©Copyright 2019
Naveen Kr. Sharma

Building Efficient Network Protocols for Data Centers using Programmable Switches

Naveen Kr. Sharma

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Arvind Krishnamurthy, Chair

Thomas Anderson

Ratul Mahajan

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Building Efficient Network Protocols for Data Centers using Programmable Switches

Naveen Kr. Sharma

Chair of the Supervisory Committee:
Arvind Krishnamurthy
Computer Science & Engineering

Historically, computer networks have been designed to have most of the complexity at the end-hosts, while the switches connecting them are simple forwarding pipes that understand a fixed, well-specified set of protocols. This simplifies switching chip design, enabling them to operate at high speeds, albeit at the cost of little to no flexibility. On the other hand, recent advances in hardware switch architectures have made it feasible to perform limited flexible packet processing without sacrificing performance. Network operators can configure switches to process custom packet headers to exercise greater control over how packets are processed and routed. However, these switches have limited state, limited per-packet computation, restricted class of operations, and support a fixed set of scheduling primitives to be able to operate at line rate.

This thesis explores various mechanisms and techniques to overcome these switch limitations and implement efficient network protocols that rely on both flexible computation and packet scheduling inside the network. First, we use approximation techniques to mask limitations on computation and network state, letting us implement rich protocols that perform complex computations inside the network. Next, we propose an approximate scheduling mechanism based on Calendar Queues that lets us implement a wide range of scheduling algorithms to achieve various end-to-end performance objectives. Finally, we implement some of these protocols on real hardware and within a packet-level simulator to demonstrate significant performance improvement over state-of-the-art techniques.

TABLE OF CONTENTS

Page

List of Figures iv

List of Tables vi

Glossary vii

Chapter 1: Introduction 1

1.1 Thesis: Realizing In-Network Protocols using Reconfigurable Switches 3

1.2 Published Work 4

1.3 Outline 4

Chapter 2: Background and Related Work 6

2.1 Programmable Networking 6

2.2 Reconfigurable Switches 7

2.2.1 Features 8

2.2.2 Limitations 11

2.2.3 Evolution and Future 13

2.3 In-network Protocols 13

2.4 Related Work 16

2.4.1 Approximating State and Computation 16

2.4.2 Programmable Scheduling 16

2.4.3 Recent work leveraging FlexSwitches 17

Chapter 3: Approximate Stateful Computation 19

3.1 Library of Building Blocks 19

3.1.1 Flow Statistics 21

3.1.2 Approximating Arithmetic 26

3.1.3 Switch Statistics 28
3.1.4 Discussion 30
3.2 Realizing Network Resource Allocation Protocols 31
3.2.1 RCP 31
3.2.2 XCP 33
3.2.3 CONGA 35
3.2.4 Other use-cases 39
3.3 Evaluation 40
3.3.1 Hardware Implementation 40
3.3.2 ns-3 Simulations 41
3.3.3 P4 Compiler-based Resource Usage 44
3.4 Summary 45

Chapter 4: Approximate Programmable Scheduling 47
4.1 Background 48
4.1.1 Current state-of-the-art Scheduling Mechanisms 49
4.1.2 Proposals for Programmable Scheduling 50
4.2 Packet Scheduling using Calendar Queues 51
4.2.1 Motivating Calendar Queues 51
4.2.2 Programmable Calendar Queues (PCQs) 54
4.2.3 Implementing Scheduling Algorithms using PCQs 55
4.3 Implementing PCQs in Hardware 58
4.3.1 Alternate ways of emulating PCQs 62
4.4 Analysis and Extensions 64
4.5 Evaluation 66
4.5.1 Hardware Prototype Implementation 66
4.5.2 Coflow Scheduling using EDF Scheduling 68
4.6 Summary 72

Chapter 5: Case Study: Weighted Fair Queueing 73
5.1 Background: Bit-by-Bit Round-Robin Algorithm 73
5.2 FlexSwitch Implementation 75
5.2.1 Storing Approximate Bid Numbers 76

5.2.2 Buffering Packets in Approximate Sorted Order 77
5.2.3 Impact of Approximations 79
5.3 Optimized End-host Flow Control Protocol 80
5.4 Evaluation 83
5.4.1 Hardware Implementation 83
5.4.2 Packet-level Simulations 87
5.4.3 P4 Implementation 95
5.5 Summary 96

Chapter 6: Conclusions and Future Work 97
6.1 Future Work 98

Bibliography 101

LIST OF FIGURES

Figure Number | Page

2.1 An abstract reconfigurable switch model 8
2.2 Timeline of in-network protocols 15
3.1 Tradeoff between accuracy and memory in the cardinality estimating building block 23
3.2 Pseudocode for the flowlet switching building block 26
3.3 Staggered evaluation of XCP control parameters. 36
3.4 Cumulative distribution of FCTs of various flow sizes 42
3.5 Performance comparison between CONGA and approximate CONGA using ns3 simulations. 43
4.1 Example of a Programmable Calendar Queue. 53
4.2 WFQ implementation using a Logical CQ. 56
4.3 EDF using a work-conserving Physical CQ. 57
4.4 A Leaky Bucket Filter using a non-work-conserving Physical CQ. 59
4.5 Steps to perform a Calendar Queue rotation 61
4.6 Latencies from a hardware testbed running EDF using CQs 67
4.7 Coflow completions times when running EDF using CQs 71
4.8 CCT and FCT when running a hybrid scheme of WFQ and FCT using CQs 71
5.1 An example of the bit-by-bit round-robin Fair Queueing algorithm 74
5.2 Pseudocode for AFQ 75
5.3 An example of the AFQ enqueue mechanism 76
5.4 An example of the AFQ dequeue mechanism. 78
5.5 Pseudocode for endhost flow control protocol 81
5.6 High-level architecture of the AFQ switch prototype. 85
5.7 FCT summary for the enterprise workload on our AFQ hardware testbed 86
5.8 Flow completion times for synthetic workload in a simulated cluster 88
5.9 Flow completion times for enterprise workload in a simulated cluster 89
5.10 Packet drops, queue lengths and buffer occupancy distribution for AFQ 89

5.11 Completion times for an incast request with varying number of senders 91
5.12 Packet drops, re-transmissions and buffer distribution across flows during incast traffic. . 91
5.13 Flow convergence test with TCP and AFQ. 92
5.14 FCT vs flow size at 70% load. 92
5.15 Deviation of various queuing mechanisms compared to ideal fair queuing 93
5.16 FCT vs number of FIFO queues. 94
5.17 AFQ with different End-hosts 95
5.18 Round Estimation vs Sketch Size 95

LIST OF TABLES

Table Number Page

3.1 Network functions that can be supported using FlexSwitch building blocks. 20
3.2 Summary of the proposed building blocks and techniques to implement them. 21
3.3 Count-min sketch size required to achieve false-positive rate below the specified threshold. 25
3.4 Number of lookup table entries required for approximation of multiplication and division 27
3.5 Flow completion times for short, medium, and long flows running RCP on Cavium hardware 41
3.6 Summary of resource usage for various use-cases. 45
5.1 Summary of resource usage for AFQ. 96

GLOSSARY

AIMD: The Additive-Increase/Multiplicative-Decrease (AIMD) algorithm, most commonly used in TCP congestion control to achieve fairness.

ECMP: Equal-Cost Multi-Path (ECMP) routing is a routing strategy where next-hop packet forwarding to a single destination can occur over multiple paths.

ECN: Explicit Congestion Notification (ECN) is an extension to IP/TCP protocol that allows marking packets using special bits to indicate congestion in the network.

GRE/NVGRE: Network Virtualization Generic Routing Encapsulation (GRE) is a simple IP packet encapsulation protocol.

IP: The Internet Protocol (IP) is a set of rules for routing and addressing packets across computer networks.

MAC: The Media Access Control (MAC) address of a device is a unique identifier assigned to a network interface controller.

MSS: The Maximum Segment Size (MSS) is the largest amount of data, that the network can handle in a single, unfragmented piece. Also called Maximum Transmission Unit (MTU).

NIC: A Network Interface Controller/Card (NIC) is a piece of hardware that connects a computer to the network.

NTP: Network Time Protocol (NTP) is a networking protocol for clock synchronization between servers over computer networks.

PFC: Priority-based Flow Control (PFC) is a link-layer flow control mechanism between directly connected nodes in a network to prevent packet loss due to congestion.

RTT: Round Trip Time (RTT) is the duration of time it takes for a packet to be sent over the network and its acknowledgement received.

SRAM: Static Random Access Memory (SRAM) is a type of memory that holds data in a static form, as long as the memory has power and does not require periodic refreshing.

TCAM: Ternary Content-Addressable Memory (TCAM) is a specialized type of high-speed memory that searches its entire contents in a single clock cycle.

TCP: Transmission Control Protocol (TCP) is a standard that defines how to establish and maintain a reliable network stream between two network servers to exchange data.

TOCTOU: Time-of-Check to Time-of-Use (TOCTOU) is a class of software bugs caused by a race condition involving the checking of the state of a part of a system and the use of the results of that check.

TTL: Time-To-Live (TTL) is a value in a IPv4 packet that tells a network entity whether or not the packet has been in the network too long and should be discarded.

UDP: User Datagram Protocol (UDP) is a protocol for exchanging data over IP networks that does not provide any reliabilty or ordering guaranteers.

VLAN: A Virtual Local Area Network (VVLAN) is a subnetwork which can group together collections of devices on separate physical local area networks.

ACKNOWLEDGMENTS

This thesis has been a long, tiring, but a gratifying journey nonetheless and would not have been possible without the support of numerous people. First, my advisor Arvind who gladly took me in his wings and patiently mentored me both inside and outside of the research environment. He has been extremely kind and encouraging, and I was always amazed by his enthusiasm and calm demeanor towards everything, even in the most stressful of times. I could not have asked for a better guiding *guru*, and he truly was the 'Krishna'murthy I needed to finish this dissertation. I am grateful to my other advisors Dan Ports and Steve Gribble who were both instrumental in me not getting lost and having a smooth start early on in my Ph.D. career. I learned a lot from them and they set me up for success by preparing me to tackle the hurdles of research life.

Several other professors in CSE, Hank Levy, Tom Anderson, and Luis Ceze have been excellent mentors to me and I cherish the sage advice I received from them. I would also like to thank Tom Anderson, Ratul Mahajan, and Sreeram Kannan for serving on my committee and providing valuable feedback and suggestions.

I've had the great pleasure and privilege of working with several amazing collaborators: Jialin Li, Irene Zhang, Adriana Szekeres, Vincent Liu, Antoine Kaufmann, Simon Peter, Jacob Nelson, Ming Liu, Ellis Michael, Changhoon Kim, Kishore Atreya, Chenxingyu Zhao and Anirudh Sivaraman. None of this work would have been possible without their efforts and I deeply value all the interactions I have had with them over the years.

A huge thanks to Lindsay, Elise, Melody, and Lisa for taking care of all administrative tasks seamlessly, so that I had the peace of mind to continue working on my research without worrying at all.

In the last 7 years, I have spent the most time with four close friends – Jialin, Irene, Adriana, and Antoine, who started as colleagues but ended up becoming my closest friends and support pillars. I have learned more from them than any one else, and I value their friendship more than any other achievement during this time. I will cherish the nights we spent in the lab before a deadline, the hikes in the mountains, the weekend brunches, and the occasional stress-busting online gaming sessions.

I am thankful to all my friends in the systems and networks lab, Pete, Katelin, Ravi, Danyang, Niel, Pedro, Yuchen, Ellis, Kaiyuan, Helga and Helgi, who provided much-needed breaks from work. I had great support outside work too, my squash partner Arun, coffee-mate Shrainik and close friend Ashish who although far away, never let me feel alone during tough times.

Last but not least, I'm forever indebted to my parents, sister Sunita and brother Praveen who have constantly supported me throughout this time. I would not be here without their persistent encouragement, love, and care; especially my elder brother Praveen who has always been there as a rock to fall back on.

Finally, thank you Varun Grover and Neeraj Ghaywan for *Masaan*.

DEDICATION

to my parents,

for their countless sacrifices

Chapter 1

INTRODUCTION

Historically, computer networks have been designed to have most of the complexity at the end-hosts generating the data packets, while the switches and routers connecting them are simple forwarding devices that implement a fixed set of well-defined protocols such as Ethernet and IP. This was primarily due to the design goal of the internet, to make it easy to interconnect a wide variety of existing networks and simple to extend them. This also made switching hardware architecture straight-forward, allowing them to operate at high speeds to achieving high bandwidth which was also a major driving factor. However this simplicity and high performance come at the cost of limited features and network functionality, and the end-host was made responsible for achieving several desirable properties such as reliable delivery, congestion control, load balancing, fair sharing and quality of service inside the network.

Unsurprisingly, there have been many proposals in literature which show that we can build more efficient protocols to solve challenging network problems such as congestion control [33, 47], fair sharing [32, 73, 80], load balancing [5, 38, 93, 71], quality of service [9, 90, 13, 40] and network monitoring [65, 91] with limited support from the network. But very few of these have adopted in practice due to several reasons. First, switches are largely fixed-function devices that do not support the complex computation and state requirements of these protocols. Second, it is often difficult to implement these complex protocols at a high bandwidth rate of terabits per second and since line-rate performance is the primary driving force, network operators do not want to sacrifice packet-forwarding rate. Finally, the adoption and deployment of new protocols is always a challenge as they must first be well defined and standardized by the community before being implemented in hardware and deployed. This usually ends up taking 2-3 years, which is too slow for today's fast-evolving networks.

On the other hand, with the rise of datacenters and advances in switching chip hardware architectures, some of these concerns have been eased and it is worth revisiting whether we can implement and take advantage of these efficient network protocols. Datacenters and enterprise networks are often managed by a single entity and have complete control over the network, making it easier to deploy and experiment with custom protocols. Moreover, recent hardware switch architectures make it feasible to perform flexible packet processing inside the network without sacrificing performance. This allows operators to configure switches to process custom packet headers in order to exercise greater control over how packets are processed and routed. However, these switches have limited state, limited per-packet computation, restricted class of operations, and support a fixed set of packet scheduling primitives to be able to operate at line rate. As a result, while these switches are fully capable of implementing flexible custom protocols such as encapsulation/decapsulation or header translations – for which they were designed, they fall short of being capable of implementing complex in-network protocols.

This dissertation explores ways to overcome these switch limitations and implement efficient network protocols that leverage flexible computation and flexible packet scheduling inside the network and further suggests ways of evolving the switching hardware to enable implementation of in-network protocols.

First, we study a wide class of network protocol proposals in the literature that require explicit support from the network to understand requirements from switches in terms of computation, state maintained and scheduling primitives. We also review the features of various programmable switches to understand their capabilities and limitations. For many complex protocols, while we cannot precisely implement them on today's programmable switches, we can use approximation techniques to realize variants of these protocols to achieve similar performance and functionality.

Next, we introduce several approximation techniques from literature to create a library of building blocks for constructing complex network protocols using in-network processors. These blocks approximate some network functionality common across several complex protocols. A common theme across many of these building blocks is the use of approximation to stay within the limited hardware resource bounds. We quantify the tradeoff between resource usage at the

switch and accuracy under various realistic workloads.

Further, we take a look at protocols requiring nontrival packet scheduling inside the network and propose an approximate scheduling mechanism based on Calendar Queues that lets up implement such protocols. It relies on the observation that most algorithms require both prioritization and implicit escalation of a packet's priority. We show how Calendar Queues can be implemented efficiently on today's programmable switches by dynamically altering priorities of queues using a combination of data-plane and control-plane operations. This enables implementation of a wide range of scheduling algorithms to achieve end-to-end network performance objectives, such as Weighted Fair Queueing [32] and Earliest Deadline First scheduling.

Finally, we implement some these network protocols on real hardware and within packet-level network simulators to show that we can indeed realize complex protocols without sacrificing packet-forwarding bandwidth and achieve significant performance gains. We also show that our approximate variants of these protocols are accurate enough for common network workloads.

1.1 Thesis: Realizing In-Network Protocols using Reconfigurable Switches

The primary hypothesis of this dissertation is that, *it is possible to approximate complex in-network protocols at line rate without sacrificing performance by leveraging limited flexibility of reconfigurable switches.* We study a wide range of protocols that require support from the network, along with the features available in the current generation of high-speed programmable switches to come up with a set of approximation techniques and hardware mechanisms that let us implement these protocols.

This dissertation makes the following technical contributions:

- a study of various classes of in-network protocols to determine their requirements in terms of state, computation and scheduling primitives inside the network.

- a library of building blocks that approximate various network state and functions that overcome the restrictions of programmable switches, thereby enabling the implementation of complex in-network protocols at terabits per second.

- a hardware proposal for flexible packet scheduling inside the network that enables approximate implementation of various classical scheduling algorithms.
- an end-to-end implementation of a weighted fair queueing protocol on high-speed reconfigurable switches using the approximation techniques described in the dissertation.

1.2 Published Work

Part of the work presented in this thesis has been published in the following papers:

- Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson and Simon Peter. **Evaluating the Power of Flexible Packet Processing for Network Resource Allocation**. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017.*
- Naveen Kr. Sharma, Ming Liu, Kishore Atreya and Arvind Krishnamurthy. **Approximating Fair Queueing on Reconfigurable Switches**. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2018.*
- Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein Govindan, Changhoon Kim, Arvind Krishnamurthy and Anirudh Sivaraman. **Programmable Calendar Queues for Packet Scheduling**. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2020.*

1.3 Outline

The rest of this thesis is organized as follows:

We begin with a brief overview of networks with a particular focus on data center networks and various network protocols that solve challenging problems in Chapter 2. We also describe the features and limitations of reconfigurable switches.

Next, in Chapter 3, we study a wide class of network protocols that require in-network support and propose a set of building blocks that let us approximately realize these complex protocols on

reconfigurable switches. Using real hardware implementations and packet-level simulations, we show we can implement such complex protocols.

Chapter 4 describes our hardware proposal for performing flexible packet scheduling. It tackles another class of in-network protocols that require per-packet scheduling inside the network. Since today's switches have fixed scheduling algorithms with little flexibility, we propose an approximate scheduling mechanism based on Calendar Queues that allows us to implement various scheduling algorithms.

Next, we take a concrete example of a complex in-network protocol, *Fair Queueing*, that requires both storing and updating state inside network switches as well as per-packet scheduling decisions. Using techniques described in the dissertation, we show an end-to-end implementation and evaluation of the fair queueing protocol.

Finally, we conclude by summarizing and discussing future work in Chapter 6.

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter, we begin by briefly describing the history of programmable networking, followed by a detailed discussion of a new class of network switches, called programmable or reconfigurable switches – their features, restrictions and, limitations. Next, we present a wide range of protocols proposed in the literature that require explicit support from the network. These protocols form the basis of our study to figure out what can and cannot be implemented on programmable switches.

2.1 Programmable Networking

The first routers used in ARPANET during the 1970s were built on top of minicomputers. The Interface Message Processor (IMP) [39] ran atop a Honeywell minicomputer and could forward up to a few hundred kilobits per second. This approach of building routers using software running on a general-purpose processor was highly flexible and programmable as it simply required updating the software. However, they provided limited maximum bandwidth and could not keep up with the growing demand of higher link speeds. By mid-1990s, companies like Cisco and Juniper started manufacturing dedicated hardware chips to forward packets at high speeds and even today specialized hardware is required to keep up with increasing network link speeds. This, however, means that these routers have limited programmability and flexibility, and adding a new protocol or feature generally requires upgrading the hardware, which has a longer cycle.

In order to overcome the limited flexibility of hardware switches, researchers proposed a separation of the control-plane and data-plane inside routers. The control-plane that ran the distributed algorithm to compute routing tables could be run on a general-purpose CPU on the router whereas the data-plane that forwarded packets by looking up routing tables could be imple-

mented in hardware for higher performance. This approach called *Software-defined Networking (SDN)* allowed network operators to implement richer functionality by programming the control-plane to provide features such as traffic engineering, access control and virtual networks. The control-plane could communicate with the data-plane using an API such as OpenFlow [61] to populate routing tables used for forwarding packets. Further, the control-planes of all routers in a network could communicate with a single central controller that could manage a large network efficiently, such as B4 [43].

However, the programmability of the data-plane was still limited and it could only process a fixed well-specified set of protocols. This is beginning to change with the introduction of a new class of switching chips which we describe next.

2.2 Reconfigurable Switches

The recent innovations in switch design have lead to not just faster but *more flexible* packet processing architectures. Whereas early generations of software-defined networking switches could specify forwarding paths on a per-flow basis, today's switches support configurable per-packet processing, including customizable packet headers and the ability to maintain state inside the switch. Examples include Intel FlexPipe [70], Texas Instruments's Reconfigurable Match Tables (RMTs) [21], the Cavium XPliant switches [25], and Barefoot's Tofino switches [16]. We term these switches *FlexSwitches*.

FlexSwitches give greater control over the network by exposing previously proprietary switching features. They can be reprogrammed to recognize, modify, and add new header fields, choose actions based on user-defined match rules that examine arbitrary components of the packet header, perform simple computations on values in packet headers, and maintain mutable state that preserves the results of computations across packets. Importantly, these advanced data-plane processing features operate at line rate on every packet, addressing a major limitation of earlier solutions such as OpenFlow [61] which could only operate on a small fraction of packets, e.g., for flow setup. FlexSwitches thus hold the promise of ushering in the new paradigm of a *software defined data-plane* that can provide datacenter applications with greater control over the

network's datapaths.

Despite their promising new functionality, FlexSwitches are not all-powerful. Per-packet processing capabilities are limited and so is stateful memory in order to achieve line-rate packet processing speeds of terabits per second. In this section, we review the features of FlexSwitches and their limitations in detail.

2.2.1 Features

We assume an abstract switch model as described in [20] and depicted in Figure 2.1.

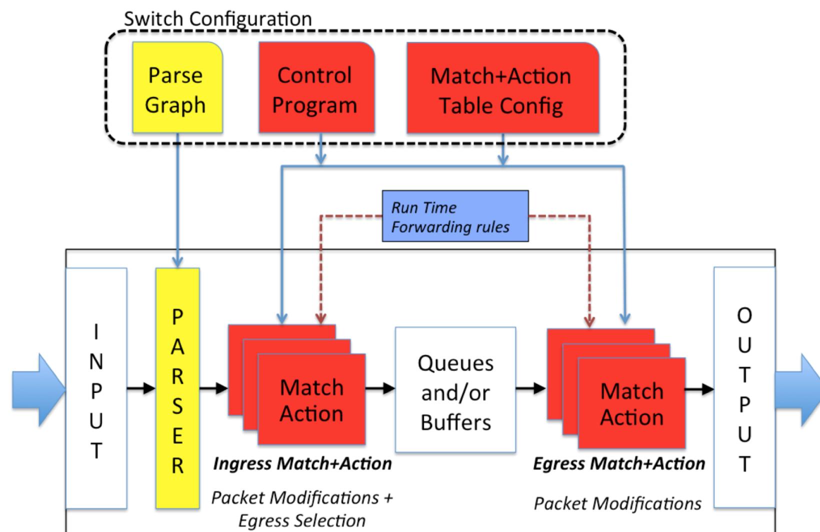


Figure 2.1: An abstract reconfigurable switch model (from [20]). The switch consists of a programmable parser that splits input data stream into individual packet header fields and passes them onto a series of match+action tables. Each table looks up into SRAM/TCAM memory and can perform computation on individual headers. Packets are then buffered into a Traffic Manager before being sent out on an output port via another set of match+action tables. The switch components are configured using a control-plane that runs on top of a switch CPU.

Rather than supporting arbitrary per-packet computation, most FlexSwitches provide a restricted *match+action* (M+A) processing model: match on arbitrary packet header fields and apply simple packet processing actions [21]. To keep up with packets arriving at line rate, switches need to operate under tight real-time constraints. Programmers configure how FlexSwitches process

and forward packets using high-level languages such as P4 [20] or Protocol Oblivious Forwarding [86].

On packet arrival, FlexSwitches parse packet headers via a user-defined parse graph. Relevant header fields along with packet meta-data are passed to an ingress pipeline of user-defined M+A tables. Each table matches on a subset of extracted fields and can apply simple processing primitives to any field, ordered by a user-defined control program. The ingress pipeline also chooses an output queue for the packet, determining its forwarding destination. Packets pass through an egress pipeline for destination-dependent modifications before output. Legacy forwarding rules (e.g., IP longest prefix match) may also impact packet forwarding and modification.

To support packet processing on the data path, FlexSwitches provide several hardware features, which we describe next.

Configurable Parser is used to break down an incoming bit stream into meaningful protocol headers, such as Ethernet/IP source and destination addresses for outgoing port lookups. This is typically achieved using a Kangaroo Parsing Unit (KPU) [34], which uses TCAM to lookahead and parse multiple protocol headers in a single step. Since the number of KPU steps are fixed in hardware, there is a limit to how many bytes can be extracted and how deep we can look into the packet.

Match+Action Tables generalize the abstraction of match+action provided by OpenFlow, allowing matches and actions on any configured packet header field. Matches are generally realized using TCAM or hash tables implemented in SRAM. For example, exact Ethernet MAC address matching can be done using SRAM hash tables and longest prefix IP address matching using TCAM. The entries in these tables can only be populated or modified using the control-plane CPU and not on a per-packet basis. Actions may add, modify, and remove fields from the packet header using data returned from the matching.

Computation Primitives that perform a limited amount of processing on header fields or stateful memory inside each stage. This includes operations such as addition, bit-shifts, hashing,

and max/min. These can be used for performing TTL decrements on packet headers or incrementing statistics counters maintained on the switch. Since these primitives can be invoked for each packet, they are both limited in number and complexity, if the switch is to achieve processing of terabits per second. For example, each stage can modify a field only once and complex operations such as multiplication, division or floating points are not available.

Stateful Memory A limited amount of *stateful memory* can maintain state across packets, such as counters, meters, and registers, that can be used while processing. These are maintained using SRAM in each stage and can be read or written as part of datapath packet processing. As we show later, this is a powerful feature that lets programmers store protocol-specific data on these switches and modify it on a per-packet basis.

Traffic Manager (TM) is a dedicated module that sits between the ingress and egress pipelines and is responsible for buffering all the packets traversing the switch. The TM maintains multiple queues, each of which can be associated with a physical port on the switch. Common hardware implementations support multiple queues per egress port with scheduling algorithms that are fixed in hardware, such as Strict Priority or Deficit Weighted Round Robin [81]. The priority level of any packet can be determined and modified by the processing pipeline, i.e., can be configured, which eventually decides which queue the packet is assigned to. The TM also maintains statistics for each queue, such as occupancy depth and congestion levels. These statistics can be accessed from an on-chip switch CPU, or from the ingress or egress pipeline at a coarse granularity and periodic time intervals.

