after applying a moving average algorithm with a window size of 10. We applied the moving average algorithm to smooth the effect of the non-determinism from the PCIe bus, which is shown as occasional spikes. The offset between a DTP daemon in software and the DTP counter in hardware was usually no more than 16 clock ticks ($\approx 102.4ns$) before smoothing, and was usually no more than 4 clock ticks ($\approx 25.6ns$) after smoothing.

Figures 4.31d-f show the measured clock offsets between each node and the grandmaster time server using PTP. Each figure shows minutes to hours of a multi-day measurement period, enough to illustrate the precision trends. We varied the load of the network from `idle` (Figure 4.31d), to `medium load` where five nodes transmitted and received at 4 Gbps (Figure 4.31e), to `heavy load` where the receive and transmit paths of all links except S11 were fully saturated at 9 Gbps (Figure 4.31f). When the network was `idle`, Figure 4.31d showed that PTP often provided hundreds of nanoseconds of precision, which matches literature [10, 25]. When the network was under `medium load`, Figure 4.31e showed the offsets of $S4 \sim S8$ became unstable and reached up to 50 microseconds. Finally, when the network was under `heavy load`, Figure 4.31f showed that the maximum offset degraded to hundreds of microseconds. Note that we

measured, but do not report the numbers from the PTP daemon, `ptpd`, because the precision with the daemon was the same as the precision with the hardware clock, PHC. Also, note that all reported PTP measurements include smoothing and filtering algorithms.

There are multiple takeaways from these results.

1. DTP more tightly synchronized clocks than PTP.
2. The precision of DTP was not affected by network workloads. The maximum offset observed in DTP did not change either when load or Ethernet frame size (the `BEACON` interval) changed. PTP, on the other hand, was greatly affected by network workloads and the precision varied from hundreds of nanoseconds to hundreds of microseconds depending on the network load.
3. DTP scales. The precision of DTP only depends on the number of hops between any two nodes in the network. The results show that precision (clock offsets) were always bounded by $4TD$ nanoseconds.
4. DTP daemons can access DTP counters with tens of nanosecond precision.
5. DTP synchronizes clocks in a short period of time, within two `BEACON` intervals. PTP, however, took about 10 minutes for a client to have an offset below one microsecond. This was likely because PTP needs history to apply filtering and smoothing effectively. We omitted these results due to limited space.
6. PTP's performance was dependent upon network conditions, configuration such as transparent clocks, and implementation.

4.6 Summary

In this chapter, we presented SoNIC which allows users to access the physical layer in realtime from software. SoNIC can generate, receive, manipulate and forward 10 GbE bitstreams at line-rate from software. Further, SoNIC gives systems programmers unprecedented precision for network measurements and research. At its heart, SoNIC utilizes commodity-off-the-shelf multi-core processors to implement part of the physical layer in software and employs an FPGA board to transmit optical signal over the wire. As a result, SoNIC allows cross-network-layer research explorations by systems programmers.

We presented two applications enabled by SoNIC: MinProbe and DTP. MinProbe is a high-fidelity, minimal-cost and userspace accessible network measurement approach that is able to accurately measure available bandwidth in 10 GbE networks. We found that MinProbe performs well even and the estimation is typically within 0.4 Gbps of the true value in a 10 GbE network. DTP tightly synchronizes clocks with zero network overhead (no Ethernet packets). It exploits the fundamental fact that two physically connected devices are already synchronized to transmit and receive bitstreams. DTP can synchronize clocks of network components at tens of nanoseconds of precision, can scale up to synchronize an entire datacenter network, and can be accessed from software with usually better than twenty five nanosecond precision. As a result, the end-to-end precision is the precision from DTP in the network (i.e. 25.6 nanoseconds for directly connected nodes and 153.6 nanoseconds for a datacenter with six hops) plus fifty nanosecond precision from software.

CHAPTER 5

RELATED WORK

5.1 Programming Network Elements

5.1.1 Hardware

Programmable network hardware allows users to experiment with novel network system architectures. Previous studies have shown that reconfigurable NICs [200] can be used to explore new I/O virtualization techniques in VMMs. Those who are conducting research on new network protocols and intrusion detection can use NetFPGA [128, 211] to experiment with FPGA-based router and switches [34, 47, 144, 210].

Furthermore, users can employ specialized NICs that are programmable and support P4 language [38] which is a dataplane programming language. In P4, forwarding elements perform user-defined actions such as modifying, discarding, or forwarding packets. Netronome's FlowNIC [145] and Xilinx's SDNet [203] support P4. While programmable NICs allow users to access the layer 2 and above, SoNIC allows users to access the PHY. In other words, SoNIC allows users to access the entire network stack in software.

5.1.2 Software

Although SoNIC is orthogonal to software routers, software routers are important because they share common techniques. SoNIC preallocates buffers to reduce memory overhead [79, 174], polls huge chunks of data from hardware to minimize interrupt

overhead [63, 79], packs packets in a fashion that resembles batching to improve performance [63,79,135,174]. Software routers normally focus on scalability, and, hence, they exploit multi-core processors and multi-queue supports from NICs to distribute packets to different cores to process. On the other hand, SoNIC pipelines multiple CPUs to handle continuous bitstreams.

5.1.3 Language

P4 Reflecting the significant interest in P4 as a development platform, several efforts are underway to implement P4 compilers and tools. Our micro-benchmarks can be compared to those of PISCES [179], which is a software hypervisor switch that extends Open vSwitch [152] with a protocol-independent design. The Open-NFP [151] organization provides a set of tools to develop network function processing logic, including a P4 compiler that targets 10, 40 and 100GbE Intelligent Server Adapters (ISAs) manufactured by Netronome. These devices are network processing units (NPUs), in contrast to P4FPGA, which targets FPGAs. The Open-NFP compiler currently does not support register-related operations and cannot parse header fields that are larger than 32 bits. Users implement actions in a MicroC code that is external to the P4 program. P4c [157] is a retargetable compiler for the P4 language that generates a high performance network switch code in C, linking against DPDK [64] libraries. DPDK provides a set of user-space libraries that bypass the Linux kernel. P4c does not yet support P4 applications that require register uses to store state. P4.org provides a reference compiler [158] that generates a software target, can be executed in a simulated environment (i.e. Mininet [142] and P4 Behavioral Model switch [156]). P4FPGA shares the same compiler front-end, but it provides a different back-end.

A P4 compiler backend targeting a programmable ASIC [90] must deal with resource constraints. The major challenge arises from mapping logical lookup tables to physical tables on an ASIC. In contrast, FPGAs can directly map logical tables into the physical substrate without the complexity of logical-to-physical table mapping, thanks to the flexible and programmable nature of FPGAs.

Perhaps the most closely related effort is Xilinx's SDNet [203]. SDNet compiles programs from the high-level PX [41] language to a data plane implementation on a Xilinx FPGA target at selectable line rates that range from 1G to 100G. A Xilinx Labs prototype P4 compiler works by translating from P4 to PX, and then it uses SDNet to map this PX to a target FPGA. The compiler implementation is not yet publicly available, and so we cannot comment on how the design or architecture compares to P4FPGA.

High Level Synthesis. FPGAs are typically programmed using hardware description languages such as Verilog of VHDL. Many software-developers find working with these languages challenging because they expose low-level hardware details to the programmer.

Consequently, there has been significant research in high-level synthesis and programming language support for FPGAs. Some well-known examples include CASH [44], which compiles C to FPGAs; Kiwi [180], which transforms .NET progrrams into FPGA circuits; and Xilinx's AccelDSP [27], which performs synthesis from MATLAB code.

P4FPGA notably relies on Bluespec [147] as a target language, and it re-uses the associated compiler and libraries to provide platform independence. As already mentioned, P4FPGA uses Connectal [96] libraries, which also are written in Bluespec, for

common hardware features.

5.2 Network Applications

5.2.1 Consensus Protocol

Prior work [56] proposed that consensus logic could be moved to forwarding devices using two approaches: (*i*) implementing Paxos in switches; and (*ii*) using a modified protocol, named *NetPaxos*, that makes assumption about packet ordering in order to solve consensus without switch-based computation. This section builds on that work by making the implementation of a switch-based Paxos concrete. István et al. [83] have also proposed implementing consensus logic in hardware, although they focus on Zookeeper's atomic broadcast written in Verilog.

Dataplane programming languages. Several recent projects have proposed domain-specific languages for dataplane programming. Notable examples include Huawei's POF [186], Xilinx's PX [41], and the P4 [39] language used throughout this section. We focus on P4 because there is a growing community of active users, and because it is relatively more mature than the other choices. However, the ideas for implementing Paxos in switches should generalize to other languages.

Replication protocols. Research on replication protocols for high availability is quite mature. Existing approaches for replication-transparent protocols, notably protocols that implement some form of strong consistency (e.g. linearizability, serializability), can be roughly divided into three classes [52]: (a) state-machine replication [104, 178], (b) primary-backup replication [150], and (c) deferred update replication [52].

Despite the long history of research on replication protocols, there exist very few examples of protocols that leverage network behavior to improve performance. We are aware of one exception: systems that exploit *spontaneous message ordering*, [107, 162, 163]. These systems check whether messages reach their destination in order; they do not assume that order must be always constructed by the protocol or incur additional message steps to achieve it. This section implements a standard Paxos protocol that does not make ordering assumptions.

5.2.2 Timestamping

The importance of timestamping has long been established in the network measurement community. Prior work either does not provide sufficiently precise timestamping, or requires special devices. Packet stamping in user-space or kernel suffers from the imprecision that is introduced by the OS layer [55]. Many commodity NICs support hardware timestamping that has levels of accuracy that range from nanoseconds [6, 7, 17] to hundreds of nanoseconds [18]. Furthermore, NICs can be combined with a GPS receiver or PTP-capability to use reference time for timestamping.

Timestamping in hardware either requires offloading the network stack to a custom processor [207] or network interface cards; the latter provide hardware timestamping capability via an external clock source [6, 17], which makes the device hard to program and inconvenient to use in a data center environment. Many commodity network interface cards support hardware timestamping. Data acquisition and generation (DAG) cards [6] additionally offer globally synchronized clocks among multiple devices, whereas SoNIC only supports delta timestamping.

Although BiFocals [67] is able to provide an exact timestamping, its limitations have

prevented it from being a portable and realtime tool. BiFocals can store and analyze only a few milliseconds worth of a bitstream at a time due to the small memory of the oscilloscope. Furthermore, it requires thousands of CPU hours to convert raw optic waveforms to packets. Finally, the physics equipment used by BiFocals is expensive and is not easily portable. Its limitations motivated us to design SoNIC to achieve the realtime exact precision timestamping. Unfortunately, both BiFocals and SoNIC only support delta timestamping.

5.2.3 Bandwidth Estimaton

Prior work in MAD [182], MGRP [160] and Periscope [80] provided a thin measurement layer (via userspace daemon and kernel module) for network measurement in a shared environment. These works complement MinProbe , which is applicable in a shared environment. We also advance the available bandwidth estimation to 10Gbps; this contrasts with most prior work, which only operated in 1Gbps or less.

We use an algorithm similar to Pathload [84] and Pathchirp [173] to estimate available bandwidth. There are many related works on both the theoretical and practical aspects of available bandwidth estimation [30, 84, 88, 123, 125, 129, 170, 173, 183, 185]. Our work contributes to the practice of available bandwidth estimation in high speed networks (10Gbps). We find that existing probing parameterizations such as probe trains are sound and applicable in high-speed networks when they are supported by the appropriate instrumentation. Furthermore, [85, 197] mentioned that burstiness of the cross traffic can negatively affect the accuracy of estimation. Although bursty traffic does introduce noise into the measurement data, we find that the noise can be filtered out using simple statistical processing, such as the moving average.

Middleboxes are popular building blocks for current network architecture [159], when operating as a middlebox, is similar to software routers in the sense that MinProbe consists of a data forwarding path and a programmable flow table. Unlike software routers, which typically focus on high throughput, MinProbe has the capability to precisely control the distance between packets; this capability is absent in most existing software routers. With regards to system design, ICIM [133] has proposed an inline network measurement mechanism that is similar to our middlebox approach. However, in contrast to ICI, which has only simulated their proposal, we implemented a prototype and performed experiment in real network. Others have investigated the effect of interrupt coalescence [171] and the memory I/O subsystem [89], which are orthogonal to our efforts. MinProbe , albeit a software tool, avoids completely the noise of OS and the network stack by operating at the physical layer of the network stack.

Another area of related work is precise traffic pacing and precise packet timestamping. These are useful system building blocks for designing a high-fidelity network measurement platform. Traditionally, traffic pacing [31] is used to smooth out the burstiness of the traffic in order to improve system performance. In MinProbe , we use traffic pacing in the opposite way to generate micro-burst of traffic to serve as probe packets. Precise timestamping has been used widely in passive network monitoring. Typically, this is achieved through the use of a dedicated hardware platform, such as Endace Data Acquisition and a Generation (DAG) card [6]. With MinProbe , we achieve the same nanosecond timestamping precision, and we are able to use the precise timestamp for active network measurement, which the traditional hardware platform cannot achieve.

5.2.4 Clock Synchronization

Clock synchronization has been extensively studied. Next, we briefly discuss clock synchronization protocols.

Because NTP normally does not provide precise clock synchronization in a local area network (LAN), much of the literature has focused on improving NTP without extra hardware. One line of work was to use `TSC` instructions to implement precise software clocks called TSCclock, which later were called RADclock [58, 161, 195]. TSC clock was designed to replace `ntpd` and `ptpd` (daemons that run NTP or PTP), and to provide sub-microsecond precision without any extra hardware support. Other software clocks include Server Time Protocol (STP) [149], Coordinated Cluster Time (CCT) [68], AP2P [181], and skewless clock synchronization [132], all of which provide microsecond precision.

The implemention of clock synchronization in hardware has been demonstrated by Fiber Channel (FC) [8] and discussed by Kopetz and Ochsenreiter [101]. FC embeds protocol messages into interpacket gaps that are similar to DTP. However, FC is not a decentralized protocol and the network fabric simply forwards protocol messages between a server and a client using physical layer encodings. Consequently, it does not eliminate non-deterministic delays which delivering protocol messages.

Synchronous optical networks (SONET/SDH) are a standard that transmits multiple bitstreams (such as Voice, Ethernet, TCP/IP) over an optical fiber. In order to reduce the buffering of data between network elements, SONET requires precise frequency synchronization (i.e. *syntonization*). An atomic clock is commonly deployed as a Primary Reference Clock (PRC) and other network elements are synchronized to it either by external timing signals or by recovering clock signals from incoming data. DTP does not

synchronize the frequency of clocks, but the values of clock counters.

Synchronous Ethernet (SyncE) [23] was introduced to enable reliable data transfer between synchronous networks (e.g. SONET/SDH) and asynchronous networks (e.g. Ethernet). Like SONET, it synchronizes the frequency of nodes in a network, rather than clocks (i.e. *syntonization*). It aims to provide a synchronization signal to all Ethernet network devices. The goal is to use the recovered clock from the receive (RX) path to drive the transmit (TX) path so that both the RX and TX paths run at the same clock frequency. As a result, each Ethernet device uses a phase locked loop to regenerate the synchronous signal. SyncE does not synchronize clocks in a network, and so PTP is often employed along with SyncE to provide tight clock synchronization. One such example is White Rabbit, which we discuss below.

White Rabbit [110, 121, 143] has by far the best precision in packet-based networks. The goal of White Rabbit (WR) [143] is to synchronize up to 1000 nodes with sub-nanosecond precision. It uses SyncE to syntonize the frequency of clocks of network devices, and WR-enabled PTP [110] to embed the phase difference between a master and a slave into PTP packets. WR demonstrated that the precision of a non-disturbed system was 0.517ns [121]. WR also requires WR-enabled switches, and synchronizes slaves that are up to four-hops apart from the timeserver. WR works on a network that has a tree topology and a limited number of levels and servers. Furthermore, it currently supports 1 Gigabit Ethernet only; because it uses PTP packets, it is not clear how WR behaves under heavy network loads. DTP does not rely on any specific network topology, and it can be extended to protocols with higher speeds.

Similarly, BroadSync [42] and ChinaMobile [118] combine SyncE and PTP to provide hundreds-of-nanosecond precision. The Data Over Cable Service Interface Specification (DOCSIS) is a frequency synchronized network that is designed to time divide

data transfers between multiple cable modems (CM) and a cable modem termination system (CMTS). The DOCSIS time protocol [51] extends DOCSIS to synchronize time by approximating the internal delay from the PHY and asymmetrical path delays between a reference CM and the CMTS. We expect that combining DTP with frequency synchronization, or SyncE, will further improve the precision of DTP to sub-nanosecond precision as it becomes possible to minimize or remove the variance of the synchronization FIFO between the DTP TX and RX paths.

CHAPTER 6

FUTURE DIRECTION

Programmable network dataplanes enrich the capabilities of networks by allowing users to deploy customized packet processing algorithms. In this dissertation, we have explored two aspects of the programmable dataplane. First we studied how to design a programmable packet processing pipeline to enable deployment of custom packet processing algorithms. Second, we researched how to design a programmable PHY to users to improve network visibility and measurement. We demonstrated an instance of each of the approaches, resulting in two systems: P4FPGA and SoNIC. We expect more systems to harness the capabilities of the work described in this dissertation. In this chapter, we review and discuss future research directions.

6.1 Rack-scale Computing

Rack-scale computing in next generation data centers advocates disaggregating resources needed for a computing task (i.e. processors, memory, networking and storage) into a pool of resources that are interconnected with high-speed network fabrics. This level of hardware resource aggregation requires a redesign of the data center network fabric to provide stronger networking support for low-latency, high-throughput and lossless properties for next generation data center networks. Fundamentally, the economy of scale in the cloud environment requires that disaggregated cloud resources to serve applications with different performance requirement and traffic patterns. One future direction is to implement network support for disaggregated storage systems.

6.2 Scaling to 100G

Data center network functions are migrating to the end host, while network speed is increasing to 50G or 100Gbps. Unfortunately, end host CPU is a scarce resource that must be used wisely. Dedicating 10s of cores for network processing is both impractical and uneconomical. A programmable dataplane in NIC is able to offload network function to the NIC to accelerate end host packet processing while maintaining software programmability. A future direction to scale programmable dataplane to support a 100Gbps line rate and design the associated end-host network stack to share the 100Gbps network capacity among 10s or 100s of CPU cores.

6.3 Packet Scheduling

A packet scheduling algorithm can achieve different desired performance objectives, such as fairness, reducing network flow tail latencies, and minimizing flow completion times. Most packet scheduling algorithms are employed on packet switching networks that have statistical multiplexing on communication links. We introduced a global clock synchronization protocol that can emulate a time division multiplexing (TDM) network on top of a packet switching network. A promising research direction is to investigate how the resulting TDM network can improve the packet scheduling algorithm to achieve less network jitter and packet loss. Moreover, future research could experiment with real hardware prototypes built with systems developed in this dissertation.

6.4 Distributed and Responsive Network Control Plane

Routing is most implemented control algorithm in a network. Routing is the act of moving packets across the network from a source to a destination. It involves two basic activities: determining the optimal routing paths and forwarding the packets through a network. The complexity of routing algorithm lies in the path determination.

An optimal routing algorithm selects the best route based on the basis of the current network conditions and the target metrics. In large-scale networks, collecting network conditions can be difficult to accomplish in realtime. A scalable and optimal routing algorithm must be able to measure the network at scale in a distributed manner and aggregate the result efficiently for a routing algorithm to consume.

