SCALABLE NETWORK SCHEDULING IN SOFTWARE

A Dissertation
Presented to
The Academic Faculty

By

Ahmed Mohamed Said Mohamed Tawfik Issa

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2019

SCALABLE NETWORK SCHEDULING IN SOFTWARE

Approved by:

Dr. Mostafa H. Ammar, Co-advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ellen W. Zegura, Co-advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Amin Vahdat
Google

Dr. Jun Xu
School of Computer Science
Georgia Institute of Technology

Date Approved: June 17th, 2019

To my wife, Heba Kamal.

ACKNOWLEDGEMENTS

This work is a culmination of work done throughout my PhD which wouldn't have been possible without the encouragement, support, and guidance of many. I am grateful to everyone who was there for me. First and foremost, I would like to express my sincere gratitude to my advisors Prof. Ellen W. Zegura and Prof. Mostafa Ammar for their dedication, availability, guidance, patience, and support. From my first day at Geogria Tech, they were accommodating of my inclination to exploration and my many side projects. Their guidance helped me to grow and develop academically and professionally. Their complimentary views and advice enriched my experience and allowed me to make more educated choices throughout my PhD. I am also extremely grateful to Prof. Khaled Harras whom without I wouldn't have applied to earn a PhD degree. I am very thankful for his wise shepherding my career since my I earned my bachelor's degree.

I am also grateful to my mentors at Google Dr. Nandita Dukkipati and Dr. Amin Vahdat. Their generosity with their time and their commitment to excellence helped shape my research approach. I am especially thankful to Dr. Dukkipati for hosting me twice as an intern and giving me the freedom to explore new research directions, while keeping me focused on practical and critical problems. I would like to also thank the rest of my thesis committee members Prof. Ada Gavrilovska and Prof. Jun Xu for their helpful comments and insights that enabled me to improve the work in this thesis.

I am also thankful for my collaborators Yimeng Zhao, Dr. Vytautas Valancius, Dr. Vinh The Lam, and Carlo Contavalli. I would also like to thank a group of wonderful colleagues in the Networking Research Group at Georgia Tech: Dr. Samantha Lo, Dr. Bilal Anwer, Tarun Mangla, Sean Donovan, Kamal Shadi, Kaesar Sabrin, Payam Siyari, and Danny Lee for their support, encouragement, fruitful discussions and above all friendship. I am especially thankful to Dr. Karim Habak whom I have shared my journey with for over thirteen years, from our first year of undergraduate class through the years of my PhD. His

friendship and support helped me through many life events.

Last, but certainly not least, I would like to thank my family for their continuous love and encouragement. I am grateful to my father who kindled and fostered by my love for science, encouraging whatever discussions and inquisitions I start, my mother who supported all my choices and has been my friend and confidant, and my wife whose support, encouragement, patience, tolerance, and initiative helped make my work possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Summary

Network scheduling determines the relative ordering and priority of different flows or packets with respect to some ranking function that is mandated by a scheduling policy. It is the core component in many recent innovations to optimize network performance and utilization. The increasing scale of cloud applications is driving the need to scale network scheduling to millions of flows and to apply complex policies. This requires software network stacks to handle a large number of flows, ranked according to operator-defined policies, and going over different types of networks. This dissertation describes three network scheduling systems that allow network operator to deploy scheduling policies efficiently in software, taking into account the diversity of the requirements of modern networks.

First, we develop a system that enforces packet shaping, the most basic scheduling policy efficiently at end hosts. The challenge of packet shaping at scale is optimizing the CPU and memory overhead of the rate limiting system. We present a framework that scales to tens of thousands of rate limiting policies and flows per server, built from the synthesis of three key ideas: i) a single queue shaper using time as the basis for releasing packets, ii) fine-grained, just-in-time freeing of resources in higher layers coupled to actual packet departures, and iii) one shaper per CPU core, with lock-free coordination.

Second, we develop a system that provides efficient and programmable packet schedulers in software. Unlike non-work-conserving shapers, programmable schedulers are challenging to implement efficiently. We overcome this challenge by exploiting underlying features of packet ranking; namely, packet ranks are integers and, at any point in time, fall within a limited range of values. To support programmability, we introduce novel programming abstractions to express scheduling policies that cannot be captured by current, state-of-the-art scheduler programming models.