Switch-local control plane CPU is a general-purpose ARM or x86 switch-local control plane CPU that can be used by the data-path pipeline for further packet processing. The CPU can perform arbitrary computations, for example on packets that do not match in the pipeline. Because computations are not limited, it cannot guarantee that forwarded packets are processed at the line-rate; instead, it drops packets when overloaded.

Switch meta-data such as queue lengths, congestion status, and bytes transferred are available from the Traffic Manager. Moreover, ingress and egress timestamps for each packet can also be used in conjunction with available stateful memory for packet processing.

Timers built into the hardware can invoke the switch-local CPU to perform periodic computation such as updating M+A table entries or exporting switch meta-data off-switch to a central controller.

2.2.2 Limitations

Although FlexSwitches are flexible and reconfigurable in several ways, they do impose several restrictions and constraints. If these limits are exceeded, then the packet processing code cannot be compiled to the FlexSwitch target. We describe these limits, providing typical values for them based on both published data [21] and information obtained from manufacturers, and note how they impact the implementation of network resource allocation algorithms.

Limited Processing Primitives

Each switch pipeline stage can execute only one ALU instruction per packet field, and the instructions are limited to signed addition and bitwise logic. Multiplication, division, and floating-point operations are not available. Hashing primitives, however, are available; this exposes the hardware that switches now use for ECMP and related load-balancing protocols. Control flow mechanisms, such as loops and pointers, are also unavailable, and entries inside M+A tables cannot generally be updated on the data path. This precludes the complex floating-point computations often employed by resource allocation algorithms from being used directly on the data-path. There is also a limit on the number of sequential processing operations per packet which is roughly equal to the number of M+A pipeline stages (generally around 10 to 20). Within a pipeline stage, rules are processed in parallel.

Constrained Stateful Memory

Generally, it is infeasible to maintain per-flow state across packets in both reconfigurable switches and their fixed-function counterparts.[1] For example, common switches support SRAM-based exact match tables of up to 12 Mb per pipeline stage. The number of rules is limited by the size of the TCAM-based ternary match tables provided per pipeline stage (typically up to 1Mb). In contrast, it is common for a datacenter switch to handle tens of thousands to hundreds of thousands of connections. This allows for a negligible amount of per-connection state, which is likely not enough for most resource allocation algorithms to perform customized flow processing (e.g., for RCP to compute a precise average RTT).

Limited State across Computational Stages

The hardware often imposes a limit on the size of the temporary *packet header vector*, used to communicate data across pipeline stages. Common implementations limit this to 512 bytes. Also, switch metadata may not be available at all processing stages. For example, cut-through switches may not have the packet length available when performing ingress processing. This precludes that information from being used on the data path, severely crimping link utilization metering and flow set size estimation as needed by RCP. Further, each stage itself might have an upper limit on the amount of state that can be accessed from within the stage.

Fixed Scheduling Primitives

While the Traffic Manager supports multiple queues for buffering packets and each of these can be assigned to any physical port on the switch, there are only a handful of scheduling algorithms available to configure these multiple queues. Common algorithms include Strict Priority, where packets from the highest priority queue are drained first, Deficit Weighted Round Robin, where each queue drains packets at a rate proportional to the configured weights, and Rate Limiting,

[1] The dramatic growth in link bandwidths coupled with much slower growth in switch state resources implies that this constraint will likely hold in future generations of datacenter switches.

where the maximum sending rate of a queue can be configured. These scheduling algorithms can be tweaked using pre-defined knobs, but there is no way for an operator to implement a completely different scheduling algorithm. This significantly restricts the types of complex network protocols that can be realized on FlexSwitches.

2.2.3 Evolution and Future

FlexSwitches are relatively new and are constantly evolving. New features are being added and it is worth asking whether the limitations described above will remain true in future iterations of the hardware or not. For example, additional computational primitives might be introduced or available stateful memory might increase. FlexSwitches today have been designed primarily to support reconfigurable routing and customized data-plane protocols, such as encapsulation/decapsulation of packets or network address translations. This thesis explores what newer primitives are required and can be added on the data-path of FlexSwitches to enable them to implement richer, more complex network protocols.

However, some of the FlexSwitch restrictions are fundamental in nature and cannot be avoided if packets must be processed in the future at a line-rate of multiple terabits per second. Today's top of the line switches process roughly 1 billion packets per second with a pipeline latency of a few hundred nanoseconds. This places restrictions on the use of per-flow state, the amount of processing that can be done, and the amount of state that can be accessed per packet. Forwarding performance is still the primary objective for chip designers and it is often desirable to have more bandwidth instead of more flexibility. Generally, more hardware can be added through parallelism, such as by adding multiple parallel match+action pipelines. However, this adds further restrictions on state and computation.

2.3 In-network Protocols

There is a broad category of network protocols that require explicit support from network elements to solve various challenging problems. We categorize and briefly discuss some of them below.

Congestion Control. Rate Control Protocol (RCP) [33] and eXplicit Control Protocol (XCP) [47] are two congestion control protocols that rely on explicit feedback from the network and are more efficient than the widely deployed TCP. In RCP, switches keep track of the fair share rate and assign it to each active flow by modifying the packet headers. The fair share rate calculation depends on the number of active flows, queue build up inside the switch and spare bandwidth. XCP, on the other hand, provides feedback by changing each flow's congestion window. For each packet traversing the switch, XCP computes positive or negative feedback depending on whether the flow is sending slower or faster than the fair share rate to keep the network link fully utilized. Both these protocols provide a good use-case for programmable switches. Other recent examples include FCP [37], MQ-ECN [14], and PDQ [40].

Fairness and Sharing. Fair sharing or fair queueing is a highly desirable property that can provide isolation and protection from ill-behaved participants. Several proposals have been made to achieve per-flow fairness at bottleneck links, such as WFQ [32], SFQ [60] and CSFQ [88]. More recently, Seawall [80], and FairCloud [73] support fairness among multiple tenants or applications using the network. Most of these approaches maintain some per-flow or per-tenant counters and drop or delay packets to ensure fairness.

Active Queue Management and QoS. A large class of protocols tries to reduce delay and achieve higher quality of service by controlling queue sizes and prioritizing packets using various heuristics to achieve end-to-end guarantees. Some examples include CoDel [68], PIAS [13], DeTail [92], pFabric [9], and qJump [35]. These algorithms and protocols make per-packet decisions, involving *computation* on switch metadata such as queue statistics or congestion level, and *scheduling* decisions to determine the relative order of packet departure.

Load Balancing. Datacenters are organized in multi-rooted tree topologies that have multiple redundant paths between any pair of nodes. In such an environment, it becomes important to balance traffic among all possible paths and route traffic in case of failure or bursty traffic imbalance. Several proposals such as CONGA [5], WCMP [93], Ananta [71], Hedera [2], Presto [38], and

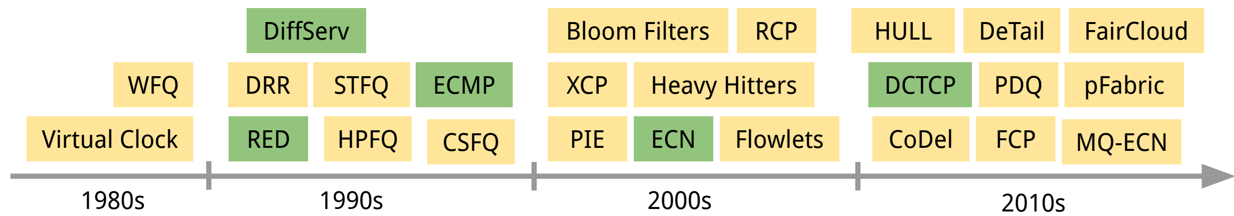


Figure 2.2: A brief timeline of some of the in-network protocols we discussed earlier (Figure adapted from [49]). The x-axis show when these protocols were proposed and only the ones shaded in green are available today in high-speed switches.

SilkRoad [62] achieve this. Again, they also require storing some port-level statistics at switches and deciding on a per-packet basis which path to use for forwarding packets. This computation involves flow size (in the case of Hedera and Ananta), or realtime congestion status (in the case of CONGA), or application information embedded in packet headers (in the case of SilkRoad).

Network Monitoring and Debugging. A large body of works has focussed on making network monitoring and debugging simpler and easier. If switches were able to maintain flexible counters that could be configured in realtime, it would give network operators rich telemetry. Several proposals have explored this such as FlowRadar [56], UnivMon [58], Everflow [94], Sonata [36], and Marple [67]. The challenge here is to come up with the right abstractions and knobs to provide the network operators, such that they can query and localize the information they are looking for. This generally involves implementing filters on switches to capture interesting events, summarizing multiple counters across switches.

Figure 2.2 shows some of these protocols on a timeline of when they were proposed. Only the ones in green have been adopted widely and are available is high-speed switches today. All these protocols and algorithms have innovative ideas that solve challenging problems, yet have proved difficult to implement in high-speed switches. Moreover, due to resource limitations, chip designers are forced to choose and implement a small subset of these protocols and cannot repurpose the same chip to realize different algorithms.

2.4 Related Work

We first describe approaches in the literature that our approximate state and computation work builds upon. Next, we summarize recent proposals on programmable scheduling inside network switches and finally list recent work that leverages FlexSwitches.

2.4.1 Approximating State and Computation

Our library of building blocks presented in Chapter 3 borrows several techniques from the streaming algorithms literature and tweak them to work in a network setting on FlexSwitches. We demonstrate how to implement the HyperLogLog [72] algorithm and count-min sketch [30] on a FlexSwitch to approximate the number and frequency of distinct elements in traffic flows. Our flow timestamps and flowlet detection building blocks are related to *Approximate Concurrent State Machines* [19], but we can design simpler solutions given that we don't need general state machines to implement the functionality. Other related efforts OpenSketch [91] and DREAM [65] propose software-defined measurement architectures. OpenSketch uses sketches implemented on top of NetFPGA, while DREAM centralizes the heavy-hitter detection at a controller while distributing the flow monitoring tasks over multiple traditional OpenFlow switches. Both works trade off accuracy for resource conservation. We build on these ideas to implement a broad set of building blocks within the constraints of the hardware model and implement resource allocation algorithms using them.

2.4.2 Programmable Scheduling

Most network switches today support a small menu of scheduling algorithms (typically priorities, weighted round-robin, and traffic shaping). Recent proposals for programmable scheduling [85, 63, 77, 87, 3, 82] propose additional switch hardware to make the scheduling decision programmable, assuming the existence of a programmable ingress and egress switch pipeline like RMT. Of the proposals for programmable scheduling, we describe the PIFO work because it targets a FlexSwitch similar to this thesis, and it is representative of the hardware considerations

associated with programmable scheduling.

PIFOs enable programmable scheduling by using a programmable priority queue to express custom scheduling algorithms. Some external computation (either on the end host or a programmable switch's ingress pipeline) sets a rank for the packet. This rank determines the packet's order in the priority queue. By writing different programs to compute different kinds of packet ranks (e.g., deadlines or virtual times), different scheduling algorithms can be expressed using PIFOs.

While PIFOs are flexible, they not only require the development of new hardware blocks to support line-rate but are also limited given their support for a finite priority range. The PIFO paper assumes that ranks within flows are naturally in strictly increasing order (i.e., flows are FIFOs), requiring the switch to only find the minimum rank among the head packets across all flows. While this reduces the sorting/ordering requirement of PIFO, sorting the total number of flows in the buffer is still challenging. The PIFO work provides a custom hardware primitive, the flow scheduler, which maintains a sorted array of a few thousand flows and can process tens of flows per output port across about 64 output ports on a single pipeline for an aggregate throughput of 640 Gbit/s. Scaling this primitive to higher speeds and a multi-pipeline switch can be challenging.

2.4.3 Recent work leveraging FlexSwitches

Applications: Several proposals aim to use FlexSwitches to build more efficient and high-performance applications. Ananta [71], Hula [48], Silkroad [62] present mechanisms for more intelligent load balancing inside the network either using realtime congestion status or application level information to choose optimal paths. NetPaxos [31], NOPaxos [54], NetChain [44] explore the possibility of implementing the widely deployed Paxos consensus protocol in network devices in order to achieve faster distributed consensus. Switch-KV [55], IncBricks [57], NetCache [45], and KV-Direct [53] implement fast key-value stores by either storing packets in the network, or routing them to optimal destination based on caching or accelerators.

Languages and Compilers: A lot of research attempts to make it easier to program FlexSwitches at a high-level. Domino [83] presents the abstraction of *packet transactions* that allows a programmer to specify packet processing in a high-level language and generates low-level instructions in terms of *atoms* to configure the hardware. SNAP [11] programs stateful data-plane algorithms using a network transaction – an atomic block of code that treats the entire network as a single big switch. It then uses a compiler to translate network transactions into rules for each switch. NetASM [78] presents a device-independent language that is expressive enough to act as the target language for high-level languages, yet low-level enough to be efficiently assembled on various FlexSwitch architectures, and enables conventional compiler optimizations to improve the performance and resource utilization of custom packet-processing pipelines. [46] explores the design of a compiler for FlexSwitches, in particular how to map logical lookup tables to physical tables, while meeting data and control dependencies in the program.

Network Monitoring and Telemetry: Several recent approaches Sonata [36], Marple [67], FlowRadar [56], and SketchVisor [41] utilize programmable counters on FlexSwitches for real-time network monitoring and debugging. Sonata partitions each telemetry query across a stream processor and the FlexSwitch data-plane to provide a wide range of functionality with high efficiency. Marple is modeled on familiar functional constructs and backed by a programmable key-value store primitive on FlexSwitches that performs flexible aggregations at line rate and scales to millions of keys. SketchVisor augments sketch-based measurement in the data-plane with a fast path that is activated under high traffic load to provide high-performance local measurement with slight accuracy degradation.

Chapter 3

APPROXIMATE STATEFUL COMPUTATION

As discussed in Chapter 2, FlexSwitches impose many constraints on data plane programmability in order to operate at line rate. These include (a) limited stateful memory, such that maintaining per-flow state might not be feasible, (b) limited number of stages or operations, such that processing has to be simple and cannot include loops or scans over data or ports, and (c) limited operators (i.e., arithmetic operations such as multiplication/division are not supported).

We perform a functional analysis of a broad class of resource allocation algorithms in the literature to identify their computational and storage requirements. Table 3.1 summarizes our findings. We can see that the requirements of most network resource allocation protocols can be distilled into a relatively small set of common *building blocks*. We show how to implement these on a FlexSwitch, and then explain how we can use these building blocks to construct more complex network protocols.

3.1 Library of Building Blocks

We first present a library of building blocks, summarized in Table 3.2 that are designed to overcome hardware limitations described in Section 2.2. In particular, we facilitate the following types of packet processing functionality: (a) flow-level measurements such as approximate maintenance of per-flow state and approximate aggregation of statistics across flows, (b) approximate multiplication and division using simpler FlexSwitch primitives, and (c) switch-level measurements such as metering of port-level statistics and approximate scans over port-level statistics.

A common theme across many of these building blocks is the use of approximation to stay within the limited hardware resource bounds. We apply techniques adapted from the streaming algorithms literature within the context of FlexSwitches. Streaming algorithms use a limited

Function	Protocol	Building Blocks Required	Implementation
Congestion Control, Scheduling	RCP [33]	Arithmetic, Cardinality, Metering	Section 3.2.1.
	XCP [47]	Arithmetic, Metering	Section 3.2.2.
	QCN [4]	Arithmetic, Metering	Meter queue sizes and calculate feedback value from switch queue length based on sampling probability.
	HULL [8]	Arithmetic, Metering	Implement *Phantom queues* by metering and marking packets based on utilization levels.
	D^3 [90]	Flow Statistics, Metering	RCP-like feedback, but prioritizes near-deadline flows.
	PIAS [13]	Flow Statistics, Balancing	Emulates shortest job next scheduling by dynamically lowering priority based on packets sent.
Load Balancing	CONGA [5]	Arithmetic, Flow Timestamps, Metering	Section 3.2.3.
	WCMP [93]	Balancing	Section 3.2.4.
	Ananta [71]	Flow Counters, Metering, Balancing	Use flow counters to realize VIP map and flow table. Use metering and balancing for packet rate fairness and to detect heavy hitters.
	Hedera [2]	Flow Counters, Balancing	Detect heavy hitters and balance them.
	Presto [38]	Flow Statistics, Balancing	Create flowlets and balance them.
QoS, Fairness	Seawall [80]	Arithmetic, Cardinality, Metering	Collect traffic statistics and provide RCP-like feedback to determine the per-link, per-entity share.
	FairCloud [73]	Arithmetic, Flow Counters	Meter number of source/destination flows and queue them into approximately prioritized queues.
	CoDel [68]	Arithmetic, Metering	Section 3.2.4.
	pFabric [9]	Arithmetic, Flow Counters	Queue packets into approximately prioritized queues using remaining flow length value in the header.
Access Control	Snort IDS [75]	Flow Counters, Cardinality	Section 3.2.4.
	OpenSketch [91]	Flow Counters, Metering	Approximate flow statistics are based on count-min sketch.

Table 3.1: Network functions that can be supported using FlexSwitch building blocks. A deeper discussion of several of the algorithms is given in Section 3.2.

amount of state to approximate digests as data is streamed through. Often, there is a tradeoff between the accuracy of the measurement and the amount of hardware resources devoted to implementing the measurement. We draw attention to these tradeoffs and evaluate them empirically. We do so by measuring the resource use of faithful C/C++ implementations of the building blocks under various configuration parameters. In later sections, we use P4 specifications of these building blocks, evaluate performance using a production FlexSwitch, and measure resource requirements in an emulator for an additional production FlexSwitch.

Building Block	**Functionality**	**Techniques**
Cardinality Estimator	Estimate #unique elements in a stream	Linear Counting, Hyperloglog
Flow Counters	Estimate per-flow bytes/packets	Count-min sketch
Flow Timestamps	Last packet sent timestamp	Sketch with timestamps
Metering queues/utilization	Estimate rates	EWMA
Port Balancing	Pick a port based on some metric	Power-of-2 choices
Approx Arithmetic	Division/Multiplication	Logarithm tables, Bit-shits

Table 3.2: Summary of the proposed building blocks and techniques to implement them.

3.1.1 Flow Statistics

Approximating Aggregate Flow Statistics

Many resource allocation algorithms require the computation of aggregate values such as the total number of active flows, the total number of sources and destinations communicating through a switch, and so on. Exact computation of these values would require large amounts of both state and per-packet computation. We, therefore, design a **cardinality estimator** building block that can approximate the number of unique elements (typically tuples of values associated with a subset of header fields) in a stream of packets in a given time frame. We expect to use this building block in contexts where there are many unique elements to track and a minor loss of accuracy can be tolerated. Thus, this building block trades accuracy for space.

Our approach is adapted from streaming algorithms [72]. We calculate a hash of each element and count the number of leading zeros n in the binary representation of the result using a TCAM. Using this approach, we compute and store the maximum number $\max(n)$ of leading zeros over all elements of the incoming stream. $2^{\max(n)}$ is then the expected number of unique elements in the stream. To implement this calculation on FlexSwitches, we use a ternary match table described by the string:

$$0^n 1 x^{N-n-1} \qquad 0 \leq n < N$$

where N is the maximum number of bits in the hash result. A stateful memory cell maintains the maximum number of leading zeros observed in the stream. This approach allows us to both update and retrieve the count n on the data plane.

This basic approach is space-efficient but exhibits high variance for the estimate. The variance can be reduced by using multiple independent hashes to derive independent estimates of the cardinality. These independent estimates are averaged to produce a final estimate. An alternative is to divide the incoming stream into k disjoint buckets. For example, we can use the last $log(k)$ bits of the hash result as a bucket index. We can estimate the cardinality of each bucket separately and then combine them to derive the final estimate by taking the harmonic mean [72].

Different traffic properties can be estimated by providing different header fields to the hash function. For example, if the header fields are the 4-tuple of source IP, destination IP, source port, and destination port, then we can estimate the number of active flows traversing a switch. Similarly, we can calculate the number of unique end-hosts/VMs/tenants traversing the switch or collect traffic statistics associated with anomalies.

The parameters are the number h of hash functions to use, the maximum number N of bits in each hash result, and the number k of hash buckets to use. Given these parameters, we can ask: how much switch state is required to achieve a given level of accuracy and how does accuracy trade-off with switch state?

Prior datacenter studies [6, 76], measure the number of concurrent flows per host to be 100-1000. This means a ToR switch can have 1,000-50,000 flows. Assuming we use a single hash function with $N = 32$, and 1 byte of memory for counting the number of leading zeroes in each

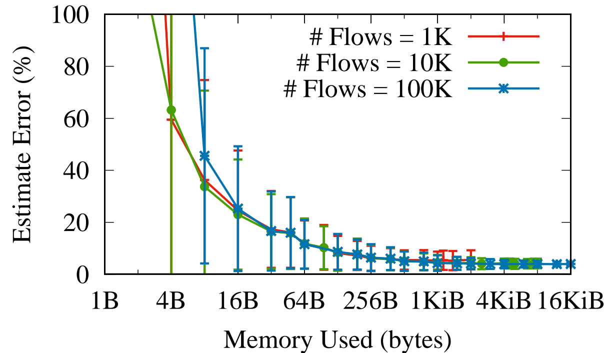


Figure 3.1: Average tradeoff between accuracy and memory used by the cardinality estimating building block for different set sizes of random flows over 1000 measurements (error bars show standard deviation).

bucket, Figure 3.1 shows the tradeoff of accuracy versus memory usage for different flow set sizes. We vary memory use, which equals $h \times k$ bytes, for $k \in [1, 2^{16}]$ and $h \in [1, 8]$ and show the tradeoff for the best choice of h and k for a particular memory size.

Accuracy improves with increased memory, but the flow set size does not impact this tradeoff significantly. Using less than 64 bytes produces error rates above 20%. Memory between 64 bytes and 1 KB has average error rates between 5% and 15%. Using more than 1 KB of memory only improves average error rates marginally (to 4%), but it can improve worst-case error rates.

We do lose accuracy for very small ($\#Flows \approx k$) and very large ($\#Flows \approx 2^N$) flow-set sizes. For very small flow sets, several buckets will remain empty. This introduces distortions which can be corrected by calculating the estimate as $\log(k/\#emptyBuckets)$ instead. For very large flow sets, frequent hash collisions cause us to underestimate the cardinality. This is not grave as the flow set size is already large, but could be avoided by using a wider hash result. We also tried using more than one hash function to improve accuracy (in order to derive independent estimates of the cardinality), but we found that accuracy is improved only marginally (within 1%) and that the accuracy increase does not justify the multi-fold increase in resource utilization.

In summary, the cardinality estimation building block can work for a wide range of packet property sets with low error and moderate resources, while providing a configurable tradeoff between accuracy and resource use.

Approximating Per-Flow Statistics

Resource limits prevent maintaining accurate per-flow state, so this class of building blocks approximates flow statistics. We discuss two building blocks. The first building block provides per-flow counters. It can be used, for example, to track the number of bytes transmitted by individual flows to identify elephant flows. The second building block tracks timestamp values for a particular flow, e.g. to detect when a flow was last seen for flowlet detection.

Per-flow Counters: For this building block we use a count-min sketch [30] to track element counts in sub-linear space. It supports two operations: `update(e,n)` adds `n` to the counter for element `e` and `read(e)` reads the counter for element `e`. This building block requires a table of r rows and c columns, and r independent hash functions. `update(e,n)` applies the r hash functions to `e` to locate a separate cell in each row, and then adds `n` to each of those r cells. `read(e)` in turn uses the r hash functions to find the same r cells for `e` and returns the minimum. The approximation of the count returned by `read` is always greater than or equal to the exact count. This can be realized on FlexSwitches by storing each row in a separate match+action stage which then requires only one computation operation per stage. To keep the sketch from saturating, we use the switch CPU to periodically multiply all values in the counter table by a factor < 1 (i.e., decay the counts across multiple intervals), or reset the whole table to 0, depending on the use-case. Table 3.3 shows an example of this building block used for heavy-hitter detection, and the size of the sketch required to achieve a specified accuracy level.

Per-Flow Timestamps: To track the timestamp of the last received packet within a flow, we modify the use of the count-min sketch to derive a *min-timestamp* sketch. Instead of adding `n` (the current time) to the selected cells, `update(e,n)` now just overwrites these cells with `n`. `read` remains unmodified and as such returns a conservative estimate for the last time an element has been seen, i.e., it is always higher than the true flow timestamp.

One use-case for this block is flowlet switching, wherein packets of a live flowlet should all be routed along the same route but new flowlets can be assigned a new route. Flowlet switching re-

N	False-positive	r	c	$r \times c$
	10%	4	512	2,048
1,000	5%	3	1,024	3,072
	1%	3	2,048	6,144
	10%	2	8,192	16,384
10,000	5%	3	8,192	24,576
	1%	4	16,384	65,536
	10%	3	65,536	196,608
100,000	5%	2	131,072	262,144
	1%	4	131,072	524,288

Table 3.3: Required count-min sketch size to achieve false-positive rate below the specified threshold. The workload consists of N flows that send a total of 10 M packets, 80% of which belong to 20% of the flows—the heavy hitters (HH)— and the HH-threshold is set based on the expected number of packets sent from a HH in this scenario.

quires two pieces of information per flow: the timestamp of the previous packet, and the assigned route. Our timestamp building block tracks flow timestamps, but storing the route information requires an extension of the building block.