A programmable dataplane can provide a flexible substrate to implement a fast and responsive routing algorithm in the dataplane in a distributed manner. Each dataplane in a network element can monitor network conditions simultaneously and, thus, alleviate the need for a powerful centralized routing control platform. Because each network element can react to traffic conditions locally, response to network events, such as failure or congestion, can be instant. A future direction could look into how to measure and implement a network routing algorithm using a programmable dataplane.

CHAPTER 7

CONCLUSIONS

At the time of writing this dissertation, network dataplane implementation often is a trade-off between flexibility and performance. Furthermore, programming a reconfigurable device, such as FPGAs, requires hardware domain expertise, which many network programmers do not have. Finally, parts of the network dataplane are not programmable at all.

We have explored a new approach to building a programmable dataplane that balances flexibility and performance. In particular, we investigate two building blocks of the network dataplane programming for network devices: the packet processing pipeline and the network device interface. By leveraging a good packet processing abstraction called the *match-action* pipeline and a domain specific language built on top of the abstraction, developers can program the network dataplane to implement a variety of novel applications. Furthermore, by opening the inner workings of the physical layer, developers can study networks and the network stack at a heretofore inaccessible level, with the precision of network measurements improved by orders of magnitude.

To validate our approach, we have described the design, implementation and evaluation of two systems that together constitute steps towards an instance of a programmable dataplane. P4FPGA is an instance of a dataplane language compiler and runtime that enables a programmable packet processing pipeline. SoNIC is an instance of a programmable physical layer that implements the physical layer of the network protocol stack in software. We also have shown a number of applications are enabled by these two systems: a network accelerated consensus protocol, a precise bandwidth estimation tool, and an accurate data center clock synchronization protocol.

A programmable network dataplane provides high-performance, customized packet processing and, therefore, it decouples the task of network programming from the hardware that realizes the implementation. This decoupling is an important step toward the further optimization of cloud computing and networking resources. A network function can be placed on any device on a network path or distributed among multiple network devices without significant engineering efforts. As the driven force behind the next generation of software defined networking, the programmable dataplane will affect not just how network devices are built, but the entire networking ecosystem.

APPENDIX A

NETWORK CONCEPTS

A basic understanding of networking is important to understand the contributions of this dissertation. In this section, we provide a brief overview of common networking concepts. We discuss basic terminology, common protocols, and the functionalities of different layers of networking.

A.1 Networking Basic Terminology

First, we define some common terms in networking.

Packet A packet is, generally speaking, the most basic unit that is transfered over a network. When communicating over a network, packets are the envelopes that carry data in pieces from one end point to the other. Packets have a header portion that contains information about the packet including the source and destination, timestamp, etc. The main portion of a packet that contains the actual data is called the body or the payload.

Protocol A protocol is a set of rules and standards that defines a language that devices can use to communicate. These are the rules or standards that define the syntax, semantics and synchronization of communication and possible error recovery methods. There are a great number of protocols in use extensively in networking, and they are often implemented in different layers. Some low level protocols are TCP, [32] UDP, [166] IP, [59, 167] and ICMP. [168] Application layer protocols can be built on top of these lower layer protocols, such as HTTP, [37, 66] SSH, [206] and FTP. [169] Protocols may be implemented by hardware, software, or a combination of both.

Connection In networking, a connection, or sometimes called a session, is a semi-permanent interactive information interchange between two or more communicating devices. A connection is built before the data transfer and then is deconstructed at the at the end of the data transfer.

A.2 Network Layering Model

The TCP/IP model, more commonly known as the Internet protocol suite, is a layering model that has been widely adopted. [103] In the TCP/IP model, five separate layers are defined.

Application Layer The application layer is responsible for creating and transmitting user data between applications. The applications can be on remote systems, and should appear to operate as if locally to the end user. The application layer is also known as Layer 5 of the network protocol stack.

Transport Layer The transport layer is responsible for communication between processes. This network layer uses ports to address different services. It can build up unreliable connections, such as UDP, (User Datagram Protocol), or reliable connections, such as TCP, (Transmission Control Protocol) depending on the type of protocol used. The transport layer is also known as the Layer 4 of the TCP/IP network protocol stack.

Network Layer The network layer is used to route data from one node to another in a network. This layer is aware of the address of the endpoints of the connections. Internet protocol (IP) addresses are commonly used as the addressing scheme. An IP

address could be 32-bit for IPv4 [167] and 128-bit for IPv6. [59] The network layer is also known as the Layer 3 of the network protocol stack.

Link Layer The link layer implements the actual topology of the local network that allows the IP in the network layer to present an addressable interface. A commonly used link layer technology is Ethernet. [139] 10Gbps Ethernet is common as of the date of this disseration. The Ethernet data communication standard is maintained and extended by the IEEE 802.3 working group. [14] The standard supports full-duplex operation, and is widely supported by network vendors. The link layer is also known as Layer 2 of the network protocol stack.

Physical Layer The physical layer deals with bit-level transmission between different devices and supports electrical or optical interfaces connecting to the physical medium for communication. The physical layer is the first layer of the network protocol stack.

A.3 Packet Encapsulation

A packet is transmitted by encapsulating the payload inside layers of packet headers. For instance, Figure A.1 shows the encapsulation of a UDP datagram as an IP packet. The UDP header is prepended to the payload of the packet as it is sent from the application layer to transport layer. Next, an IP header is prepended to the packet in the network layer. Finally, the packet is encapsulated inside an Ethernet frame, which includes an Ethernet header, a preamble and and a 32-bit Cyclic Redundancy Check, (CRC) which is used to detect bit corruption during transmission.

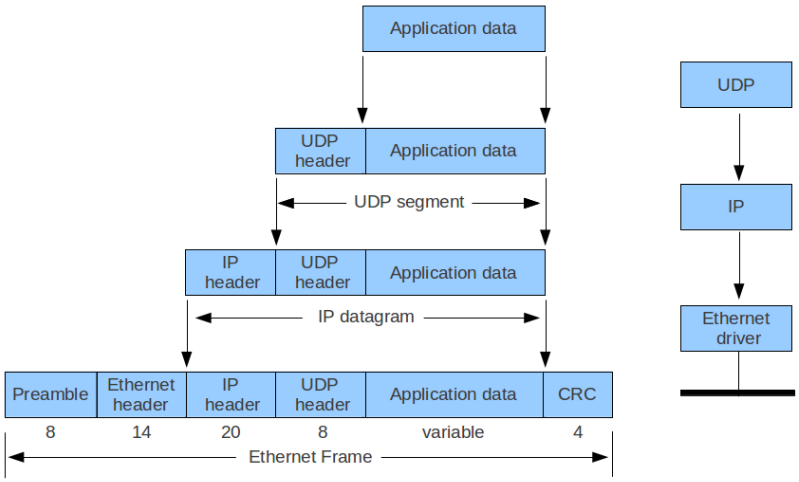


Figure A.1: Encapsulation of data as it goes down the network stack.

A.4 Switch and Router

Switches and routers are both computer networking devices that allow one or more computers to be connected to other computers, networked devices, or to other networks.

A router is a device that forwards data packets along networks. A router is connected to at least two networks, commonly two LANs or WANs. Routers are located at gateways, the places where two or more networks connect. Routers use headers and forwarding tables to determine the best path for forwarding the packets, and they use protocols such as ICMP [168] to communicate with each other and configure the best route between any two hosts.

A switch is a device that filters and forwards packets between LAN segments. Switches operate at the data link layer (Layer 2) and sometimes the network layer (Layer

3) of the OSI Reference Model [212] and therefore support any packet protocol. LANs that use switches to join segments are called switched LANs or, in the case of Ethernet networks, switched Ethernet LANs.

In the context of this dissertation, we used the term switch and router interchangably.

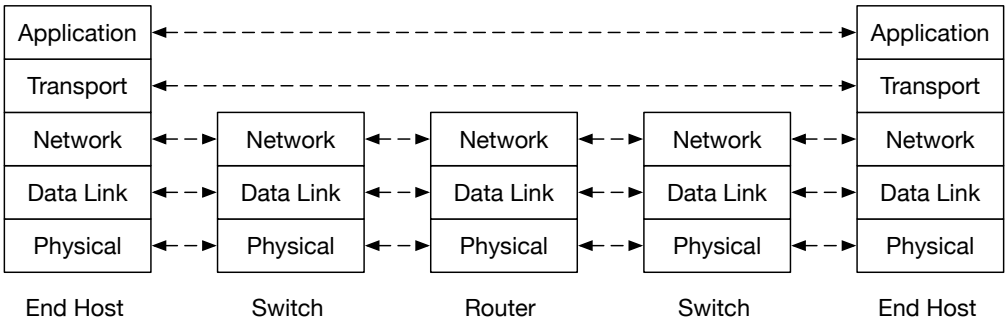


Figure A.2: Network Dataplane Built with Endhosts, Switches and Routers.

As shown in Figure A.2, a dataplane typically consists of more than one network devices. The endhost implements all five layers of the network protocol stack, which communicates with the network protocol stack on another endhost through a series of network devices, e.g. switches and routers. The switches and routers implement the first two or three layers of the network stack. The IP routers receive TCP/IP packets, look inside each packet to identify the source and target IP addresses, and then forward these packets as needed to ensure the data reaches its final destination.

APPENDIX B

IEEE 802.3 STANDARD

In this section, we discuss the physical layer of the network stack in detail. The physical layer is the lowest layer of the network stack (Appendix A.2), and is often implemented in hardware. According to the IEEE 802.3 standard, [13] the physical layer, (PHY) of 10 GbE consists of three sublayers: the Physical Coding Sublayer, (PCS) the Physical Medium Attachment (PMA) sublayer and the Physical Medium Dependent (PMD) sublayer. (See Figure B.1) The PMD sublayer is responsible for transmitting the outgoing symbolstream over the physical medium and receiving the incoming symbolstream from the medium. The PMA sublayer is responsible for clock recovery and (de-)serializing the bitstream. The PCS performs the blocksync and gearbox, (we call this PCS1) scramble/descramble, (PCS2) and encode/decode (PCS3) operations on every Ethernet frame. The IEEE 802.3 Clause 49 explains the PCS sublayer in further detail, but we will summarize below.

When Ethernet frames are passed from the data link layer to the PHY, they are reformatted before being sent across the physical medium. On the transmit (TX) path, the PCS performs 64b/66b encoding and encodes every 64-bit of an Ethernet frame into a 66-bit block (PCS3), which consists of a two-bit *synchronization header* (syncheader) and a 64-bit *payload*. As a result, a 10 GbE link actually operates at 10.3125 Gbaud. ($10G \times \frac{66}{64}$) Syncheaders are used for block synchronization by the remote receive (RX) path.

There are two types of 66-bit *blocks*: A data block and a control block. The data block (/D/) is shown in the third row of Figure B.2 and conveys data characters from Ethernet frames. All other blocks in Figure B.2 are control blocks, which contain a combination of data and control characters. Each rectangle labeled by a D_i represents

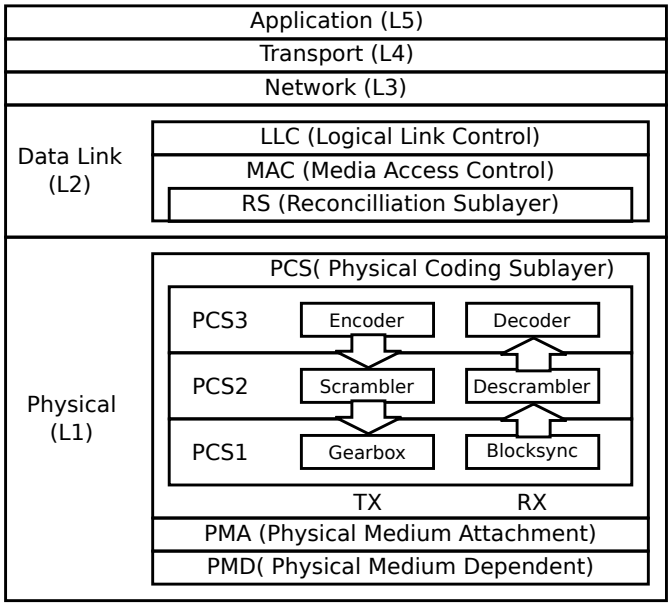


Figure B.1: IEEE 802.3 10 Gigabit Ethernet Network stack.

8-bit data characters, and C_i represents 7-bit control characters. An Idle character (/I/) is one of control characters that are used to fill any gaps between two Ethernet frames. Given an Ethernet frame, the PCS first encodes a *Start* control block, (S_0, and S_4 in Figure B.2) followed by multiple data blocks. At the end of the packet, the PCS encodes a *Terminate* control block (T_0 to T_7) to indicate the end of the Ethernet frame. Note that one of Start control blocks (S_4) and most of Terminate control blocks (T_0 to T_7) have one to seven control characters. These control characters are normally filled with Idle characters. (zeros)

There is a 66-bit Control block, (/E/) which encodes eight seven-bit Idle characters. (/I/) As the standard requires at least twelve /I/sin an interpacket gap, it is *guaranteed* to have at least one /E/ block preceding any Ethernet frame[1]. Moreover, when there is no Ethernet frame, there are always /E/ blocks: 10 GbE is always sending at 10 Gbps and sends /E/ blocks continuously if there are no Ethernet frames to send.

[1]Full-duplex Ethernet standards such as 1, 10, 40, 100 GbE send at least twelve /I/s(at least one /E/) between every Ethernet frame.

	sync	Block Payload							
	0 1	2 Bit Position 65							
/D/	0 1	D0	D1	D2	D3	D4	D5	D6	D7
		BlockType							
/E/	1 0	0x1e	C0	C1	C2	C3	C4	C5	C6	C7
/S0/	1 0	0x78	D1	D2	D3	D4	D5	D6	D7
/S4/	1 0	0x33	C0	C1	C2	C3		D5	D6	D7
/T0/	1 0	0x87		C1	C2	C3	C4	C5	C6	C7
/T1/	1 0	0x99	D0		C2	C3	C4	C5	C6	C7
/T2/	1 0	0xaa	D0	D1		C3	C4	C5	C6	C7
/T3/	1 0	0xb4	D0	D1	D2		C4	C5	C6	C7
/T4/	1 0	0xcc	D0	D1	D2	D3		C5	C6	C7
/T5/	1 0	0xd2	D0	D1	D2	D3	D4		C6	C7
/T6/	1 0	0xe1	D0	D1	D2	D3	D4	D5		C7
/T7/	1 0	0xff	D0	D1	D2	D3	D4	D5	D6	

Figure B.2: IEEE 802.3 64b/66b block format.

The PCS scrambles each encoded 66-bit block (PCS2) to maintain DC balance[2] and adapts the 66-bit width of the block to the 16-bit width of the PMA interface (PCS1; the gearbox converts the bit width from 66- to 16-bit width,) before passing it down the network stack. The entire 66-bit block is transmitted as a continuous stream of *symbols* which a 10 GbE network transmits over a physical medium. (PMA & PMD) On the receive (RX) path, the PCS performs block synchronization based on two-bit syncheaders, (PCS1) descrambles each 66-bit block (PCS2) before decoding it. (PCS3)

Above the PHY is the Media Access Control (MAC) sublayer and Reconciliation Sublayer. (RS) The 10 GbE MAC operates in full duplex mode; it does not handle collisions. Consequently, it only performs data encapsulation/decapsulation and media access management. Data encapsulation includes framing as well as error detection. A Cyclic Redundancy Check (CRC) is used to detect bit corruption. Media access management inserts at least 96 bits (twelve `/I/`characters) between two Ethernet frames. The RS is responsible for additionally inserting or removing idle characters (`/I/`s) between Ethernet frames to maintain the link speed. The channel between the RS and the PHY is called the 10 gigabit media independent interface, (XGMII) and is four bytes wide. (Figure B.1) Data is transferred at every rising and falling edge. As a result, the

[2]Direct current (DC) balance ensures a mix of 1's and 0's is sent.

channel operates at 156.25 MHz. (= 10Gb/32bit/2)

On the TX path, upon receiving a layer 3 packet, the MAC prepends a preamble, start frame delimiter, (SFD) and an Ethernet header to the beginning of the frame. It also pads the Ethernet payload to satisfy a minimum frame-size requirement, (64 bytes) computes a CRC value, and places the value in the Frame Check Sequence (FCS) field. On the RX path, the MAC checks the CRC value and passes the Ethernet header and payload to higher layers while discarding the preamble and SFD.

APPENDIX C

LANGUAGE AND FRAMEWORKS

C.1 Bluespec System Verilog

We used Bluespec System Verilog (BSV) [147] as our main implementation language on FPGA. BSV is a high-level language that is also fully synthesizable to hardware. BSV is used in many design activities: specifications, architectural modeling, design and implementation, verification.

The behavioral model of BSV is popular among formal specification languages for complex concurrent systems, called *Atomic Rules and Interfaces*, which is also known as *Term Rewriting Systems*, *Guarded Atomic Actions*. The model of atomic rules is fundamentally parallel, which is suited for the fine-grain parallelism in complex hardware design. The atomicity of rules is a powerful tool in thinking about correct behavior in the presence of complex parallelism.

BSV's high-level programming languages builds on and borrows ideas from System Verilog and Haskell. It uses ideas from System Verilog for modules and module hierarchy; separation of interfaces from modules; syntax for literals, scalars, expressions, blocks, loops; syntax for user-defined types, (enums, structs, tagged unions, arrays) and so on. BSV uses Haskell for more advanced types, parameterization and static elaboration.

BSV is superior than high level synthesis in C++ in the following ways:

- Architecturally transparent.
- Higher-level abstraction for concurrency.

- Parameterization
- Strong data type system
- Strong static checking
- Fully synthesizable

As a result, BSV enables a single, unified language for designing synthesizable executable specs, synthesizable (fast, scalable) architecture models, synthesizable design and implementation, and synthesizable test environments. All of these can be run on FPGA platforms.

A detailed documentation on the language features of Bluespec can be found in Bluespec Reference Guide. [147] Bluespec beginner is suggested to read the book "Bluespec by example," which has a more general treatment of the language with illustrative examples.

C.2 Connectal Framework

Most modern systems are composed of both hardware and software components. For example, the system prototypes described in this dissertation consist of a host system with the multi-core CPU, system memory, and PCI Express (PCIe) bus, and a PCIe expansion card containing (among other things) a high-performance FPGA chip. The FPGA board was used as a platform to prototype packet processors and low level network stack, such as physical layers.

A common problem when developing with a system architecture as above is the task of enabling communication between host and FPGA board via PCIe bus. The existing

solutions are often ad hoc, tedious, fragile, and difficult to maintain. Without a consistent framework and tool chain for managing the components of the hardware/software boundary, designers are prone to make simple errors which can be expensive to debug.

Connectal is a software-driven hardware development framework. Connectal consists of a fully-scripted tool chain and a collection of libraries which can be used to develop production quality applications comprised of software components running on CPUs communicating with hardware components implemented in FPGA or ASIC.

We give a brief overview of the Connectal work flow and tool chain. The complete tool chain, libraries, and many running examples may be obtained at www.connectal.org. Most of the content below can be found in the introductory paper for Connectal, [96] which we reproduced below.

C.2.1 Top level structure of Connectal applications

The simplest Connectal application consists of 4 files:

Makefile The top-level Makefile defines parameters for the entire application build process. In its simplest form, it specifies which Bluespec interfaces to use as portals, the hardware and software source files, and the libraries to use for the hardware and software compilation.

Application Hardware Connectal applications typically have at least one BSV file containing declarations of the interfaces being exposed as portals, along with the implementation of the application hardware itself.