Third, we present a congestion control approach that is designed to handle datacenter LAN traffic and WAN traffic simultaneously. This work is motivated by our observation

that customization of congestion control algorithms causes the WAN traffic to negatively impact the performance of the LAN traffic. This customization is based on the differences between WAN and LAN characteristics. However, it overlooks that both types of traffic compete at the near-source bottlenecks where the two types of traffic overlap. We propose deploying two interacting congestion control loops: one handles near-source congestion leveraging local network feedback, while the other handles congestion on the rest of the path using conventional LAN and WAN congestion control schemes.

CHAPTER 1
INTRODUCTION

For years, improved chips added more cores per chip, rather than increasing the processing capacity per core [1, 2]. This means that rather than relying on free performance improvements through improved chips, parallel execution became the main way of making use of the new chips [3]. Moving forward beyond parallelization, performance improvements will have to come from specialized hardware (e.g., accelerators, FPGAs, or ASICs) connected to general purpose compute over the network [4]. From the perspective of the networking stack, this meant that rather than having to serve a few connections per machine, new networking stacks have to cope up with requirements of multiple network connections running in parallel [5, 6]. We can already see the exponential growth of number of network connections handled by a single host driven by innovations in systems architectures. For instance, virtualization allows having multiple users simultaneously using the same end host, each competing for the network resources of the physical machine [7]. Furthermore, massively parallel applications led to having communication over the network as an essential component of task execution [8]. Moreover, disaggregation of resources means that access to storage, memory, and special hardware (e.g., GPUs) is performed over the network [9]. This means that the number of network connections per end host, and consequently the total number of connections in the network, will keep growing for the foreseeable future.

Growth in network scale in terms of both number of connections, as well as traffic volume, prompted innovation in the design and implementation of the various network functions. *Our focus in this dissertation is on the design and implementation of efficient and flexible network scheduling in software, taking into account the diversity of the requirements of modern networks.* While there are several network functions that can be optimized to operate at scale, our focus on network scheduling is motivated by its significance in the

development and deployment of modern networks. In particular, network scheduling is the core component in many recent innovations to optimize network performance and utilization. Typically, packet scheduling targets network-wide objectives (e.g., meeting strict deadlines of flows [10], reducing flow completion time [11]), or provides isolation and differentiation of service (e.g., through bandwidth allocation [12, 13] or Type of Service levels [14, 15, 16]). It is also used for resource allocation within the packet processing system (e.g., fair CPU utilization in middleboxes [17, 18] and software switches [19]).

Network scheduling determines the ordering of packets in a *queuing data structure* with respect to some *ranking function* that is mandated by a *scheduling policy*. In particular, as packets arrive at the scheduler they are *enqueued*, a process that involves ranking based on the scheduling policy and ordering the packets according to the rank. Then, periodically, packets are *dequeued* according to the packet ordering. In general, the dequeuing of a packet might, for some scheduling policies, prompt recalculation of ranks and a reordering of the remaining packets in the queue. A packet scheduler should be *efficient* by performing a minimal number of operations on packet enqueue and dequeue thus enabling the handling of packets at high rates. It should also be *flexible* by providing the necessary abstractions to implement as many scheduling policies as possible. Schedulers can be *work-conserving*, meaning a packets will be dequeued back-to-back as long as there scheduler is occupied, or *non-work-conserving*, where packets are dequeued according to some inter-departure timing constraints dictated by traffic pacing or shaping requirements.

Network functions, including scheduling, are implemented at many points on the path of a packet. End hosts, switches, and programmable middleboxes are currently the primary locations where network functions are implemented. In modern networks, hardware and software implementation of network functions both play an important role [20]. Despite hardware implementation of network functionality being faster than its corresponding software implementation, our focus in this dissertation on software implementation of scheduling is motivated by its several advantages. First, software has a short development cycle

as well as flexibility in development and deployment, making it an attractive replacement or precursor for functionality implementation in hardware. This velocity allows network operators to experiment with new functions and policies rapidly at a significantly lower cost. Second, memory and processing capacity available to hardware implementations typically lag behind behind network needs. For instance, three years ago, network needs were estimated to be in the tens of thousands of rate limiters [21] while hardware network cards offered 10-128 queues [22]. Third, software network functions has the promise of "build once, deploy many". In particular, an efficient software implementation of a network function can be deployed in multiple platforms and locations, including middleboxes as Virtual Network Functions and end hosts (e.g., implementation based on BESS [19], or OpenVSwitch [23]).