The extended building block supports the following two operations: `update(e,n,r)` sets the timestamp of element `e` to `n` and, if the flowlet timestamp had expired, sets the route to `r`; and `read(e)` returns both the last timestamp and the route for element `e`. We assume that the chosen route for a flow can be represented as a small integer that represents its index in a deterministic set of candidate routes for the flow. Both `update` and `read` still use the same mechanism for finding r cells for a particular `e`, but we extend those cells to store both the timestamp and a route integer. `read` returns the minimum of the timestamps and *the sum* of the route integers from the cells. `update` still updates the timestamps in all selected cells, but will only modify the route information in cells where the timestamp is older than the configured flowlet timeout. If no cell has a timestamp older than the timeout, then the flowlet is considered live and thus the route is not modified. Any non-live flowlet will have at least one cell that has a timestamp older than the

timeout, thus the route values stored in these cells can be modified so that the sum of all route values equals the specified r. The pseudocode for this building block is shown in Figure 3.2.

```
sketch = {ts, route_id}[2][N]

elements(five_tuple):
  h1, h2 = hashes(five_tuple)
  e1 = sketch[0][h1 % N]
  e2 = sketch[1][h2 % N]

read(five_tuple):
  e1, e2 = elements(five_tuple)
  ts = min(e1.ts, e2.ts)
  route = e1.route_id + e2.route_id
  return (ts, route)
```

```
update(five_tuple, ts, route_id):
  e1, e2 = elements(five_tuple)
  cutoff = ts - TIMEOUT
  if (e1.ts < cutoff && e2.ts < cutoff)
    e1.route_id = rand()
    e2.route_id = route_id - e1.route_id
  else if (e1.ts < cutoff)
    e1.route_id = route_id - e2.route_id
  else if (e2.ts < cutoff)
    e2.route_id = route_id - e1.route_id

  e1.ts = e2.ts = ts
```

Figure 3.2: Pseudocode for the flowlet switching building block. This particular sketch uses two hash functions and two rows of N counters each, but generalizes to multiple hash functions in a straightforward manner.

The key observation for correctness is that the route value in a cell with a live timestamp will never be modified, which guarantees that none of the route information for a live flowlet is modified, because the flowlet's cells will all have live timestamps.

3.1.2 Approximating Arithmetic

FlexSwitches do not support operations such as multiplication and division in hardware. We design a class of building blocks that provide an exact implementation of complex arithmetic where possible and approximations for the general case. We can exactly compute multiplication and division with a constant operand using bit shifts and addition. We resort to approximation only when operands cannot be represented as a short sequence of additions. Our approximate arithmetic supports two variable operands and relies on lookup tables and addition.

Bit shifts: Multiplication and division by powers of 2 can be implemented as bit shifts. If the constant operand is not a power of 2, adding or subtracting the results from multiple bit shifts provides the correct result, e.g., $A \times 6$ can be rewritten as $A \times 2 + A \times 4$. Similarly, a division can

be implemented by rewriting as multiplication with the inverse. Resource constraints (such as the number of pipeline stages) limit the number of shifts and additions that can be executed for a given packet.

Logarithm lookup tables: Where multiplication and division cannot be carried out exactly (e.g., if both operands are variables), we use the fact that $A \times B = \exp(\log A + \log B)$. Given log and exp, we can reduce multiplication and division to addition and subtraction. Because FlexSwitches do not provide log and exp natively, we approximate them using lookup tables.

N	Error	m	l	exp (SRAM)	log (TCAM)
	10%	3	6	64	59
16	5%	4	7	128	111
	1%	6	9	512	383
	10%	3	7	128	123
32	5%	4	8	256	239
	1%	6	10	1024	895
	10%	3	9	512	251
64	5%	4	9	512	495
	1%	6	11	2048	1919

Table 3.4: Required number of lookup table entries for an approximation of multiplication/division for different size operands (N in bits) for the smallest configuration (m and l which control precision) with a mean relative error below the specified threshold.

We use a ternary match table to implement logarithms. For N-bit numbers and a window of calculation accuracy of size m, with $1 \leq m \leq N$, table entries match all bit-strings of the form:

$$0^n 1(0|1)^{min(m-1,N-n-1)} x^{max(0,N-n-m)} \qquad 0 \leq n < N$$

where x is the wildcard symbol. These entries map to the l-bit log value—represented as a fixed point integer—of the average number covered by the match. For example, for $N = 3$ and $m = 1$, the entries are $\{001, 01x, 1xx\}$, and for $m = 2$ the table entries are $\{001, 010, 011, 10x, 11x\}$. exp

is calculated using an exact match table that maps logarithms back to the corresponding integer value. The parameters m and l control the space/accuracy tradeoff for this building block. For N-bit operands, table size is approximately $N \times 2^m$ for the log table and precisely 2^l for the exp table. Table 3.4 shows the minimal values of m and l to achieve a mean relative error below the specified thresholds. Note that even for 64-bit numbers a mean relative error below 1% can be achieved within 2048 exact match and ternary match table entries.

3.1.3 Switch Statistics

Metering queue lengths

Network protocols often require information about queue lengths. FlexSwitches provide queue length information as part of the packet metadata, but only in the egress pipeline and only for the queue just traversed. This building block tracks queue lengths and makes them available through the entire processing pipeline. When a packet arrives in the egress pipeline we record the length of queue traversed in a register for that queue. Depending on the use-case, this building block can be configured to track the minimal queue length seen in a specified time interval, and using a timer to reset the register at the end of every interval. A variant of this building block is to calculate a continuous exponentially weighted moving average (EWMA) of queue lengths, and this utilizes the approximate arithmetic building block to perform the weighting.

Metering Rates

Similarly, data rates are an important metric in resource allocation schemes. FlexSwitches provide metering functionality in hardware, but the output is generally limited to colors that are assigned based on thresholds as described in the P4 language. This building block can measure rates for arbitrary input events in a more general fashion. We provide two configurations: the first measures within time intervals, and the other measures continuously. We describe the latter as the former is straightforward.

Timer-based Metering: This configuration is applicable if a new estimation every timer interval is sufficient. A counter or a register with addition is used to aggregate the input events during a timer interval. When the timer expires, we divide the aggregate value by a configurable decay factor. We store the resulting rate in the output register and reset the counter to zero. This division can be implemented with a bit shift.

Continuous Metering: In order to support continuous measurement of the rate during the last time interval T, we use two registers to store the rate and the timestamp of the last update. When updating the estimate, we first calculate the time passed since the last update as t_Δ. Because there were no events during the last t_Δ ticks, we can calculate how many events occurred during the last time interval T based on the previous rate R as $Y = R \times (T - t_\Delta)$ (or 0 if $t_\Delta > T$). Based on that and the number of events x to be added with the current operation we update the rate to $\frac{Y+x}{T}$, and also update the timestamp register with the current timestamp. The multiplication with R for calculating Y is implemented using the second method described in the approximate multiplication building block, while the division by T can be implemented using a right shift.

Approximate Port Balancing

A common task is to assign packets to one of multiple destinations (e.g., links or servers) to balance the load between them. We provide a class of building blocks that implement different balancing mechanisms. The building blocks in this class fall in two separate categories, static balancing steers obliviously to the current load while dynamic balancing takes the current load into account.

Static balancing: Equal-cost multi-path (ECMP) is an example of static balancing: The switch calculates a hash over the flow's 5-tuple (protocol, source/destination IP and port) and uses this hash to consistently assign packets of a flow to a destination, thereby avoiding reordering. Additional inputs can be provided to this building block to be included in the hash, e.g. a flowlet number. Instead of using the hash directly, this building block can use an additional lookup table

to lookup (part of) the hash and possibly other fields – e.g., an destination address prefix – for determining the destination. This provides more flexibility to choose the destination based on packet criteria and to assign different weights to destinations by adding more table entries for some destinations.

Dynamic balancing: Making a load balancing decision while taking into account the current load avoids the imbalances common with static load balancing schemes. Computational limitations on FlexSwitches make it infeasible to pick the least loaded destination from a set of candidates larger than 2-4. Previous work [12] has shown that picking the least loaded destination from a small subset – even just 2 – of destinations chosen at random, significantly reduces load imbalances. Picking random candidates can be implemented on FlexSwitches by calculating two or more hashes on high-entropy inputs (timestamp, or other metadata fields). Information about the load of different destinations can be obtained from the *metering queue lengths* or *metering rates* building blocks.

3.1.4 Discussion

We note that our building blocks address both short and long-term limitations associated with FlexSwitches. For example, some of our building blocks emulate complex arithmetic using simpler primitives and help make the case for supporting some of these complex operations in future versions of FlexSwitches. The rest of the building blocks address fundamental constraints associated with switch state and operation count, thereby allowing the realization of a broader class of protocols that are robust to approximations (as we will see in the next section).

We provide the building blocks in a *template* form, parameterized by zero or more packet header fields. Each block either rewrites the packet header or maintains state variables that are available to subsequent building blocks in the pipeline. For example, the cardinality estimator can be parameterized by the 5-tuple or just the source address and exposes a single variable called ‘cardinality’. This variable can be re-used when blocks are chained within a pipeline.

There is a class of complex protocols that cannot be supported by our building blocks. These

include algorithms that maintain per-flow state in sorted order and require updates for every packet arrival. This is because it requires a potentially unbounded number of state lookups and updates for processing a single packet. One such example is PDQ [40].

Tables 3.2 and 3.1 summarize our building blocks and the different classes of protocols we can support using them. We can realize a variety of schemes ranging from classical congestion control and scheduling to load balancing, QoS, and fairness. This shows our blocks are general enough to be reused across multiple protocols and sufficient to implement a broad class of resource allocation protocols.

3.2 Realizing Network Resource Allocation Protocols

In this section, we describe several network resource allocation protocols that can be built using the building blocks described in Section 3.1. The network protocols that we target fall into the following broad categories: congestion control, load balancing, QoS/Fairness, and IDS/Monitoring.

3.2.1 RCP

Rate Control Protocol (RCP) [33] is a feedback-based congestion control algorithm that relies on explicit network feedback to reduce flow completion times. RCP attempts to emulate processor sharing on every switch and assigns a single maximum throughput rate to all flows traversing each switch. In an ideal scenario, the rate assigned at time t is simply $R(t) = C/N(t)$, where C is the link capacity and $N(t)$ is the number of ongoing flows, and the rate of each flow is the minimum across the path. Each switch computes $R(t)$ every *control interval*, which is typically set to be the average round-trip time (RTT) of active flows.

In RCP, every packet header carries a rate field R_p, initialized by the sender to its desired sending rate. Every switch on the path to the destination updates the rate field to its own $R(t)$ if $R(t) < R_p$. The receiver returns R_p to the source, which throttles its sending rate accordingly. The packet header also carries the source's current estimate of the flow RTT. This is used by each

switch to compute the control interval. For a precise calculation, per-flow RTTs need to be kept[1].

The original RCP algorithm computes the fair rate $R(t)$ using the equation

$$R(t) = R(t-d) + \frac{\alpha \cdot S - \beta \cdot \frac{Q}{d}}{\hat{N}(t)}$$

where d is the control interval, S is the spare bandwidth, Q is the persistent queue size and α, β are stability constants. $\hat{N}(t)$ is the estimated number of ongoing flows. This *congestion controller* maximizes the link utilization while minimizing queue depth (and drop rates). If there is spare capacity available (i.e., $S > 0$), RCP increases the traffic allocation. If queueing is persistent, RCP decreases rates until queues drain.

FlexSwitch Implementation

We now illustrate how to orchestrate several of the building blocks described in Section 3.1 to implement RCP. Staying true to our design principles, we employ approximation when necessary to make the implementation possible within limited resource bounds. We recall: To implement RCP we require a way of metering average utilization and queueing delay over a specified control interval. We also need to determine the number of active flows for each outgoing port over the same interval.

First, we estimate the spare capacity and the persistent queues built up inside the switch using the *metering utilization and queueing* building block. We use the building block to maintain per-link meta-data for the number of bytes received and the minimum queue length observed during a control period. When a packet arrives, we update Q by taking the minimum of the previous Q value and the current queue occupancy size as measured by our building block. Similarly, a counter B accumulates the number of bytes sent over the link. A timer is initialized to d, the expected steady-state RTT of most flows. When the timer expires, we calculate the spare bandwidth as $S = C - B/d$, where C is the total capacity of the link. d is rounded down to a power of two so that the division operation can be replaced by a bit-shift operation. This use of a slightly smaller

[1] However, it is possible to approximate the average RTT and only keep an aggregate of the number of flows.

d is consistent with RCP's recommendation of using control periods that are roughly comparable to the average RTT.

RCP approximates the number of flows using a circular definition: $\hat{N}(t) = \frac{C}{R(t-d)}$. This essentially estimates the number of flows using the rate computed in the previous control interval. We use a more direct approximation of the number of unique flows by employing the *cardinality estimator* building block.

Finally, with estimates of utilization, queueing, and the number of flows, we can use the RCP formula for calculating $R(t)$. Given that RCP is stable for a wide range of $\alpha, \beta > 0$, we pick fractional values that can be approximated by bit shift operations. For the division by $N(t)$, a general division is required because both sides are variables. We use the *approximate divider* building block to perform the division with sufficient accuracy.

3.2.2 XCP

Like RCP, the eXplicit Control Protocol (XCP) [47] is a congestion control system that relies on explicit feedback from the network, but optimizes fairness and efficiency over high bandwidth-delay links. An XCP-capable router maintains two control algorithms that are executed periodically on each output port: a congestion controller and a fairness controller. The congestion controller is similar to RCP's—it computes the desired increase or decrease in the aggregate traffic (in bytes) over the next control interval as $\phi = \alpha \cdot d \cdot S - \beta \cdot Q$, where S, Q, and d are defined as before.

The fairness controller distributes the aggregate feedback ϕ among individual packets to achieve per-flow fairness. It uses the same additive increase, multiplicative decrease (AIMD) [10] principle as TCP. If $\phi > 0$, it increases the throughput of all flows by the same uniform amount. If $\phi < 0$, it decreases the throughput of a flow by a value proportional to the flow's current throughput. XCP achieves AIMD control without requiring per-flow state by sending feedback in terms of changes in the congestion window, and through a formulation of the feedback values designed to normalize feedback to account for variations in flow rates, packet sizes, and RTTs.

In particular, given a packet i of size s_i corresponding to a flow with current congestion window $cwnd_i$ and RTT rtt_i, XCP computes the positive feedback p_i (when $\phi > 0$) and the negative

feedback n_i (when $\phi < 0$) as:

$$p_i = \xi_p \cdot \frac{rtt_i^2 \cdot s_i}{cwnd_i} \qquad\qquad n_i = \xi_n \cdot rtt_i \cdot s_i$$

where ξ_p and ξ_n are constants for a given control interval. Observe that the negative feedback is simply a uniform constant across packets if all flows have the same RTT and send the same sized packets. As a consequence, flows that send more packets (and hence operate at a higher rate) will get proportionally higher negative feedback and will multiplicatively decrease their sending rates in the next control interval. Similarly, the structure of p_i results in an additive increase as the per-packet feedback is inversely proportional to $cwnd_i$. Finally, the per-interval constants ξ_p and ξ_n are computed such that the aggregate feedback provided across all packets equals ϕ, with L being the set of packets seen by the router in the control interval:

$$\xi_p = \frac{\phi}{d \cdot \sum_L \frac{rtt_i \cdot s_i}{cwnd_i}} \qquad\qquad \xi_n = \frac{-\phi}{d \cdot \sum_L s_i}$$

FlexSwitch Implementation

The core of the XCP protocol requires each switch to (i) calculate and store at every control interval the values ϕ, ξ_p, and ξ_n, and (ii) compute for every packet the positive or negative feedback values (p_i or n_i). p_i and n_i are communicated back to the sender, while $cwnd_i$ and rtt_i are communicated to other routers to allow them to compute ξ_p and ξ_n. Given the programmable parser in a FlexSwitch, it is straightforward to extend the packet header to include fields corresponding to $cwnd_i$, rtt_i, and either p_i or n_i. However, the XCP equations described above require complex calculations. We outline a sequence of refinements designed to address this.

Approximate computations: As with RCP, we make use of the *metering utilization and queueing* building block and then suitably choose the stability constants and the control interval period to simplify computations in the processing pipeline. First, we set the stability constants α, β to be negative powers of 2, such that ϕ can be calculated using bit shifts. XCP is stable for $0 < \alpha < \frac{\pi}{4\sqrt{2}}$ and $\beta = \alpha^2\sqrt{2}$, which makes this simplification feasible. Next, we approximate the control inter-

val d to be a power of two that approximates the average RTT in the datacenter. We can then compute ϕ/d every d microseconds using integer counters and bit-shifts without incurring the loss in precision associated with the approximate division building block.

End-host computations: In XCP, end-hosts send rtt_i and $cwnd_i$ values with every packet and receive from the switches a per-packet feedback p_i, n_i. In our FlexSwitch implementation, we offload more of the computation to the end-hosts. First, we require the end-host to send to switches the computed value $\frac{rtt_i \cdot s_i}{cwnd_i}$. Second, instead of calculating the per-packet feedback p_i, n_i at the switch, we simply send the positive and negative feedback coefficients ξ_p, ξ_n to the end-host. The end-host can then calculate the necessary feedback by computing p_i, n_i based on its local flow state. This results in a subtle approximation as we can no longer keep track of aggregate feedback at the router. It is possible that we give out more aggregate feedback than the target ϕ if a large burst of packets arrives during a control interval. This can temporarily cause a queue buildup inside the switch, but XCP's negative feedback will quickly dissipate the queue. We did not see large queue buildups in our simulations.

Approximate control intervals: Finally, instead of computing the positive and negative feedback coefficients ξ_p, ξ_n along with ϕ at the end of every control interval, we compute them whenever the denominator reaches a value close to a power of 2. In our case, we approximate $\sum_L \frac{rtt_i \cdot s_i}{cwnd_i}$ or $\sum_L s_i$ to a power of 2 for positive and negative feedback coefficients respectively, both of which are in the XCP congestion header and are accumulated in integer counters inside switch memory. As a result, calculating ξ_p, ξ_n requires only a bit-shift operation. It also means that we are not calculating all XCP parameters synchronously at every control interval, but rather at staggered intervals as shown in Figure 3.3.

3.2.3 CONGA

CONGA [5] is a congestion-aware load balancing system that splits TCP flows into flowlets and allocates them to paths based on congestion feedback from remote switches. Congestion is commu-

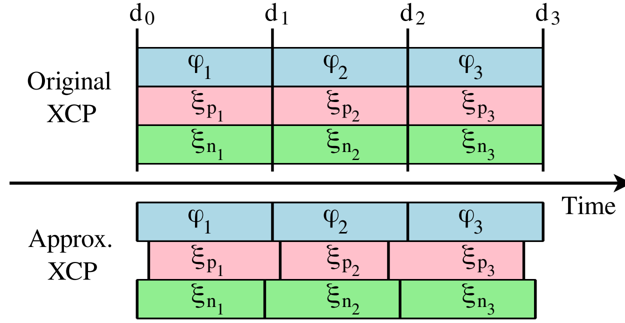


Figure 3.3: Staggered evaluation of XCP control parameters.

nicated among switches with each payload packet by embedding congestion information within unused bits of the VXLAN [59] overlay network header that is common in datacenters today.

CONGA operates primarily at leaf switches and does load balancing based on per-uplink congestion. Each leaf switch holds two tables that contain congestion information along all possible uplink choices from and to other leaf switches and are updated by information from these other leaf switches. To forward a packet, a leaf switch picks an outgoing link and records its choice in the packet header. Core switches record the maximum congestion along the path to the packet's destination by updating a header field with the maximum of their local congestion and that already in the field. Finally, the destination leaf switch updates its congestion-from-leaf table according to the recorded congestion along the sender's uplink port choice. To relay congestion information back to sender switches, each switch additionally chooses one entry in its congestion-from-leaf table for that switch and transmits it using another set of CONGA header fields within each payload packet.

To estimate local link congestion, all switches use a discounting rate estimator (DRE). DREs use a single register X to keep track of bytes sent along a link over a window of time. For each sent packet, the register is incremented by that packet's size. The register is decremented periodically with a multiplicative factor α between 0 and 1: $X \longleftarrow X \times (1 - \alpha)$.

CONGA's load-balancing algorithm is triggered for each new flowlet. It first computes for each uplink the maximum of the locally measured congestion of the uplink and the congestion

feedback received for the path from the destination switch. It then chooses the uplink with the minimum congestion for the flowlet.

To recognize TCP flowlets, leaf switches keep flowlet tables. Each table entry contains an uplink number, a valid bit, and an age bit. For each incoming packet, the corresponding flowlet is determined via a hash on the packet's connection 5-tuple that indexes into the table. If the flowlet is valid (valid bit set), we simply forward the packet on the recorded port. If it is not valid, we set the valid bit and determine the uplink port via the load-balancing algorithm. Each incoming packet additionally resets the age bit of its corresponding flowlet table entry. A timer periodically checks the age bit of all flowlets before setting it. If a timer encounters an already set age bit, then the flowlet is timed out by resetting the valid bit. To accurately detect enough flowlets, CONGA switches have on the order of 64K flowlet entries in a table.

FlexSwitch Implementation

Relaying CONGA congestion information is straight-forward in FlexSwitches: We simply add the required fields to the VXLAN headers. Since CONGA is intended to scale only within 2-layer cluster sub-topologies, congestion tables are also small enough to be stored entirely within Flex-Switches. To implement the remaining bits of CONGA, we need the following building blocks:

Measuring Sending Rates: We need to measure the rate of bytes sent on a particular port, which CONGA implements with DREs. We use the timer-based rate measurement building block for this task. We set up one block for each egress port to track the number of transmitted bytes along that port and set the timeout and multiplicative decrease factor to CONGA-specific values.

Maintaining congestion information: We can simply fix the size of congestion tables to a power of 2 and use precise multiplication via bit shifts and addition to index into the 2-dimensional congestion tables.

Flowlet detection: To identify TCP flowlets and remember their associated paths we use the flow statistics building block. We mimic CONGA's flowlet switching without a need for valid or age bits.

CONGA flowlet switching: To remember the initially chosen uplink for each flowlet, we store the uplink number as a tag inside the high-order bits of the flow's associated counter. We use this tag directly for routing. At the same time, we ensure that per-flow counters are reset frequently enough so that the tag value will never be overwritten due to the flow packet counter growing too large. To do so, we simply calculate the maximum number of min-sized packets that can traverse an uplink within a given time frame and ensure that this number stays below the allocated counter size (32 bits), minus the space reserved for the tag (2 bits in current ToR switches). For 100Gb/s uplinks, more than 5 seconds may pass sending min-sized packets at line-rate before counter overflow.

CONGA's load balancing requires a complex calculation of minimums and maximums—too many to be realized efficiently on a FlexSwitch. Rather than faithfully replicating the algorithm and determining the best uplink port upon each new flowlet, we keep a running tally of the minimally-congested uplink choice for each leaf switch. Upon a new flowlet, we simply forward along the corresponding minimally-congested uplink.

Our running tally needs to be updated each time a corresponding congestion metric is updated. If this results in a new minimum, we simply update the running minimum. However, if an update causes our current minimum to fall out of favor, we might need to find the new current minimum. This would require re-computation of the current congestion metric for all possibilities—an operation we want to avoid. Instead, we simply update our current "minimum" to the value we have at hand as a best guess and wait for further congestion updates from remote switches to tell us the true new minimum. In our current implementation, we also do not update our tally when local rate measurements are updated. This spares further resources at minimal accuracy loss.

3.2.4 Other use-cases

The simple use-cases typically apply our building blocks in a direct manner to achieve their goals. We provide here a few examples to give insight into how our building blocks apply to a wide variety of different network applications.

WCMP: Weighted Cost Multipath (WCMP) routing [93] is an improvement over ECMP that can balance traffic even when underlying network performance is not symmetric. For example, heterogeneous network components and/or failures might make some paths more desirable than others. Furthermore, network component failure can dynamically change which paths are more desirable. WCMP uses *weights* to express path preferences that consequently impact load balance. To implement WCMP using FlexSwitches, we use the approximate port balancing building block over a next-hop hash table and replicate next-hop entries according to their weight. The WCMP weight reduction algorithm [93] applies in the same way to reduce table entries.

CoDel: This QoS mechanism [68] monitors the minimum observed per-packet queueing delay during a time interval and drops the very last packet in the interval if this minimum observed delay is above a threshold. If there is a packet drop, the scheme provides a formula for calculating a shorter interval for the next time period. This scheme can be easily implemented on a FlexSwitch using metering, a small amount of state for maintaining the time period, approximate arithmetic for computing the next interval, and timers.

TCP port scan detection: To detect TCP port scans, we filter packets for set SYN flags and use the cardinality estimation building block to estimate the number of distinct port numbers observed. If this estimate exceeds a set threshold, we report a scan event to the control plane.

NTP amplification attack detection: Similarly, to detect NTP amplification attacks, we detect NTP packets (UDP port 123) and use cardinality estimation of distinct source IP addresses. If the number is high, we conclude that a large number of senders is emitting NTP requests over a

small window of time and report an attack event.

3.3 Evaluation

To show that we achieve our goal of implementing complex network resource allocation problems on limited hardware resources using only approximating building blocks, we first implement RCP on top of a real FlexSwitch. We use the Cavium Xpliant CNX880xx [24], a fully flexible switch that provides several of the features described in Section 2.2 while processing up to 3.2 Tb/s. We then evaluate the accuracy of our implementations versus the non-approximating originals using ns-3 simulations. Finally, we evaluate the resource usage of our implementations and discuss whether they fit within the limited resources that are expected of FlexSwitches.

3.3.1 Hardware Implementation

Our hardware implementation of RCP uses several features of the XPliant CNX880xx switch. First, we use the configurable counters to build the cardinality estimator. An array of counters indexed by the packet hash is incremented on each packet arrival. The counter values are read periodically to estimate the number of on-going flows. Next, we use the metering block to maintain port level statistics such as bytes transmitted and queue lengths. For periodically computing RCP rates, we utilize an on-switch service CPU to compute and store the rates inside the pipeline lookup memory. Finally, we program the reconfigurable pipeline to correctly identify and parse an RCP packet, extract the rate and rewrite it with the switch rate if it is lower.

To measure the performance of our RCP implementation against other protocols, we emulate a 2-level FatTree topology consisting of 8 servers, 4 ToRs and 2 core switches by creating appropriate VLANs on the switch and directly interconnect the corresponding switch ports. This way, the same switch emulates all switches in the topology. All links operate at 10Gbps. We generate flows from all servers towards a designated target server with Poisson arrivals such that the ToR links are 50-60% utilized. The flows are Pareto distributed ($\alpha = 1.2$), with a mean size of 25 packets. We measure the flow completion times and compare it with the default Linux TCP-cubic implementation.

Table 3.5 shows the flow completion times for various flow sizes. As expected, RCP benefits significantly from the switch notifying the end-hosts the precise rate to transmit at. This avoids the need for slow-start and keeps the queue occupancy low at the switches, leading to lower flow completion times.