Top.bsv In this file, the developer instantiates the application hardware modules, connecting them to the generated wrappers and proxies for the portals exported to software. To connect to the host processor bus, a parameterized standard interface is used, making it easy to synthesize the application for different CPUs or for simulation. If CPU specific interface signals are needed by the design, (for example, extra clocks that are generated by the PCIe core,) then an optional CPU-specific interface can also be used.

If the design uses multiple clock domains or additional pins on the FPGA, those connections are also made here by exporting a Pins interface. The Bluespec compiler generates a Verilog module from the top level BSV module, in which the methods of exposed interfaces are implemented as Verilog ports. Those ports are associated to physical pins on the FPGA using a physical constraints file.

Application CPP The software portion of a Connectal application generally consists of at least one C++ file, which instantiates the generated software portal wrapper and proxies. The application software is also responsible for implementing main.

C.2.2 Development Cycles

After creating or editing the source code for the application, the development cycle consists of four steps: generating makefiles, compiling the interface, building the application, and running the application.

Generating Makefiles Given the parameters specified in the application Makefile and a platform target specified at the command line, Connectal generates a target-specific Makefile to control the build process. This Makefile contains the complete dependency

information for the generation of wrappers/proxies, the use of these wrappers/proxies in compiling both the software and hardware, and the collection of build artifacts into a package that can be either run locally or over a network to a remote device under test machine.

Compiling the Interface The Connectal interface compiler generates the C++ and BSV files to implement wrappers and proxies for all interfaces specified in the application Makefile. Human-readable JSON is used as an intermediate representation of portal interfaces, exposing a useful debugging window as well as a path for future support of additional languages and IDLs.

Building the Application A target in the generated Makefile invokes GCC to compile the software components of the application. The Bluespec Compiler (BSC) is then used to compile the hardware components to Verilog. A parameterized Tcl scripts is used to drive Vivado to build the Xilinx FPGA configuration bitstream for the design. The framework also supports Altera tool chains.

Running the Application As part of our goal to have a fully scripted design flow, the generated Makefile includes a run target that will program the FPGA and launch the specified application or test bench. In order to support shared target hardware resources, the developer can direct the run to particular machines, which can be accessed over the network.

APPENDIX D

FPGA IMPLEMENTATION

In this appendix, we describe the FPGA hardware implementations of the P4FPGA, (D.1) SoNIC, (D.2) and DTP (D.3) platforms.

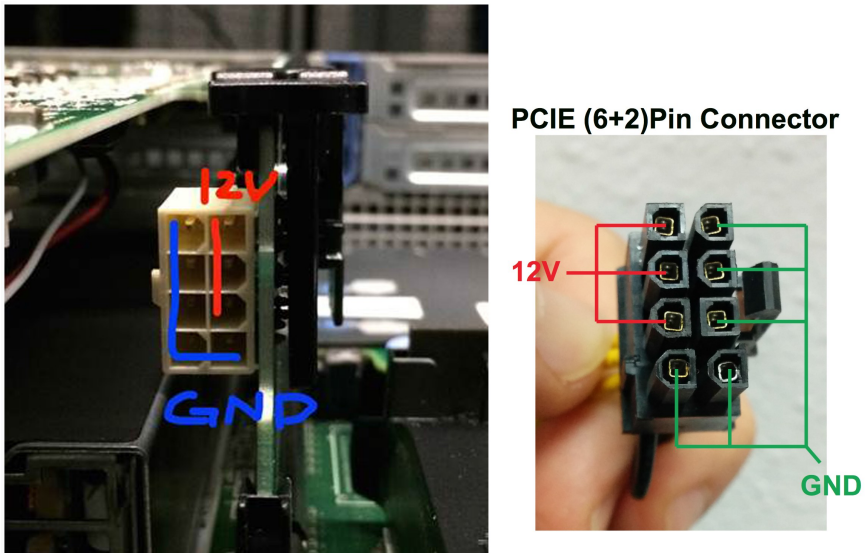
D.1 P4FPGA Hardware Implementation

First, we describe the design and implementation of the P4FPGA platform. We describe the hardware platform on which P4FPGA is developed, the hardware and software interface, and the building blocks of hardware implementation in Bluespec.

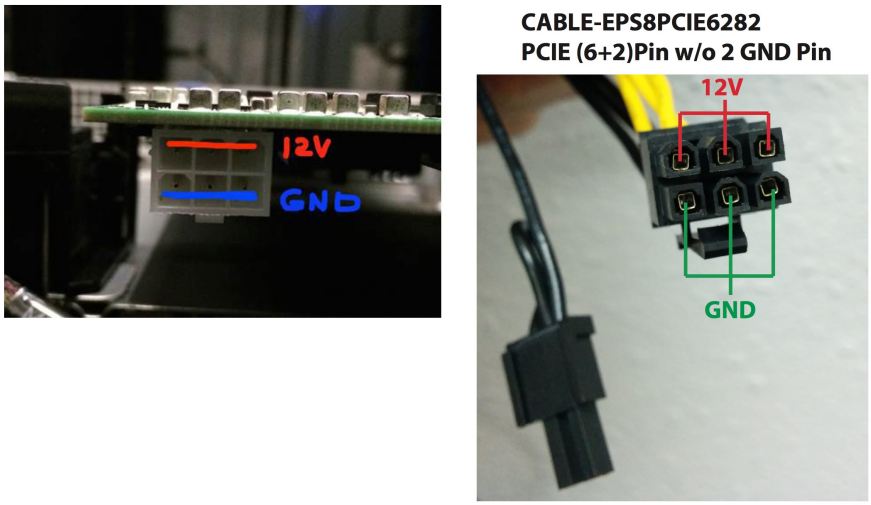
D.1.1 NetFPGA SUME

The NetFPGA SUME [211] platform contains a Xilinx Virtex-7 XC7V690T FPGA [205] that can be programmed with user-defined logic and implements the PCI interface to a host processor. The platform includes three 72 Mbits of Static RAM (SRAM) with 36bit wide buses operating at 500MHz. The platform also contains two 4GB third-generation Double Data Rate (DDR3) Synchronous DRAM (SDRAM) devices that operate at 850Hz. Four Small Form-factor Pluggable (SFP) transceivers are provided to support 10Gbps Ethernet with four globally unique MAC addresses.

The NetFPGA SUME platform receives power via a 2x4 pin PCI Express Auxiliary Power Connector. The 2x4 pin PCI Express Auxiliary Power receptacle can accept both 2x3 and 2x4 pin PCI Express Auxiliary Power Plugs found on a standard Advanced Technology eXtended (ATX) power supply. When used in servers that do not provide standard ATX power supply, e.g. PowerEdge R720, [60] the Graphical Processing Unit



(a) 8-pin end



(b) 6-pin end

Figure D.1: Cable Pin-out to use GPU power source for PCI Express Power Connector.

(GPU) power supply can be used with a special adapter cable. The pin-out of the special adapter cable is shown in figure D.1. It is recommended to always check the correctness of the pin-out with a multi-meter using the connectivity test mode. Off-the-shelf PCI power supply cables may not be compatible with a GPU power source, and may cause damage to the power system of the servers. Cables can be purchased from `http://athenapower.us`.

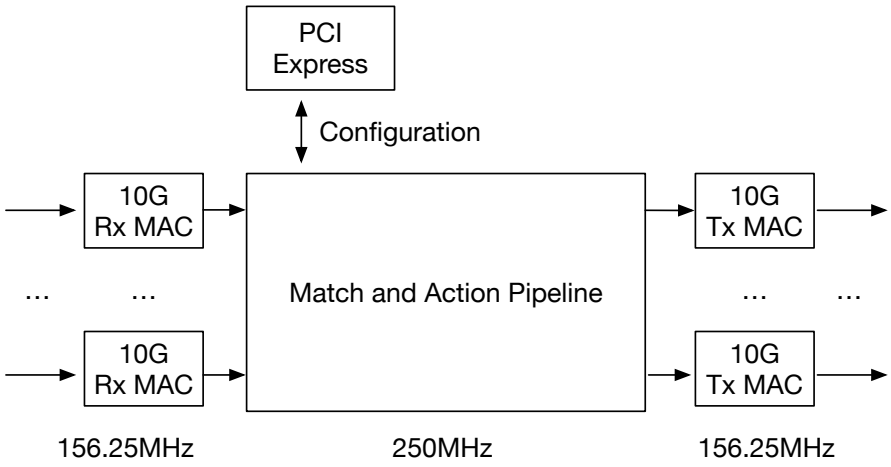


Figure D.2: P4FPGA Clock Domains.

The Xilinx tools typically communicate with FPGAs using the Test Access Port and Boundary-Scan Architecture, commonly referred to as Joint Test Action Group. (JTAG) During JTAG programming, a `.bit` file is transferred from the PC to the FPGA using the on-board USB-JTAG circuitry or an external JTAG programmer. Programming the NetFPGA-SUME with an uncompressed bitstream using the on-board USB-JTAG circuitry usually takes around a minute. With P4FPGA, the programming is done through an open-source tool called `fpgajtag` which is available at `https://github.com/cambridgehackers/fpgajtag`. The `fpgajtag` tool allows developers to program Xilinx FPGAs without installing the vendor-specific tools, such as the hardware server or the iMPACT tool in Vivado. [204]

The P4FPGA has two clock domains: the PCI express clock domain running at 250 Mhz and the 10Gb Ethernet clock domain running at 156.25 Mhz. The PCI express clock domain is used to drive a packet processing pipeline. It has a 512 bit width datapath. The overall theoretical single pipeline throughput is 128Gbps. The 10Gb Ethernet clock domain is used to drive the Ethernet sub-system, including the MAC block and PHY block. (See Figure D.2)

```
1 interface SimpleRequest;
2     method Action read_counter_req(Bit#(32) idx);
3 endinterface
4
5 interface SimpleIndication;
6     method Action read_counter_rsp(Bit#(32) val);
7 endinterface
```

Figure D.3: Simple Connectal Example.

D.1.2 Hardware Software Interface

The hardware-software interface in P4FPGA is implemented using Connectal. [96] (See Appendix C.2) Communications between hardware and software are provided by a bidirectional flow of events and regions of memory shared between hardware and software. Events from software to hardware are called *requests* and events from hardware to software are called *indications*.

We describe the interface between hardware and software components using Bluespec as an Interface Description Language (IDL). A Connectal framework script generates the implementation of the interface, and combines it with libraries to coordinate the data-flow between software and hardware together. A logical request/indication channel is referred to as a *portal*. Each request and indication channel is called a *portal method*. For example, a portal can be used to implement a channel to access hardware counters. Figure D.3 illustrates the definition of a hardware-software interface to read a 32-bit counter from an array of counters in hardware. Line 2 shows the `read_counter_req` function that takes the index of the counter as an input parameter. Line 6 represents the `read_counter_rsp` function that return the value of the counter which is an input to the function. The request function is invoked by a software process on the host, and the response function is invoked by hardware on FPGA.

```
1 interface MemoryTestRequest;
2    method Action read_version();
3    // Packet Generator and Capturer APIs
4    method Action writePacketData(Vector#(2, Bit#(64)) data,
     Vector#(2, Bit#(8)) mask, Bit#(1) sop, Bit#(1) eop);
5    method Action start_pktgent(Bit#(32) iteration, Bit#(32)
     ipg);
6    method Action stop_pktgen();
7    method Action start_pktcap(Bit#(32) iteration);
8    method Action stop_pktcap();
9    // Register Access APIs
10   method Action wr_role_reg(Role role);
11   method Action rd_role_reg();
12   method Action wr_datapath_id_reg(Bit#(DatapathSize)
     datapath);
13   method Action wr_instance_reg(Bit#(InstanceSize) inst);
14   method Action wr_value_reg(Bit#(10) inst, Vector#(8, Bit
     #(32)) value);
15   method Action wr_round_reg(Bit#(10) inst, Bit#(RoundSize)
      round);
16   method Action wr_vround_reg(Bit#(10) inst, Bit#(RoundSize
     ) round);
17   // Table Access APIs
18   method Action sequenceTable_add_entry(Bit#(16) msgtype,
     SequenceTblActionT action);
19   method Action acceptorTable_add_entry(Bit#(16) msgtype,
     AcceptorTblActionT action);
20   method Action dmacTable_add_entry(Bit#(48) mac, Bit#(9)
     port);
21   // Performance Measurement APIs
22   method Action read_ingress_info();
23   method Action read_hostchan_info();
24   method Action read_txchan_info();
25   method Action read_rxchan_info();
26   method Action read_ingress_perf();
27   method Action read_parser_perf();
28   method Action read_pktcap_perf();
29   method Action read_deparser_perf();
30   method Action set_verbosity(Bit#(32) v);
31 endinterface
```

Figure D.4: Paxos Request Interface Definition in Connectal.

```
typedef struct {
   LUInt pktIn;
   LUInt pktOut;
} TableDbgRec deriving (Bits, Eq, FShow);

typedef struct {
   LUInt fwdCount;
   TableDbgRec accTbl;
   TableDbgRec seqTbl;
   TableDbgRec dmacTbl;
} IngressDbgRec deriving (Bits, Eq, FShow);

interface PaxosIndication;
   method Action rd_version_resp(Bit#(32) version);
   method Action rd_ingress_resp(IngressDbgRec rec);
   method Action rd_hostchan_resp(HostChannelDbgRec rec);
   method Action rd_txchan_resp(TxChannelDbgRec rec);
   method Action rd_rxchan_resp(HostChannelDbgRec rec);
   method Action rd_role_resp(Role role);
   method Action rd_ingress_perf_resp(IngressPerfRec rec);
   method Action rd_parser_perf_resp(ParserPerfRec rec);
   method Action rd_pktcap_perf_resp(PktCapRec rec);
   method Action rd_deparser_perf_resp(DeparserPerfRec rec);
endinterface
```

Figure D.5: Paxos Indication Interface Definition in Connectal.

We implemented the control channel between a packet processing pipeline and software network controllers using the hardware-software interface. For example, Figure D.4 and D.5 illustrates the control plane interface for P4Paxos. (described in Section 3.5)

Request APIs Figure D.4 shows the definition for the *request* interfaces. The *request* interfaces specify the commands and associated data sent by the software controller to the FPGA hardware. For example, the P4Paxos *request* interfaces consist of four categories of commands: packet generator commands, register access commands, table access commands and performance measurement commands.

P4Paxos has a built-in packet generator and capturer which can replay packet traces stored in a special packet buffer. The packet generator and capturer are controlled by the APIs shown in Figure D.4 from line 4 to 8. The packet trace in the special packet buffer is programmed via the `writePacketData` function show in line 4 of Figure D.4. The `writePacketData` command takes four parameters: `data`, `mask`, `sop`, `eop`, which respresents a single flit of a packet. A flit of the packet is a chunk of data that can be transferred within one clock cycle. Typically, the amount of bits in a flit is equal to the datapath width. In this case, the flit is 128-bit or 16 bytes wide. The `mask` contains one valid bit for each byte of `data`, and is 16-bit in total. The `sop` and `eop` represent the start flit and end flit of a packet. A 64-byte packet is divided into four flits. The first flit has `sop` set to 1, and the last has `eop` set to 1. The commands in line 5 to 8 control the start and stop of the packet generator and capturer. The `start_pktgen` command takes two parameters: `iteration` and `ipg`. The `iteration` parameter control how many packets to be generated by the packet generator. The `ipg` parameter controls the inter-packet gap between each generated packet to control the generated packet rate.

The P4Paxos program requires stateful memory, such as registers to keep track of the state in the consensus protocol instance. Lines 10 to 16 show the APIs supported by P4Paxos for register access. The software control program can use the `wr_*` APIs to set the initial values in the stateful registers. The software control program can also use the `rd_*` APIs to read the current value in the stateful registers, as shown in line 11 `rd_role_reg`. The value of role register is returned through the *indication* APIs, which we will explain later.

The tables in P4Paxos program are programmed through the table access commands. Lines 18 to 20 show the APIs for adding entries to the tables in P4Paxos. The software control program can call the `sequenceTable_add_entry` command with two pa-

rameters to program the sequence table. (described in Section 3.5.2) The first parameter `msgtype` represents that type of consensus message to match with. The second parameter represents the associated action upon matching the message type.

Finally, the performance of P4Paxos program can be measured by polling the statistics counters embedded in the dataplane. The commands are shown in lines 22 to 29 in Figure D.4. A software control program can invoke these commands to retrieve the corresponding counter values in hardware. The values are returned through the *indication* APIs in Figure D.5.

Indication APIs The indication APIs are used by FPGA hardware to send messages to software. The most common use case is to retrieve various counter values for performance measurement and debugging. The value returned by the indication APIs are defined as a custom *struct* type, (line 1 to 11 in Figure D.5) which may contain more than one counter value. For example, the `rd_ingress_resp` command in line 15 in Figure D.5 returns a `IngressDbgRec` struct, which is defined in line 6 to 11 in Figure D.5. The `IngressDbgRec` struct contains a `LUInt` (64 bit) integer (`fwdCount`) and three `TableDbgRec` structs. (`accTbl, seqTbl, dmacTbl`.) The `TableDbgRec` struct is defined in line 1 to 4 in Figure D.5, which contains two `LUInt`-type integer. (`pktIn, pktOut`.) In total, the `rd_ingress_resp` command sends seven `LUInt`-type integers to software.

D.1.3 Packet Processing Pipeline Templates

We will now discuss how to map P4 into Bluespec. Importantly, P4FPGA requires platform specific details such as data path widths, whereas the P4 language does not re-

quire these details. As a result, the design of P4FPGA additionally requires considering hardware implementation-specific details.

Packet Representation First, We will discuss packet representation used in the P4FPGA pipeline. Fixed length headers are represented as a *struct* in P4FPGA. As shown below in lines 1-4, *Header_t* is a direct translation of the P4 definition into Bluespec.

```
1 typedef struct {
2         Bit#(64) preamble;
3         Bit#(32) numValid;
4 } Header_t;
5
6 Reg#(Header_t) header = mkReg(unpack(0));
7 Reg#(Bit#(96)) bits = mkReg(pack(header));
8 Maybe#(Header_t) hv = tagged Valid header;
```

Figure D.6: An example packet header in P4FPGA in Bluespec.

A header instance may be checked if it is *valid*. We use the *Maybe* type in Bluespec for tagging *valid* or *invalid* to a data type. (line 8) For variable-length packet headers, at most one field may have undefined width, and the total length must be specified to give a upper bound on the size of the header. We use *tagged union* to ensure the variable-length field also has a well-defined type. In addition, conversion between type and canonical bit-level representation is performed by *pack* and *unpack* functions in Bluespec. Note that in the code snippet in Figure D.6, *Reg* is polymorphic.

Parser When a network packet arrives at a receiver MAC layer or PCIe channel, it is first parsed by a parser to extract packet headers for the Match-Action pipeline to be processed. There is one parser for each channel. Therefore, all packets arriving at the same channel are processed in order.

```
1 // Assume fifo_in contains incoming data
2 FIFO#(Bit#(128)) fifo_in <- mkFIFO;
3
4 Stmt parse_head =
5 seq
6         action
7                 let data <- toGet(fifo_in).get;
8                 // Assume data[95:0] contains Header_t
9                 Hdr_t src_route = unpack(data[95:0]);
10                Bit#(32) remaining_bits = data[128:96];
11        endaction
12        action
13        ...
14        endaction
15 endseq
16
17 FSM fsm <- mkFSM(parse_head)
18 Once run_once <- mkOnce(fsm.start);
```

Figure D.7: Parser implementation.