A major challenge is enforcing policies accurately. For instance, as networks run at higher levels of utilization, accurate rate limiting to a target rate is increasingly important to efficient network operation. Inaccurate rate limiting causes bursty transmissions from a flow's target rate which can lead to: i) packet loss, ii) less accurate bandwidth calculations for competing flows, and iii) increasing round trip times. Packet loss reduces goodput and confuses transport protocols attempting to disambiguate fair-share available-capacity signals from bursty traffic sources. One could argue that deep buffers are the solution, but we find that the resulting increased latency leads to poor experience for users. Worse, high latency reduces application performance in common cases where compute is blocked on, for example, the completion of an RPC. Similarly, the performance of consistent storage systems is dependent on network round trip times.

Another challenge is developing a CPU-efficient implementation of even the simplest scheduling policies. For instance, kernel packet pacing can cost CPU utilization of up to 10% [24] and up to 12% for hierarchical weighted fair queuing scheduling in NetIOC of VMware's hypervisor [25]. This overhead will only grow as more programmability is added to the scheduler, assuming basic building blocks remain the same (e.g., Open-

Queue [26]). The inefficiency of the discussed systems stems from relying on $O(\log n)$ comparison-based priority queues. At a fundamental level, a scheduling policy that has m ranking functions associated with a packet (e.g., pacing rate, policy-based rate limit, weight-based share, and deadline-based ordering) typically requires m priority queues in which this packet needs to be enqueued and dequeued [27], which translates roughly to $O(m \log n)$ operations per packet for a scheduler with n packets enqueued. This simplifies the challenge to the reduction of this overhead to $O(m)$ for any scheduling policy (i.e., constant overhead per ranking function).

These challenges are exacerbated by the need for coexistence between different types of traffic at in the same software stack. This requires network operators to define complex policies that cater to the needs to individual network tenants. This requirement fueled the need for programmable schedulers as more sophisticated policies are required of networks [28, 29] with schedulers deployed at multiple points on a packet's path. For instance, Inter-datacenter (WAN) and intra-datacenter (LAN) traffic will typically share their traffic source as well as network bottlenecks. Yet, they are generally different both in the nature of the applications using them and the network equipment carrying their traffic. WAN traffic generally faces large RTTs and jitter in latency, while having small per flow throughput. Furthermore, WAN equipment, used in access networks, is typically expensive with deep buffers. On the other hand, LAN equipment is typically more cost effective with shallow buffers, where traffic is expected to have very small RTTs. This divergence between WAN and LAN traffic motivated the development of customized congestion control protocols tailored for the requirements of each type of traffic. Intra-datacenter protocols, including DCTCP [30], TIMELY [31], and DCQCN [32], aim at estimating queue lengths and maintaining short queues. While WAN congestion control protocols (e.g., BBR [33], and Copa [34]) are typically more aggressive, or have a "competitive mode, to provide robustness to loss and avoid long ramp up times caused by long RTTs. This aggressive behavior is typically motivated by the need to compete with buffer filling protocols (e.g., CUBIC [35]

and NewReno [36]) and is enabled by the deeper buffers supported by WAN equipment. However, WAN and LAN traffic share their first few hops which, by design, allows WAN traffic to get more bandwidth compared to LAN traffic due to its aggressive nature. This problem can be generalized to isolation between different types of traffic to avoid cases where bandwidth allocated to one type of traffic is mandated by its transmission strategy rather than policies defined by the network operator. Enforcing such isolation is typically achieved through careful scheduling at the traffic source based on network feedback.

1.1 Primary Contributions

This dissertation considers the problem of designing efficient and flexible networks scheduling systems. My thesis is that *it is possible to implement programmable network schedulers in software accurately and efficiently, if the underlying data structures are optimized and the proper network signals are used*. Optimized data structures, tailored to the packet scheduling use case, allow for CPU efficiency in software. This further improves scheduler programmability by providing an efficient building block that enable more complex policies. Furthermore, network signals (e.g., congestion notifications) allow for more accurate scheduling, based on network state.