Flow Size	**TCP**			**RCP**		
	Mean	$50^{th}\%$	$95^{th}\%$	Mean	$50^{th}\%$	$95^{th}\%$
Short	10.32	0.85	2.92	0.85	0.73	2.05
Medium	67.97	5.33	216.99	5.07	3.18	14.85
Long	649.25	59.73	3559.85	50.42	36.85	137.26

Table 3.5: Flow completion times for short, medium, and long flows (< 50, < 500, and ≥ 500 packets) in milli-seconds on a two-tier topology running RCP on Cavium hardware.

3.3.2 ns-3 Simulations

To evaluate the accuracy of our use cases, we implement them within the ns-3 network simulator [69] version 3.23. We simulate a datacenter network topology consisting of 2560 servers and a total of 112 switches. The switches are configured in a 3-level FatTree topology with a total over-subscription ratio of 1:4. The core and aggregation switches each have 16 × 10Gbps ports while the ToRs each have 4 × 10Gbps ports and 40 × 1Gbps ports. All servers follow an on-off traffic pattern, sending every new flow to a random server in the datacenter at a rate such that the core link utilization is approximately 30%. Flows are generated using a Pareto distribution ($\alpha = 1.2$), and a mean flow size of 25 packets. Simulations are run for long enough that flow properties can be evaluated.

To measure the impact of approximation on the accuracy of our FlexSwitch implementation of RCP, we compare our implementation (RCP-Approx) to an original implementation (RCP).

Figure 3.4 shows the result as a number of CDFs of flow completion time (FCT) for flows of various lengths. We can see that RCP-Approx matches the performance of RCP closely, for all three types of traffic flows. We also compare to the performance of TCP to validate that our

simulation adequately models the performance of RCP. We see that this is indeed the case as RCP performance exceeds TCP performance for shorter flows, as shown in [33].

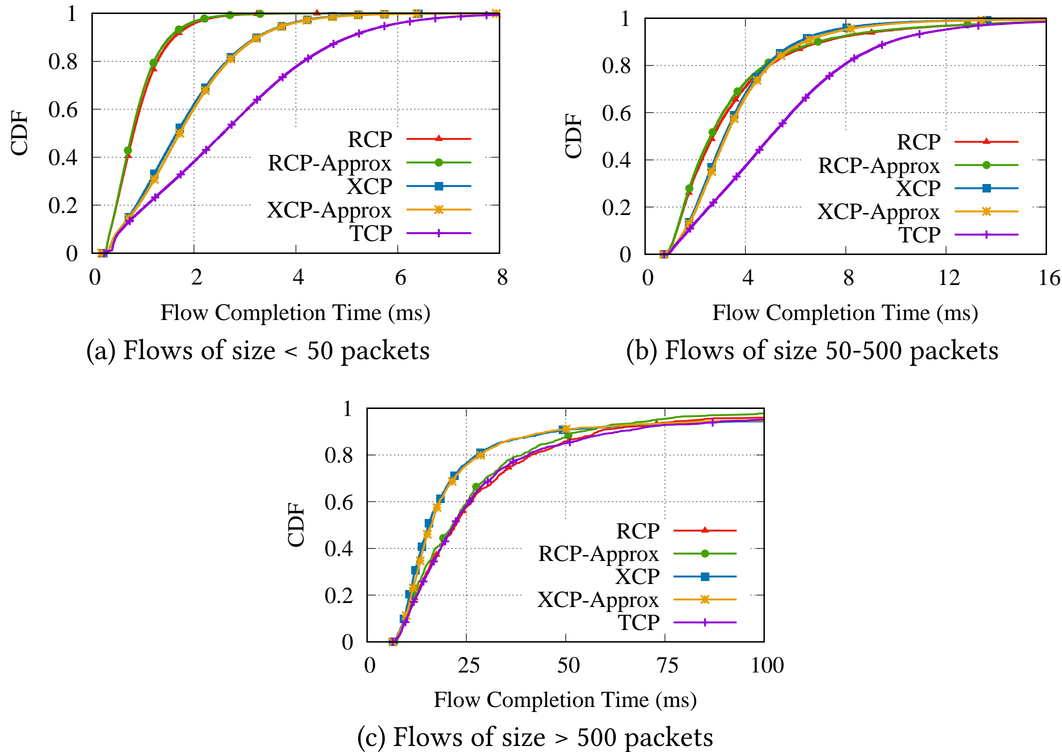


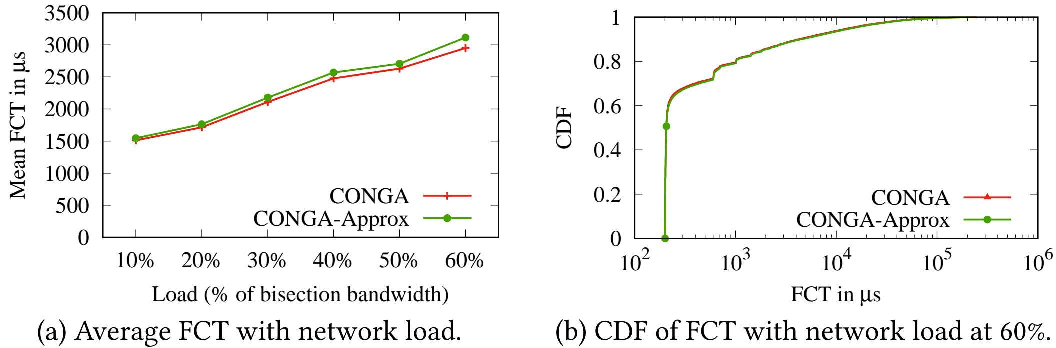
Figure 3.4: Cumulative distribution of FCTs of various flow sizes for TCP as well as both precise and approximate RCP and XCP.

Similarly, we measure the impact of approximation on the accuracy of our XCP FlexSwitch implementation by comparing it (XCP-Approx) to an original implementation of XCP using the same simulated workload. From Figure 3.4 we can see that XCP-Approx also closely matches the performance of XCP, for all three types of traffic flows. Both implementations outperform RCP for long flows and perform worse than RCP for short flows. This validates our simulation as XCP is optimized for long flows over high bandwidth-delay links, while RCP is optimized for short flows.

Next, we compare the performance of approximate CONGA implemented using our building blocks to the original version of CONGA. We simulate the same 4-switch (2 leaves, 2 spines), 64

server topology in [5] with $50\mu s$ link latency and the default CONGA parameters: $Q = 3$, $\tau = 160\mu s$, and flowlet gap $T_{fl} = 500\mu s$. We run the enterprise workload described in the same paper, and vary the arrival rate to achieve a target network load, which we measure as a percentage of the total bisection bandwidth.

First, we measure the change in average flow completion time as we increase the load in the network. Figure 3.5a shows that our approximate implementation closely follows the original protocol. We sometimes see a slightly higher FCT primarily because we perform an approximate *minimum* over the ports when we have to assign a new flowlet. Current restrictions on Flex-Switches don't allow us to scan all ports to pick the least loaded one, so we keep a running approximate minimum. This results in some flowlets not getting placed optimally to the least loaded link. In all other cases, we implement CONGA's protocols accurately.



(a) Average FCT with network load.

(b) CDF of FCT with network load at 60%.

Figure 3.5: Performance comparison between CONGA and approximate CONGA using ns3 simulations.

Figure 3.5b shows the CDF of flow completion times for all flows when the network is loaded at 60%. Again, the approximate implementation of CONGA matches very closely with that of the original CONGA protocol. A majority of the flows are short and hence are not affected by the approximate minimum selection of ports.

3.3.3 P4 Compiler-based Resource Usage

To evaluate the resource use of our use cases on an actual FlexSwitch, we implement our use cases in the P4 programming language and compile them to a production switch target. The compiler implements the functionality proposed in [46] and compiles to the hardware model described in Section 2.2. It reports the hardware resource usage for the entire use case, including memory used for data and code.

Our compiler-based evaluation serves to quantify the increased resource usage of our congestion control implementations when added to a baseline switch implementation based upon [84] that provides the common functionality of today's datacenter switches. The baseline switch implementation provides basic L2 switching (flooding, learning, and STP), basic L3 routing (IPv4, IPv6, and VRF), link aggregation groups (LAGs), ECMP routing, VXLAN, NVGRE, Geneve and GRE tunneling, and basic statistics collection. We intentionally do not add more functionality to the baseline to highlight the additional resources consumed by our implementations.

To measure the additional hardware resources used for RCP-Approx, we integrate RCP-Approx into our baseline switch implementation and compile using the compiler described above. Table 3.6 shows the additional hardware resources used compared to the baseline switch. We can see that additional resource use is small—not exceeding 6% for all resources but pipeline stages and requiring an additional stage.

We conclude that RCP can indeed be implemented with adequate accuracy and limited additional resource usage. This gives us confidence that other resource allocation algorithms might be implementable as well and we do so in the following subsections.

Table 3.6 shows the additional hardware resources used by XCP-Approx when integrated into the baseline switch. XCP requires almost twice the computational resources of RCP—both in terms of ALU instructions and state carried among pipeline stages in the packet header vector. This is expected, as XCP-Approx computes 2 counter values for every packet and 3 parameters every control interval, while RCP carries out only 1 computation per interval and 1 per packet arrival. Conversely, SRAM use is diminished versus RCP, as we can carry out multiplication/division

solely via bit-shifts, while we require more TCAM entries to identify when a variable is an approximate power of 2.

Resource	Baseline	+RCP		+XCP		+CONGA	
Pkt Hdr Vector	187	191	+2%	195	+4%	199	+6%
Pipeline Stages	9	10	+11%	9	+0%	11	+22%
Match Crossbar	462	473	+2%	471	+2%	478	+3%
Hash Bits	1050	1115	+6%	1058	+1%	1137	+8%
SRAM	165	175	+6%	172	+4%	213	+29%
TCAM	43	44	+2%	45	+5%	44	+2%
ALU Instruction	83	88	+6%	92	+11%	98	+18%

Table 3.6: Summary of resource usage for various use-cases.

Table 3.6 shows the resources used when our CONGA implementation is added to our baseline switch implementation. We can see that an additional 29% of SRAM to store the additional congestion and flowlet tables is the main contribution. Also, we require 2 extra pipeline stages and 18% more ALU instructions to realize CONGA computationally, for example to approximate multiplication and division. Other resource usage increases minimally, below 10%.

We conclude that even complex load balancing protocols can be implemented on FlexSwitches using our building blocks. The additional resource use is moderate, taking up less than a third of the baseline amount of stages, SRAM bytes, and ALU instructions and less than 10% of the baseline for the other switch resources.

3.4 Summary

Recent hardware switch architectures make it feasible to perform flexible packet processing inside the network. This allows operators to configure switches to parse and process custom packet headers using flexible match+action tables in order to exercise control over how packets are processed and routed. However, flexible switches have limited state, support limited types of operations, and limit per-packet computation in order to be able to operate at line-rate. The work

presented in this chapter addresses these limitations by providing a set of general building blocks that mask these limitations using approximation techniques and thereby enabling the implementation of realistic network protocols. We develop a number of building blocks that are broadly applicable to a wide range of network allocation and management applications. Our evaluations show that these approximations are accurate and that they do not exceed the hardware resource limits associated with these flexible switches.

Chapter 4

APPROXIMATE PROGRAMMABLE SCHEDULING

Many network scheduling algorithms today require a notion of *dynamic priority*, where the priority of individual packets within a flow varies over the lifetime of the flow. These packet priorities generally change with either the number of bytes sent by the flow, how fast the flow is transmitting, or time spent by the packet inside the network. Such scheduling algorithms enable richer application-level prioritization and performance guarantees, such as shortest job next to minimize average flow completion time, earliest deadline first to enable timely delivery of all messages, or fair resource allocation, as illustrated by a long list of proposed scheduling algorithms (e.g., pFabric, PIAS, WFQ, FIFO+, LSTF, and EDF).

Switch-level support for multiple fine-grained priority levels (as in PIFO [85], pHeap [18]) can aid the realization of these scheduling algorithms, but there are still several challenges in faithfully implementing these algorithms efficiently and at line rate. First, implementing strict and fine-grained priority levels is expensive, especially at scale involving high bandwidths and hundreds or thousands of unique flows at multiple terabits per second. Second, and more crucially, existing switch support for priorities does not allow for dynamic changes to the priority of a packet during its stint inside the switch buffer. Fixed packet priorities, therefore, cannot effectively emulate the *ageing* property required by many of the scheduling algorithms, wherein the priority of a packet increases with the time it spends inside a queue.

Consider, for instance, the Least Slack Time First (LSTF) scheduling discipline wherein each packet maintains a delivery deadline, and the switch emits from its buffer the packet with the least slack at a given instant. LSTF cannot be realized using fixed packet priorities that are determined when the packets are inserted into a priority queue. Notably, given a packet that has a deadline of $current_time + slack$, a switch scheduler that maps this deadline to a priority level would quickly

exhaust the priority level space; as long as there are packets buffered inside the switch, packets received with later deadlines would have to be assigned progressively lower priority levels, and the switch will eventually run out of priority levels to use for incoming packets.

In this chapter, we argue that what is needed is a scheduling mechanism that supports both prioritization and implicit escalation of a packet's priority as it spends more time inside the switch buffer. We observe that a mechanism similar to Calendar Queues would be a more appropriate fit for implementing these scheduling algorithms. They allow events (or packets) to be enqueued at a priority level or rank corresponding to a future time, and this rank gradually changes with logical or physical time. A scheduling algorithm simply decides how far in the future a packet must be processed, and periodically increments time forward at some rate. We show that several classical scheduling algorithms can be realized using this paradigm and mapped onto a calendar queue with some approximation that is a function of the number of queues that constitute a calendar queue.

Calendar queues have certain properties that make it amenable for efficient hardware realization, especially on upcoming programmable switch hardware. At any point in time, only one of the queues in a calendar queue is active. Further, a calendar queue imposes a fixed rotation order for activating queues. We describe how the activation of queues in a fixed order can be achieved by periodically modifying queue priorities and active status, either through data-plane primitives expected to be available in future programmable switches or through today's control-plane reconfiguration operations, albeit at a higher latency. When combined with the stateful and customizable packet processing capabilities of a programmable switch (such as Barefoot's Tofino and Cavium's Xpliant), we can customize the calendar queue abstraction to realize a broad class of scheduling algorithms that capture both physical and logical notions of time.

4.1 Background

In this section, we review current start-of-the-art packet scheduling mechanisms available in today's switches, and several recent proposals that attempt to provide more powerful scheduling features.

4.1.1 Current state-of-the-art Scheduling Mechanisms

We briefly describe the architecture of the traffic manager (TM) on merchant-silicon switches (e.g., Barefoot's Tofino and Broadcom's Trident series). The TM is responsible for two tasks: (1) buffering packets when more than one input port is trying to send packets to the same output port simultaneously, and (2) scheduling packets at each output port when the link attached to the port is ready to accept another packet.

Buffering: The TM is organized as a fixed number of first-in, first-out (FIFO) queues per output port. The ingress pipeline of the switch is responsible for determining both the FIFO queue and the output port that the packet should go to. Once the packet exits the ingress pipeline, the TM first checks if there is sufficient space in the packet buffer to admit the new packet. This check can be based on static limits per output queue/port. Alternatively, the check can be based on a dynamically changing limit that allows an output port to take up a larger share of the buffer if others are not utilizing it [28]. In either case, once the packet has been admitted to the buffer, it can not be dropped because dropping a previously enqueued packet requires additional memory accesses to the packet buffer, which is expensive at line-rate. In lossless networks, where PFC is enabled, admission of a packet can also cause the buffer to be filled up by packets from a particular upstream switch or PFC class, resulting in a pause message to the upstream switch.

Scheduling: Packets are eventually dequeued from the buffer. The dequeue process is initiated when the link attached to an output port goes idle and requests a new packet. During dequeue, the TM has to pick a particular FIFO at that output port, remove the earliest packet at that queue, and transmit it. The TM uses a combination of factors to determine which queue to dequeue from. First, each queue has a priority; queues with higher priority are strictly preferred to those with lower priorities. Next, within a priority level, the queues are scheduled in weighted round-round robin order (using an algorithm like DRR [81]). Lastly, each queue can be limited to a maximum rate and is paused and removed from consideration for scheduling if it has exceeded this rate over some time interval. Queues can also be paused due to the receipt of a PFC pause frame from a downstream switch.

To perform buffering and scheduling, the TM maintains a per-queue priority level, a per-queue pause status flag, and counters that track buffer occupancy on a per-input-port, per-PFC-class, per-output-port, and per-output-FIFO basis. These are required both to decide when and whether to admit packets and which queues to schedule. Because the status flag and counters support limited operations (e.g., toggling or increment/decrement), they can be implemented very efficiently in hardware, allowing simultaneous access from multiple ingress and egress switch pipelines in a single clock cycle (unlike state within the pipeline that can only be accessed once per clock cycle). Metadata fields in the packet header allow the traffic manager to determine which input port's buffer occupancy to check for PFC functionality and which output port/FIFO the packet should go to. Metadata fields also allow the traffic manager to write the queue occupancy experienced by the packet into the packet itself (e.g., INT [29]). t is important to note that these hardware mechanisms appear in traffic managers in both fixed-function and programmable switches.

4.1.2 Proposals for Programmable Scheduling

The discussion above focused on a fixed-function TM that supports a small menu of scheduling algorithms (typically priorities, weighted round-robin, and traffic shaping). Recent proposals for programmable scheduling [85, 63, 77, 87, 3, 82] propose additional switch hardware in the TM to make the scheduling decision programmable, assuming the existence of a programmable ingress and egress switch pipeline like RMT. Of the proposals for programmable scheduling, we describe the PIFO work because it targets a switch similar to our work, and it is representative of the hardware considerations associated with programmable scheduling. We defer a detailed comparison of both expressiveness and feasibility with prior work to the related work section.

PIFOs enable programmable scheduling by using a programmable priority queue to express custom scheduling algorithms. Some external computation (either on the end host or a programmable switch's ingress pipeline) sets a rank for the packet. This rank determines the packet's order in the priority queue. By writing different programs to compute different kinds of packet ranks (e.g., deadlines or virtual times), different scheduling algorithms can be expressed using PIFOs.

While PIFOs are flexible, they not only require the development of new hardware blocks to support line rate (as we discuss below) but are also limited given their support for a finite priority range (as we discuss in the next section). The PIFO paper assumes that ranks within flows are naturally in strictly increasing order (i.e., flows are FIFOs), requiring the switch to only find the minimum rank among the head packets across all flows. While this reduces the sorting/ordering requirement of PIFO, sorting the total number of flows in the buffer is still challenging. The PIFO work provides a custom hardware primitive, the flow scheduler, which maintains a sorted array of a few thousand flows and can process tens of flows per output port across about 64 output ports on a single pipeline for an aggregate throughput of 640 Gbit/s. Scaling this primitive to higher speeds and a multi-pipeline switch can be challenging. Thus, a key requirement for any programmable scheduling proposal is that it should be realized by piggybacking on top of existing TM implementations.

4.2 *Packet Scheduling using Calendar Queues*

In this section, we begin by describing the Calendar Queue concept as introduced by Randy Brown in 1988 [22]. We then consider the abstraction of a Programmable Calendar Queue that combines the calendar queue scheduling mechanism with programmable packet processing pipelines. This combination allows for extensibility and customizability, and we show how we can instantiate different variants of Programmable Calendar Queues to emulate different scheduling algorithms that appear in the literature.

4.2.1 Motivating Calendar Queues

Calendar Queues: The Calendar Queue was first introduced for organizing the pending event set in a discrete event simulation. It is a type of priority queue implementation that has low insertion and removal costs for certain priority distributions. Calendar Queues (CQs) are analogous to desk calendars used for storing future events for the next year in an ordered manner. A CQ consists of an array of buckets or queues, each of which stores events for a particular *day* in sorted order. Events can be scheduled for a future date by inserting the event in the bucket

corresponding to the date. At any point, events are dequeued and processed from the current day in sorted order. Once all events are processed from the current day, we stop processing events for the current day and move onto the next day. The emptied bucket is then used to store tasks that need to be performed a year from now.

Drawbacks of existing priority queueing schemes: Prior work has made the observation that, scheduling algorithms make two decisions: in what order packets should be scheduled (in the case of work-conserving algorithms) or when they should be scheduled (in the case of non-work-conserving algorithms). For most scheduling algorithms, these decisions can be made at packet enqueue time. Comparison-based fine-grained priority queueing schemes, such as pHeap or PIFO, can realize some of these algorithms by computing an immutable *rank* for a packet at packet enqueue time and dequeuing packets in increasing rank order. Eiffel further makes the observation that often packet ranks have a specific range, can be expressed as integers, and a large number of packets share the same rank. These characteristics make a bucket-based priority queue an efficient and feasible solution for implementing various scheduling algorithms.

We make the observation that many scheduling algorithms cannot be realized using fine-grained priority queuing schemes if the computed rank is constrained to a finite or bounded range of values. Consider the example of fair queuing (WFQ or STFQ which emulate bit-by-bit round-robin algorithm), where for each arriving packet a finishing round is computed based on the current round number and the finishing round of the previous packet in that packet's flow. Packets are then transmitted in order of increasing finishing round numbers. Further, the algorithm periodically increases the current round number. One could attempt to realize fair queuing using a fine-grained priority queuing scheme by mapping a packet's finishing round number to an immutable rank. Since the ranks of buffered packets cannot be changed, the mapping function needs to be monotonic, i.e., it needs to map higher finishing rounds to higher ranks. The mapping function would then exhaust any finite range of ranks, and the switch would then not be able to attribute a meaningful rank for incoming packets.

Similarly, in the case of the earliest deadline first (EDF), each packet in a flow is associated

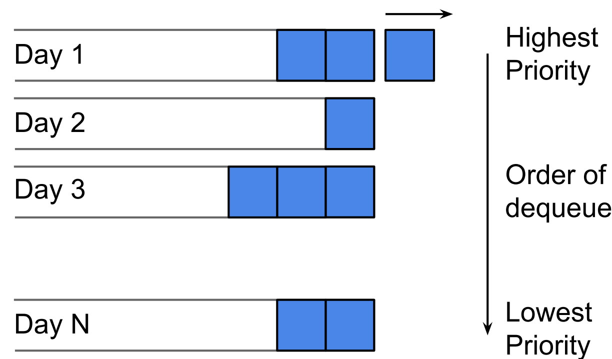


Figure 4.1: Example of a Programmable Calendar Queue.

with a wall-clock deadline, and packets need to be scheduled in increasing order of deadlines. If one were to compute the rank of a packet as a monotonic function of the packet's deadline, the switch would exhaust the rank space as the wall-clock time progresses.

It is worth noting that, when the switch has no buffered packets, the mapping function could execute a "reset" and start reusing lower ranks. However, an implementation cannot assume that the switch would ever enter such a state, let alone periodically (i.e., within a bounded period of time before the rank space is exhausted).

Utility of Calendar Queues: We propose to use the Calendar Queues abstraction as a mechanism to realize scheduling algorithms such as EDF and fair queuing. A CQ is an attractive option for implementing these algorithms as it allows for implicit and en-masse escalation of the priorities of buffered packets when the CQ moves from one day to another. For instance, when a CQ completes processing the events for Day k and *rotation* to Day $k + 1$, it implicitly increases the priority of all days except Day k, which now occupies the lowest priority in the priority range. This rotation mechanism allows scheduling algorithms to escalate the priorities of buffered packets with time (as is the case with EDF and fair queuing) and reuse emptied buckets for incoming packets with low priority.

4.2.2 Programmable Calendar Queues (PCQs)

We now describe Programmable Calendar Queues in the context of reconfigurable switches. The programmable packet processing pipelines on these switches allow us to customize not only the rank computation but also the CQ rotation process. Just like a calendar has 365 days, we assume our Calendar Queue abstraction has a fixed number of buckets or FIFO queues, say N, each of which stores packets scheduled for next N periods. Any network scheduling algorithm using CQs must then make the following key decisions. First, the scheduling algorithm must decide how far in the future the incoming packet should be scheduled, i.e., choose a future period from [0, N-1] to enqueue the packet into. This is similar to rank computation in PIFO. Second, it must periodically decide when and how to advance time, i.e., decide when a period is over and move onto the next period. This stops the enqueueing of packets in the current period and allows the reuse of the corresponding queue resource for the period that is *N* periods into the future. Third, when the CQ advances to the next period, the pipeline state has to be suitably modified to ensure the appropriate computation of ranks for incoming packets.

The advancing of time can be done either using a physical clock, i.e., the CQ moves onto the next queue after a fixed time interval periodically; we call this a *Physical* Calendar Queue. Alternatively, the CQ can advance to the next queue whenever the current queue is empty, i.e., it happens logically depending on metrics such as bytes sent, or sending rate; we call this a *Logical* Calendar Queue. A Physical Calendar Queue lets us implement non-work-conserving schemes, such as Leaky Bucket Filter, Jitter EDD, and Stop-and-Go, whereas a Logical Calendar Queue can implement work-conserving schemes, such as LSTF, WFQ, and EDD.

We now list the interface methods exposed to the packet processing pipelines that enable these forms of customizations.

- **CQ.enqueue(n)**: Used by the ingress pipeline to schedule the current packet *n* periods into the future.

- **CQ.dequeue()**: Used by the egress pipeline to obtain a buffered packet, if any, for the

current period.

- **CQ.rotate()**: Used by the pipelines to advance the CQ so that it can start providing packets for the next period.

We observe that PCQs have certain properties that make them amenable to efficient implementations. (In Section 4.3, we describe how to realize this abstraction in hardware.) When individual CQ periods are mapped to separate queues, a CQ scheduler needs to maintain state only at the granularity of queues (e.g., the queue corresponding to the current period). The scheduler does not require expensive sorting or comparisons to determine packet transmission order. More importantly, a CQ rotation involves a deterministic and predictable transition from one queue to another at the end of each period. This transition can be realized either using data-plane primitives in upcoming reconfigurable hardware (as we discuss in Section 4.3) or through the switch's control plane (as is the case with our prototype).

4.2.3 Implementing Scheduling Algorithms using PCQs

We now show how various scheduling algorithms can be realized using Calendar Queues in conjunction with a programmable packet processing pipeline. We describe three different algorithms, each of which differs in the way it utilizes CQs. First, an approximate variant of WFQ that uses a *Logical* CQ. Next, we implement approximate EDF using LSTF scheduling that uses a work-conserving *Physical* CQ. Finally, we realize a Leaky Bucket Filter that utilizes a non-work-conserving *Physical* CQ.