The parser implementation uses an acyclic directed graph, i.e. parse graph, to represent packet headers. [70] In the parse graph, a vertex represents a header field, and an edge represents a transition between header fields. Implementing a parse graph is equivalent to generating a Finite State Machine (FSM). We optimized the implementation of the FSM for performance, resource efficiency and code simplicity. To achieve low latency and high throughput, the parser must process a packet as soon as a sufficient number of bytes have arrived, even if the entire packet has not yet arrived. For example, the parser can start processing the Ethernet header, as soon as the first flit (128-bit) of the packet has been received. This is different from a software implementation of packet parser, such as the ones implemented in software switch, e.g. OpenvSwitch. [164] In a software packet parser, it usually assumes the entire packet is received in a packet buffer before the parsing begins. However, storing the entire packet in the packet buffer in hardware implementation is very costly and inefficient use of on-chip memory resource.

Bluespec provides a convenient language feature to express FSMs, as shown in Fig-

ure D.7. `Stmt` creates a multi-cycle statement (each `action` block is one-cycle) and assigns the statement to a variable `parse_head`, lines 4-15. `parse_head` is used to create a FSM instance `fsm`, line 17, and the execution of `fsm` can be explicitly controlled by certain conditions, line 18, e.g. enough bits are buffered for parsing. Multiple FSMs can be composed sequentially to form a larger FSM, or they can operate in parallel to implement look-ahead or speculative parsers [102]. During parser generation, we need to take into account the physical constraints of the hardware platform, e.g. the width of data path. In our platform, the data-path width is either 128-bits or 256-bit due to the width of the PCI Express engine.

Match and Action Engine In P4FPGA, match table inputs are typed. The implementation of internal match memory depends on the type of match table. For instance, we have implemented binary content addressable memory (BCAM) and hash table to implement an exact match table.

A match table exposes two types of interfaces: dataplane and control-plane interfaces. The match table dataplane interface accepts a match key input and generates an output match result. For instance, an example match table in Figure D.8 accepts the validity of a packet header as the match key and generates a corresponding *drop* or *route* action to the action engine, shown in lines 3-9, 17-18. The match table control plane interface inserts run-time rules into the match table. The definition of a match entry is shown in lines 11-14 and the corresponding match table interface is shown in line 19.

Actions are implemented using atomic-guarded rules in Bluespec. Each *rule* implements a primitive action. Interestingly, the semantics of *rule* in a Bluespec module are identical to the parallel semantics of actions in P4. For example, applying `add_to_field` to the TTL field of an IPv4 packet is shown in line 30-34. An-

```
typedef enum {Drop, Route} ActionType;

typedef struct {
    Bool valid;
} MatchKey deriving (Bits, Eq);

typedef struct{
    ActionType action;
} ActionOut deriving (Bits, Eq);

typedef struct {
    Bool valid;
    ActionType action;
} MatchEntry deriving (Bits, Eq);

interface MatchTable;
    interface Put#(MatchKey) keyIn;
    interface Get#(ActionOut) actionOut;
    interface Put#(MatchEntry) putEntry;
    ...
endinterface

module mkMatchTable(MatchTable);
    ... instantiate CAM to match valid...
    ... instantiate RAM to output action...
endmodule

module mkActionEngine;
    ... internal fifo ...
    rule add_to_field;
        Bit#(8) ttl_now <- toGet(ttl_in).get;
        Bit#(8) ttl_next = ttl_now - 1;
        ttl_out.enq(ttl_next);
    endrule
    ... intf definition ...
endmodule

MatchTable match <- mkMatchTable;
ActionEngine action <- mkActionEngine;
mkConnection(match.actionOut, action.actionIn);
```

Figure D.8: Match table implementation.

other category of primitive actions modifies the layout of a packet. For example, `remove_header` changes the packet layout by removing a header. Removing a header from a packet is equivalent to invalidating the header from the packet buffer. During deparsing, we need to enforce that the deparser skips the invalidated header field.

Control Flow The control flow specifies the order of pipeline stages and thus the order in which a packet would flow through the pipeline stages. The P4FPGA compiler must ensure that match and action stages are connected in the correct order, and that the tables are applied to a packet in the sequence specified in P4 program. Bluespec provides a useful library to connect hardware elements in a specified order. For example, if a match table provides a *Get* interface for data type *ActionOut* and action engine provides a *Put* interface of the same type, then the P4FPGA Bluespec compiler will ensure that the *Put* occurs before the *Get*. We can use *mkConnection* to connect match table and action engine as shown in line 38-40 of Figure D.8.

D.1.4 Code Generation

Configuration The following data summarize the generated P4FPGA configuration.

Ports:

- 4 of 10G Ethernet
- 1 of 8 lane PCIe Gen2.0/3.0 for management and debugging.

Pipeline:

- 1 Pipeline with two stages. (Ingress and Egress.)

- Pipeline supports 5 parsers, each parser is able to parse at 10Gb/s.
- Pipeline has 5 channels. (4 MACs and 1 PCIe.)

Shared Buffer Memory:

- 65KBytes on-chip memory buffer.

Packet size:

- 2048B, no Jumbo packet support.

Parser Generation The generated parser state machine consists of one state for each packet header to be extracted. Within each header state, there are two sub-states: load state and extract state. The code for generating these two states are shown in Figure D.9 and Figure D.10.

Figure D.9 shows the function to generate a Bluespec *rule* for a header state. A *rule* in Bluespec is an atomic unit of execution which completes in one clock cycle **if** the predicate of that rule is true for the clock cycle. Line 11 shows the signature of the function `genLoadRule`. The function returns a *rule* data type as indicated by the `Rules` keyword in front of the function name. The function takes two arguments: `state` and `len`. The `state` argument is a constant value from an *enum* struct which contains all possible header states of the parser, as shown in line 1 to 9 in Figure D.9. The `len` argument is the bit length of the header to be extracted. For example, to generate a load state for Ethernet header, the `len` argument is equal to 112 bit.

The body of the `genLoadRule` function is a `return` statement that returns a rule named `rl_load`. The `rl_load` is predicated by the result of two expressions, as shown

```
 1 typedef enum {
 2   StateStart,
 3   StateParseEthernet,
 4   StateParseIpv4,
 5   StateParseIpv6,
 6   StateParseIcmp,
 7   StateParseTcp,
 8   StateParseUdp
 9 } ParserState;
10
11 function Rules genLoadRule (ParserState state, Integer len);
12   return (rules
13     rule rl_load if ((parse_state_ff.first == state) &&
     rg_buffered < len);
14       data_in_ff.deq;
15       let data = zeroExtend(data_this_cycle) << rg_buffered
     | rg_tmp;
16       rg_tmp <= zeroExtend(data);
17       move_shift_amt(128);
18     endrule
19   endrules);
20 endfunction
```

Figure D.9: Load State Rule Generation.

in line 13. The first expression compares whether the current parser state is to parse the particular header. The second expression checks if enough data have arrived from the receiver MAC layer to be parsed. The current state and buffered data count are stored in global registers that can be accessed by all parser states. The body of the generated `rl_load` rule reads data from input fifo `data_in_ff`, shifts the data by current offset `rg_buffered` in the data buffer `rg_tmp`, and concatenates the data buffer with the new shifted data. (See line 17)

Figure D.10 shows the function to generate the extract state. The signature of the `genExtractRule` function is the same as the `genLoadRule` function. The predicate of the `genExtractRule` function checks if enough data has arrived. If so, the body of `rl_extract` rule invokes the `extract_header` function to extract the

```
1 function Rules genExtractRule (ParserState state, Integer
    len);
2   return (rules
3     rule rl_extract if ((parse_state_ff.first == state) && (
    rg_buffered >= len));
4       let data = rg_tmp;
5       extract_header(state, data);
6       rg_tmp <= zeroExtend(data >> len);
7       succeed_and_next(len);
8       parse_state_ff.deq;
9     endrule
10  endrules);
11 endfunction
```

Figure D.10: Extract State Rule Generation.

```
1 `define COLLECT_RULE(collectrule, rl) collectrule = List::
    cons (rl, collectrule)
2
3 module mkParser(Parser);
4   List#(Rules) parse_fsm = List::nil;
5   `COLLECT_RULE(parse_fsm, joinRules(vec(genLoadRule(
    StateParseEthernet, valueOf(ParseEthernetSz)))));
6   `COLLECT_RULE(parse_fsm, joinRules(vec(genExtractRule(
    StateParseEthernet, valueOf(ParseEthernetSz)))));
7   Vector#(2, Rules) fsmRules = toVector(parse_fsm);
8   Empty fsmrl <- addRules(foldl(rJoin, emptyRules, fsmRules)
    );
9 endmodule
```

Figure D.11: Collect All Generated Rules to State Machine.

packet header from the buffer `rg_tmp` (See line 5) and updates the buffer by shifting out the extract data. (See line 6). Finally, the extract rule changes the current parser state to next available parser state in `parse_state_ff` fifo. (See line 8).

Finally, all generated rules are added to the generated parser module, as shown in Figure D.11. Rules are stored in a *List* called `parse_fsm` which is initialized to an empty list `List::nil`. (See line 4). We wrote a helper function `COLLECT_RULE` to append a rule to a list of rules. Line 8 joins all generated rules with the `foldl` function

and adds the result to the parser module with the built-in `addRules` function.

Match Table Generation Match tables share similar interfaces and have similar behaviors. For example, they all perform look up on a particular match key and execute a pre-defined set of operations on a set of packet headers. In Bluespec, we leverage the *typeclass* to implement a generic match table. Specific match tables are generated as instances of generic implementation.

The code for match table typeclass and instantiation is shown in Figure D.12. Line 1 shows the type for a generic match table, which consists of five generic types: `nact`, `metaI`, `actI`, `keyT`, `valT`. The `nact` represents the number of P4 actions that a match table entry can execute upon matching the key, e.g. a *drop* action or a *forward* action. The `metaI` represents the packet header extracted from the packet. The `actI` represents the tagged union of all possible parameters used by the P4 action. (defined in line 9 to 16, used in line 26). The `keyT` represents the match key *struct*. The `valT` represents the action and parameters stored in the match table, e.g. an entry in a L2 forwarding table may contain an action of *forward* and an additional parameter of outgoing port number.

The typeclass for a match table is defined from lines 2 to 6. The definition takes three parameters: function `table_request`, function `execute_action` and a Content Addressable Memory (CAM). The function `table_request` is another typeclass that must be instantiated by each match table to specify how key look up is implemented. The function `execute_action` is a typeclass that must also be instantiated by each match table to specify the operation on packets after each table lookup. The typeclass definition of each function is shown in lines 18 to 24.

Lines 31 and 32 shows how to declare an instance of the match table from the generic

```
1 typeclass MkTable #(numeric type nact, type metaI, type actI
    , type keyT, type valT);
2   module mkTable#(function keyT table_request(metaI data),
3     function Action execute_action(valT data, metaI md,
4       Vector#(nact, FIFOF#(Tuple2#(metaI, actI))) fifo),
5       MatchTable#(a__, b__, g__, SizeOf#(keyT), SizeOf#(
    valT)) matchTable)
6   (Table#(nact, metaI, actI, keyT, valT));
7 endtypeclass
8
9 typedef union tagged {
10    struct {
11        Bit#(48) dmac;
12    } SetDmacReqT;
13    struct {
14        Bit#(0) unused;
15    } Drop2ReqT;
16 } ForwardParam;
17
18 typeclass Table_request #(type reqT);
19   function reqT table_request (MetadataRequest data);
20 endtypeclass
21
22 typeclass Table_execute #(type rspT, type paramT, numeric
    type num);
23   function Action table_execute (rspT rsp, MetadataRequest
    meta, Vector#(num, FIFOF#(Tuple2#(MetadataRequest, paramT
    ))) fifos);
24 endtypeclass
25
26 typedef Table#(3, MetadataRequest, ForwardParam,
    ConnectalTypes::ForwardReqT, ConnectalTypes::ForwardRspT)
     ForwardTable;
27 typedef MatchTable#(256, SizeOf#(ConnectalTypes::ForwardReqT
    ), SizeOf#(ConnectalTypes::ForwardRspT))
    ForwardMatchTable;
28
29 `SynthBuildModule1(mkMatchTable, String, ForwardMatchTable,
    mkMatchTable_Forward)
30
31  ForwardMatchTable forward_table <- mkMatchTable_Forward("
    forward");
32  Control::ForwardTable forward <- mkTable(table_request,
    table_execute, forward_table);
```

Figure D.12: Match Table Typeclasses and Instances.

typeclass implementation. Line 31 declares a `forward_table` which is the CAM in a match table. Line 32 declares the Match **and** Action stages with instances of the `table_request` and `table_execute` functions, and the `forward_table` CAM.

Control Flow Generation The control flow is generated as part of the ingress and egress pipelines as shown in Figure D.13. Lines 1 to 4 show the interface of the `ingress` module. The ingress module has a `PipeIn` interface that receives `Packet` to be processed. The `Packet` type contains information extracted from a packet by the parser. The `Packet` tokens flow through the ingress pipeline. The exact path of `Packet` token is controlled by the control flow rules shown in lines 17 to 31. For example, a token is always enqueued to the `entry_req_ff` FIFO first. Rule `rl_entry` handles the new token as soon as the fifo is not empty, as predicated by the notEmpty flag of the FIFO shown in line 17. The token is dequeued from the entry fifo, and enqueued to the next stage. A conditional path is implemented with a *if* statement shown in lines 29 to 33. Finally, the token is enqueued to the exit fifo, which is connected to the queueing sub-system or egress pipeline.

Library P4FPGA also implemented a set of library modules for various datapath components. We briefly describe some of them below.

- `PacketBuffer.bsv` implemented a generic ring buffer using block ram on FPGA, which can store arbitrary elements and has a configurable depth.
- `PacketGen.bsv` implemented a packet generator that can replay arbitrary packet traces as user specified speeds up to 10Gbps.
- `SharedBuff.bsv` implemented a shared packet buffer with multiple read and write ports.

```
interface Ingress;
    interface PipeIn#(Packet) prev;
    interface PipeOut#(Packet) next;
endinterface
module mkIngress (Ingress);
    FIFOF#(Packet) entry_req_ff <- mkFIFOF;
    FIFOF#(Packet) entry_rsp_ff <- mkFIFOF;
    FIFOF#(Packet) ipv4_lpm_req_ff <- mkFIFOF;
    FIFOF#(Packet) ipv4_lpm_rsp_ff <- mkFIFOF;
    ... ...
    FIFOF#(Packet) exit_req_ff <- mkFIFOF;
    FIFOF#(Packet) exit_rsp_ff <- mkFIFOF;
    mkConnection(toClient(ipv4_lpm_req_ff, ipv4_lpm_rsp_ff),
    ipv4_lpm.prev_control_state);
    mkConnection(ipv4_lpm.next_control_state[0],
    setnhop_action.prev_control_state);
    mkConnection(ipv4_lpm.next_control_state[1],
    drop1_action.prev_control_state);
    ... ...
    rule rl_entry if (entry_req_ff.notEmpty);
        entry_req_ff.deq;
        let _req = entry_req_ff.first;
        let meta = _req.meta;
        let pkt = _req.pkt;
        Packet req = Packet {pkt: pkt, meta: meta};
        node_2_req_ff.enq(req);
    endrule
    rule rl_node_2 if (node_2_req_ff.notEmpty);
        node_2_req_ff.deq;
        let _req = node_2_req_ff.first;
        let meta = _req.meta;
        if (meta.hdr.ipv4 matches tagged Valid .h &&& h.hdr.
    ttl > 0) begin
            ipv4_lpm_req_ff.enq(_req);
        end
        else begin
            exit_req_ff.enq(_req);
        end
    endrule
    ... ...
    interface prev = toPipeIn(entry_req_ff);
    interface next = toPipeOut(exit_req_ff);
endmodule
```

Figure D.13: Generated Control Flow.

- `SharedBuffMMU.bsv` implemented a memory management unit on top of a shared memory to support fixed-size packet cells, and provide a *malloc* and *free* interface.
- `XBar.bsv` implemented a butterfly crossbar that supports numbers of clients that are powers of two. Latency through XBar scales logarithmically with respect to the number of clients.

In summary, the design of P4FPGA is flexible and efficient to be used to implement a variety of applications. The source code of the generated applications can be found at `https://www.p4fpga.org`.

D.2 SoNIC Hardware Implementation

A high level overview of the SoNIC architecture is shown in Figure 4.3. The SoNIC architecture includes an FPGA firmware to realize the lower parts (Gearbox and BlockSync) of the network physical layer, and a kernel-space software stack that implements the upper parts (Physical Coding Sublayer) of the network physical layer. In this section, we describe the implementation of SoNIC FPGA firmware.

D.2.1 Hardware Overview

The FPGA firmware is built on a HiTech Global Stratix IV platform which contains a Altera Stratix IV EP4S100G2 FPGA. [3] The platform provides one 8-lane PCI Express 2.0 edge connector, and two SFP+ transceivers are provided to support 10Gbps Ethernet. The PCI express block provides features to implement the transaction, data link, and

physical layers of the PCI express protocol. The PCI express block is parameterized to include two virtual channels with Gen2 protocol support.

The FPGA design consists of multiple clock domains. A PCI express clock domain contains direct memory access (DMA) engine and device management registers. The Ethernet physical layer implementation uses a 644.53125 MHz clock which is derived from a on-board programmable oscillator.

Overall, the FPGA firmware includes five major building blocks: PCI Express block, DMA engine, circular packet ring buffers, parts of network physical layer and high-speed transceivers. Among them, the PCI Express block and high-speed transceivers are built-in functionalities in the FPGA, which requires additional configuration. The DMA engine, circular buffer and physical layer design are written in Verilog and synthesized to FPGA firmware.

D.2.2 PCI Express

Figure D.14 shows the configuration for SoNIC PCI Express block. The PCI Express block can be created by entering the configuration through the Mega Wizard GUI or through command line version of the same tool via a tcl script that contains the configuration key value pairs. In particular, we generated a PCI Express Gen 2.0 block for Stratix IV GX device with 8 lanes. The total theoretical bandwidth is 32Gbps. The PCI Express block exposes a 128-bit avalon streaming interface to FPGA application logic. The host can communicate with the PCI Express block via a set of memory-mapped control registers. We use all six available control register regions of the PCI Express block. Base Address Register (BAR) 0-1 and 4-5 are used to implement a DMA channel. Each DMA channel requires two Base Address Register (BAR) regions to implement

```
 1 p_pcie_enable_hip=1,
 2 p_pcie_number_of_lanes=x8,
 3 p_pcie_phy=Stratix IV GX,
 4 p_pcie_rate=Gen2 (5.0 Gbps),
 5 p_pcie_txrx_clock=100 MHz,
 6 p_pcie_app_signal_interface=AvalonST128
 7 p_pcie_bar_size_bar_0="256␣MBytes␣-␣28␣bits"
 8 p_pcie_bar_size_bar_1="N/A"
 9 p_pcie_bar_size_bar_2="256␣KBytes␣-␣18␣bits"
10 p_pcie_bar_size_bar_3e="256␣KBytes␣-␣18␣bits"
11 p_pcie_bar_size_bar_4="256␣MBytes␣-␣28␣bits"
12 p_pcie_bar_size_bar_5="N/A"
13 p_pcie_bar_type_bar_0="64-bit␣Prefetchable␣Memory"
14 p_pcie_bar_type_bar_1="N/A"
15 p_pcie_bar_type_bar_2="32-bit␣Non-Prefetchable␣Memory"
16 p_pcie_bar_type_bar_3="32-bit␣Non-Prefetchable␣Memory"
17 p_pcie_bar_type_bar_4="64-bit␣Prefetchable␣Memory"
18 p_pcie_bar_type_bar_5e="N/A"
19 p_pcie_bar_used_bar_0e="1"
20 p_pcie_bar_used_bar_1="1"
21 p_pcie_bar_used_bar_2="1"
22 p_pcie_bar_used_bar_3="1"
23 p_pcie_bar_used_bar_4="1"
24 p_pcie_bar_used_bar_5="1"
```

Figure D.14: SoNIC PCI Express Configuration.

a 256MByte memory-mapped region. (line 7-8, line 11-12) SoNIC implemented two DMA channels for the two Ethernet ports on board. BAR 2 and 3 are memory-mapped regions of 256 bytes each. They are used to implement a register-based control interface for each DMA channel. For example, the DMA descriptors are transmitted from host to FPGA through these control registers.