In combining optimized data structures and network signals, we develop scheduling systems that overcomes the aforementioned challenges. We present three systems, aiming at a three different scheduling objectives: non-work conserving scheduling, generic programmable packet scheduling, and near-source bottleneck scheduling from the end host. In particular, we make the following contributions, summarized in §1.2, §1.3, and §1.4:

- **Scalable Traffic Shaping:** We address the challenge of rate limiting tens of thousand of flows while minimizing CPU and memory overhead of the rate limiting system. The challenge is exacerbated by the requirement of avoiding packet drops and head of line blocking. This challenge is addressed through the combination of three key ideas: i) a single queue shaper using time as the basis for releasing packets, ii) fine-

grained, just-in-time freeing of resources in higher layers coupled to actual packet departures, and iii) one shaper per CPU core, with lock-free coordination.

- **Scalable and Programmable Packet Scheduling:** We address two challenges: i) developing data structures that can be employed for any scheduling algorithm providing O(1) processing overhead per packet, and ii) provide a fully expressive scheduler programming abstraction by augmenting state of the art models. We achieve this by exploiting underlying features of packet ranking; namely, packet ranks are integers and, at any point in time, fall within a limited range of values.

- **A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates:** We address the challenge caused by the disparity in reaction between WAN congestion control and datacenter LAN congestion control. This challenge manifests at near-source bottlenecks shared by both types of traffic. This challenge is addressed by adding a second congestion control loop that only handles near-source congestion based on network feedback, while relying on typical WAN and LAN congestion control algorithms to handle congestion on the rest of the packet path.

This thesis work has been published in part in the following publications: [24] and [37].

1.2 Scalable Traffic Shaping

In Chapter 3, we build a CPU-efficient traffic shaping mechanism designed based on the following requirements:

1) Work compatibly with higher level congestion control mechanisms such as TCP. This requirements implies that the shaper should pace packets correctly to avoid bursts and unnecessary delays. It also requires the shaper to provide backpressure and avoid packet drops. Delaying packets because of rate limiting should quickly result in slowing down the end application. Each additional packet generated by the application will otherwise need to be buffered or dropped in the shaper, wasting memory and CPU to either queue or regener-

ate the packet. Additionally, loss-based congestion control algorithms often react to packet loss by significantly slowing down the corresponding flows, requiring more time to reach the original bandwidth. Finally, the shaper should avoid head of line blocking by making sure that shaping packets belonging to a traffic aggregate does not delay packets from other aggregates.

2) Use CPU and memory efficiently: Rate limiting should consume a very small portion of the server CPU and memory. This means avoiding the processing overhead of multiple queues, as well as the associated synchronization costs across CPUs in multi-core systems.

We find that these requirements are satisfied by a shaper based on three simple tenets that are the basis of our shaper design.

1.2.1 Single Queue Shaping

Relying on a single queue to shape packets alleviates the inefficiency and overhead with multi-queue systems which use token buckets, because these require a queue per rate limit. We employ a single queue indexed by time, e.g., a Calendar Queue [38] or Timing Wheel [39]. We insert all packets from different flows into the same queue and extract them based on their transmission schedule. We set a send-time timestamp for each packet, based on a combination of its flows rate limit, pacing rate, and bandwidth sharing policy.

1.2.2 Deferred Completions

The sender must limit the number of packets in flight per flow, potentially using existing mechanisms such as TCP Small Queues or the congestion window. Consequently, there must be a mechanism for the network stack to ask the sender to queue more packets for a specific flow, e.g., completions or acknowledgments. With appropriate backpressure such as our proposed *Deferred Completions*, we reduce both head of line blocking and memory pressure. Deferred Completions is a generalization of the layer 4 mechanism of TCP Small Queues to be a more general layer 3 software switch mechanism in a hypervisor/container

software switch.

1.2.3 Silos of one shaper per-core

Siloing the single queue shaper to a core alleviates the CPU inefficiency that results from locking and synchronization. We scale to multi-core systems by using independent single-queue shapers, one shaper per CPU. Note that a system can have as few as one shaper assigned to a specific CPU, i.e. not every CPU requires a shaper. To implement shared rates across cores we use a re-balancing algorithm that periodically redistributes the rates across CPUs.