Weighted Fair Queueing

Weighted Fair Queueing (WFQ) scheduling achieves max-min fair allocation among flows traversing a link by emulating a bit-by-bit round-robin scheme where each active flow transmits a single bit of data each round. This is realized at packet granularity by assigning each incoming packet a departure *round number* based on the current round number and the total bytes sent by the flow. All buffered packets are dequeued in order of increasing departure round numbers.

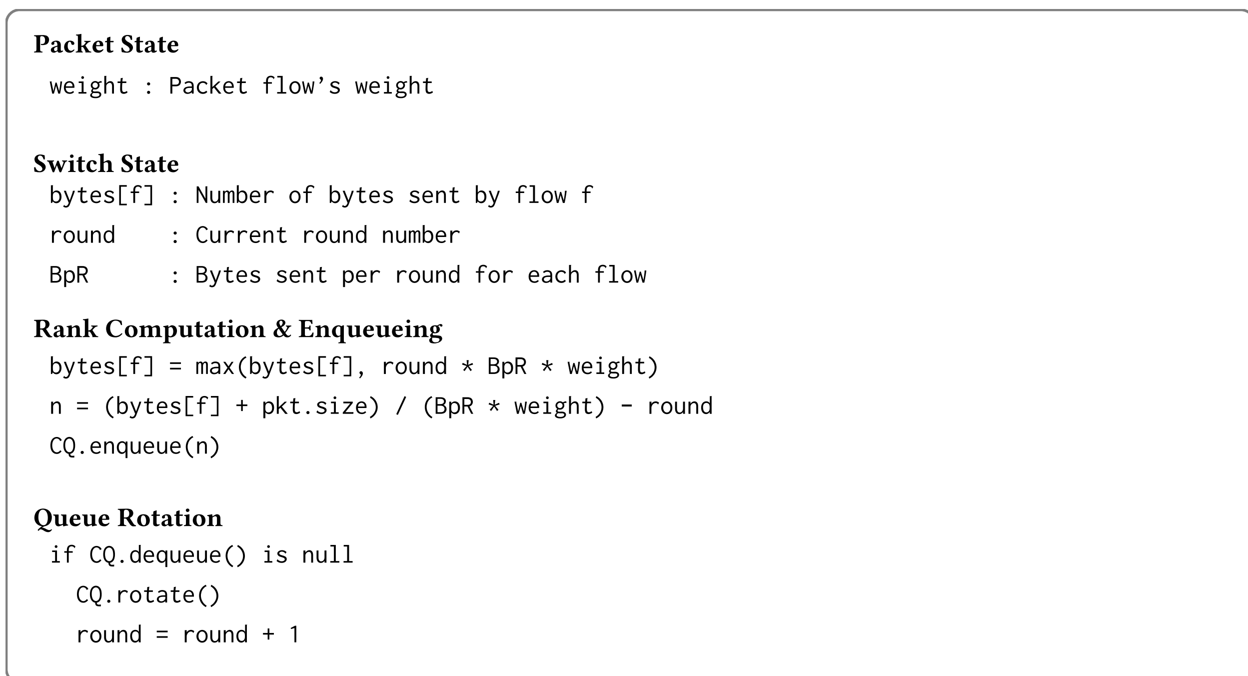
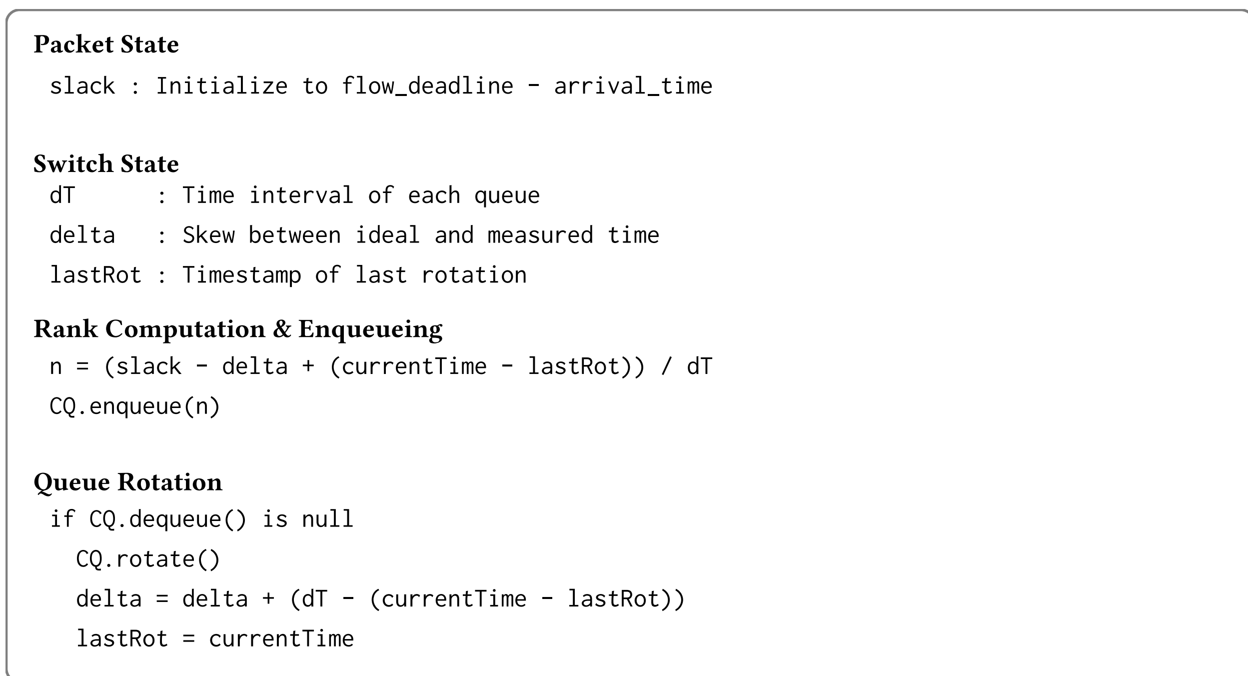


Figure 4.2: WFQ implementation using a Logical CQ.

We implement WFQ using Logical CQs closely following the round number approximation described in [79]. We use coarse-grain rounds that are only incremented after all active flows have transmitted a configurable quantum of bytes. The rank computation is done in such a way that each fair queuing round is mapped to a day (queue) in the Calendar Queue and, whenever a day finishes (i.e., the queue is drained completely), the round number is incremented by one. The complete switch state and computation required is shown in Figure 4.2. Note that this is an approximation of the WFQ algorithm where the round numbers are not as precise or faithful to the original algorithm, and there can be situations in which packets are transmitted in an *unfair* order. However, this *unfairness* has an upper-bound and controlled by the BpR variable in the rank computation.



Packet State

```
slack : Initialize to flow_deadline - arrival_time
```

Switch State

```
dT      : Time interval of each queue
delta   : Skew between ideal and measured time
lastRot : Timestamp of last rotation
```

Rank Computation & Enqueueing

```
n = (slack - delta + (currentTime - lastRot)) / dT
CQ.enqueue(n)
```

Queue Rotation

```
if CQ.dequeue() is null
  CQ.rotate()
  delta = delta + (dT - (currentTime - lastRot))
  lastRot = currentTime
```

Figure 4.3: EDF using a work-conserving Physical CQ.

Earliest Deadline First

In Earliest Deadline First (EDF) scheduling, each packet from a flow is assigned a deadline or expected time of reception. At each network hop, the packet with the closest deadline is transmitted first. We implement EDF using Least Slack Time First scheduling, where each packet carries a *slack* value that is a measure of the time remaining till its deadline. The slack is initialized to `deadline-arrivalTime` at the source and updated at each hop along the way (i.e., each switch subtracts the time spent at the hop from the slack). The implementation uses a Physical CQ as shown in Figure 4.3, which we describe next.

We choose a fixed time interval for each *day* or queue for our Physical CQ, say dT. Packets with an effective slack of $0 - dT$ are assigned to queue 1, slack of $dT - 2 \cdot dT$ are assigned to queue 2, and so on. This assignment ensures that packets with closer deadlines are prioritized. Queue rotation occurs when the current queue becomes empty. Since we can spend a longer or shorter time than dT in any queue depending on the traffic pattern, we require some extra state to

ensure that new packets are inserted in the correct queue with respect to the deadlines of already enqueued packets. The `delta` variable keeps track of how far ahead the CQ is compared to the ideal time. If we spend less than dT for a queue, $delta$ increases and if we spend more than dT, it decreases. The $delta$ is then incorporated in the rank computation and is reset to 0 whenever there are zero buffered packets.

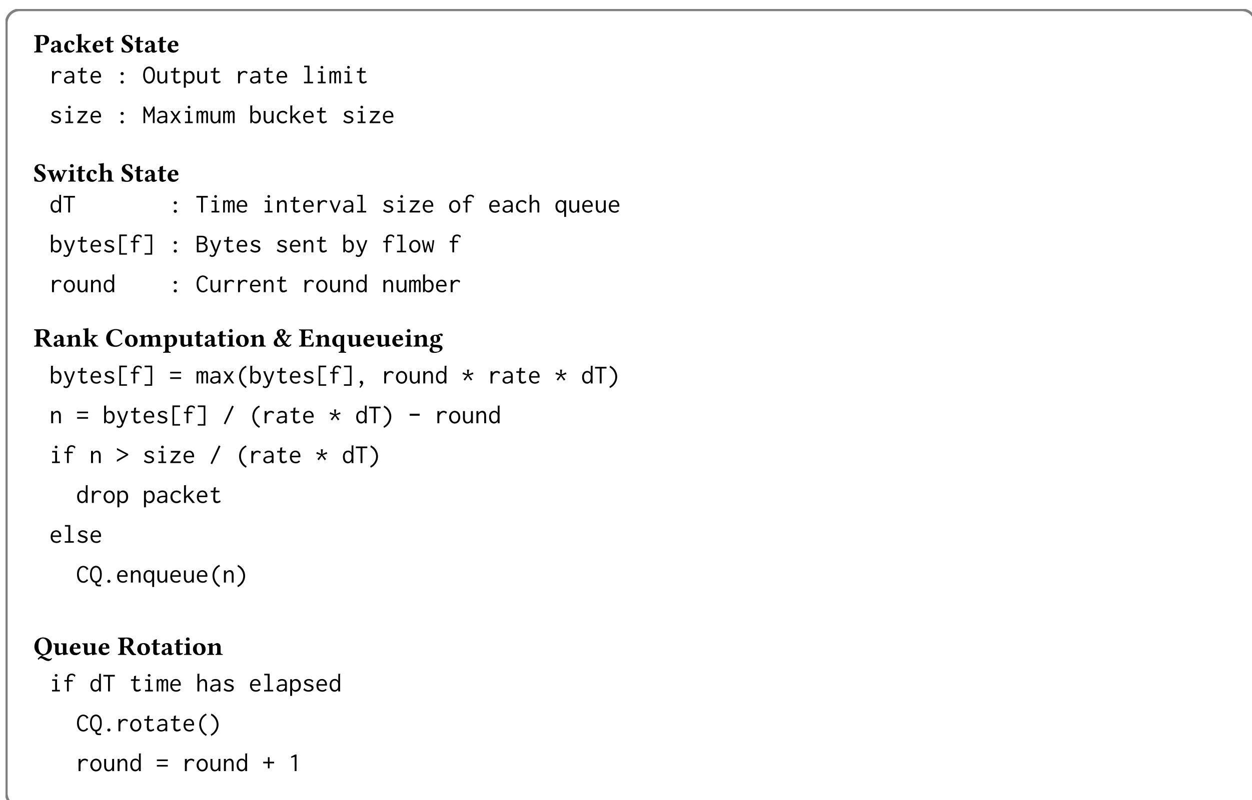
Leaky Bucket Filter

A Leaky Bucket Filter (LBF) is a non-work-conserving scheduling algorithm that rate limits a flow to a specified bandwidth and a maximum backlog buffer size. An LBF can be realized using a Physical Calendar Queue by storing a fixed quantum of bytes per flow in each queue and rotating queues at fixed time intervals, very similar to the WFQ example discussed earlier. However, we do not dequeue packets from the next queue even if the current queue is empty – which gives us the desired non-work-conserving behavior. The byte quantum depends on the rate limit set for the flow and the configured time interval dT of each queue. If the number of enqueued bytes for a flow exceeds the bucket size, we simply drop the packet. This scheme lets us realize multiple filters using the same underlying CQ, as shown in Figure 4.4.

We assume the configured `rate` and `size` parameters for the filter are in the packet header, but they can be stored at the switch as well. For each flow, we keep track of bytes sent so far and compute the queue id by dividing it with the quantum, which is the configured filter rate times the queue interval dT. We assume that the cumulative rate of all flows does not exceed the line rate at the switch; if that happens all flows will be slowed down in proportion to their configured rates equally.

4.3 Implementing PCQs in Hardware

We now describe how Calendar Queues can be implemented on programmable switches. We assume an RMT model switch (as described in Section 2) with an ingress pipeline, followed by the traffic manager, which maintains multiple packet queues, and finally an egress pipeline. Implementing a CQ in this model is non-trivial because the packet enqueue decision (i.e., which



Packet State
```
rate : Output rate limit
size : Maximum bucket size
```
Switch State
```
dT       : Time interval size of each queue
bytes[f] : Bytes sent by flow f
round    : Current round number
```
Rank Computation & Enqueueing
```
bytes[f] = max(bytes[f], round * rate * dT)
n = bytes[f] / (rate * dT) - round
if n > size / (rate * dT)
  drop packet
else
  CQ.enqueue(n)
```
Queue Rotation
```
if dT time has elapsed
  CQ.rotate()
  round = round + 1
```

Figure 4.4: A Leaky Bucket Filter using a non-work-conserving Physical CQ.

queue to insert the packet into) is made in the ingress pipeline, but the queue status (i.e., occupancy, depth) is available in the egress pipeline after the packet traverses the traffic manager. Since these modules are implemented as separate hardware blocks, we need to *synchronize* state among them to achieve the CQ abstraction.

We can realize CQs on programmable switches using mutable switch state, multiple FIFO queues, the ability to create and recirculate special packets, and a unique Barefoot feature, expected to be available in upcoming versions of Tofino, that allows data-plane packets to pause/resume queues and alter queue priorities. This new feature—relatively easy to bolt-on—is the addition of a metadata field containing commands to instruct the TM to change the priorities or toggle the active status of queues [15]. This feature permits changes of queue metadata in the data plane,

which is currently possible only through the control plane.[1] In the absence of data plane support for priority changes, we can still approximate this functionality using the control-plane (as we do in our testbed).

Implementation Overview: We first provide a high-level description of our scheme. Each *period* in the CQ is mapped to a single FIFO queue within a set of queues associated with the outgoing port. The ingress pipeline computes which *period* or queue each incoming packet is enqueued into. We assume a TM that allows the ingress pipeline to enqueue incoming packets into any of the available FIFO queues. At any given time, the queue settings satisfy the following properties: (a) The queue corresponding to the current period has the highest priority level, so that its packets can be transmitted immediately. We refer to this queue as the head queue. (b) The queue corresponding to the next period has a lower priority level and is active/unpaused. (c) All other queues corresponding to future periods are at the lowest priority level and are paused. This specific configuration is maintained so that each CQ rotation can be handled with a small number of changes to queue priorities and active status.

The CQ cycles or rotates through available queues one at a time, making each queue the head queue. The rotation is triggered when the head queue is empty (in case of a logical CQ) or the CQ time interval has elapsed (in case of a physical CQ). When a rotation happens, we need to make sure all packets from the head queue are drained completely and that it is empty before changing its priority to lowest. Once the priority is set, the head queue can be reused to store packets for future periods.

Implementation Details: We break down the implementation of CQs into the following three steps.

Step 1 - Initiate Rotation: This step detects when a rotation needs to happen and initiates the rotation by informing the ingress pipeline using a recirculation packet. In the case of a logical

---

[1] Note that PFC already require the switch to toggle queue status based on protocol messages. This new feature exposes a similar functionality in a programmatic way.

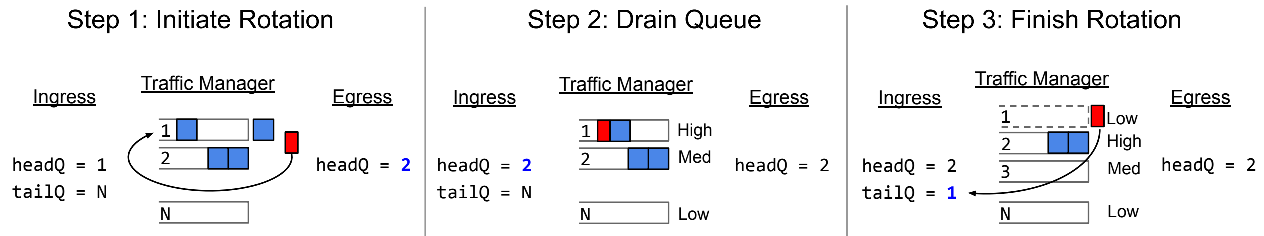


Figure 4.5: Step 1, the egress pipeline tries to dequeue from the current headQ and if it's the last packet, it initiates rotation by creating and recirculating a `rotate` (red packet) to the ingress pipeline. Step 2, the ingress pipeline on receiving this packet, updates the headQ and enqueues a `marker` packet as the last packet into the old headQ. Finally in Step 3, when the egress pipeline receives the `marker` packet, it updates the queue priority and notifies the ingress pipeline that it is safe to re-use the queue for future packets by updating the tailQ.

CQ, we initiate rotation when the head queue is empty. This can be detected in two ways. First, by checking the traffic manager metadata available at the egress pipeline, which tells the queue depth from the which the packet was dequeued. If it is zero, this is the last packet from the head queue, and we initiate rotation. Second, we can check the queue id from which the packet was dequeued to infer whether the head queue is empty. Since the successor head queue is also unpaused with a lower priority, a packet dequeued from it implies the head queue is empty. In the case of a physical CQ, the rotation happens at fixed time intervals and is configured through timers or packet generators. When a rotation begins, we recirculate a special `rotate` packet to the ingress pipeline so that it stops enqueuing packets in the head queue and begins draining it, which happens in Step 2.

Step 2 - Drain Queue: This step ensures that the head queue is completely drained and no more packets are enqueued into it till the rotation finishes. On receiving the `rotate` recirculation packet, the ingress pipeline advances the head queue pointer, essentially stopping any new packets from being enqueued into it. But, there could still be some packets in the pipeline currently making their way into the head queue. To make sure these are transmitted in the right order, the ingress enqueues a special `marker` packet into the head queue after updating the head of the calendar queue. This packet is the last packet to be enqueued into the head queue, and its arrival at the egress pipeline means the queue is completely drained, and we can proceed with finishing the

rotation described in Step 3.

Step 3 - Finish Rotation: The `marker` packet is recirculated back to the ingress pipeline, and this informs the ingress pipeline that it is safe to reuse the queue for future periods. The ingress changes the priority of the just emptied queue to lowest and also pauses it, essentially pushing the queue to the end of the CQ. The ingress also unpauses the queue associated with the next period to ensure that there are no transmission stalls after the current period ends. The queue configuration change can be achieved in two ways depending on the underlying hardware support. The `marker` packet can be pushed up to the control plane CPU, which can alter the queue configurations using traffic manager APIs. This approach incurs a latency overhead before the drained queue can be used for packets associated with future periods. Alternatively, if the hardware supports priority change in the datapath, the processing of the `marker` packet with the appropriate metadata tags affects the configuration change almost immediately.

We now briefly highlight some of the attributes of PCQ that aid in efficient hardware realization. First, CQs maintain state at the granularity of queues instead of individual packets or flows. Second, at any given point in time, there is a designated head queue that is responsible for providing the packets that are to be transmitted. Third, the rotation operation involves changing just the metadata of queue and that too of at most three queues. These combinations of factors allow us to bolt-on the PCQ abstraction on to a traditional TM.

4.3.1 Alternate ways of emulating PCQs

Using a Generic DRR Scheduler. A Deficit Round Robin (DRR) [81] is a scheduling algorithm that guarantees isolation and fairness across all queues serviced. It proceeds in rounds; in each round, it scans all non-empty queues in sequence and transmits up to a configurable quantum of bytes from each queue. Any deficit carries over to the next round unless the queue is empty, in which case the deficit is set to zero. We note that a Logical Calendar Queue is simply a version of DRR with the quantum set to a large value that is an upper-bound on the number of bytes transmitted in a single round. With a very high quantum, a queue serviced in DRR is never serviced again until all other queues have been serviced. This is equivalent to demoting the

currently serviced queue to the lowest priority.

Crucially, the DRR emulation approach indicates that the hardware costs of realizing a Calendar Queue should be minimal since we can emulate its functionality using a mechanism that has been implemented in switches today. However, we note that many modern switches implement a more advanced version of DRR, called Shaped DWRR, a variant that performs round-robin scheduling of packets from queues with non-zero deficit counters in order to avoid long delays and unfairness. Unfortunately, the Calendar Queue mechanism cannot be emulated directly using DWRR due to its use of round-robin scheduling across active queues.

Using Switch CPU and Inbuilt Priorities. We now consider another emulation strategy that uses periodic involvement of the switch-local control plane CPU to alter the priority levels of the available egress queues. When the priority level for a queue is changed, typically through a PCIe write operation, the switch hardware instantaneously uses the queue's new priority level to determine packet schedules. The challenge here is that the switch CPU cannot make repeated updates to the priority levels given its clock speed and the PCIe throughput. We, therefore, designed a mechanism that requires less frequent updates to the priority levels (e.g., two PCIe operations every 10us) using hierarchical schedulers.

Our emulation approach splits the FIFO queues into two strict priority groups and defines hierarchical priority over the two groups. All priority level updates are made by switching the upper-level priority of the two sets of queues; these updates are made only after the system processes a certain number of rounds. Suppose we have $2 \times n$ queues split into two groups (G^1, G^2) of n queues each. In each group, all n queues are serviced using strict priority. Initially, G^1 has strict priority over G^2. Packets with round number $1 \longrightarrow n$ are enqueued in $G^1_{1\rightarrow n}$, whereas packets with round $(n+1) \longrightarrow 2n$ are enqueued in $G^2_{1\rightarrow n}$. Packets with a round number greater than that are dropped. After a period τ, or when all queues in G^1 are empty, we switch the priorities of G^1 and G^2, making all queues of G^2 higher priority than G^1. Queues in each group retain their strict priority ordering. After the switch, we allow packets to be enqueued on G^1's queues for rounds corresponding to $(2n + 1) \longrightarrow 3n$. This approach is feasible using hierarchical schedulers

available in most ToR switches today. It reduces the number of priority transitions the switch must make and is implementable with the help of the management/service CPU on the switch. The time period τ depends on the link-rate and number of queues. Our experiments with the Cavium Xpliant switch indicate that $\tau = 10\mu s$ is both sufficient and supportable using the switch CPU. The disadvantage of this emulation approach is that the number of active queues the system can use could drop from $2n$ to n at certain points in time.

4.4 Analysis and Extensions

We now analyze our PCQ abstraction and compare it to both fine-grained priority queuing schemes and an ideal Calendar Queue along different dimensions. We also provide an extension that expands the scheduling capability of the PCQ.

Expressiveness: From a theoretical perspective, a static priority mechanism (e.g., PIFO) with infinite priority levels is equivalent to a Calendar Queue with infinite buckets, and most scheduling algorithms can be expressed in both these hypothetical schemes. However, a practical PIFO has finite priority levels and a practical CQ has finite buckets, which affects the feasibility and the fidelity of scheduling algorithms. An algorithm can be implemented using PIFOs if all packets throughout the "lifetime" of the algorithm have ranks strictly in the priority queue range. This is true for algorithms like pFabric where packet rank is solely a function of flow size, but not true for WFQ or EDF where packet ranks are computed based on a round number or current time, which is monotonically increasing. As discussed earlier, the priority-level space will roll over eventually, and the ordering of enqueued packets will be violated. On the other hand, our realization of PCQs requires that the enqueued packets' ranks at any instant fit within the available buckets or queues, which makes it challenging to implement algorithms that require both a large packet rank range as well as high fidelity in distinguishing between the packet ranks; we can extend the range of a CQ by bucketing several packet ranks together, but that introduces approximations which we discuss next.

Approximations: There are two sources of approximations that arise in a PCQ. First, inversions within a FIFO queue. The original CQ maintains events in a single bucket in sorted order, whereas we simply use a FIFO queue. This can lead to inversions if multiple ranks are assigned to the same bucket, i.e., a higher priority packet is scheduled after a lower priority packet. This presents a feasibility vs. accuracy trade-off for the scheduling algorithm, and if the bucket intervals are chosen carefully, the approximation is acceptable as we show later in the evaluation. It is worth noting that one could borrow some of the mechanisms from the SP-PIFO work [3] to reduce rank inversions, but we leave that to future work. Second, the PCQ imposes a limit on the range of the CQ. Since the number of FIFO queues is limited, there is a possibility that packets will arrive with a rank beyond the range of the CQ. One can theoretically increase the bucket size to include a larger priority schedule such packets that are very far in the future. However, this will lead to an increase in inversions and reduce the accuracy of the priority queuing mechanism. Another option is to store overflowing packets into a separate queue and recirculate them into their appropriate queue when they get close to their service time. In fact, the range of a CQ can be significantly increased by employing a hierarchical structure, and we describe this next.

Implementation Complexity: Priority queues are a fundamental data-structure for implementing various scheduling algorithms. PIFO and pHeap provide a fine-grained comparison-based priority queuing mechanism, which requires O(logN) operations where N is the number of packets or flows. This makes PIFOs scale to much lower aggregate link speed (i.e., single pipeline switches) and number of flows (O(1000) active flows at any point for the entire switch), compared to programmable calendar queues which can scale to an arbitrary number of flows and pipelines, because CQs maintain state at the granularity of number of queues instead of individual packets or flows. The calendar queues also have a much better incremental pathway to deployment as we later show in the implementation section.