The PCI root complex device connects the processor and memory subsystem to the PCI Express switch fabric composed of one or more switch devices. A PCI root complex is set to low speed configuration (2.5 GT/s on Dell T7500 workstation) by default. We use the following commands to configure the PCI root complex to speed by setting the configuration registers in the chipset. This configuration is necessary to run the PCI bus

at full bandwidth.

`sudo setpci -v -G -s 00:03.0 C0.w=2:F` //set target link speed to 5GT/s

`sudo setpci -v -G -s 00:03.0 A0.b=20:20` //retrain link

D.2.3 Transceivers

The FPGA firmware implements the PMA and PMD sublayer of the 10GBase-R PHY. (Appendix B) These sublayers are implemented using Altera transceivers. The Altera transceiver data interface is 40 bits wide and has a throughput of 10312.5Mbps with a transmit (TX) and receive (RX) clock rate of 257.8125MHz.

To generate the 10G PMA transceiver for Stratix IV, we used the Qsys flow in Altera Quartus. [2] The design of a custom PHY is described in the *Altera Transceiver PHY IP Core user guide*. We integrated a PMA-only transceiver, a register management module and a reset controller to the transceiver design. The FPGA transceiver implemented the transmit(TX) phase compensation FIFO, byte serializer on transmit path, and the receive(RX) phase compensation FIFO, clock data recovery (CDR) circuitry, byte deserializer on the receiving path. The TX phase compensation FIFO connects the transmitter channel PCS and the FPGA fabric PCIe interface. It compensates for the phase difference between the low-speed parallel clock and the FPGA fabric interface clock. The byte serializer divides the input datapath by two. This allows the transceiver channel to run at higher data rates while keeping the FPGA fabric interface at lower frequency. On the receive path, the receiver channel has an independent CDR unit to recover the clock from the incoming serial data stream. The high-speed and low-speed

recovered clocks are used to clock the receiver PMA and SoNIC application logic. The deserializer block clocks in serial input data from the receiver buffer using the high-speed serial-recovered clock and deserializes it using the low-speed parallel-recovered clock. It forwards the deserialized data to the SoNIC application logic. The receiver phase compensation FIFO ensures reliable transfer of data and status signals between the receiver channel and the FPGA fabric. The receiver phase compensation FIFO compensates for the phase difference between the parallel receiver PCS clock (FIFO write clock) and the FPGA fabric clock (FIFO read clock).

Figure D.15 shows the configuration of Altera Stratix IV transceivers generated from the Altera Megawizard tool. The transceiver is configured to operate in GT low-latency mode. (line 4). The input bit width is 1 bit and the output bit width is 40 bit. (line 8). We instantiated two lanes (line 6), with each lane operating at 10312.5 Mbps. (line 11). The transceiver block is driven by a clock of 644.53125 MHz. (line 12). We used default analog settings for the transceiver. (lines 29 to 40), but the settings can be updated at runtime using a management interface running at 50MHz. (line 23).

D.2.4 DMA Engine

The high-speed DMA engine is capable of sustaining 32Gbps of raw throughput, which was state-of-art at the time of implementation. The DMA engine implements a chaining DMA native endpoint, which supports simultanenous DMA read and write transactions. The write operations transmit data from the endpoint memory to the FPGA memory. The read operation implements the data path from the FPGA memory to the endpoint memory.

The chaining DMA engine uses a high-performance architecture that is capable of

```
altera_xcvr_low_latency_phy
  #(
    .device_family           ("Stratix␣IV"),
    .intended_device_variant ("GT"),
    .operation_mode          ("DUPLEX"),
    .lanes                   (2),
    .bonded_mode             ("FALSE"),
    .serialization_factor    (40),
    .pll_type                ("CMU"),
    .data_rate               ("10312.5␣Mbps"),
    .base_data_rate          ("10312.5␣Mbps"),
    .pll_refclk_freq         ("644.53125␣MHz"),
    .bonded_group_size       (1),
    .select_10g_pcs          (0),
    .tx_use_coreclk          (0),
    .rx_use_coreclk          (0),
    .tx_bitslip_enable       (0),
    .tx_bitslip_width        (7),
    .rx_bitslip_enable       (0),
    .phase_comp_fifo_mode    ("EMBEDDED"),
    .loopback_mode           ("SLB"),
    .use_double_data_mode    ("true"),
    .mgmt_clk_in_mhz         (50),
    .gxb_analog_power        ("AUTO"),
    .pll_lock_speed          ("AUTO"),
    .tx_analog_power         ("AUTO"),
    .tx_slew_rate            ("OFF"),
    .tx_termination          ("OCT_100_OHMS"),
    .tx_preemp_pretap        (0),
    .tx_preemp_pretap_inv    ("false"),
    .tx_preemp_tap_1         (0),
    .tx_preemp_tap_2         (0),
    .tx_preemp_tap_2_inv     ("false"),
    .tx_vod_selection        (2),
    .tx_common_mode          ("0.65V"),
    .rx_pll_lock_speed       ("AUTO"),
    .rx_common_mode          ("0.82V"),
    .rx_termination          ("OCT_100_OHMS"),
    .rx_eq_dc_gain           (1),
    .rx_eq_ctrl              (16),
    .starting_channel_number (0),
    .pll_refclk_cnt          (1),
    .plls                    (1)
    )
```

Figure D.15: SoNIC PMA Configuration.

transferring a large amount of fragmented memory without having to access the DMA register for every memory block. The engine uses a descriptor table which contains the length of the transfer, the address of the source and destination of the transfer, and the control bits.

Before initiating a DMA transfer, the control software writes the descriptor tables into the shared memory region between FPGA and host. After that, the control software programs the DMA engine's control register to inform the DMA engine about the total number of descriptor tables and the memory address of the first descriptor table. After programming the DMA control register, the DMA engine continously fetches descriptors from the shared memory region for both the read and write operations, and then performs the data transfer according to each descriptor.

Figure D.16 shows an example DMA descriptor table, which is stored in shared host memory. It consists of four-dword descriptor header and a contiguous list of four-dward descriptors. The chaining DMA engine iteratively fetches four-dword descriptors to start a DMA transfer. After completion of a DMA transfer, it sends an update to the host to record the number of descriptors completed to the descriptor header. The host software can poll on the `EPLAST` field of the descriptor header to decide whether the chain of DMA transfers have been completed.

A reference design of the DMA engine is generated using tools in Altera Quartus. [2]

D.2.5 Ring Buffer

The buffer implementation has been optimized for SoNIC. (Section 4.2.2). Ring buffers are used as temporary buffer space for sent and received packets. They are required

Address offset	Type	Description
0x0	Header	Reserved
0x4		Reserved
0x8		Reserved
0xC		EPLAST, when enabled, this location records the number of the last descriptor completed by the DMA engine
0x10	Descriptor 0	Control field, DMA Length
0x14		Endpoint address
0x18		RC address upper dword
0x1C		RC address lower dword
0x20	Descriptor 1	Control field, DMA Length
0x24		Endpoint address
0x28		RC address upper dword
0x2C		RC address lower dword
…		
0x..0	Descriptor <n>	Control field, DMA Length
0x..4		Endpoint address
0x..8		RC address upper dword
0x..C		RC address lower dword

Figure D.16: SoNIC DMA Descriptor Format.

because of the data-rate mismatch between the DMA engine and the transceivers. The DMA engine operates at 32Gbps throughput, which means the instantenous data transfer rate of the DMA engine from the host to FPGA can be as fast as 32Gbps, and vice versa. If there is no buffering between the DMA engine and the rest of the FPGA system, data will be lost on the transmit path. Similarly, if there is no buffering on the receiving path, packets may be dropped. The transmit ring buffer is written by the DMA engine and read by the transceiver control block. The receive ring buffer is written by the transceiver

Vaddr	Vaddr(bin)	Paddr	Paddr(bin)	Types
0	0b0000000	0	0b0000000	First sync header byte
1	0b0000001	1	0b0000001	...
2	0b0000010	2	0b0000010	...
...	...	...	...	...
123	0b1111011	123	0b1111011	Last sync header byte
124	reserved	...	...	...
125	reserved	...	...	...
126	reserved	...	...	...
127	reserved	...	...	...
128	0b0000_1000_0000	0	0b0000_0000_0000	First data byte
...	...	...	...	...
4094	...	...	...	...
4095	0b1111_1111_1111	3967	0b1111_0111_1111	Last data byte

Table D.1: DMA Memory Page Layout

control block and read by the DMA engine.

The design of the ring buffer is a trade-off between DMA driver performance and hardware control logic simplicity. In particular, the linux driver is optimized to request data in 4KB pages through the DMA engine. Each address issued by DMA points to 128-bit data block, because the data width of the DMA engine is 128 bits. For 4KB pages, this corresponds to 4096B/16B or 256 address offsets for each DMA transfer. To further optimize software performance, memory access must be 64-bit aligned. We used a customized memory page layout as shown in Table D.1.

For each 4KB memory page, the segment at the top of the page is used to store 2-bit sync-headers for each 66-bit data in the PCS layer. The rest of the page stores data blocks associated with the sync headers. As a result, in each memory page, 124 bytes are used to store the sync headers, 3968 bytes are used to store data blocks and 4 bytes are reserved. In total, 496 66-bit blocks are stored in a single page. The ring buffer

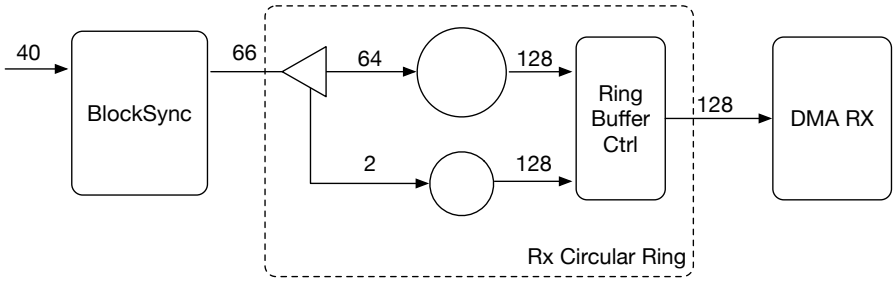


Figure D.17: Rx Circular Ring.

memory contains 8192 128-bit entries, which means it could store up to 32 pages.

Rx Ring Buffer Figure D.17 shows the design of Rx ring buffer, and how it is connected to the BlockSync module on the network side and the DMA Rx module on the host side. The Rx circular ring receives inputs from blocksync, which are 66-bit blocks per cycle. It then outputs 128-bit blocks to the DMA engine. Therefore, the Rx ring buffer is also called a 66-to-128 ring buffer.

Inside the 66-to-128 ring buffer, there are two smaller ring buffers. The first one is the `sync_ring` buffer, which is used to store sync-headers. It has a 2 bit-wide input and 128 bit-wide output. The `sync_ring` buffer needs 124/8 = 15.5 entries to store all the sync headers. We round the space requirement up to 128 bytes or 16 128-bit entries. The other ring buffer is the `data_ring` buffer. It has 64 bit-wide input and 128 bit-wide output. The `data_ring` is used to store the 64 bit data block associated with each 2-bit sync-header in a 66-bit block.

In each DMA transaction, one page is requested, with sync headers stored in the first 128 bytes. The valid sync headers are from byte 0 to byte 123. In addition, byte 128 to byte 4095 contain the correponding 64-bit data-block of the 66-bit block. As previously metioned, each page fits 496 64-bit blocks, or 248 128-bit blocks, as shown in the following example:

- Page 0
  - 128 bytes of sync header from `sync_ring` address [0:15].
  - 3968 bytes of data block from `data_ring` address [16:256].
- Page 1:
  - 128 bytes of sync header from `sync_ring` address [16:31].
  - 3968 bytes of data block from `data_ring` address [272:512].
- Page 2:
  - 128 bytes of sync header from `sync_ring` address [32:47].
  - 3968 bytes of data block from `data_ring` address [528:756].
- ...
- Page 31:
  - 128 bytes of sync header from `sync_ring` address [496:512].
  - 3968 bytes of data block from `data_ring` address [7944:8192].

Since it is possible to store up to 32 pages with the current ring buffer size, there are 256 (8192 - 7936) 128-bit unused entries in the `data_ring` memory. To align things better, the unused entries of `data_ring` memory are distributed sparsely in each page. For example, the first 16 128-bit entries in the `data_ring` memory are not used, because they are shadowed by the `sync_header` ring. We store the first entries of 64-bit data block in entry 16 in the `data_ring` buffer.

From the DMA engine point of view, the two rings (`sync_ring` and `data_ring`) appear to be a single ring buffer. The address mapping from DMA address to ring address is handled by the ring buffer control module.

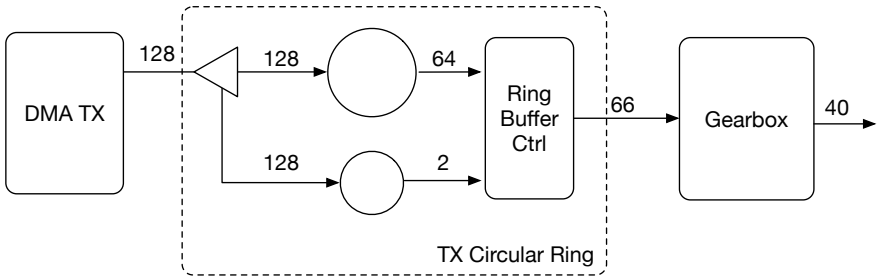


Figure D.18: Tx Circular Ring.

From the blocksync module point of view, the ring buffer is also a single entity, because the 66-bit incoming block is split by the ring buffer control module, and the 2-bit sync headers are stored in `sync_ring`. 64-bit data blocks are stored in `data_ring`.

From the driver's perspective, it only issues DMA transactions if there is a full page worth of data available in the Rx ring. As a result, the Rx ring buffer needs to indicate to the driver the amount of available data in the unit of pages. The driver keeps track of the available space in the ring buffer and specifies the proper DMA address offset to the ring buffer during DMA transaction.

Tx Ring Buffer Figure D.18 shows the implementation of the transmit circular buffer. Every DMA transaction transfers data in pages of 4096B to the Tx ring. The start address of each page is used as offset into the Tx ring. For example, the first page starts at offset 0x0, the second page starts at offset 0x100, the third page starts at offset 0x200 and the last page is at offset 0x1F00. The format of the memory page layout is the same as the Rx ring, as shown in Table D.1. Each page has 496 or 0x1F0 entries. When translating the read address 0x0 to the memory address of the data ring, 0x0 maps to 0x10 in the `data_ring`, and 0x10 in the `sync_ring`. The conversion is simple: first, the bit address of the sync header i in the page is computed as $32 + 2i$, since the first sync header starts at bit offset 32. The bit address of the data block i in the page is $(32 + 2i) * 32$, since the first data block starts at bit offset 1024. The read address to sync

ring is $(32 + 2i)/2$ or $16 + i$. The read address to data ring is $(32 + 2i) * 32/64$ or $16 + i$. Therefore, the addresses to index the sync and data ring are always the same.

The external circuitry does not observe the offset in the sync/data ring. Therefore, when a read request to address 0 arrives at the tx ring, a ring buffer control block translates the external address 0 to internal address 0x10. The external address 0x1F0 is translated to internal address 0x200. Symbolically, the address translation unit implements the following:

$$external\ address + 16 * pagenumber = internal\ address$$

Tx ring waits for an entire page to be received, before it sends the data to the network. The delay of sending packets is at most the time it takes to send a page-worth of data from the host to the Tx ring.

D.2.6 BlockSync and Gearbox

BlockSync In the ingress direction, the block synchronization (blockSync) is responsible for identifying and locking on each of the 66-bit blocks in the bit stream using the sync headers. The physical layer transmits data in blocks of 66 bits. The first two bits of a block are the synchronization header. Blocks are either data blocks or control blocks. The sync header is 01 for data blocks and 10 for control blocks. Lock is obtained as specified in the block lock state machine shown in Figure D.19.