1.3 Scalable and Programmable Packet Scheduling

In Chapter 4, we explain our approach to providing both flexibility and efficiency in software packet schedulers is two fold. We start by developing efficient priority queues tailored for the task of ordering packets. We observe that packet ranks can be represented as integers that at any point in time fall within a limited window of values. However, existing software schedulers do not make use of this property and rely on generic comparison-based priority queues with a worst case performance of $O(\log n)$. We exploit this property to employ integer priority queues that have $O(1)$ overhead for packet insertion and extraction. We achieve this by proposing a modification to priority queues based on the Find First Set (FFS) instruction, found in most CPUs, to support a wide range of policies and ranking functions efficiently. We also propose a new approximate priority queue that can outperform FFS-based queues for some scenarios. Second, we observe that packet scheduling programming models (i.e., PIFO [29] and OpenQueue [26]) do not support per-flow packet scheduling nor do they support reordering of packets on a dequeue operation. We augment the PIFO scheduler programming model to capture these two abstractions.

1.4 Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates

In Chapter 5, we explore the interaction between, independently congestion controlled, LAN and WAN traffic. In particular, we look at cases when the two types of traffic co-exist in the same datacenter network. We base our findings on measurements at a large cloud operator over a period of one month. We find that the difference in congestion control schemes and traffic characteristics cause the WAN traffic to negatively impact the performance of the LAN traffic. Such cases occur when the type types of traffic share a bottleneck. We also observe that such bottlenecks are typically close to the traffic source.Based on our observations, we design, implement and evaluate Annulus, a single congestion control approach that is designed to handle the two types of traffic simultaneously. Annulus achieves this by deploying two interacting congestion control loops: one handles near-source congestion leveraging local network feedback, while the other handles congestion on the rest of the path using conventional LAN and WAN congestion control schemes.

CHAPTER 2

RELATED WORK

2.1 Traffic Shaping

In this dissertation, we use traffic *shaping* broadly to refer to either pacing or rate limiting, where *pacing* refers to injecting inter-packet gaps to smooth traffic within a single connection, and *rate limiting* refers to enforcing a rate on a flow-aggregate consisting of one or more individual connections. VirtualClock [40] seminal work relies on transmission time to control traffic. The focus in VirtualClock is on setting timestamps for specific rates that can be specified by a network controller. Since then, several developments have been made in improving the efficiency of traffic shapers. SENIC and vShaper introduce hybrid software/hardware shapers to reduce CPU overhead. SENIC allows the hosts CPU to classify and enqueue packets while the NIC is responsible for pulling packets when it is time to transmit them [21]. vShaper allows for sharing queues across different flows by estimating the share each will take based on their demand [41]. Pacing was first suggested to have a per-flow timer which made pacing unattractive due to the excessive CPU overhead [42]. FQ/pacing is a linux kernel Queuing Discipline (Qdisc) that uses only one timer and still imposes non-trivial CPU overhead.

2.2 Flexibility of Programmable Packet Schedulers

There has been recent interest in developing flexible, programmable, packet schedulers [29, 26]. This line of work is motivated by the support for programmability in all aspects of modern networks. Specifically, several packet scheduling algorithms have been proposed each aiming at achieving a specific and diverse set of objectives. Some of such policies include per flow rate limiting [24], hierarchical rate limiting with strict or relative

prioritization [43, 44], optimization of flow completion time [11], and joint network and CPU scheduling [18], several of which are used in practice.

Work on programmable schedulers focuses on providing the infrastructure for network operators to define their own scheduling policies. This approach improves on the current standard approach of providing a small fixed set of scheduling policies as currently provided in modern switches. A programmable scheduler provides building blocks for customizing packet ranking and transmission timing. Proposed programmable schedulers differ based on the flexibility of their building blocks. A flexible scheduler allows a network operator to specify policies according to the following specifications:

- *Unit of Scheduling:* Scheduling policies operate either on per packet basis (e.g., pacing) or on per flow basis (e.g., fair queuing). This requires a model that provides abstractions for both.
- *Work Conservation:* Scheduling policies can also be work-conserving or non-work-conserving.
- *Ranking Trigger:* Efficient implementation of policies can require ranking packets on their enqueue, dequeue, or both.