Hierarchical Calendar Queues: One way to extend the range of a CQ is to employ a hierarchical structure among the available FIFO queues, similar to hierarchical timing wheels, at the

cost of recirculating some data packets. To create a 2-level hierarchical calendar queue (HCQ), we split the N FIFO queues into two groups of sizes n_1 and n_2 respectively. The two groups run independent calendar queues CQ1 and CQ2 on top of them, however with different bucket intervals: CQ1 having an interval of one time period and CQ2 having an interval of n_1 time periods. The idea is that a single queue of CQ2 has an interval equivalent to the full rotation of all n_1 queues of CQ1. A packet with a scheduled time between 1 to n_1, is inserted into the appropriate queue in CQ1, packets with time between $n_1 + 1$ to $2 \times n_1$ are inserted into the first queue of CQ2, packets with $2 \times n_1 + 1$ to $3 \times n_1$ are inserted into the second queue of CQ2, and so on. This approach provides a total range of $n_1 \times n_2$ time periods, whereas just using a single CQ over N queues would give a range of just $n_1 + n_2$.

However, this comes at the cost of recirculating any data packet that is enqueued in CQ2. When a full rotation of all n_1 queues in CQ1 finishes, all packets from the head queue of CQ2 are recirculated and deposited into appropriate queues in CQ1. Note that this approach is still significantly better than the approach described in [22], where packets scheduled too far in the future are simply enqueued in the scheduled queue modulo N, and recirculated if the scheduled time has not arrived; Brown's scheme can recirculate a packet multiple times, whereas an HCQ recirculates a packet only once leading to more efficient use of bandwidth. Implementing HCQs also requires storing and managing extra state for both CQs and more complex computations when determining the destination queue for a packet. With 32 FIFO queues, a 2-level HCQ can be implemented with 16×16 queues to achieve a reach of 256 time periods or a 3-level HCQ with $16 \times 8 \times 8$ queues with a total reach of 1024, which is significantly larger than 32.

4.5 Evaluation

4.5.1 Hardware Prototype Implementation

We implement and evaluate our programmable Calendar Queues on the Barefoot Tofino 100BF-32X switch. As the current Tofino switch does not support updating a queue's priority on the datapath, we implement Calendar Queues using a combination of in-built packet generator, packet

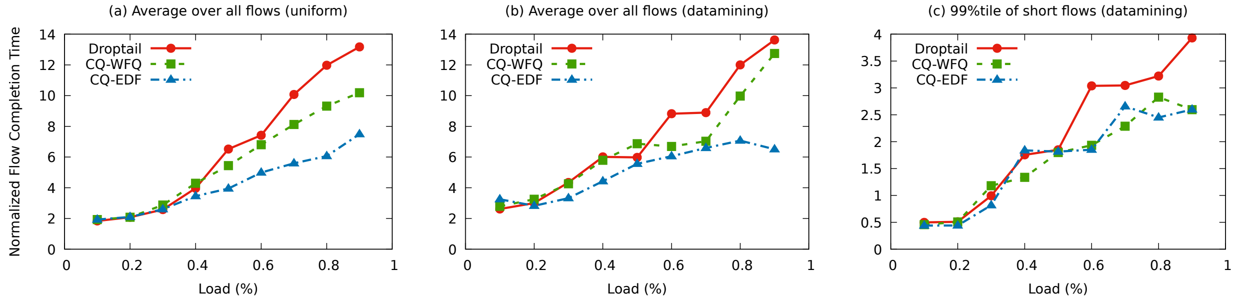


Figure 4.6: Average and tail latencies from the hardware testbed running a synthetic and datamining workload.

re-circulation, and control plane operations to drain packets in the correct order.

First, we use the in-built packet generator on the switch to periodically generate probe packets and detect when a queue rotation needs to be performed. The egress pipeline tracks the current head of the CQ, and when a packet is dequeued from the next queue, it re-circulates the probe packet back to ingress to initiate a rotation. Next, the ingress pipeline updates the current head of the CQ and enqueues the probe packet into the queue being rotated to drain it out completely. When the egress pipeline receives the probe packet again, it is safe to update the priority of the drained queue, and we achieve this by setting a flag in the egress pipeline. Finally, the control CPU polls on this flag variable, and when set, it makes an API call to update the queue's priority and notifies the ingress pipeline that it is safe to use this queue to store future packets.

Testbed and Workload We implement two scheduling algorithms, WFQ and EDF using Calendar Queues and compare them against standard FIFO drop-tail scheduling in a 2-level fat-tree topology, consisting of 2 ToR switches, 2 aggregation switches, and 4 servers by using VLANs to divide the 32 physical ports into multiple switches. All links in the network are 40Gbps with 80µs end-to-end latency. Each server opens 80 concurrent long-running connections to other servers, and requests flows according to a Poisson process at a rate configured to achieve the desired network load. We tested a synthetic workload that draws flow sizes at uniform with a max size of 12.5MB and the data-mining workload from [9].

Performance Figure 4.6 shows the average and 99th percentile latencies. Across both workloads, we can see the WFQ and EDF implementations performing better than simple drop-tail queues. The difference is more significant at higher network load when queues build up at the switch due to bursty arrivals. This is when the prioritization and correct scheduling of packets leads to a visible difference in FCT.

4.5.2 *Coflow Scheduling using EDF Scheduling*

Several distributed applications running inside datacenters rely on network traffic patterns that require optimizing the performance on a collection of flows rather than individual flows, e.g., partition-aggregate or bulk synchronous programming tasks such as multi-get Memcached queries and MapReduce jobs. The performance of such applications depends on the last finishing flow among the collection and prior work [1] has shown that near-optimal performance can be achieved by ordering coflows using a Shortest Remaining Processing Time (SRPT) first mechanism and ensuring that any packet from any flow of a coflow X ordered before coflow Y is transmitted before any packet from coflow Y.

We implement the above approach using LSTF scheduling on top of Calendar Queues to optimize coflow completion times (CCT). However, instead of using the complex BSSI algorithm in [1] which decides priorities based on other coflows in the system, we choose a much simpler heuristic to order coflows as they arrive. We compute a deadline for the whole coflow, assuming the largest sub-flow in the coflow is the bottleneck and will be the last to finish. Therefore, the deadline is simply the largest sub-flow size divided by end-host link speed. All we need to do is assign this deadline to all packets of all flows in the coflow and ensure that packets with the earliest deadline are transmitted at any switch.

We calculate each packet's slack as the time remaining until the deadline of its corresponding coflow. The slack is initialized in the packet header at the end-host, and as the packet traverses the network, each switch enqueues the packet in the Calendar Queue based on this slack value. The higher the slack value, the farther in future the packet is scheduled for transmission. On departure, the switch deducts the time spent at the switch from the slack and updates the packet

header. As a result, critical flows with lower slack values and closer deadlines are dynamically prioritized over non-critical flows with larger slack values and farther deadlines. Note that, this scheme could have been implemented using an exact priority queue (such as PIFO) with absolute deadlines embedded in the packet header, but that would require clocks to be synchronized. More importantly, the switch would run out of priority level eventually and would not be able to enforce deadlines. We, therefore, implement the scheme using LSTF on top of Calendar Queues.

We measure the performance of the above coflow scheduling mechanism using event-driven simulations and compare it with the following queueing schemes:

- **Drop-tail**: Traditional switch with a single FIFO queue that drops packets from the tail when full.
- **Fair Queue**: Bit-by-bit round-robin algorithm from [32] that achieves max-min fairness.
- **Ideal Calendar Queue**: A CQ with *infinite* buckets that also transmits packets in sorted order within each bucket.
- **Approx Calendar Queue**: Our implementation of CQs that uses 32 FIFO queues and 10µs round interval.

In all the above schemes, we use the same end-host flow control protocol, DCTCP, with the additional embedding of *slack* value based on the coflow deadline.

Workload and Performance Metric We use a mix of background traffic, which is the enterprise workload in [5], and a synthetic coflow workload derived from a Facebook trace used in [1]. Coflows on average have ten sub-flows and a total size of 100KB. The ratio of background traffic to coflow traffic is 3:1. Flows and coflows arrive according to a Poisson process at randomly and independently chosen source-destination server pairs with an arrival rate chosen to achieve the desired level of utilization in the aggregation-core switch links. We evaluate the performance in terms of flow completion time (FCT) or in case of coflows, the coflow completion time (CCT)

which is the maximum FCT of comprising sub-flows. We report both average and 99 percentile latencies.

Since we have a mix of background flows and coflows, we must decide how they co-exist together and are scheduled inside the network. One trivial way is to treat background traffic as a separate class with a lower priority than coflow traffic. We ran and measured this configuration in Setup 1. Another option made possible by CQs is to treat background traffic as fair-queued and coflow traffic as deadline-aware using the same underlying CQ to schedule packets belonging to both classes. This demonstrates the flexibility of CQs in realizing multiple scheduling policies at once, and we study this configuration in Setup 2.

Setup 1: Coflows have priority over background traffic. We treat background flows as a different traffic class with lower priority than coflows by using a separate queue with strictly lower priority. As a result, the background traffic performance is not affected by the coflow scheduling policy, and we report coflow completion times in Figure 4.7. The average CCT improves by up to 3x and 99th percentile CCT by 5x at high network loads. This improvement is both because we are emulating an SRPT policy as well as we de-prioritize shorter sub-flows within a coflow over other more critical flows. Both drop-tail and fair-queuing finish short flows within a coflow quickly although they have a significant *slack* till the deadline. Moreover, our CQ implementation is able to accurately emulate an ideal calendar queue mechanism using a limited number of FIFO queues, and the gap between ideal CQ and our CQ is fairly negligible.

Setup 2: Both coflows and background traffic scheduled on the same CQ. We use the same underlying Calendar Queue to schedule background flows as fair-queued traffic and coflows as deadline or slack based traffic. For a background flow packet, which is fair-queued, we compute its departure queue based on bytes enqueued by the flow using the WFQ implementation described in Section 4.2.3. For a deadline-aware coflow packet, we calculate its departure queue using the slack inside the packet header by dividing it with the configured bucket interval as described in Section 4.2.3. We can control the relative priority of background vs. coflow traffic by

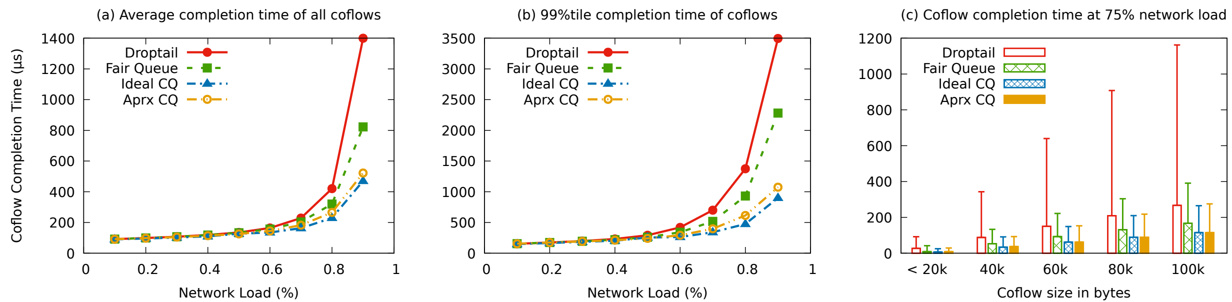


Figure 4.7: Coflow completion time when running a mix of background and coflow traffic with coflows prioritized over background flows. (a) average CCT for all coflows, (b) 99th percentile CCT for all coflows, and (c) average and 99th percentile (using error bar) for various coflow size buckets at 75% network load.

changing the bucket interval. A higher bucket interval will accommodate more bytes from deadline traffic compared to fair-queued traffic. We use a default value of 10μs as the bucket interval and 1 MSS as the bytes quantum per round for fair queueing.

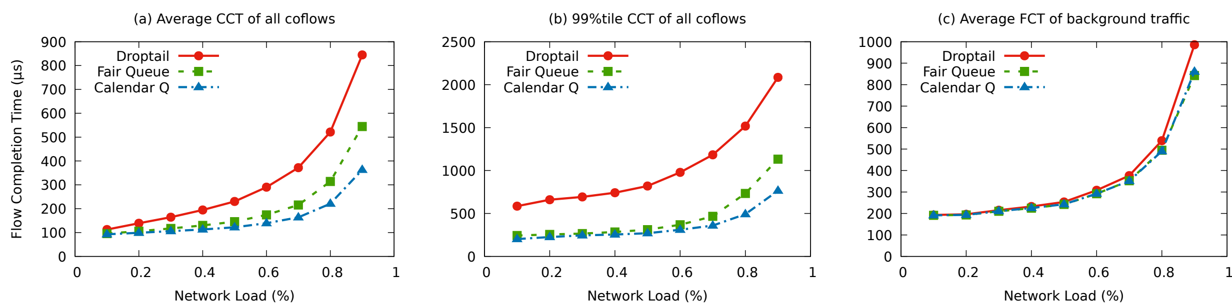


Figure 4.8: CCT and FCT when scheduling background traffic as fair-queued and coflow traffic as deadline-aware using the same Calendar Queue.

Figure 4.8 shows the coflow completion times and FCT of background flows in this setup. The average CCT shows up to 2.5x and 1.5x improvement compared to drop-tail and fair queuing, respectively. This benefit again comes from the fact that we are able to schedule shorter coflows before longer coflows, as well as de-prioritize shorter sub-flows within a coflow over other more critical sub-flows. The 99th percentile shows a similar improvement of 3x over drop-tail queues. Moreover, the average FCT of background flows stays roughly the same and is unaffected by the

coflow scheduling being done by the Calendar Queue.

4.6 Summary

Scheduling is often realized through queue-level priorities, as in today's switches, or through fine-grained packet-level priorities, as in recent proposals such as PIFO. Unfortunately, fixed packet priorities determined when a packet is received by the traffic manager is not sufficient to support a broad class of scheduling algorithms that require the priorities of packets to change as a function of the time it has spent inside the network. In this paper, we revisit the Calendar Queue abstraction and show that it is an appropriate fit for scheduling algorithms that not only require prioritization but also perform dynamic escalation of packet priorities. We show that the calendar queue abstraction can be realized using either data-plane primitives or control-plane reconfigurations that dynamically modifying the scheduling status of queues. Further, when paired with programmable switch pipelines, we can realize extensible and customizable calendar queues that can emulate a diverse set of scheduling policies.

Chapter 5

CASE STUDY: WEIGHTED FAIR QUEUEING

In this Chapter, we take a classical networking problem – *Fair Queueing*, that has proved challenging to implement in high speed switches and show that by using techniques described earlier in the thesis we can realize it on today's programmable high-speed switches.

The idea of enforcing fair bandwidth allocation inside the network has been well studied and shown to offer several desirable properties. A straight-forward way of achieving such allocation is to have per-flow queues, as proposed by Nagle [66], serviced in a round robin manner. This is clearly impractical given today's network speeds and workload complexities. An efficient algorithm, called *bit-by-bit round-robin (BR)*, proposed in [32], achieves ideal fair queueing behavior without requiring expensive per-flow queues. We describe this approach next since it forms the basis of our Approximate Fair Queueing (AFQ) mechanism using Calendar Queues.

5.1 Background: Bit-by-Bit Round-Robin Algorithm

The *bit-by-bit round-robin* algorithm achieves per-flow fair queueing using a round-robin scheme wherein each active flow transmits a single bit of data every round. Then, the round ends, and the round number is incremented by one. Since it is impractical to build such a system, the BR algorithm "simulates" this scheme at packet granularity using the following steps.

- For every packet, the switch computes a *bid number* that estimates the time (round) when the packet would have departed.
- All packets are then buffered in a *sorted priority queue* based on their bid numbers, which allows dequeuing and transmission of the packet with the lowest bid number at any time.

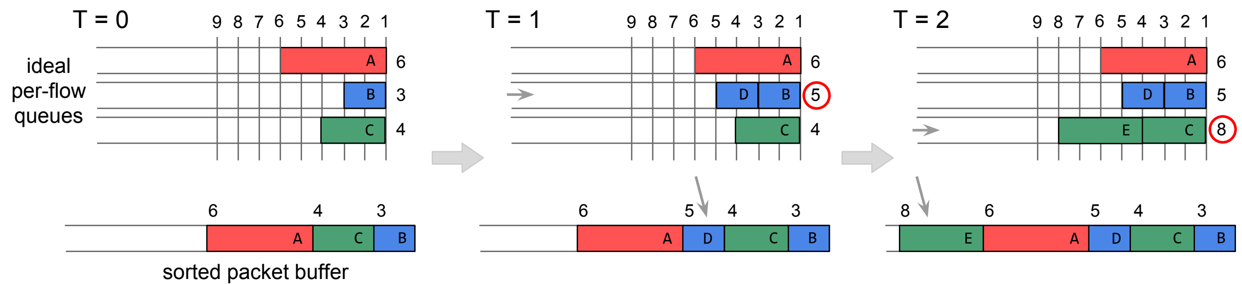


Figure 5.1: An example of the bit-by-bit round-robin Fair Queueing algorithm. The algorithm buffers all packets in sorted order based on their departure round. When a blue packet D of size 2 arrives at T = 1, its departure round is calculated as 5 and is placed between packets A and C in the sorted buffer. Similarly, when a green packet of size 4 arrives at T = 2, its departure round is 8, and it is placed at the end of the departure queue.

Figure 5.1 shows a simple example of this approach. Although the BR algorithm achieves ideal fair queuing behavior, several factors make it challenging to implement given today's line-rate, 3-6 Tbps switches. First, to compute bid numbers for each packet, the switch must maintain the *finish round number* for each active flow. This is equal to the round when the flow's last byte will be transmitted and must be updated after each packet's arrival. Today's switches carry hundreds to thousands of concurrent flows [17, 76]. Their limited amount of stateful memory makes it difficult to store and update per-flow bid numbers.

Second, inserting packets into an ordered queue is an expensive $O(logN)$ operation, where N is the maximum buffer size in number of packets. Given the 12-20MB packet buffers available in today's switches, this operation is challenging to implement at a line-rate of billions of packets per second. Finally, switches need to store and update the current round number periodically using non-trivial computation involving: (1) time elapsed since last update, (2) number of active flows, and (3) link speed, as described in [51]. Today's line-rate switches lack the capability to perform such complex computations on a per-packet basis.

```
/* AFQ parameters */

S[][] : sketch for bid numbers
nH    : # of hashes in sketch
nB    : # of buckets in sketch
nQ    : # of FIFO queues
BpR   : bytes sent in each round

/* Count-min sketch functions */
func read_sketch(pkt):
   val = INT_MAX
   for i = 1 to nH:
      h = hash_i(pkt) % nB
      val = min(S[i][h], val)
   return val

func update_sketch(pkt, val):
   for i = 1 to nH:
      h = hash_i(pkt) % nB
      S[i][h] = max(S[i][h], val)
```

```
/* Enqueue Module */

R : Current round (shared w/ dequeue)

On packet arrival:
   bid = read_sketch(pkt)

   // If flow hasn't sent in a while,
   // bump it's round to current round.
   bid = max(bid, R * BpR)

   bid = bid + pkt.size
   pkt_round = bid / BpR

   // If round too far ahead, drop pkt.
   if (pkt_round - R) > nQ:
      drop(pkt)
   else:
      enqueue(pkt_round % nQ, pkt)
      update_sketch(pkt, bid)
```

```
/* Dequeue Module */

R : Current round number (shared)
i : Current queue being serviced

while True:
   // If no packets to send, spin.
   if buffer.empty()
      continue;

   // Drain i'th queue till empty.
   while !queue[i].empty():
      pkt = dequeue(i)
      send(pkt)

   // Move onto next queue,
   // increment round number.
   i = (i + 1) % nQ
   R = R + 1
```

Figure 5.2: Pseudocode for AFQ

5.2 FlexSwitch Implementation

Our design emulates the ideal BR algorithm described earlier. Like that algorithm, AFQ proceeds in a round-robin manner, where every flow transmits a fixed number of bytes in each round. On arrival, each packet is assigned a departure round number based on how many bytes the flow has sent in the past, and packets are scheduled to be transmitted in increasing round numbers. Implementing this scheme requires AFQ to store the finish round number for every active flow at the switch and schedule buffered packets in sorted order. It must also store and update the current round number periodically at the switch.

We approximate fair queueing using three key ideas. First, we store approximate flow bid numbers in sub-linear space using a variant of the count-min sketch, letting AFQ maintain state for a large number of flows with limited switch memory. This is made feasible by the availability of read-write registers on the datapath of reconfigurable switches. Second, AFQ uses coarser grain rounds that are incremented only after all active flows have transmitted a configurable number of bytes through an output port. Third, AFQ schedules packets to depart in an approximately sorted manner using a Logical Calendar Queue that uses multiple FIFO queues available at each

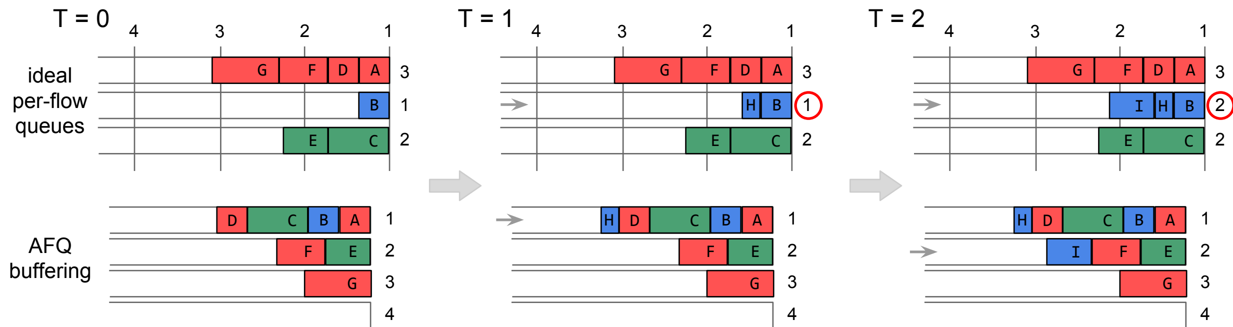


Figure 5.3: An example of the AFQ enqueue mechanism. As packets arrive, their bid numbers are estimated, and they are placed in an available FIFO queues. When a blue packet H arrives at T = 1, its bid number falls within round 1 and is placed in the first FIFO queue servicing round 1. When a subsequent blue packet I arrives at T = 2, its bid number falls in round 2; hence, it is placed is the second FIFO queue. For both packets H and I, we can see the approximation effects of using a large quantum of bytes per round and FIFO queues. An ideal FQ scheme using BR would transmit packet H before packets C and D, and packet I before E and F, as their last bytes are enqueued before the other packets in the per-flow queue. However, this reordering is upper-bounded by the number of active flows multiplied by the round quantum.

port. Combining these techniques yields schedules that approximate those produced by a fair queueing switch. However, we show that AFQ provides performance that is comparable to fair queueing for today's datacenter workloads despite these approximations. Figure 5.2 shows the pseudocode describing AFQ's main components, which we explain in more detail next.

5.2.1 *Storing Approximate Bid Numbers*

A flow's bid number in the BR algorithm is its finish-round number, which estimates when the flow's last enqueued byte will depart from the switch. The bid number of a flow's packet is a function of both the current active round number as well as the bid number associated with the flow's previous packet, and it is used to determine the packet's transmission order. AFQ stores each active flow's bid number in a count-min sketch-like data-structure described in Section 3.1 to reduce the stateful memory footprint on the switch since such memory is a limited resource. To recap, the count-min sketch building block is a 2D array of counters that supports two operations: (a) `inc(e,n)`, which increments the counter for element `e` by `n`, and (b) `read(e)`, which returns the

counter for element e. For a sketch with r rows and c columns, inc(e,n) applies r independent hash functions to e to locate a cell in each row and increments the cell by n. The operation read(e) applies the same r hash functions to locate the same r cells and returns the minimum among them. The approximate counter value always exceeds or equals the exact value, letting us store flow bid numbers efficiently in sub-linear space. Theoretically, to get an ϵ approximation, – i.e., $error < \epsilon \times K$ with probability $1 - \delta$, where K is the number of increments to the sketch, – we need $c = e/\epsilon$ and $r = log(1/\delta)$ [30].

In hardware, a sketch is realized using a simple *increment by x* primitive and predicated read-write registers (as described in [83]), both of which are available in reconfigurable switches. On packet arrival, r hashes of the flow's 5-tuple are computed to index into the register arrays and estimate the flow's finish round number, which is used to determine the packet's transmission schedule. In practice, AFQ re-uses one of several hashes that are already computed by the switch for Link Aggregation and ECMP. Today's devices support up to 64K register entries per stage and 12-16 stages [45], which is sufficient for a reasonably sized sketch per port to achieve good approximation, as we show later in the evaluation 5.4.2.

5.2.2 *Buffering Packets in Approximate Sorted Order*

The BR fair queuing algorithm ensures that the packet with the lowest bid number is transmitted next at any point of time using a sorted queue. Since maintaining such a sorted queue is expensive, AFQ instead uses a Calendar Queue that leverages the multiple FIFO queues available per port to approximate ordered departure of buffered packets.

Assume there are N FIFO queues available at each egress port of the switch. AFQ uses each queue to buffer packets scheduled to depart within the next N rounds, wherein each round, every active flow can send a fixed number of bytes, i.e., BpR bytes (bytes per round). We next describe how packets are enqueued and dequeued in approximate sorted order using these multiple queues.

Enqueue Module: The enqueue module decides which FIFO queue to assign to each packet. On arrival, the module retrieves the bid number associated with the flow's previous packet from the sketch. If it is lower than the starting bid number for the current round, the bid is pushed up to match the current round. The packet's bid number is then obtained by adding the packet's size to the previous bid number, and the packet's departure round number is computed as the packet's bid number divided by BpR. If this departure round exceeds N rounds in the future, the packet is dropped, else it is enqueued in the queue corresponding to the computed round number. Note that the current round number is a shared variable that the dequeue module updates after it finishes draining a queue. Finally, the enqueue module updates the sketch to reflect the bid number computed for the current packet. Figure 5.3 shows an example of how AFQ works when various flows with variable packet sizes arrive at the same egress port.

Clearly, having more FIFO queues leads to finer ordering granularity and a better approximation of fair queuing. Switches available today support 24-32 queues per port [21, 25], which we show is sufficient for datacenter workloads. AFQ assumes that the total buffer assigned to each port can be dynamically assigned to any queue associated with that port. This lets AFQ to absorb a burst of new flow arrivals when several packets are scheduled for the same round number. Most switches already implement this functionality via dynamic buffer sharing [28].