The implementation of the blocksync state machine is shown in Figure D.20 and Figure D.21. When system reset is asserted, or the transceiver signal is invalid, the blocksync state machine enters the init state `LOCK_INIT`. As soon as the link status becomes ready, the blocksync state machine performs an unconditional transition (UCT)

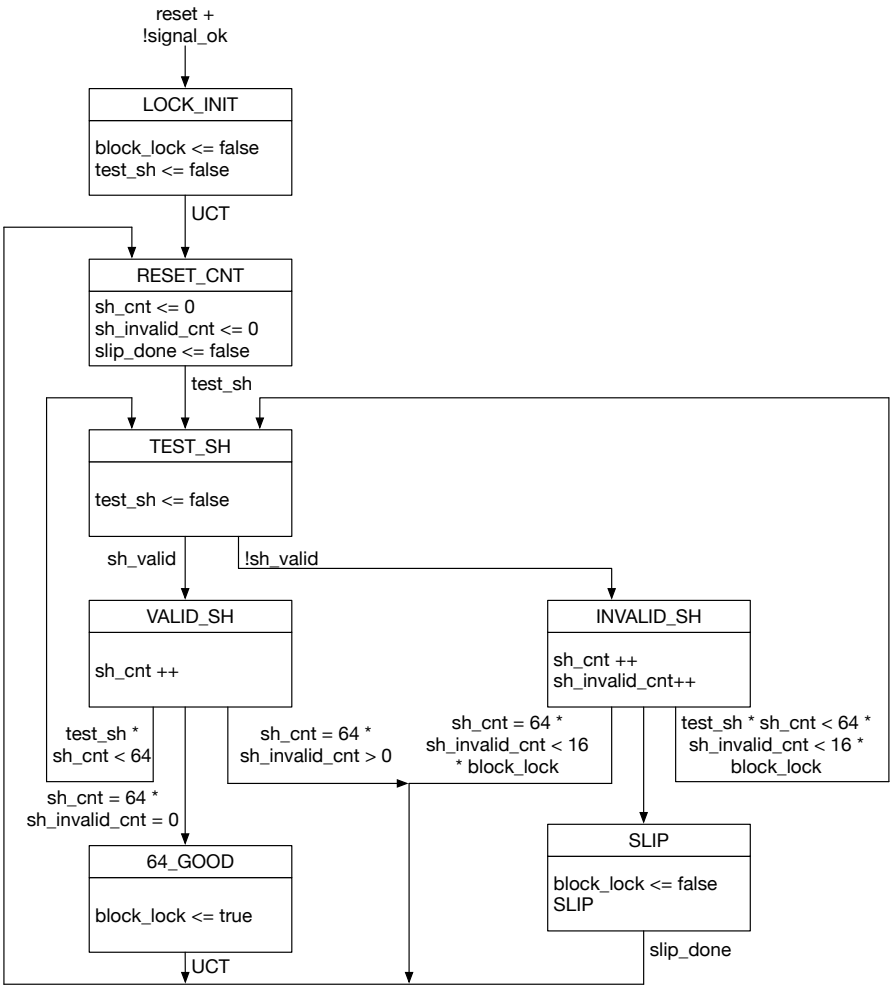


Figure D.19: BlockSync State Machine.

to state RESET_CNT. The RESET_CNT state resets three state variables to their initial values (lines 11 to 14 in Figure D.20). Next, the test_sh boolean variable that is set *true* when a new sync header is available for testing, and *false* when TEST_SH state is entered. If the received block has valid sync header bits, the sh_valid boolean indication is true, and the state machine transits to the VALID_SH state. Otherwise, the state machine transits to the INVALID_SH state. In the VALID_SH state, the count of the number of sync headers checked within the current 64 block window is incremented. If the count is less than 64, then state machine returns to TEST_SH state, and examines the next data block. If the count is equal to 64, and the count of the number of

```
1   /* output depends on state */
2   always @ (posedge clk) begin
3      case (state)
4        LOCK_INIT: begin
5           block_lock = 0;
6           offset = 0;
7           test_sh = 0;
8        end
9
10       RESET_CNT: begin
11          sh_cnt = 0;
12          sh_invalid_cnt = 0;
13          slip_done = 0;
14          test_sh = valid;
15       end
16
17       TEST_SH: begin
18          test_sh = 0;
19       end
20
21       VALID_SH: begin
22          sh_cnt = sh_cnt + 1;
23          test_sh = valid;
24       end
25
26       INVALID_SH: begin
27          sh_cnt = sh_cnt + 1;
28          sh_invalid_cnt = sh_invalid_cnt + 1;
29          test_sh = valid;
30       end
31
32       GOOD_64: begin
33          block_lock = 1;
34          test_sh = valid;
35       end
36
37       SLIP: begin
38          if (offset >= 66) offset = 0;
39          else offset = offset + 1;
40          slip_done = 1;
41          block_lock = 0;
42          test_sh = valid;
43       end
44     endcase // case (state)
45  end
```

Figure D.20: BlockSync State Machine Output Transition.

```
1 /* determine next state */
2 always @ (posedge clk or posedge reset) begin
3    if (reset) begin
4       state <= LOCK_INIT;
5    end
6    else begin
7       case (state)
8         LOCK_INIT:
9           state <= RESET_CNT;
10        RESET_CNT:
11          if (test_sh) state <= TEST_SH;
12        TEST_SH:
13          if (sh_valid) state <= VALID_SH;
14          else state <= INVALID_SH;
15        VALID_SH:
16          if (test_sh & (sh_cnt < 64)) begin
17             state <= TEST_SH;
18          end
19          else if (sh_cnt == 64 & sh_invalid_cnt == 0) begin
20             state <= GOOD_64;
21          end
22          else if (sh_cnt == 64 & sh_invalid_cnt > 0) begin
23             state <= RESET_CNT;
24          end
25        INVALID_SH:
26          if (sh_cnt == 64 & sh_invalid_cnt < 16 &
    block_lock) begin
27             state <= RESET_CNT;
28          end
29          else if (sh_invalid_cnt==16|!block_lock) begin
30             state <= SLIP;
31          end
32          else if (test_sh & sh_cnt < 64 & sh_invalid_cnt <
    16 & block_lock) begin
33             state <= TEST_SH;
34          end
35        GOOD_64:
36          state <= RESET_CNT;
37        SLIP:
38          if (slip_done) state <= RESET_CNT;
39      endcase // case (state)
40   end
41 end
```

Figure D.21: BlockSync State Machine Next State Computation.

invalid sync headers within the current 64 block window is larger than 0, then the state machine transits to RESET_CNT state to restart a new 64 block window. Lastly, if 64 consecutive valid sync headers are received, and there are no invalid sync headers in the 64 block window, the state machine has successfully locked to the bit stream, the block_lock signal is set to true, and the state machine restarts a new 64 block window. If the TEST_SH state transits to INVALID_SH state, the count of the number of invalid sync headers within the current 64 block window is incremented. If the total number of sync headers is 64 and the number of invalid sync header is less than 16 and the previous 64 block window has successfully locked the bit stream, then the state machine transits to TEST_SH state. Otherwise, if the current 64 block window has not filled and the number of invalid sync header is less than 16 and the previous 64 block window has locked on the bit stream, then the state machine transits to TEST_SH state. Finally, the state machine releases the lock on the bit stream and shifts to a new offset in the bit stream to verify the sync header validity.

Gearbox 10G Ethernet data is 64B/66B encoded in the egress direction and 66B/64B decoded in the ingress direction. Each data block from or into the PCS block is 66 bits wide. When configured at 10.3125 Gbps for 10G Ethernet, the Stratix IV GT transceiver channel supports a parallel data width of 40 bits.

In the egress direction, the gearbox logic is responsible for translating each 66 bit-wide data block from the PCS into 40 bit-wide data to the Stratix IV GT transmitter channel. In the ingress direction, the gearbox logic is responsible for translating the 40 bit-wide data from each Stratix IV GT receiver channel into 66 bit-wide data to the PCS.

The 40-bit-to-66-bit gearbox implementation consists of two shift registers: SR0 and SR1. Each shift register is 66 bits wide. Figure D.22 shows the logic for implementing

```
 1 |St| SR0 (66 bits)  |         SR1 (66 bits)          |V|
 2   0  ----------------------------[39              0] 0
 3   1  --------[13   0] [65     40][39              0] 1
 4   2  ------------------------[53          14][13  0] 0
 5   3  -----[27      0] [65:54][53          14][13  0] 1
 6   4  ----------[1:0] [65          28][27         0] 1
 7   5  ---------------------------[41          2][1:0] 0
 8   6  -------[15    0] [65    42][41          2][1:0] 1
 9   7  ------------------------[55         16][15   0] 0
10   8  ------[29     0] [65:56][55         16][15   0] 1
11   9  ----------[3:0] [65           30][29         0] 1
12  10  ------------------------[43         4][3     0] 0
13  11  ---    [17    0] [65 44][43         4][3     0] 1
14  12  ------------------------[58      18][17      0] 0
15  13  -----[31      0] [65 58][57      18][17      0] 1
16  14  ---------- [5:0] [65           32][31         0] 1
17  15  -------------------------[45         6][5    0] 0
18  16  -------[19    0] [65  46][45         6][5    0] 1
19  17  ------------------------[59      20][19      0] 0
20  18  -----[33      0] [65  60][59     20][19      0] 1
21  19  ----------[7:0] [65        34][33           0] 1
22  20  ------------------------[47        8][7      0] 0
23  21  -------[21    0] [65   48][47      8][7      0] 1
24  22  --------------------[61      22][21           0] 0
25  23  -[35           0] [65:62][61  22][21          0] 1
26  24  ---------[9   0] [65      36][35              0] 1
27  25  ----------------------[49          10][9     0] 0
28  26  -------[23    0] [65   50][49      10][9     0] 1
29  27  --------------------[63        24][23         0] 0
30  28  [37            0] [65:64][63    24][23         0] 1
31  29  --------[11   0] [65         38][37           0] 1
32  30  ----------------------[51        12][11      0] 0
33  31  ----[25       0] [65  52][51     12][11      0] 1
34  32  -----------------[65          26][25          0] 1
```

Figure D.22: 40 bit to 66 bit gearbox logic.

the 40-to-66 gearbox. The logic in Figure D.22 is implemented as a state machine, identified with the number in the first column `St`. The state machine has 33 states, from 0 to 32. Each state generates a 66-bit output with a valid signal to represent whether the data output is valid to use. Data enters the gearbox with LSB first. The data is shifted to the left by the number of bits stored in SR1 and appended to SR1. The overflow bits from SR1 are stored in SR0. When all 66 bits in SR1 contain received data, the stored data is sent to output and remaining bits are shifted to the right by 66 bits.

The 66-bit-to-40-bit gearbox implementation consists of two registers: SR0 and SR1. Figure D.23 shows the logic for implementing the 66-to-40 gearbox. The logic in Figure D.23 is also implemented as a state machine with 33 states, identified with the number in the first column `St`. The numbers in each row of Figure D.23 represent the subset of bits in each register that are the valid output bits for this state. For example, in state 0, the lower 40 bits in SR0, i.e. SR0[39:0] are the output bits of this state. The 66-to-40 gearbox generates valid output in every cycle. New data is always stored in SR0. The data in SR0 and SR1 are shifted when the status variable `sh` is 1. In addition, the gearbox can issue a read signal to buffer memory to read new data by setting the `rd` signal to 1. When the state machine reaches state 32, the next state wraps around to state 0.

In summary, the SoNIC FPGA firmware implements a high performance DMA engine that operates at Gen2.0 x8 speed. The DMA transfers data between the host and the FPGA ring buffers at 32 Gbps, which is able to support two 10G Ethernet ports at line rate. To optimize software performance, the DMA engine employs a custom memory layout in host, as well as offloading part of the physical layer functionality to hardware, e.g. gearbox and blocksync. Overall, the SoNIC FPGA firmware enables the capability of processing every bit in physical layer in software, which enables the programmable

```
|St| sr0 (66 bits) |         sr1   (66 bits)   |rd|sh|
  0 **[39         0] ************************** 1  1
  1 *********[13  0] [65       40]************ 1  1
  2 ***************** *****[53           14]**** 0  0
  3 *****[27      0] [65:54]******************* 1  1
  4 ***********[1:0] [65             28]******* 1  1
  5 ***************** *********[41            2] 0  0
  6 ********[15   0] [65     42]*************** 1  1
  7 ***************** ********[55         16]*** 0  0
  8 *****[29      0] [65:56]******************* 1  1
  9 ***********[3:0] [65            30]******** 1  1
 10 ***************** **********[43        4]*** 0  0
 11 ******[17     0] [65   44]***************** 1  1
 12 ***************** ******[58          18]**** 0  0
 13 ****[31       0] [65  59]****************** 1  1
 14 ***********[5:0] [65            32]******** 1  1
 15 ***************** *********[45         6]*** 0  0
 16 ***** [19     0] [65   46]***************** 1  1
 17 ***************** ******[59         20]***** 0  0
 18 ****[33       0] [65   60]***************** 1  1
 19 ***********[7:0] [65        34]************ 1  1
 20 ***************** *********[47         8]*** 0  0
 21 ******[21     0] [65   48]***************** 1  1
 22 ***************** ******[61         22]***** 0  0
 23 **[35         0] [65:62]******************* 1  1
 24 ***********[9:0] [65         36]*********** 1  1
 25 ***************** *******[49          10]*** 0  0
 26 *****[23      0] [65   50]***************** 1  1
 27 ***************** ***[63           24]****** 0  0
 28 [37           0] [65:64] ****************** 1  1
 29 **********[11:0] [65         38]*********** 1  1
 30 ***************** *******[51          12]*** 0  0
 31 *****[25      0] [65   52]***************** 1  1
 32 ***************** [65              26]****** 0  0
```

Figure D.23: 66 bit to 40 bit gearbox logic.

PHY.

D.3 DTP Hardware Implementation

In this section, we describe DTP hardware implementation. We describe the hardware platform, the implementation of the Physical Coding Sublayer (PCS) in FPGA, the DTP protocol sublayer, and the control interface based on the Connectal framework.

D.3.1 Altera DE5

DTP uses the Terasic DE5-Net Stratix V GX FPGA Development Kit. [191] The DE5 platform contains four external 10G Small form-factor pluggable (SFP+) modules, and a PCI Express 3.0 x8 edge connector. The DE5 platform has four Independent 550MHz SRAM, with 18-bits data bus and 72Mbit for each SRAM. It also contains two Independent DDR3 SODIMM Socket with up to 8GB 800 MHz or 4GB 1066 MHz for each socket.

D.3.2 Physical Layer Implementation

As shown in Figure 4.28, the DTP hardware implementation consists of the standard Ethernet PCS sublayer and DTP sublayer. The PCS sublayer consists of four building blocks: encoder, decoder, scrambler and descrambler. Note, for SoNIC in Section 4 and Appendix D.2, these four blocks of the PCS were implemented in software, but for DTP, we implemented them in hardware. In hardware, all four blocks are implemented using

Bluespec language and synthesized to Altera FPGA using the Quartus FPGA design suite.

Encoder and Decoder The *encoder* and *decoder* blocks implement 64b/66b line code. The line code transforms 64-bit data to 66-bit line code to provide enough state changes to allow reasonable clock recovery and facilitate alignment of the data stream at the receiver. The *encoder* implements 64-bit to 66-bit conversion. The 66 bit entity is made by prefixing one of two possible two-bit preambles to the 64 bits to be transmitted. If the preamble is 01, the 64 bits are entirely data. If the preamble is 10, an eight-bit type field follows, plus 56 bits of control information and/or data. The preambles 00 and 11 are not used, and generate an error if used. The input interface to *encoder* block is the 10 Gigabit Media Independent Interface (XGMII) interface, which is a standard interface between the MAC and PHY layer of 10 Gigabit Ethernet (10GbE). The XGMII interface consists of 72 bits, where 64 of them are data bits and 8 of them are control bits. The control bits are used to indicate whether the data bits are data block or control block. The output interface of the *encoder* block is 66 bits wide and connected to the DTP sublayer. The *decoder* block implements the reverse process of decoding a 66-bit line code to 64-bit data. The input to the *decoder* block is from DTP sublayer, and the output of *decoder* block is connected to MAC layer.

Scrambler and Descrambler The scrambler block converts an input string into a seemingly random output string of the same length to avoid long sequences of bits of the same value. Because of this, a scrambler is also referred to as a randomizer. The purpose of scrambling is to enable accurate timing recovery on receiver equipment without resorting to redundant line coding. It facilitates the work of a timing recovery circuit. There can be many types of scramblers. For 10G Ethernet, we implemented the follow-

ing polynomial:

$$X = x^{58} + x^{39} + 1$$

The input to the scrambler is a 66-bit data block and the scrambler produces a scrambled data block of the same length. The data block may contain DTP timestamp messages. The descrambler performs the reverse process of descrambling a pseudo-random data block. As a result, if a DTP message is embedded into a data block before scrambling, the same message will be recovered after descrambling at the receiver end.

DTP sublayer The DTP sublayer implements the DTP algorithm as shown in Algorithm 3. The implementation of the algorithm contains two modules in two different clock domains. The module on receive path (Rx module) is controlled by receiving clock domain. The Rx module implements a multiplexer based on received data block type. If a block is a data block type $/D/$ or any of the control block types except the idle type $/E/$, then the block is forwarded as is. However, if the data block is of type $/E/$, the received block is further processed by the received module as a DTP message. If a valid DTP message is found, the associated timestamp will be extracted and forwarded to the DTP logic on the transmission path. (Tx module). The Tx module performs two tasks. First, it computes the local timestamp by comparing the timestamp received from Rx module with the local timestamp, and selects the maximum value to update the local timestamp. Second, the Tx module inserts a local timestamp to a DTP message to transmit to the remote node. Further, if a Tx module is used in a DTP-enabled switch, the module will guarantee all ports share the same timestamps by computing the maximum of local timestamps from all ports, and using the maximum value as global timestamp.

D.3.3 Control Interface

The DTP sublayer is accessible from host for debugging and management. We used Connectal [96] (see Appendix C.2) to implement a set of APIs to interact with DTP sublayer. For example, the user can configure the rate at which clock synchronization messages are exchanged between two nodes by setting the `dtp_frequency` counter via a `set_frequency` API method. The user can also read the current state of DTP protocol sublayer, the measured latency between two DTP-enabled ports, the local timestamp on each DTP-enabled port, and the global timestamp in a DTP-enabled switch via *get* API methods. Finally, the DTP timestamps are exposed to a userspace daemon to provide the synchronized clock service to distributed applications.

D.3.4 Bluespec Implementation

We implemented the DTP PHY in Bluespec. Figure D.24 shows the top level module for the DTP PHY. The top level module contains the instantiation of two sub-modules: `DTPController` and `EthPhy`. The `DTPController` implements the control and management interface for DTP PHY. The details about the control interface functions are explained later. The `EthPhy` implements the physical layer datapath of 10G Ethernet, including the encoder, decoder, scrambler, descrambler and the DTP sub-layer. The physical layer communicates with the upper layer in the network stack through the 10G Media Independent Interface (XGMII). In this example, the Tx path of the XGMII interface (line 19-21) transmits IDLE frame as encoded as an 72-bit constant number in hexdecimal format. (0x83c1e0f0783c1e0f07). The Rx path of the XGMII interface receives 72-bit every cycle, (lines 23-25), and the bits are dropped by default. If there exists an MAC layer to generate Ethernet frames, the XGMII interfaces of the PHY will

```
1 `define IDLE_FRAME 0x83c1e0f0783c1e0f07
2
3 module mkDtpTop#(DtpIndication indication)(DtpTop);
4    Clock defaultClock <- exposeCurrentClock();
5    Reset defaultReset <- exposeCurrentReset();
6
7    De5Clocks clocks <- mkDe5Clocks();
8    De5SfpCtrl#(4) sfpctrl <- mkDe5SfpCtrl();
9    Clock txClock = clocks.clock_156_25;
10   Clock phyClock = clocks.clock_644_53;
11   Clock mgmtClock = clocks.clock_50;
12
13   Reset txReset <- mkAsyncReset(2, defaultReset, txClock);
14
15   DtpController dtpCtrl <- mkDtpController(indication,
     txClock, txReset);
16   DtpPhyIfc dtpPhy <- mkEthPhy(mgmtClock, txClock, phyClock
     , clocked_by txClock, reset_by dtp_rst);
17   mkConnection(dtpPhy.api, dtpCtrl.ifc);
18
19   rule send_idle_frame;
20      dtpPhy.phys.tx.put(IDLE_FRAME);
21   endrule
22
23   rule recv_idle_frame;
24      let v <- dtpPhy.phys.rx.get;
25   endrule
26
27   interface request = dtpCtrl.request;
28   interface pins = mkDE5Pins(defaultClock, defaultReset,
     clocks, dtpPhy.phys, leds, sfpctrl, buttons);
29 endmodule
```

Figure D.24: DTP Top Level Module in Bluespec.

be connected to the corresponding XGMII interface of the MAC layer.

DTP Controller Figure D.25 shows the software to hardware interface definition. Software can retrieve various status register values from the DTP sub-layer. For example, `dtp_read_delay` (line 5) will return the measured latency between the local DTP node to its neighboring DTP node. `dtp_read_state` will return the current state

```
 1 interface DtpRequest;
 2    method Action dtp_read_version();
 3    method Action dtp_reset(Bit#(32) len);
 4    method Action dtp_set_cnt(Bit#(8) port_no, Bit#(64) c);
 5    method Action dtp_read_delay(Bit#(8) port_no);
 6    method Action dtp_read_state(Bit#(8) port_no);
 7    method Action dtp_read_error(Bit#(8) port_no);
 8    method Action dtp_read_cnt(Bit#(8) cmd);
 9    method Action dtp_logger_write_cnt(Bit#(8) port_no, Bit
      #(64) local_cnt);
10    method Action dtp_logger_read_cnt(Bit#(8) port_no);
11    method Action dtp_read_local_cnt(Bit#(8) port_no);
12    method Action dtp_read_global_cnt();
13    method Action dtp_set_beacon_interval(Bit#(8) port_no,
      Bit#(32) interval);
14    method Action dtp_read_beacon_interval(Bit#(8) port_no);
15    method Action dtp_debug_rcvd_msg(Bit#(8) port_no);
16    method Action dtp_debug_sent_msg(Bit#(8) port_no);
17    method Action dtp_debug_rcvd_err(Bit#(8) port_no);
18    method Action dtp_debug_tx_pcs(Bit#(8) port_no);
19    method Action dtp_debug_rx_pcs(Bit#(8) port_no);
20    method Action dtp_set_mode(Bit#(8) mode);
21    method Action dtp_get_mode();
22 endinterface
```

Figure D.25: DTP Control API definition in Connectal.

of the DTP state machine. The control interface also provides debugging functions to the counters at different points in the datapath (lines 15-19) to monitor message performance and loss rate. Finally, the control interface provides configuration functions the control the behavior of the DTP state machine. For example, it can control the mode of the DTP state machine (switch or NIC) (lines 20-21). The user can also configure the frequency at which DTP messages are generated using the `set_beacon_interval` command. (line 13). Most commands can be applied to one of the four DTP ports on the FPGA board. Therefore, there is only one parameter to specify the port number.