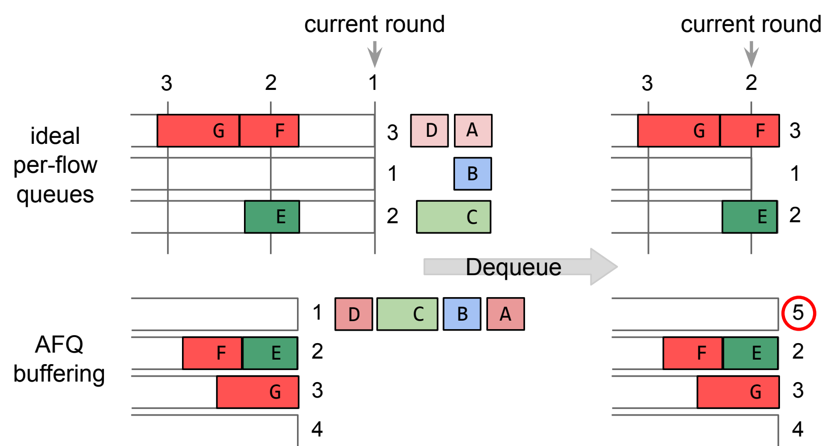


Figure 5.4: An example of the AFQ dequeue mechanism.

Dequeue Module: The dequeue module transmits the packet with the smallest departure round number. Since the enqueue module already stores packets belonging to a given round number in a separate queue, AFQ must only drain the queue with the smallest round number. This is achieved by arranging all queues in strict priority, with the queue having the lowest round number assigned the highest priority. However, once empty, the queue must be bumped down to the lowest priority and the current round number incremented by 1. Note that this round number is shared with the enqueue module, which can then adjust its queueing behavior. The just-emptied queue is then used to store packets belonging to a future round number that is N higher than the current round number. Figure 5.4 shows the priority change and round assignment that occurs when a queue is drained to empty by a *Rotating Strict Priority* (RSP) scheduler.

An important implication of this design is that updating the current round number becomes trivial – increment by 1 whenever the current queue drains completely. Unlike the BR fair queuing algorithm, which must update the round number on every packet arrival, this coarse increment does not involve any complex computations or extra packet state, making it much more feasible to implement on reconfigurable switches.

5.2.3 Impact of Approximations

First, using a count-min sketch means that AFQ can over-estimate a packet's bid number in case of collisions. As the number of active flows grows beyond the size of the sketch, the probability of collisions increases, causing packets to be scheduled later than expected. However, as we show in the Appendix 5.4.2, the sketch must be sufficiently large to store state only for active flows that have a packet enqueued at the switch, not all flows traversing the switch, including dormant ones that have not transmitted recently.

Second, unlike the BR fair queueing algorithm, which transmits one bit from each flow per round, AFQ lets active flows send multiple bytes per round. Since this departure round number is coarser than the bid number and AFQ buffers packets with the same round number in FIFO order, packets with higher bid numbers might be transmitted before packets with lower bid numbers if the switch received them earlier. This reordering can lead to unfairness within the round, but is

bounded by the number of active flows times BpR in the worst case.

BpR trade-off: Since AFQ buffers packets for the next N rounds only, the BpR must be chosen carefully to balance fairness and efficient use of the switch buffer. If BpR is too large, a single flow can occupy a large portion of the buffer, causing unfair packet delays and drops. If it is too small, AFQ will drop packets from a single flow burst despite having sufficient space to buffer them. The choice of BpR depends on network parameters, such as round trip times and link speeds, switch parameters, such as the number of FIFO queues per port and the total amount of packet buffer, as well as the end-host flow control protocol. We discuss how to set the BpR parameter after we describe the end-host transport protocol, which prescribes the rate adaptation mechanisms and determines the desired queue buildup at the switch.

5.3 Optimized End-host Flow Control Protocol

Although AFQ is solely a switch-based mechanism that can be deployed without modifying existing end-hosts to achieve significant performance improvement, a network-enforced fair queuing mechanism lets us optimize the end-host flow control protocol to extract even more gains. This section describes our approach, adapted from literature, for performing fast ramp-ups and keeping queue sizes small at switches. If all network switches provided a fair allocation of bandwidth, the bottleneck bandwidth can be measured using the packet-pair approach [52], which sends a pair of packets back-to-back and measures the inter-arrival gap.

Packet-pair flow control: We briefly describe the packet-pair flow control algorithm. At startup, two packets are sent back-to-back at line-rate, and the returning ACK separation is measured to get an initial estimate of the channel RTT and bottleneck bandwidth. Normal transmission begins by sending packet-pairs paced at a rate equal to the estimated bandwidth. For every packet-pair ACK received during normal transmission, the bandwidth estimate is updated and the packet sending rate adjusted accordingly. If the bandwidth estimate decreases, a few transmission cycles are skipped, proportional to the rate decrease, to avoid queue buildup. Similarly,

when the bandwidth estimate increases, a few packets, again proportional to the rate increase, are injected immediately to maintain high link utilization as described in [50], which also studies the stability of such a control-theoretic flow control.

Although this approach works well for an ideal fair-queuing network, we need to make some modifications for it to be robust against approximations introduced by AFQ. The complete pseudocode of our flow control protocol is shown in Figure 5.5.

```
SENDER PROTOCOL

Startup():
  state = STARTUP
  SendPacketPair()

SendPacketPair():
  /* Bound inflight bytes to roughly bdp. */
  if (inflight > CWND_FACTOR * bdp):
   /* Wait for ack or retransmission timeout. */
    return

  packet1 = nextPacket()
  packet1.first = true
  send(packet1)

  /* Add delay if necessary. */

  packet2 = nextPacket()
  send(packet2)

  if (state == STARTUP):
    Wait for AckReceive()
  else:
    nextSendTime = now() + 2 * MSS / rate
   scheduleTimer(SendPacketPair, nextSendTime)

On AckReceive(pktpair, rtt):
  newGap = pktpair.gap

  if (rtt < minRTT):
    minRTT = rtt

  if state == STARTUP:
    /* Start normal packet transmission. */
    state = NORMAL
    gap = newGap
    SendPacketPair()
  else:
    /* Update rate estimate. */
    gap = (1 - GAIN) * gap + GAIN * newGap
    linkRate = MSS / gap
    bdp = linkRate * minRTT

    /* Throttle rate based on ECN marks. */
    rate = linkRate * (1 - alpha / 2)

RECEIVER PROTOCOL

OnPacketReceive (packet):
  if (packet.first == true):
    first_pktpair_time = now()
    pktpair_ts = packet.sendTime
  else:
    gap = now - first_pktpair_time
    ack = nextAck()
    ack.sendTime = pktpair_ts
    ack.gap = gap
    send(ack)
```

Figure 5.5: Pseudocode for endhost flow control protocol

Robust bandwidth estimation: Since AFQ transmits multiple bytes in a single round, the packet-pair approach can incorrectly estimate bottleneck bandwidth if two back-to-back packets are enqueued in the same round and transmitted one after the other. This is not an issue if the BpR is less than or equal to 1 MSS, where MSS is the maximum segment size of the packets in the pair, and it holds true for our testbed and simulations. However, if the BpR is greater than twice the MSS, we must ensure that the very first packet-pair associated with a flow maps onto different rounds to get a reasonable bandwidth estimate using the inter-arrival delay. We accomplished this by adding a delay of `BpR-MSS` bytes at line-rate in between the packet-pairs at the end-host. This careful spacing mechanism, described in [89] measures the cross-traffic observed in a short interval and extrapolates it to identify the number of flows traversing the switch at that juncture. The protocol records the packet-pair arrival gap at the receiver and piggybacks on the acknowledgment to avoid noise and congestion on the reverse path. To further reduce variance, the protocol keeps a running EWMA of bandwidth estimates in the last RTT and uses the average for pacing packet transmission.

Per-flow ECN marking: Unlike an ideal fair-queueing mechanism, where the packet with the largest round number is dropped on buffer overflow, AFQ never drops packets that have already been enqueued. As a result, AFQ must maintain short queues to absorb bursty arrival of new flows. To dissipate standing queues and keep them short, we rely on a DCTCP-like ECN marking mechanism. Each sender keeps track of the fraction of marked packets and instead of transmitting packets at the estimated rate, the protocol sends packets at estimated rate times $(1 - \alpha/2)$. This optimization ensures that any standing queue is quickly dissipated. Further, unlike simple drop-tail queues, AFQ lets us perform per-flow ECN marking, which we exploit by marking packets when the enqueued bytes for a flow exceed a threshold round number. We set this number to 8 rounds in our simulations, which keeps per-flow queues very short without sacrificing throughput.

Bounding burstiness: Finally, since we have a fairly accurate estimate of the base RTT and the fair-share rate for each flow, we bound the number of inflight packets to a small multiple of the

available bandwidth-delay product (BDP) – similar to the rate based TCP BBR [23], – currently set to 1.5x the BDP in our implementation. This reduces network burstiness, especially when new flows arrive, by forcing older flows to stop transmitting due to their reduced BDP. This optimization keeps queues short, avoiding unnecessary queue buildup and packet drops.

We now perform a simple back of the envelope calculation to determine how to set the `BpR` parameter. As noted, we can use any end-host mechanism with AFQ, including standard ones such as TCP and DCTCP. Prior work has shown that DCTCP requires a queue of size roughly $1/6^{th}$ of the bandwidth-delay product for efficient link utilization [7]. If the average round-trip latency of the datacenter network is d and the peak line rate is l, then we require $d \times l/6$ amount of buffering for a single flow to ensure maximum link utilization. Further, if we have `nQ` queues in the system, then we set `BpR` to $d \times l/(6 \times nQ)$. In practice, this is less than a MSS for a 40 Gbps link, 20 us RTT, and 10-20 queues. Further, the amount of buffering required by a single flow can be even lower by using an end-host protocol that leverages packet-pair measurements (such as that described above). Section 5.4.2 provide empirical data from our experiments to show that our end-host protocol does indeed maintain lower levels of per-flow buffer buildup than traditional protocols and that packet drops are rare.

5.4 Evaluation

We evaluated AFQ's overall performance, fairness guarantees and feasibility using: (1) a hardware prototype based on a Cavium network processor within a small cluster, (2) large-scale packet-level simulations, and (3) a programmable switch implementation in P4.

5.4.1 Hardware Implementation

Since existing reconfigurable switches do not expose the programmability of internal queues, we therefore built a prototype of an AFQ switch using a programmable network processor. The Cavium OCTEON platform [27] has a multi-core MIPS64 processor with on-board memory and 4x10Gbps network I/O ports alongside several hardware-assisted network/application acceleration units, such as a traffic manager, packet buffer management units, and security co-processors.

All of these components are connected via fast on-chip interconnects providing high performance, low latency, and programmability for network applications ranging from 100Mbps to 200Gbps.

AFQ Switch Implementation We built a 4-port AFQ switch on top of the network processor using the Cavium Development Kit [26]. Figure 5.6 shows the high-level architecture, which includes 4 ingress pipelines, 4 egress pipelines, 32 FIFO packet queues, and a count-min sketch table containing 4 rows and 16K columns. The number of ports was fixed due to hardware limitations while all other individual components, such as ingress/egress pipelines, queue, and table sizes were configured based on available resources.

Each ingress and egress pipeline instance runs on a dedicated core, sharing access to packet buffer queues and the count-min sketch stored on the on-board DRAM. The ingress pipeline implements most of the AFQ functionality. First, it parses the packet and computes multiple hashes using on-chip accelerators for indexing into the count-min sketch. Next, it estimates the current round number for the packet using the algorithm shown in Figure 5.2. Finally, it updates the count-min sketch and enqueues the packet in the queue corresponding to the estimated round number. The egress simply dequeues packets from the queue corresponding to the current round being serviced, re-encapsulates the packets and transmits them to the specific port based on a pre-loaded MAC table.

Each packet queue maintains a shared lock to avoid race conditions arising from concurrent accesses of the ingress and egress cores. Other queue state updates and sketch table reads/writes use lock-free operations. We use the software reference counting technique to avoid TOCTOU race conditions.

End-host Protocol Implementation We implemented the packet-pair flow control protocol (Section 5.3) in user-space on top of UDP and integrated it with our workload generator. The implementation uses hardware timestamps from the NIC to measure the spacing between packet-pairs to accurately obtain bandwidth estimate and RTT samples, similar to prior work [64]. The

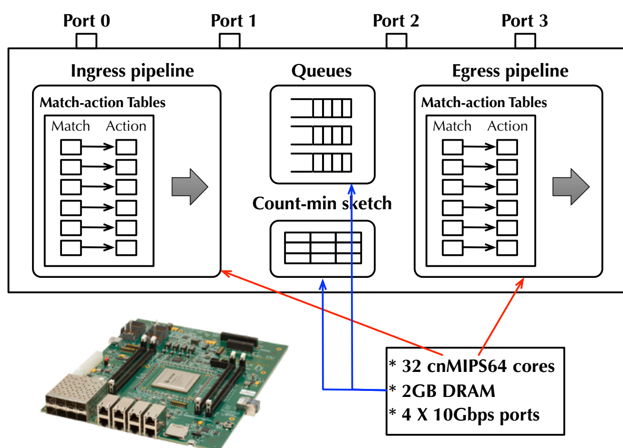


Figure 5.6: High-level architecture of the AFQ switch prototype.

flow control re-implements standard TCP sequencing, fast retransmit, and recovery in user-space atop UDP.

Hardware Testbed and Workload Our testbed includes 8 Supermicro servers, 2 Cavium XPliant switches and the prototype AFQ switch atop the network processor described above. All servers are equipped with 2x10Gbps port NICs. We created a 2-level topology using VLANs to divide the physical ports on the two switches. We integrated the prototyped AFQ switch into the aggregation switch which runs the AFQ mechanism at the second layer of the topology. The end-to-end latency is approximately $200\mu s$, most of which is spent inside the network processor.

We set up 4 clients and 4 servers that generated traffic using the enterprise workload described in [5], such that all traffic traversed the AFQ switch in the aggregation layer. Each client opened 25 concurrent long-running connections to each server, and requested flows according to a Poisson process at a rate configured to achieve desired network load. We compared four schemes,

- Default Linux TCP CUBIC with drop-tail queues
- DCTCP [6] with ECN marking drop-tail queues
- DCTCP with our AFQ mechanism

- Our packet-pair flow control with AFQ mechanism

For DCTCP, we enabled the default kernel DCTCP module and set the ECN marking threshold to $K = 65$ packets. For a fair comparison, we relayed the TCP and DCTCP traffic through our emulated switch.

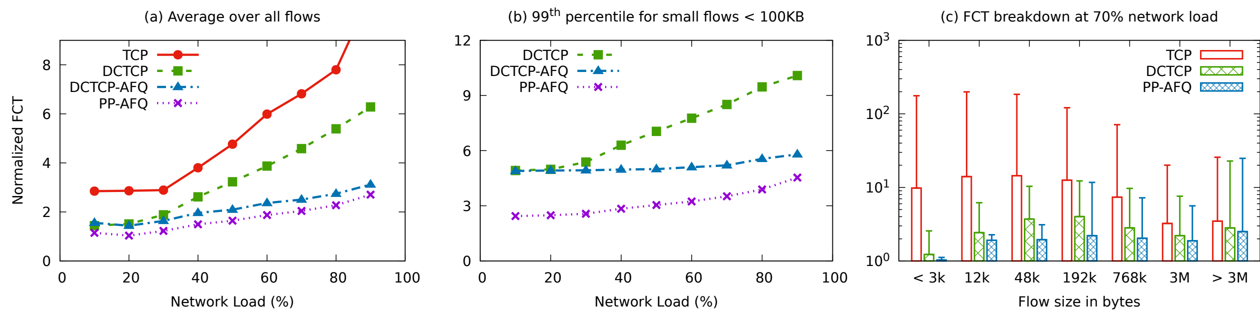


Figure 5.7: FCT summary for the enterprise workload on our hardware testbed. (a) average FCT for all flows, (b) tail latency for short flows, and (c) average and 99$^{\text{th}}$ percentile (using error bar) for various flow sizes. Note, TCP does not appear in (b) as its performance is outside the plotted range.

Overall Performance We use flow completion time (FCT) as the evaluation metric and report the average and 99$^{\text{th}}$ percentile latency over a period of 60 seconds. Figure 5.7 shows FCT statistics for various flow sizes as we increase the network load; data points are normalized to the average FCT achieved in an idle network. AFQ improves DCTCP performance by 2x and TCP performance by 10x for both average and tail flow completion times. The benefits of AFQ are more visible at high network loads when there is substantial cross-traffic with high churn. In such a scenario, TCP and DCTCP take multiple RTTs to achieve fair bandwidth allocation, and suffer long queueing delays behind bursty traffic; whereas AFQ lets new flows achieve their fair share immediately and isolates them from other concurrent flows, leading to significantly more predictable performance.

Figure 5.7(b) also shows the improvement from our packet-pair end-host flow control over DCTCP, as the packet-pair approach avoids slow-start and begins transmitting at fair bandwidth allocation immediately after the first ACK. This fast ramp-up along with fair allocation at AFQ

switches translates to significant FCT improvement, especially for short flows, as shown in Figure 5.7(c).

5.4.2 Packet-level Simulations

We also studied AFQ's performance in a large-scale cluster deployment using an event-driven, packet-level simulator. We extended the mptcp-htsim simulator [74] to implement AFQ and several other comparison schemes.

Simulation Topology and Workload We simulated a cluster of 288 servers connected in a leaf-spine topology, with 9 leaf and 4 spine switches. Each leaf switch is connected to 32 servers using 10Gbps links; and each spine switch is connected to each leaf using 40Gbps links. All leaf and spine switches have a fixed-sized buffer of 512KB and 1MB per port respectively. The end-to-end round-trip latency across the spine (4 hops) is $\approx 10\mu s$. All flows are ECMP load-balanced across all spine switches. We use a small value of minRTO = $200\mu s$ for all schemes, as suggested in [9].

We used both synthetic and empirical workloads derived from traffic patterns observed in production datacenters. The synthetic workload generates Pareto distributed ($\alpha = 1.1$) flows with mean flow size 30KB. The empirical workload is based on an enterprise cluster reported in [5]. Flows arrive according to a Poisson process at randomly and independently chosen source-destination server pairs from all servers. The arrival rate is chosen to achieve the desired level of utilization in the spine links. Both workloads are heavy-tailed with majority bytes coming from a small fraction of large flows; both also have a diverse mix of short and long flows, with the enterprise workload having more short flows.

Comparison Schemes

- **TCP**: Standard TCP-Reno with fast retransmit and recovery, but without SACKs, running on switches with traditional drop-tail queues.

- **DCTCP**: The DCTCP [6] congestion control algorithm with drop-tail queues supporting ECN marking on all switches; marking threshold set to 20 packets for 10Gbps links and 80 packets for 40Gbps links.

- **SFQ**: Same TCP-Reno as above with Stochastic Fair Queueing [60] using DRR [81] on all switch ports; with 32 FIFO queues available at each switch port.

- **AFQ**: Our packet-pair flow control with AFQ switches using 32 FIFO queues per port, a count-min sketch of size 2x16384, and a `BpR` of 1 MSS.

- **Ideal-FQ**: An ideal fair queueing router that implements the BR algorithm (described in [32]) and uses our packet-pair flow control at the end-host.

Overall Performance We compared the overall performance of various schemes in the simulated topology by measuring the FCT of all flows that finished over a period of 10 seconds in the simulation. Figures 5.8 and 5.9 show the normalized FCT (normalized to the average FCT achieved in an idle network) for all flows, short flows (<100KB) and flows bucketed across different sizes at varying network loads and workloads.

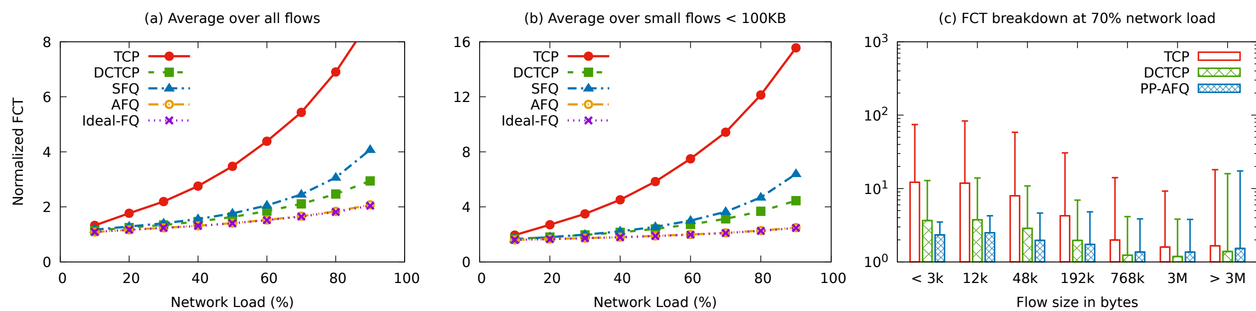


Figure 5.8: Flow completion times for synthetic workload in the cluster. (a) average FCT for all flows, (b) average FCT for flows shorter than 100KB, and (c) average and 99th percentile (using error bar) for various flow size buckets at 70% network load.

Our simulation results match previous emulated observations. As expected, most schemes perform close to optimal at low network load, but quickly diverge as network traffic increases.

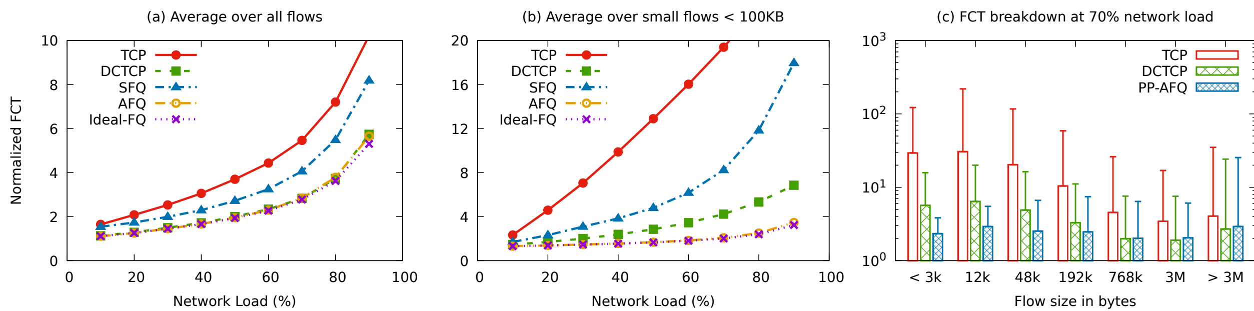


Figure 5.9: Flow completion times for enterprise workload, with each graph showing the same metrics as in Figure 5.8.

Traditional TCP keeps switch buffers full, leading to long queueing delays, especially for shorter flows. DCTCP improves the performance of short flows significantly since it maintains shorter queues, but is still a factor of 2-4x away from ideal fair-queuing behavior. SFQ works very well at low network loads when the number of active flows is comparable to number of queues, however as network traffic increases, collisions within a single queue become more frequent leading to poor performance. AFQ achieves close to ideal fair queuing performance for all network load, which is 3-5x better than TCP and DCTCP for tail latency of short flows: irrespective of other network traffic, all flows immediately get their fair share of the network without waiting behind other packets. This leads to a significant performance benefit for shorter flows, which do not have to wait behind bursty traffic.

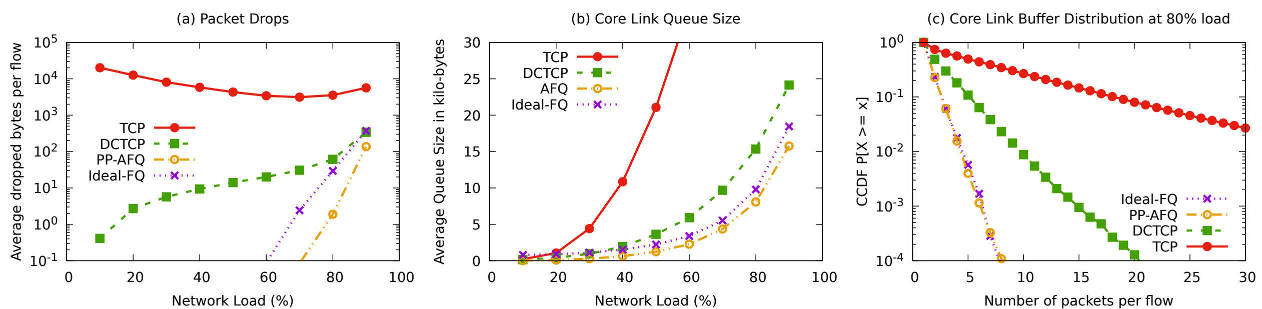


Figure 5.10: Packet drops, queue lengths and buffer occupancy distribution for enterprise workload in the cluster.

To further understand the performance gains, we measured several other metrics, such as packet drops, re-transmissions, average queue lengths, and buffer occupancy distribution during the experiment. Figure 5.10(a) shows the average bytes dropped per flow for each scheme. As expected, standard TCP drops on average one packet per flow, and DCTCP has negligible drops at low network load. However, at higher loads, drops are more frequent, leading to occasional re-transmission and performance penalty. This is also reflected in the average queue length shown in Figure 5.10(b). Both DCTCP and packet pair with AFQ are able to maintain very short queues, but with an interesting difference in the buffer occupancy distribution as shown in Figure 5.10(c). We took periodic snapshots on the queue every $100\mu s$, to count how many packets belong to each flow in the buffer and plotted the CCDF of the number of packets per flow across all snapshots. AFQ with packet-pair flow control rarely has more than 5 packets enqueued per flow at the core links, whereas DCTCP and TCP have many more packets buffered per flow. This can lead to unfairness when bursty traffic arrives, such as during an incast, which we discuss next. In summary, AFQ achieves similar performance to DCTCP for all flows, and 2x better performance for short flows while maintaining shorter queues and suffering fewer drops by ensuring fair allocation of bandwidth and buffers.

Incast Patterns Incast patterns, common in datacenters, often suffer performance degradation due to poor isolation. In this setup, we started a client on every end-host which requests a chunk of data distributed over N other servers. Each sender replies back with $1/N$ of the data at the same time. We report the total transfer time of the chunk of data with a varying number of senders averaged over multiple runs. Simultaneous multiple senders can cause unfair packet drops for flow arriving later, causing timeouts that delay some flows and increase overall completion time. An ideal fair-queuing scheme would allocate equal bandwidth and buffer to each sender, hence finishing all transfers at roughly the same time.