Standard PHY functionalities Standard physical layer functionalities, such as encoder, decoder, scrambler and descrambler, are implemented in Bluespec for perfor-

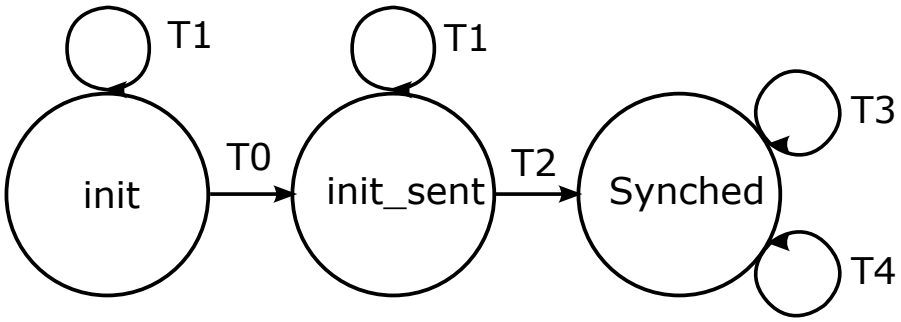


Figure D.26: DTP State Machine.

mance reason. In particular, we need the datapath to perform at line rate (10Gbps) with deterministic latency. Implementing the physical layer can be done in software at line rate, as we have shown in SoNIC. But the end-to-end latency has too much variance for the precise clock synchronization. In the interest of space, we will not show the source code for these modules. We further note that these modules do not take part in the DTP operation.

DTP state machine There are three states in the DTP state machine: INIT, SENT, and SYNC, three of which are used in the initial phase to measure point-to-point latency between two DTP-enabled PHYs. The state machine diagram is shown in Figure D.26. Upon reset, the DTP state machine is initialized to INIT state. It will transition to SENT state when the state machine has sent an `INIT` message to its neighbor, as shown in Figure D.27. In the SENT state, the state machine will monitor the received data until it receives an `ACK` message from the neighbor node. Upon receiving an `ACK` message, the state machine will transit to SYNC state. Otherwise, the state machine will stay in SENT state. (lines 1-15 in Figure D.28.) Finally, once the state machine arrives at SYNC state, it will remain in SYNC state until a reset event. (lines 17-27 in Figure D.28.)

The beacon messages are sent periodically. The period is controlled by a register `sync_timeout`, which will increment every cycle until it reaches to a configurable

```
1   rule state_init (curr_state == INIT);
2      let init_type = fromInteger(valueOf(INIT_TYPE));
3      cfFifo.deq;
4      // update states
5      if (txMuxSelFifo.notEmpty && is_idle) begin
6         if (txMuxSelFifo.first == init_type) begin
7            curr_state <= SENT;
8         end
9      end
10     else begin
11        curr_state <= INIT;
12     end
13  endrule
```

Figure D.27: DTP INIT State Implementation.

```
1   rule state_sent (curr_state == SENT);
2      let init_type = fromInteger(valueOf(INIT_TYPE));
3      cfFifo.deq;
4
5      // update states
6      if (init_rcvd) begin
7         curr_state <= SENT;
8      end
9      else if (ack_rcvd) begin
10        curr_state <= SYNC;
11     end
12     else begin
13        curr_state <= SENT;
14     end
15  endrule
16
17  rule state_sync (curr_state == SYNC);
18     cfFifo.deq;
19
20     // update states
21     if (init_rcvd) begin
22        curr_state <= SYNC;
23     end
24     else begin
25        curr_state <= SYNC;
26     end
27  endrule
```

Figure D.28: DTP SENT State and SYNC State Implementation.

```
1    rule beacon(curr_state == SYNC);
2       dmFifo.deq;
3       let sync_timeout = interval_reg._read;
4       let beacon_type = fromInteger(valueOf(BEACON_TYPE));
5       let ack_type = fromInteger(valueOf(ACK_TYPE));
6       let rxtx_delay = fromInteger(valueOf(RXTX_DELAY));
7       if (timeout_count_sync >= sync_timeout-1) begin
8          if (is_idle) begin
9             timeout_count_sync <= 0;
10         end
11         else begin
12            timeout_count_sync <= timeout_count_sync + 1;
13         end
14         if (txMuxSelFifo.notFull) begin
15            txMuxSelFifo.enq(beacon_type);
16         end
17      end
18      else if (init_rcvd) begin
19         if (txMuxSelFifo.notFull) begin
20            txMuxSelFifo.enq(ack_type);
21         end
22         timeout_count_sync <= timeout_count_sync + 1;
23      end
24      else begin
25         timeout_count_sync <= timeout_count_sync + 1;
26      end
27   endrule
```

Figure D.29: DTP Beacon Message Generation.

time-out value. Upon time out, a beacon message will be generated and sent, and the time out counter will be cleared. The logic for beacon message generation is shown in Figure D.29.

The point-to-point delay is measured by subtracting the sent timestamp of a SYNC message from the receive timestamp of the ACK message. The logic is shown in Figure D.30. In order to accurately measure the point-to-point, the computed timestamp difference must be compensated by the internal delay of the DTP layer, as well as divided by two. This is because the timestamp difference measures the round-trip time, instead of one-way delay.

```
1    // delay measurement
2    rule delay_measurment(curr_state == INIT || curr_state ==
      SENT);
3       let init_timeout = interval_reg._read;
4       let init_type = fromInteger(valueOf(INIT_TYPE));
5       let ack_type = fromInteger(valueOf(ACK_TYPE));
6       let rxtx_delay = fromInteger(valueOf(RXTX_DELAY));
7       dmFifo.deq;
8       // compute delay
9       if (ack_rcvd) begin
10         let temp <- toGet(ackTimestampFifo).get;
11         Bit#(53) tmp2 = zeroExtend(cycle);
12         delay <= (tmp2 - temp - 3) >> 1;
13      end
14   endrule
```

Figure D.30: DTP Delay Measurement Implementation.

Figure D.31 illustrates how DTP messages are inserted into the IDLE frame at the transmit path. In particular, lines 10 to 27 show the implementation of inserting INIT, ACK and BEACON messages into an IDLE frame. If an encoded frame is IDLE, then part of the IDLE frame will be reused to carry DTP-specific fields, such as INIT message, (line 14), ACK message, (line 19), or BEACON message, (line 22), otherwise, if there is no DTP message to send, the encoded frame will be forwarded as is. (line 25). In addition, the DTP transmit module also implements the ability to send LOG messages for monitoring purposes. Lines 29 to 40 implement the LOG message operation. Host can send a timestamp through the DTP datapath to a collector using a special LOG message type.

Figure D.32 shows the implementation of the receive path. The task of the receive datapath is to parse the DTP messages and extract the associated fields, e.g. timestamps, in the message. Lines 4 to 34 implement the parsing logic for each of the four types of messages mentioned above. If an INIT message is received, the logic will extract the timestamp embedded in the INIT message, and send the timestamp to the local DTP time

```
1   rule tx_stage;
2       let val <- toGet(stageOneFifo).get;
3       let v <- toGet(dtpTxInPipelineFifo).get();
4       let mux_sel = val.mux_sel;
5       let c_local = val.c_local;
6       let parity = val.parity;
7       Bit#(10) block_type;
8       Bit#(66) encodeOut; block_type = v[9:0];
9       if (mux_sel && txMuxSelFifo.notEmpty) begin
10         let sel = txMuxSelFifo.first;
11         if (sel == init_type) begin
12            Bit#(53)    tmp = zeroExtend(cycle);
13            encodeOut = {tmp+1, parity, INIT_TYPE,
    block_type};
14         end
15         else if (sel == ack_type) begin
16            let init_timestamp <- toGet(initTimestampFifo).
    get;
17            let init_parity <- toGet(initParityFifo).get;
18            encodeOut = {init_timestamp, init_parity,
    ACK_TYPE, block_type};
19         end
20         else if (sel == beacon_type) begin
21            encodeOut = {c_local+1, parity, BEACON_TYPE,
    block_type};
22         end
23         else encodeOut = v;
24         txMuxSelFifo.deq;
25      end
26      else if (mux_sel && fromHostFifo.notEmpty) begin
27         let host_data = fromHostFifo.first;
28         debug_from_host <= host_data;
29         if (host_data[52] == 0) begin
30             Bit#(52) tmp = c_local[51:0] + 1;
31             encodeOut = {0, tmp, log_type, block_type};
32         end
33         else encodeOut = {host_data, LOG_TYPE, block_type};
34         fromHostFifo.deq;
35      end
36      else encodeOut = v;
37      dtpTxOutFifo.enq(encodeOut);
38  endrule
```

Figure D.31: DTP Transmit Path Implementation.

synchronization logic to compare with the local timestamp. Similarly, if a BEACON or ACK message is received, the correponding timestamp will be sent to the handler logic of such events.

Finally, the time synchronization logic simply computes the maximum value among the global, local or remote timestamps for the switch mode, and computes the maximum value among local or remote timestamps for the NIC mode. The implementation is shown in Figure D.33.

In summary, the DTP sublayer consists of a state machine and message parsing and generation logic, all of which can be concisely expressed in Bluespec. The resulting implementation runs at 10Gbps and is capable of sending millions of synchronization messages at no network layer overhead, and achieve 10s of nanosecond clock synchronization precision.

```
1    rule rx_stage;
2       if (dtpEventInFifo.notEmpty) begin
3          let v <- toGet(dtpEventInFifo).get;
4          if ((v.e == INIT_TYPE)) begin
5             let parity = ^(v.t);
6             if (initTimestampFifo.notFull && initParityFifo.
    notFull) begin
7                initTimestampFifo.enq(v.t);
8                initParityFifo.enq(parity);
9             end
10            init_rcvd_next = True;
11         end
12         else if (v.e == ACK_TYPE) begin
13            ack_rcvd_next = True;
14            // append received timestamp to fifo
15            if (ackTimestampFifo.notFull)
16               ackTimestampFifo.enq(v.t);
17         end
18         else if (v.e == BEACON_TYPE) begin
19            beacon_rcvd_next = True;
20            localCompareRemoteFifo.enq(c_local+1);
21            remoteCompareLocalFifo.enq(v.t+1);
22            if (is_switch_mode) begin
23               localCompareGlobalFifo.enq(c_local+1);
24               remoteCompareGlobalFifo.enq(v.t+1);
25            end
26         end
27         else if (v.e == LOG_TYPE) begin
28            log_rcvd_next = True;
29            if (toHostFifo.notFull) begin
30               toHostFifo.enq(v.t);
31            end
32         end
33         dtpEventFifo.enq(v.e);
34      end
35      else begin
36         dtpEventFifo.enq(0);
37      end
38   endrule
```

Figure D.32: DTP Receive Path Implementation.

```
1   rule compare_global_remote (is_switch_mode);
2      let global_delay = fromInteger(valueOf(GLOBAL_DELAY));
3      let v_global <- toGet(globalCompareRemoteFifo).get();
4      let v_remote <- toGet(remoteCompareGlobalFifo).get();
5      if (v_global + global_delay <= v_remote + delay) begin
6         globalLeRemoteFifo.enq(True);
7         globalGtRemoteFifo.enq(False);
8      end
9      else begin
10        globalLeRemoteFifo.enq(False);
11        globalGtRemoteFifo.enq(True);
12     end
13     globalOutputFifo.enq(v_global + 1);
14  endrule
15
16  rule compare_global_local (is_switch_mode);
17     let global_delay = fromInteger(valueOf(GLOBAL_DELAY));
18     let v_global <- toGet(globalCompareLocalFifo).get();
19     let v_local <- toGet(localCompareGlobalFifo).get();
20     if (v_global + global_delay <= v_local + 1) begin
21        globalLeLocalFifo.enq(True);
22        globalGtLocalFifo.enq(False);
23     end
24     else begin
25        globalLeLocalFifo.enq(False);
26        globalGtLocalFifo.enq(True);
27     end
28  endrule
```

Figure D.33: DTP Timestamp Comparison Logic.

APPENDIX E

GLOSSARY OF TERMS

Accuracy Closeness of a measured value to a standard or known value. In clock synchronization, it is the closeness of a read time to a reference time. In available bandwidth estimation, it is the closeness of an estimated bandwidth to the actual bandwidth available. See also *clock synchronizatoin* and *availabld bandwidth*.

Available bandwidth The maximum data rate that a system can send down a network path to another system without going over the capacity between the two.

Bit error rate (BER) The ratio of the number of bits incorrectly delivered from the number of bits transmitted.

Bitstream A sequence of bits.

Clock domain crossing (CDC) delivering a signal from one clock comain into another in a digital circuit.

Clock recovery In high-speed data communications such as Ethernet, bitstreams do not carry clock signals. As a result, the receiving device recovers clock from the transitions in the received bitstreams. There must be enough transitions (one to zero or zero to one) for easier clock synchronization, which is achieved by scrambler in Ethernet.

Clock skew The time difference between two clocks. See also *offset*.

Computer network A collection of computers and network devices that communicate data via data links.

Covert channel A channel that is not intended for information transfer, but can leak sensitive information.

Covert storage channel See also *covert channel*.

Covert timing channel See also *covert channel*.

Cut-through switch Switch starts forwarding a frame once the destination port is known without checking CRC value. It does not store an entire frame before forwarding. Switching latency is lower than store-and-forward switches. See also *Cyclic Redundancy Check*.

Cyclic redundancy check (CRC) A check value that is computed from data and sent along with data to detect data corruption at the receiver.

Datacenter A facility that consists of racks of servers and a network connecting servers along with the physical infrastructure and power.

Device driver A computer program that controls a hardware device attached to a system.

Direct memory access (DMA) Transfer of data in and out of main memory without involving central processing unit.

Ethernet frame A unit of data in Ethernet See also *Ethernet*

Ethernet A family of standards specified in IEEE 802.3 that is used for data communication between network components in local area networks.

Field-programmable gate array (FPGA) An intergrated circuit that can be configured and programmed after manufacture.

First in first out (FIFO) A data structure where the oldest entry is processed first.

Hardware timestamping See also *timestamp*.

Homogeneous packet stream See also *interpacket delay*, *interpacket gap* and *network packet*.

Interpacket delay (IPD) The time difference the first bits of two successive packets.

Interpacket gap (IPG) The time difference between the last bit of the first packet and the first bit of the next packet.

Kernel The core component of operating systems, managing systems resources and hardware devices, providing interface to userspace programs for accessing system resources.

Metastability An unstable state in a digital circuit where the state does not settle onto '0' or '1' within a clock cycle.

Network application A program running on one host that is communicating with other programs running on other machines over a network.

Network component Network interface cards, switches, routers, any other devices that build a network and send, receive or process network packets. See also *computer network* and *network packet*

Network covert channel A network covert channel sends hidden messages over legitimate packets by modifying packet headers or by modulating interpacket delays. See also *covert channel*.

Network device See *network component*

Network interface card (NIC) Device attached to a computer that connects the host computer to a network.

Network measurement Measuring the amount and type of network traffic on a network.

Network node See*network component*

Network packet A unit of data being transferred in a network.

Network switch Multi-port network device that forwards network packets to other network devices at the data link layer (Layer 2). Some switches also support Layer 3 forwarding. See also *network stack*.

Network traffic Data being moved in a network.

Network See *computer network*.

Offset In clock synchronization, offset means the time difference between two clocks.

One way delay (OWD) The time it takes for a message to travel across a network from source to destination.

Operating system (OS) A software that manages hardware and software resources and provides abstractions and services to userspace programs.

Oscillator A circuit or device that generates a priodically oscillating signal.

Pacing Controlling time gaps between network packets.

Packet See *network packet*.

Peripheral component interconnect express (PCIe) A standard for serial communication bus between a computer and peripheral devices.

Peripheral device Hardware device that is attached to a computer.

Precision Closeness of two or more measurements to each other. In clock synchronization, it is the degree to which clocks are synchronized, or the maximum offset between any two clocks. In timestamping packets, it is the closeness of the timestamp to the actual time it was received. In pacing packets, it is the closeness of an intended time gap to the actual time gap between two messages. See also *offset*, *timestamp*, and *pacing*.

Process An instance of program that is being executed.

Robustness is how to deliver messages with minimum errors.

Round trip time (RTT) The sum of the length of time it takes for a request to be sent and the length of time it takes for a response to be received.

Router Network device that forwards network packets to other network devices at the network layer (Layer 3). See also *network stack*.

Store-and-forward switch Switch stores an entire frame, verifies CRC value, and forwards it to destination port. See also *Cyclic Redundancy Check*.

Symbol stream A sequence of symbols. See also *bitstream* and *symbol*.

Symbol A pulse in the communication channel that persists for a fixed period of time and that represents some number of bits.

Synchronization FIFO A special FIFO that delivers data between two clock domains. See also *clock domain crossing* and *first-in-first-out*.

System clock The number of seconds since the epoch (1 January 1970 00:00:00 UTC).

System time What a system clock reads. See also *system clock*

Time synchronization protocol A protocol that achieves clock synchronization in a network.

Timeserver A server that reads the reference time from an atomic clock or a GPS receiver and distributes the time to other systems in a network.

Timestamp counter (TSC) A register in Intel processors that counts the number of cycles since reset or boot.

Timestamp The time at which an event is recorded, such as when a packet is transmitted or received.

Transceiver A device that can both transmit and receive signals.

Undetectability is how to hide the existence of a covert chennel.

Userspace All code that run outside the operating systems' kernel. It includes programs and libraries that provide interface to the operating system. See also *operating system* and *kernel*

BIBLIOGRAPHY

[1] Altera. PCI Express High Performance Reference Design. http://www.altera.com/literature/an/an456.pdf.

[2] Altera Quartus II. http://www.altera.com/products/software/quartus-ii/subscription-edition.

[3] Altera Stratix IV FPGA. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/stxiv-index.jsp.

[4] Amazon Web Services. http://amazonaws.com.

[5] Broadcom. http://http://www.broadcom.com/products/Switching/Data-Center.

[6] Endace DAG Network Cards. http://www.endace.com/endace-dag-high-speed-packet-capture-cards.html.

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

[8] Fibre Channel. http://fibrechannel.org.

[9] High Frequency sFlow v5 Counter Sampling. ftp://ftp.netperf.org/papers/high_freq_sflow/hf_sflow_counters.pdf.

[10] Highly Accurate Time Synchronization with ConnectX-3 and Timekeeper. http://www.mellanox.com/pdf/whitepapers/WP_Highly_Accurate_Time_Synchronization.pdf.