Figure 5.11 shows the total completion time of various schemes for a total chunk size 1.5MB with varying number of senders. The receiver link has approximately 300KB of buffer, roughly around 200 packets. Most schemes perform well with few senders but degrade when the number

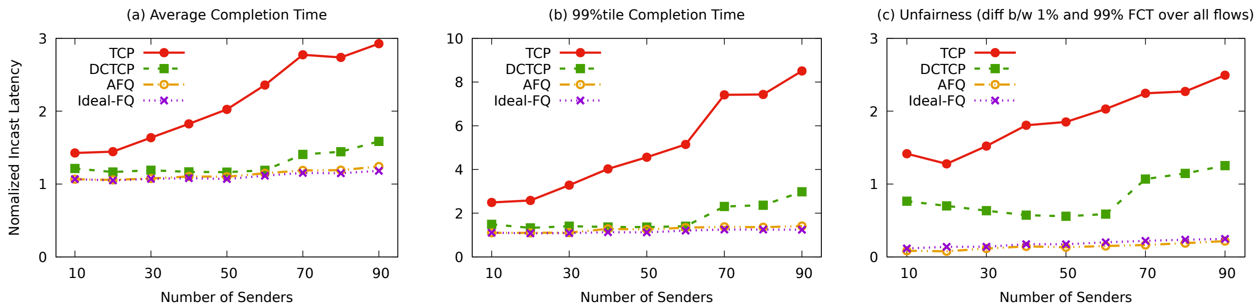


Figure 5.11: Completion time summary for an incast request of size 1.5MB from a varying number of senders.

of senders overwhelms the receiver link buffer. This leads to packet drops for flows arriving later in a traditional drop-tail queue, sending them into timeouts. AFQ achieves close to optimal request completion time, even with large senders because it ensures each flow gets fair buffer allocation regardless of when it arrives. As a result packet drop are minimal, leading to fewer re-transmissions and lower completion time. Figure 5.12 shows the number of packet drops observed during the incast, packet re-transmissions, and buffer occupancy, which confirm the preceding observation. As expected, TCP drops several packets throughout the incast experiment, causing several re-transmissions. DCTCP performs much better and suffers zero packet drops until the number of senders exceeds the link buffer capacity. AFQ has even fewer drops than DCTCP because it fairly distributes the available buffer space among all flows, as shown in Figure 5.12(c).

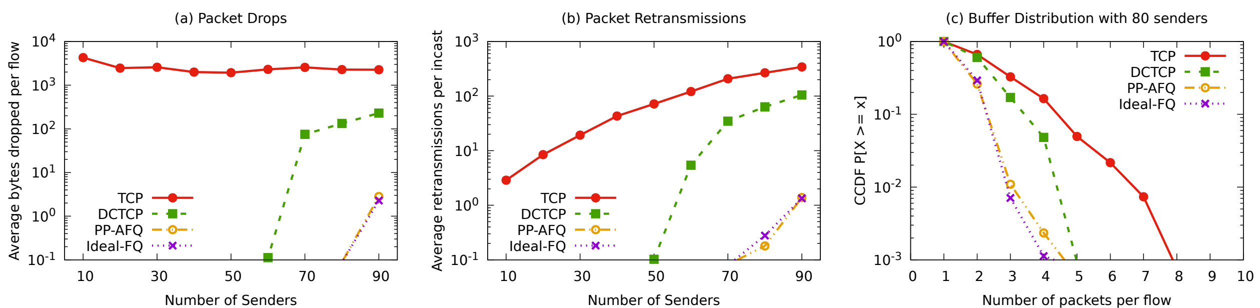


Figure 5.12: Packet drops, re-transmissions and buffer distribution across flows during incast traffic.

Convergence and Fairness To demonstrate that AFQ does indeed assign each flow its fair share rapidly, we connected two hosts via a 10Gbps, 10μs RTT link and sequentially started-stopped flows at 1-second intervals. We used standard TCP end-hosts, and change the queueing mechanism from drop-tail to AFQ. The time series in Figure 5.13 shows the throughput achieved by each flow as they enter and exit the link. AFQ assigns each flow its fair share immediately, while a drop-tail queue exhibits high variance in throughput.

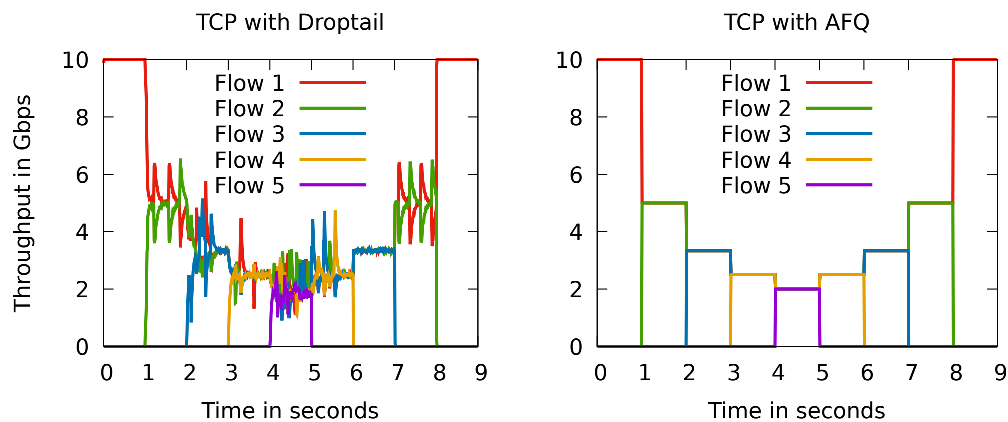


Figure 5.13: Flow convergence test with TCP and AFQ.

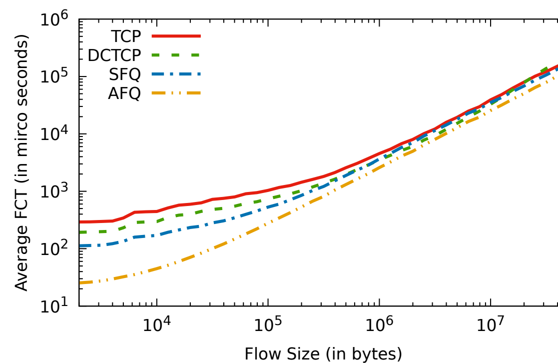


Figure 5.14: FCT vs flow size at 70% load.

Next, we plot the FCT versus flow size from our cluster simulations in Figure 5.14 to demonstrate how fair each scheme is with respect to flow size. An ideal fair queuing scheme would be

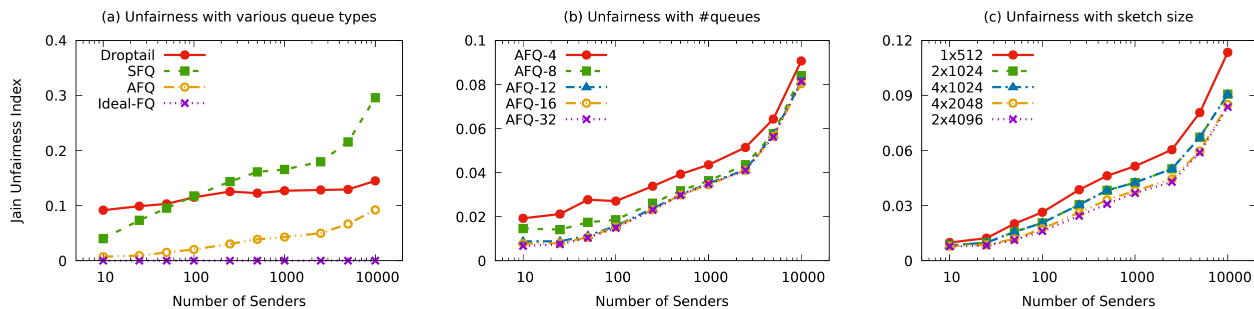


Figure 5.15: Micro-benchmarks showing deviation of various queuing mechanism compared to ideal fair queuing using a DCTCP end-host.

a straight line from the origin. All schemes achieve fairness over a period of time for long flows, but are significantly unfair to short flows either due to slow-start or queueing behind other flows in the network. AFQ lets all flows, regardless of size to achieve their fair share within an RTT, leading to better fairness.

To further study AFQ's fairness guarantees, we simulated a 10Gbps, 25μs RTT link and increased the number of on-off senders transmitting concurrent flows on the link. We measured the Jain Unfairness index (1–Jain Fairness [42]) across all flows. Figure 5.15(a) shows the unfairness across different queueing schemes. AFQ has better fairness than other schemes, until the number of concurrent flows exceeds the sketch size. Figures 5.15(b) and (c), plot the same metric while varying the sketch-size and number of FIFO queues available to AFQ.

Impact of Number of Queues on FCT AFQ uses multiple FIFO queues to store packets in an approximate sorted order. To understand how many FIFO queues are required per-port to get accurate fair-queueing behavior, we ran the same enterprise workload while varying the number of FIFO queues available to the AFQ implementation and keeping BpR fixed at 1 MSS. Figure 5.16 shows the impact on average FCT of all flows as we varied the number of queues from 4 to 32. When fewer queues are available, AFQ buffers packets for very few rounds at any given time. This causes unnecessary packet drops during bursty arrivals, and also leads to poor bandwidth estimation at the end-host. Once there are sufficient queues to absorb packet bursts and accurately

estimate bottleneck bandwidth, AFQ achieves near-ideal fair queueing behavior, which occurred around 16-20 queues. This is not surprising, given the analysis from [6], a queue of size roughly $1/6^{th}$ of the bandwidth-delay product is required for efficient link utilization. For our testbed with 40 Gbps links and $20\mu s$ RTT, this value is ≈20KB, translating to about 15 queues.

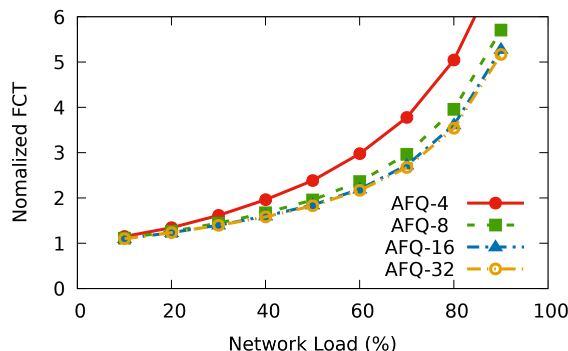


Figure 5.16: FCT vs number of FIFO queues.

AFQ with Other End-host Protocols As an in-network switch mechanism, AFQ can be deployed without modifying the end-host to achieve significant performance gains. To quantify the benefits, we simulate the same enterprise workload using TCP, DCTCP end-hosts with all switches implementing the AFQ mechanism. Figure 5.17 shows the significant improvement in average FCT when switching from drop-tail to AFQ behavior inside the network. Moving to DCTCP gives another small improvement due to shorter queues; finally, using packet-pair flow control eliminates slow-start behaviors, further reducing FCT. This matches our observations from the hardware prototype emulation.

Impact of Sketch Size AFQ stores bid numbers in a count-min sketch, trading off space for accuracy. To determine how large a sketch is required to achieve sufficient accuracy without affecting performance, we re-ran the enterprise workload in the leaf-spine topology while tracking exact bid numbers and those returned by the count-min sketch. During the 10 second simulation run, we count how many times a packet bid number was misestimated and enqueued in a

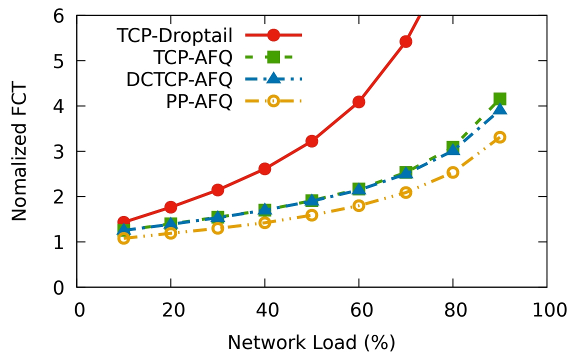


Figure 5.17: AFQ with different End-hosts

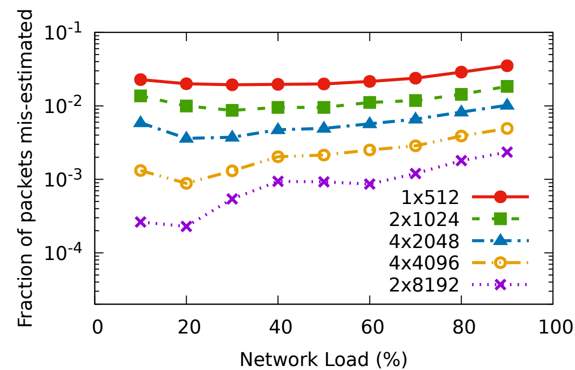


Figure 5.18: Round Estimation vs Sketch Size

later-than-expected queue. Figure 5.18 shows the misestimation rate as we change the sketch size. This is less than 1% using a relatively small sketch of 2x1024. This is not surprising since the collision probability is proportional to the number of active flows that have packets enqueued at the switch, which is generally a few tens to hundreds. It is not affected by the total number of ongoing flows, which could be several thousands. Such a low rate of misestimation does not significantly impact the flow-level performance, because a bad estimate delays the packet by only a small amount of time. Further, increasing the number of cells in each row has a more significant impact on the accuracy than increasing the number of rows.

5.4.3 P4 Implementation

To evaluate the overhead of implementing AFQ on an actual reconfigurable switch, we expressed AFQ in the P4 programming language and compiled it to a production switch target. The P4 code ran on top of a baseline switch implementation [84] that provides common functionality of today's datacenter switches, such as basic L2 switching (flooding, learning, and STP), basic L3 routing (IPv4, IPv6, and VRF), link aggregation groups (LAGs), ECMP routing, VXLAN, NVGRE, Geneve and GRE tunneling, and basic statistics collection. The compiler implements the functionality proposed in [46] and compiles to the hardware model described in Section 2.2. It reports the hardware usage of various resources for the entire implementation.

Table 5.1 shows the additional overhead of implementing two variants of AFQ as reported by

Resource	Baseline	+AFQ		+AFQ-Large	
Pkt Header Vector	187	191	+2%	191	+2%
Pipeline Stages	9	12	+33%	12	+33%
Match Crossbar	462	465	+1%	465	+1%
Hash Bits	1050	1082	+3%	1092	+4%
SRAM	165	178	+8%	190	+15%
TCAM	43	44	+2%	44	+2%
ALU Instruction	83	90	+8%	90	+8%

Table 5.1: Summary of resource usage for AFQ.

the compiler. AFQ uses a count-min sketch of size 2x2048, while AFQ-Large uses a sketch of size 3x16384. We can see the extra overhead is small for most resources. We need more pipeline stages to traverse the count-min sketch and keep a running minimum, and more SRAM to store all the flow counters. We also use extra ALU units to perform per-packet increments and bit-shifts to divide by `BpR`.

5.5 *Summary*

In this chapter, we showed how to implement a fair bandwidth allocation mechanism called Approximate Fair Queueing (AFQ), on emerging reconfigurable switches. We approximate the various mechanisms of a fair queueing scheduler using features available on reconfigurable switches. Specifically, we approximate the per-flow state regarding the number and timing of its previously transmitted packets using mutable switch state; we perform limited computation for each packet to compute its position in the output schedule; we dynamically determine which egress queue to use for a given packet; and we leverage Calendar Queues, described earlier to transmit packets in approximate sorted order. Using a networking-processor-based prototype in a real hardware testbed and large scale simulations, we showed that AFQ approximates ideal queuing behavior accurately, improving performance significantly over existing schemes.

Chapter 6

CONCLUSIONS AND FUTURE WORK

As network switching technology evolves to provide flexible match+action processing and configurable scheduling for each forwarded packet, it holds the promise of making a software-defined data-plane a reality. This thesis presents a feasibility study of a wide class of in-network protocols on emerging reconfigurable switches or FlexSwitches.

While hardware constraints make the precise implementation of resource allocation protocols and scheduling algorithms difficult, we show that it is possible to approximate several popular algorithms with acceptable accuracy, by careful design of approximate data structures and co-design of network computation with end-host processing to reduce the computational demand on the FlexSwitches. We develop a library of building blocks that are broadly applicable to a wide range of network allocation and management applications, allowing us to implement efficient in-network protocols that solve important problems like congestion control, load balancing, fair sharing and QoS management. Our approximate variants of these protocols are able to accurately emulate their original counterparts for common data center workloads.

Next, we propose a flexible packet scheduler for line-rate hardware switches, called Programmable Calendar Queues, that enables the efficient realization of several classical scheduling algorithms. It relies on the observation that most algorithms require both prioritization and implicit escalation of a packet's priority. We show how they can be implemented efficiently on today's programmable switches by dynamically changing the priority of queues using either data-plane primitives or control-plane operations. We demonstrate that PCQs can be used to realize interesting variants of LSTF, Fair Queueing, and pFabric to provide stronger delay guarantees, burst-friendly fairness, and starvation-free prioritization of short flows, respectively.

Finally, we take a fair bandwidth allocation mechanism and, using techniques developed in

the dissertation, implement a variant called Approximate Fair Queueing (AFQ), designed to run on emerging reconfigurable switches. We approximate the various mechanisms of a fair queueing scheduler using features available on reconfigurable switches. Specifically, we approximate the per-flow state regarding the number and timing of its previously transmitted packets using mutable switch state; we perform limited computation for each packet to compute its position in the output schedule; we dynamically determine which egress queue to use for a given packet; and transmit packets in approximate sorted order using Calendar Queues. This demonstrates the utility of techniques developed in this dissertation.

6.1 Future Work

This dissertation takes a first step towards understanding the flexibility and limitations of FlexSwitches in the context of classic networking problems, such as network resource allocation and packet scheduling inside the network. We focus on these problems as they have a rich literature that advocates per-packet data-plane processing and scheduling in the network. Researchers have used data-plane processing to provide rate adjustments to end-hosts (congestion control), determine meaningful paths through the network (load balancing), schedule or drop packets (QoS, fairness), monitor flows to detect anomalies and resource exhaustion attacks (IDS), and so on.

However, several other areas are worth investigating which will determine how successful FlexSwitches become in the future. We briefly describe some of these below.

Sharing FlexSwitches among multiple tenants or applications. Most of the work presented in this thesis assumes exclusive and complete control over the switch. However, in most deployments, network resources are shared among multiple tenants or applications running inside the data center. It is not clear how switch resources should be shared or efficiently multiplexed among competing entities, each of which have their own demands. Exposing the right set of abstractions and providing precise guarantees to the users or applications of the network is going to be crucial in this regard.

Programming the whole network. In this thesis, we have focused on FlexSwitches as a single entity in isolation and programmed a single device at a time. But any network is composed of multiple such switches and end-hosts, and most networks are interested in end-to-end properties. A natural next step is to explore ways to take a high-level end-to-end specification for a network and automatically program individual devices in the network to achieve end-to-end objectives.

Integration with existing programming languages. All the implementation on real hardware demonstrated in this thesis has been done in low-level device-specific programming languages. There are multiple language proposals such as P4 [20] and PoF [86] that aim to provide a high-level programming environment for FlexSwitches. It is worth investigating how our library of building blocks or programmable calendar queue abstraction can be integrated with these existing programming languages.

Verification Tools. While flexibility of high-speed switches enables us to do interesting things inside the network, it also opens up a new world of bugs that can have severe impact. Till now, hardware manufacturers were responsible for verifying that the switching chip accurately implements standardized protocols and handles all edge-cases. But with FlexSwitches, network operators will have to make sure the code they compile on individual switches is correct and bug-free. It will be crucial to develop right verification tools to help network operators achieve this.

Monitoring, tracing and debugging networks. Another major use-case for FlexSwitches that we have not explored in detail in this thesis is that of network monitoring and tracing. FlexSwitches provide a huge level of insight into the network, right down to packet level. With the right instrumentation tools and filters at switches, we can build powerful network debugging tools that can operate in realtime. Today, it can take several hours to diagnose network failures and localize issues. With enhanced visibility and programability inside the network, FlexSwitches can make network debugging much faster.

Deployment Model. FlexSwitches are still being actively developed and it is not clear how they are going to be deployed or used within data centers or the internet in general. Will Flex-Switches replace just a portion of the network, say only top-of-rack switches or all switches inside a datacenter? It is unlikely core switches and routers will be replaced by FlexSwitches as they have different requirements and operate at much higher bandwidths. In such scenarios, can we still extract the benefits of in-network protocols if only a fraction of the network supports programmability and flexibity? As FlexSwitches evolve, we'll have to study and understand how to make the best use of their programmability.

FlexSwitch Evolution. FlexSwitches are relatively new and will continue evolving significantly in the near future. Potential challenges in this area will be to figure out which features and primitives should be added onto reconfigurable switches to provide the most benefit in terms of not just expressiveness, but performance as well. This thesis proposes several primitives and ideas in this direction that enable the implementation of a wide range of in-network protocols. However, there is a large class of rich and complex protocols that will require even more innovation in switching chip design to make them feasible, especially in the face of ever-increasing link bandwidth demand in the industry.

BIBLIOGRAPHY

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal Network Design for Coflows. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2018.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2010.

[3] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, Santa Clara, CA, 2020.

[4] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, Rong Pan, B. Prabhakar, and M. Seaman. Data Center Transport Mechanisms: Congestion Control Theory and IEEE standardization. In *Annual Allerton Conference on Communication, Control, and Computing*, 2008.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2014.

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

[7] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: Stability, Convergence, and Fairness. In *ACM Conference on Measurement and Modeling of Computer Systems, SIGMETRICS*, 2011.

[8] 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 *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2012.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2013.

[10] M Allman, V Paxson, and E Blanton. TCP Congestion Control. RFC 5681, 2009.

[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2016.

[12] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced Allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[13] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2015.

[14] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2016.

[15] Barefoot Networks. Personal communication.

[16] Barefoot Networks. Tofino Programmable Switch. `https://www.barefootnetworks.com/technology/`.

[17] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM Internet Measurement Conference, IMC*, 2010.

[18] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *IEEE Conference on Computer Communications, INFOCOM*, 2000.

[19] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2006.

[20] 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*, 44(3):87–95, July 2014.

[21] 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 *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2013.

[22] R. Brown. Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*, 31:1220–1227, 1988.

[23] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5):50:20–50:53, October 2016.

[24] Cavium. CNX880XX_PB_p1 rev1 - Cavium. `http://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf`.

[25] Cavium. XPliant Ethernet Switch Product Family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[26] Cavium. OCTEON Development Kits, 2016. `http://www.cavium.com/octeon_software_develop_kit.html`.

[27] Cavium. Cavium OCTEON SoC Development Board, 2017. `http://www.cavium.com/OCTEON_MIPS64.html`.

[28] A. K. Choudhury and E. L. Hahne. Dynamic Queue Length Thresholds for Shared-memory Packet Switches. *IEEE/ACM Transactions on Networking*, 6(2):130–140, 1998.

[29] P4 Consortium. In-band Network Telemetry (INT). `https://p4.org/assets/INT-current-spec.pdf`.

[30] Graham Cormode and S Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[31] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *ACM Symposium on SDN Research, SOSR*, 2015.

[32] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 1989.

[33] Nandita Dukkipati. *Rate Control Protocol (RCP): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2007.

[34] Gibb, Glen and Varghese, George and Horowitz, Mark and McKeown, Nick. Design principles for packet parsers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS*, 2013.

[35] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2015.

[36] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2018.

[37] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. FCP: A Flexible Transport Framework for Accommodating Diversity. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2013.

[38] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2015.

[39] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden. The Interface Message Processor for the ARPA Computer Network. In *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, 1970.

[40] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2012.

[41] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2017.

[42] Raj Jain, Dah-Ming Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. *Computing Research Repository, CoRR*, cs.NI/9809099, 1998.

[43] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2013.

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2018.

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

[46] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2015.

[47] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2002.

[48] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM Symposium on SDN Research, SOSR*, 2016.

[49] Anirudh Sivaraman Kaushalram. *Designing Fast and Programmable Routers*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2017.

[50] Srinivasan Keshav. A Control-theoretic Approach to Flow Control. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 1991.

[51] Srinivasan Keshav. On the Efficient Implementation of Fair Queueing. *Journal of Internetworking: Research and Experience*, 2:157–173, 1991.

[52] Srinivasan Keshav. The Packet Pair Flow Control Protocol. Technical Report 91-028, ICSI Berkeley, 1991.

[53] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *ACM Symposium on Operating Systems Principles, SOSP*, 2017.

[54] 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 *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.

[55] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2016.

[56] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2016.

[57] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2017.

[58] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2016.

[59] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, 2014.

[60] Paul E McKenney. Stochastic Fairness Queueing. In *IEEE Conference on Computer Communications, INFOCOM*, 1990.

[61] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[62] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2017.

[63] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal Packet Scheduling. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, Santa Clara, CA, 2016.

[64] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2015.

[65] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2014.

[66] J. B. Nagle. On Packet Switches with Infinite Storage. In *Innovations in Internetworking*, pages 136–139. Artech House, Inc., 1988.

[67] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2017.

[68] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[69] NS-3 Consortium. ns-3 Network Simulator. `http://www.nsnam.org/`.

[70] Recep Ozdag. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf`.

[71] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2013.

[72] Philippe Flajolet and Éric Fusy and Olivier Gandouet and Frédéric Meunier. HyperLogLog: The Analysis of a Near-optimal Cardinality Estimation Algorithm. *HAL CCSD; Discrete Mathematics and Theoretical Computer Science*, pages 137–156, June 2007.

[73] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *ACM Workshop on Hot Topics in Networks, HotNets*, 2011.

[74] Costin Raiciu. MPTCP htsim simulator. `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html`.

[75] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX Conference on System Administration*, 1999.

[76] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2015.

[77] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and Flexible Software Packet Scheduling. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, Boston, MA, 2019.

[78] Muhammad Shahbaz and Nick Feamster. The Case for an Intermediate Representation for Programmable Data Planes. In *ACM Symposium on SDN Research, SOSR*, 2015.

[79] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, Renton, WA, 2018.

[80] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2011.

[81] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 1995.

[82] Vishal Shrivastav. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2019.

[83] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2016.

[84] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. DC.P4: Programming the Forwarding Plane of a Data-center Switch. In *ACM Symposium on SDN Research, SOSR*, 2015.

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

[86] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN*, 2013.

[87] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, Boston, MA, 2019.

[88] Ion Stoica, Scott Shenker, and Hui Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 1998.

[89] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *ACM Internet Measurement Conference, IMC*, 2003.

[90] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2011.

[91] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2013.

[92] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2012.

[93] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *European Conference on Computer Systems, EuroSys*, 2014.

[94] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM Conference on Data Communication, SIGCOMM*, 2015.