[11] HiTech Global, LLC. http://hitechglobal.com.

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

[13] IEEE Standard 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[14] IEEE Standard 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[15] Intel Westmere Processor. http://ark.intel.com/products/codename/33174/Westmere-EP.

[16] Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/netlink.7.html.

[17] Mellanox. www.mellanox.com.

[18] Myricom Sniffer10G. http://www.myricom.com/sniffer.html.

[19] sFlow, Version 5. http://www.sflow.org/sflow_version_5.txt.

[20] Stratix V FPGA. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp.

[21] The CAIDA Anonymized Internet Traces. http://www.caida.org/datasets/.

[22] Timekeeper. http://www.fsmlabs.com/timekeeper.

[23] Timing characteristics of a synchronous Ethernet equipment slave clock. http://www.itu.int/rec/T-REC-G.8262.

[24] Xilinx. http://www.xilinx.com/.

[25] IEEE 1588 PTP and Analytics on the Cisco Nexus 3548 Switch. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper-c11-731501.html, 2014.

[26] Ameer M.S. Abdelhadi and Guy G.F. Lemieux. Modular SRAM-Based Binary Content-Addressable Memories. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015.

[27] AccelDSP Synthesis Tool. http://www.xilinx.com/tools/acceldsp.htm.

[28] Mohammad Al-Fares, Rishi Kapoor, George Porter, Sambit Das, Hakim Weatherspoon, Balaji Prabhakar, and Amin Vahdat. Netbump: User-extensible active queue management with bumps on the wire. In *Proceedings of the Eighth*

ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '12, New York, NY, USA, 2012. ACM.

[29] Algorithms in Logic. `www.algo-logic.com`.

[30] Ahmed Ait Ali, Fabien Michaut, and Francis Lepage. End-to-End Available Bandwidth Measurement Tools : A Comparative Evaluation of Performances. *arXiv.org*, June 2007.

[31] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Procs. NSDI*, 2012.

[32] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[33] Bilal Anwer, Theophilus Benson, Nick Feamster, Dave Levin, and Jennifer Rexford. A slick control plane for network middleboxes. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[34] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[35] Axonerve. Axonerve Low Latency Matching Engine Synthesizable IP Core.

[36] Barefoot. Barefoot Networks. `http://www.barefootnetworks.com/`.

[37] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol–http/1.0. Technical report, 1996.

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

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

[40] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, August 2013.

[41] Gordon Brebner and Weirong Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE/ACM International Symposium on Microarchitecture*, 34(1):8–18, January 2014.

[42] Broadcom. Ethernet time synchronization. `http://www.broadcom.com/collateral/wp/StrataXGSIV-WP100-R.pdf`.

[43] Stephen Brown and Jonathan Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[44] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–26, 2004.

[45] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, November 2006.

[46] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP covert timing channels: Design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[47] Martin Casado. Reconfigurable networking hardware: A classroom tool. In *Proceedings of Hot Interconnects 13*, 2005.

[48] Ceph. `http://ceph.com`.

[49] Samarjit Chakraborty, Simon Künzli, Lothar Thiele, Andreas Herkersdorf, and Patricia Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5):641–665, 2003.

[50] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, August 2007.

[51] John T. Chapman, Rakesh Chopra, and Laurent Montini. The DOCSIS timing protocol (DTP) generating precision timing services from a DOCSIS system. In *Proceedings of the Spring Technical Forum*, 2011.

[52] Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[53] Corsa. Corsa DP6420 OpenFlow data plane. `http://www.corsa.com/products/dp6420`.

[54] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, September 1989.

[55] Mark Crovella and Balachander Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. John Wiley and Sons, Inc, 2006.

[56] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR)*, 44:87–95, April 2016.

[57] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM SOSR*, pages 59–73, June 2015.

[58] Matthew Davis, Benjamin Villain, Julien Ridoux, Anne-Cecile Orgerie, and Darryl Veitch. An IEEE-1588 Compatible RADclock. In *Proceedings of International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2012.

[59] Stephen E Deering. Internet protocol, version 6 (ipv6) specification. 1998.

[60] Dell. PowerEdge R720 rack server. `http://www.dell.com/us/dfb/p/poweredge-r720/pd`.

[61] Udit Dhawan and André Dehon. Area-efficient near-associative memories on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):30:1–30:22, January 2015.

[62] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proc. SOSP*, 2009.

[63] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[64] DPDK. `http://dpdk.org/`.

[65] Nandita Dukkipati, Glen Gibb, Nick McKeown, and Jiang Zhu. Building a rcp (rate control protocol) test network. In *Hot Interconnects*, volume 15, 2007.

[66] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[67] Daniel A. Freedman, Tudor Marian, Jennifer H. Lee, Ken Birman, Hakim Weatherspoon, and Chris Xu. Exact temporal characterization of 10 Gbps optical wide-area network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[68] Steven Froehlich, Michel Hack, Xiaoqiao Meng, and Li Zhang. Achieving precise coordinated cluster time in a cluster environment. In *Proceedings of International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2008.

[69] Eli Gafni and Leslie Lamport. Disk Paxos. In *Distributed Computing*, LNCS, pages 330–344, February 2000.

[70] Glen Gibb. *Reconfigurable Hardware for Software-defined Networks*. PhD thesis, Stanford University, 2013.

[71] Peter Gomber, Björn Arndt, Marco Lutat, and Tim Uhle. High-frequency trading. *Available at SSRN 1858626*, 2011.

[72] Vinodh Gopal, Erdinc Ozturk, Jim Guilford, Gil Wolrich, Wajdi Feghali, Martin Dixon, and Deniz Karakoyunlu. Fast CRC computation for generic polynomials using PCLMULQDQ instruction. White paper, Intel, `http://download.intel.com/design/intarch/papers/323102.pdf`, December 2009.

[73] Gotthard: Moving Transaction Logic to the Switch. `https://github.com/usi-systems/gotthard`.

[74] Albert Greenberg. Networking the cloud. In *ICDCS*, page 264, 2009.

[75] Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. White paper, Intel, `http://software.intel.com/file/24918`, January 2010.

[76] Cesar D. Guerrero and Miguel A. Labrador. On the applicability of available bandwidth estimation techniques and tools. *Computer Communication*, 33(1):11–22, January 2010.

[77] Riccardo Gusella and Stefano Zatti. The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, July 1989.

[78] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos, Gianni Antichi, Nirav H. Dave, Andrew William Moore, and Peter G. Neumann. Blueswitch: Enabling provably consistent configuration of network switches. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 17–27, Washington, DC, USA, 2015. IEEE Computer Society.

[79] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[80] Khaled Harfoush, Azer Bestavros, and John Byers. Periscope: An active internet probing and measurement api. Technical report, Boston University Computer Science Department, 2002.

[81] Ningning Hu, Li Erran Li, Zhuoqing Morley Mao, Peter Steenkiste, and Jia Wang. Locating internet bottlenecks: algorithms, measurements, and implications. In *Proceedings of the ACM SIGCOMM 2004 Conference*, 2004.

[82] Pter Hga, Attila Psztor, Darryl Veitch, and Istvn Csabai. Pathsensor: Towards efficient available bandwidth measurement. In *Proceedings of IPS-MoMe 2005*, 2005.

[83] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolić. Consensus in a Box: Inexpensive Coordination in Hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, March 2016.

[84] Manish Jain and Constantinos Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *In Proceedings of Passive and Active Measurements (PAM) Workshop*, pages 14–25, 2002.

[85] Manish Jain and Constantinos Dovrolis. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proc. IMC*, 2004.

[86] Raj Jain, , and Shawn A. Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal On Selected Areas in Communications*, 4:986–995, 1986.

[87] Hao Jiang and Constantinos Dovrolis. Why is the internet traffic bursty in short time scales? In *Proc. SIGMETRICS*, 2005.

[88] Guojun Jin and Brian L. Tierney. System capability effects on algorithms for network bandwidth measurement. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, pages 27–38, New York, NY, USA, 2003. ACM.

[89] Guojun Jin and Brian L. Tierney. System capability effects on algorithms for network bandwidth measurement. In *Proc. IMC*, 2003.

[90] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115, Oakland, CA, May 2015. USENIX Association.

[91] Christoforos Kachris, Keren Bergman, and Ioannis Tomkos. *Optical Interconnects for Future Data Center Networks*. Springer, 2013.

[92] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictble low latency for data center applications. In *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[93] Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[94] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. ACM Symposium on SDN Research*, 2016.

[95] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[96] Myron King, Jamey Hicks, and John Ankcorn. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, New York, NY, USA, 2015. ACM.

[97] David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[98] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 2000.

[99] Ramana Rao Kompella, Kirill Levchenko, Alex C. Snoeren, and George Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.

[100] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 100(8):933–940, 1987.

[101] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36:933–940, Aug 1987.

[102] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, Piscataway, NJ, USA, 2010. IEEE Press.

[103] James F Kurose. *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[104] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21:558–565, July 1978.

[105] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16:133–169, May 1998.

[106] Leslie Lamport. Paxos Made Simple. *SIGACT*, 32(4):18–25, December 2001.

[107] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19:79–103, October 2006.

[108] Leslie Lamport and Mike Massa. Cheap Paxos. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[109] Leslie Lamport and P. M. Melliar-Smith. Byzantine Clock Synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, 1984.

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

[111] Kang Lee, John C Eidson, Hans Weibel, and Dirk Mohl. Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems. In *Conference on IEEE*, volume 1588, page 2, 2005.

[112] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the ACM conference on Special Interest Group on Data Communication*, 2016.

[113] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. SoNIC: Precise Realtime Software Access and Control of Wired Networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[114] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. Phy covert channels: Can you see the idles? In *Proc. NSDI*, 2014.

[115] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transaction on Networking*, 2(1), February 1994.

[116] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural enhancements in stratix v. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 147–156. ACM, 2013.

[117] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. ACM.

[118] Han Li. IEEE 1588 time synchronization deployment for mobile backhaul in China Mobile, 2014. Keynote speech in the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication.

[119] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *OSDI16*, pages 467–483. USENIX Association, November 2016.

[120] Chiun Lin Lim, Ki Suh Lee, Han Wang, Hakim Weatherspoon, and Ao Tang. Packet clustering introduced by routers: Modeling, analysis and experiments. In *Proceedings of the 48th Annual Conference on Information Sciences and Systems*, 2014.

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

[122] Xiliang Liu, Kaliappa Ravindran, Benyuan Liu, and Dmitri Loguinov. Single-hop probing asymptotics in available bandwidth estimation: sample-path analysis. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[123] Xiliang Liu, Kaliappa Ravindran, Benyuan Liu, and Dmitri Loguinov. Single-hop probing asymptotics in available bandwidth estimation: sample-path analysis. In *Proc. IMC*, 2004.

[124] Xiliang Liu, Kaliappa Ravindran, and Dmitri Loguinov. Multi-hop probing asymptotics in available bandwidth estimation: stochastic analysis. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, 2005.

[125] Xiliang Liu, Kaliappa Ravindran, and Dmitri Loguinov. Multi-hop probing asymptotics in available bandwidth estimation: stochastic analysis. In *Proc. IMC*, October 2005.

[126] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Hide and seek in time: Robust covert timing channels. In *Proceedings of the 14th European conference on Research in computer security*, 2009.

[127] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Robust and undetectable steganographic timing channels for i.i.d. traffic. In *Proceedings of the 12th international conference on Information hiding*, 2010.

[128] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proceedings of Microelectronics Systems Education*, 2007.

[129] Sridhar Machiraju. *Theory and Practice of Non-Intrusive Active Network Measurements*. PhD thesis, EECS Department, University of California, Berkeley, May 2006.

[130] Sridhar Machiraju and Darryl Veitch. A measurement-friendly network (mfn) architecture. In *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*, 2006.

[131] Mallik Mahalingam, Dinesh Dutt, and Kenneth Duda. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. *VXLAN: A Framework for Overlaying Virtualized Layer*, 2, 2012.

[132] Enrique Mallada, Xiaoqiao Meng, Michel Hack, Li Zhang, and Ao Tang. Skewless Network Clock Synchronization. In *Proceedings of the 21st IEEE International Conference on Network Protocols*, 2013.

[133] Cao Le Thanh Man, G. Hasegawa, and M. Murata. Icim: An inline network measurement mechanism for highspeed networks. In *Proceedings of the 4th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, April 2006.

[134] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[135] Tudor Marian, Ki Suh Lee, and Hakim Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2012.

[136] David Mazieres. Paxos Made Practical. Unpublished manuscript, January 2007.

[137] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[138] Bjrn. Melander, Mats Bjorkman, and Per Gunningberg. A New End-to-end Prob-

ing and Analysis Method for Estimating Bandwidth Bottlenecks. In *Proceedings of the IEEE Global Telecommunications Conference*, 2000.

[139] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[140] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[141] David L. Mills. Internet time synchronization: the network time protocol. *IEEE transactions on Communications*, 39:1482–1493, October 1991.

[142] Mininet. `http://mininet.org`.

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

[144] Jad Naous, David Erickson, Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[145] Netronome. FlowNIC. `http://netronome.com/product/flownics/`.

[146] Peter Newman, Greg Minshall, Tom Lyon, and Larry Huston. Ip switching and gigabit routers. *IEEE Communications magazine*, 35(1):64–69, 1997.

[147] Rishiyur S. Nikhil and Kathy Czeck. BSV by Example. CreateSpace, 2010.

[148] NLR. National Lambda Rail. `http://www.nlr.net/`.

[149] Bill Ogden, Jose Fadel, and Bill White. IBM System Z9 109 Technical Introduction. July 2005.

[150] Brian Oki and Barbara H. Liskov. Viewstamped Replication: A General Primary-Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, August 1988.

[151] Open-NFP. `http://open-nfp.org/`.

[152] Open vSwitch. http://www.openvswitch.org.

[153] OpenReplica. http://openreplica.org.

[154] P4. P4 Behavioral Model. https://github.com/p4lang/p4c-bm.

[155] P4. P4 Specification. http://p4.org/spec/.

[156] P4 Behavioral Model. https://github.com/p4lang.

[157] P4@ELTE. http://p4.elte.hu/.

[158] P4.org. http://p4.org.

[159] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, 2015. ACM.

[160] Pavlos Papageorge, Justin McCann, and Michael Hicks. Passive aggressive measurement with mgrp. *SIGCOMM Computer Communication Review*, 39(4):279–290, August 2009.

[161] Attila Pásztor and Darryl Veitch. PC Based Precision Timing Without GPS. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[162] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *European Dependable Computing Conference (EDCC)*, pages 44–61, October 2002.

[163] Fernando Pedone and Andre Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *Theoretical Computer Science*, 291:79–101, January 2003.

[164] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[165] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data

Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, March 2015.

[166] Jon Postel. User datagram protocol. Technical report, 1980.

[167] Jon Postel. Internet protocol. 1981.

[168] Jon Postel. Rfc 792: Internet control message protocol. *InterNet Network Working Group*, 1981.

[169] Jon Postel and Joyce Reynolds. File transfer protocol. 1985.

[170] Ravi Prasad, Constantine Dovrolis, and Kathy Claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *Network*, 2003.

[171] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. In *Proceedings of Passive and Active Measurements Workshop*, 2004.

[172] Andrew Putnam, Adrian Caulfield, Eric Chung, and Derek Chiou. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE Press, June 2014.

[173] Vinay Ribeiro, Rudolf H. Riedi, Richard G. Baraniuk, Jiri Navratil, and Les Cottrell. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Proceedings of Passive and Active Measurements Workshop*, 2003.

[174] Luigi Rizzo. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[175] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[176] David Schneider. The Microsecond Market. *IEEE Spectrum*, 49(6):66–81, 2012.

[177] Fred B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical Report TR87-859, Cornell University, August 1987.

[178] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22:299–319, December 1990.

[179] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *SIGCOMM16*, pages 525–538, 2016.

[180] Satnam Singh and David J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2008.

[181] Ahmed Sobeih, Michel Hack, Zhen Liu, and Li Zhang. Almost Peer-to-Peer Clock Synchronization. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2007.

[182] Joel Sommers and Paul Barford. An active measurement system for shared environments. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, 2007.

[183] Joel Sommers and Paul Barford. An active measurement system for shared environments. In *Proc. IMC, 2007*, October 2007.

[184] Joel Sommers, Paul Barford, and Mark Crovella. Router primitives for programmable active measurement. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, 2009.

[185] Joel Sommers, Paul Barford, and Walter Willinger. Laboratory-based Calibration of Available Bandwidth Estimation Tools. *Microprocessors and Microsystems*, 31(4):222–235, 2007.

[186] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Workshop on Hot Topics in Software Defined Networking*, pages 127–132, August 2013.

[187] Carly Stoughton. Getting started with cisco nexus 9000 series switches in the small-to-midsize commercial data center. 2015.

[188] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *Proceedings of the ACM SIGCOMM 2003 Conference*, 2003.

[189] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *Proceedings of*

the 6th USENIX symposium on Networked systems design and implementation, 2009.

[190] Lloyd S Tenny. Chicago mercantile exchange. *The ANNALS of the American Academy of Political and Social Science*, 155(1):133–135, 1931.

[191] Terasic. DE5-Net FPGA Development Kit. `http://de5-net.terasic.com`.

[192] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf. `http://dast.nlanr.net/Projects/Iperf/`.

[193] Tolly. IBM Bnt RackSwitch G8264: Competitive Performance Evaluations vs. Cisco Systems, Inc. Nexus 5548P, Arista Networkrs 7148SX, and Juniper Networks EX 4500. `http://www.tolly.com/DocDetail.aspx?DocNumber=211108`, 2011.

[194] Robbert Van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, February 2015.

[195] Darryl Veitch, Satish Babu, and Attila Pàsztor. Robust Synchronization of Software Clocks Across the Internet. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, 2004.

[196] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *Proceedings of the ACM SIGCOMM*, 2006.

[197] Kashi Venkatesh Vishwanath and Amin Vahdat. Evaluating distributed systems: Does background traffic matter? In *Proceedings of USENIX 2008 Annual Technical Conference*, 2008.

[198] Rick Walker, Birdy Amrutur, and Tom Knotts. 64b/66b coding update. `grouper.ieee.org/groups/802/3/ae/public/mar00/walker_1_0300.pdf`.

[199] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 1995.

[200] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual ma-

chine monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[201] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[202] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. *SIGCOMM Computer Communication Review*, 35(4):37–48, August 2005.

[203] Xilinx. SDNet. `http://www.xilinx.com/products/design-tools/software-zone/sdnet.html`.

[204] Xilinx. Vivado Design Suite - HLx Editions. `https://www.xilinx.com/products/design-tools/vivado.html`.

[205] Xilinx. Xilinx Virtex-7 FPGA. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html`.

[206] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) protocol architecture. 2006.

[207] Takeshi Yoshino, Yutaka Sugawara, Katsushi Inagami, Junji Tamatsukuri, Mary Inaba, and Kei Hiraki. Performance optimization of TCP/IP over 10 gigabit Ethernet by precise instrumentation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[208] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *Proceedings of Passive and Active Measurements Workshop*, 2013.

[209] Ryan Zarick, Mikkel Hagen, and Radim Bartos. Transparent clocks vs. enterprise ethernet switches. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011.

[210] Hongyi Zeng, John W. Lockwood, Adam Covington, and Alexander Tudor. AirFPGA: A software defined radio platform based on NetFPGA. In *NetFPGA Developers Workshop*, 2009.

[211] Noa Zilberman, Yury Audzevich, Adam Covington, and Andrew W Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34:32–41, September 2014.

[212] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.