Recent programmable packet schedulers export primitives that enable the specification of a scheduling policy and its parameters, often within limits. The PIFO scheduler programming model is the most prominent example [29]. It is implemented in hardware relying on Push-In-First-Out (PIFO) building blocks where packets are ranked only on enqueue. The scheduler is programmed by arranging the blocks to implement different scheduling policies. Due to its hardware implementation, the PIFO model employs compact constructs with considerable flexibility. However, PIFO remains very limited in its capacity (i.e., PIFO can handle a a maximum of 2048 flows at line rate), and expressiveness (i.e., PIFO can't express per flow scheduling). OpenQueue is an example of a flexible programmable packet scheduler in software [26]. However, the flexibility of OpenQueue comes at the expense of having three of its building blocks as priority queues, namely queues, buffers, and

Table 2.1: Summary of the state of the art in scheduling

System	Efficiency	HW/SW	Flexibility			
			Unit of Scheduling	Work-Conserving	Supports Shaping	Programmable
FQ/Pacing qdisc [45]	$O(\log n)$	SW	Flows	No	**Yes**	No
hClock [43]	$O(\log n)$	SW	Flows	**Yes**	**Yes**	No
Carousel [24]	$O(1)$	SW	Packets	No	**Yes**	No
OpenQueue [26]	$O(\log n)$	SW	Packets & Flows	**Yes**	No	On enq/deq
PIFO [29]	$O(1)$	HW	Packets	**Yes**	**Yes**	On enq

ports. This overhead, even in the presence of efficient priority queues, will form a memory and processing overhead. Furthermore, OpenQueue does not support non-work-conserving schedules. Table 2.1 summarizes the discussed related work.

2.3 Congestion Control

Data Center TCP (DCTCP) [30], HULL [46], D2TCP [47] rely on Explicit Congestion Notification (ECN) marks aggregated over several packets due to obtain fine-grained congestion information. TCP Bolt [48] also relies on ECN and is essentially DCTCP without slow start. However, averaging ECN marks to derive congestion information over several RTTs could be slow and lead to delayed reaction to congestion. Further, ECN marking is generally not supported end-to-end outside the datacenter fabric and in fact, the ECN marks could be overwritten at external switches. This makes ECN based techniques unsuitable for dealing with congestion at the datacenter edge. TIMELY [31] and DCQCN [32] are two recent proposals for congestion control for RDMA deployments. While TIMELY relies on accurate RTT measurements, DCQCN relies on ECN marks as a proxy for QCN feedback. However, such end-to-end congestion methods may require several RTTs to converge and may lead to packet losses especially for small flows. Further, TIMELY cannot be supported without specialized hardware to guarantee high fidelity RTT information.

ICMP Source Quench [49] relies on notifications from switches for managing congestion. However, due to challenges associated with practical deployments, its deployment never took off and has been recently deprecated. FastLane [50] leverages in-network notifications to avoid packet losses for small flows but it is not a congestion control scheme

and it requires changes to switches. Several proposals have attempted to reduce latency of TCP flows in datacenter and other environments. Some examples include pFabric [11], Detail [51], Fastpass [52], RCP [53]. These proposals require significant changes to switch architectures, and to our knowledge, are not being deployed on a large scale. Congestion control for datacenter-edge links has not been widely addressed. Most of the existing schemes rely on minimizing congestion at inter-datacenter or WAN links through traffic engineering, e.g., SWAN [13] and B4 [54]. TCP Fast Open [55] proposes techniques to reduce completion time of short flows over long RTTs by enabling data exchange during the handshake phase.

2.4 Overview of Priority Queuing

Priority queues are the main data structure needed for scheduling (i.e., ordering elements). A priority queue maintains a list of elements, each tagged with a priority value. A priority queue supports one of two operations efficiently: `ExtractMin` or `ExtractMax` to get the element with minimum or maximum priority respectively. In this section, we give a brief overview of theoretical results on priority queue complexity as well as modifications and customization applied to such priority queues for networking and systems applications.

2.4.1 Exact priority queuing

A well-known efficient priority queuing data structure is the van Emde Boas tree [56]. The tree requires $O(\log \log C)$ for both `Insert` and `ExtractMin`, where C is the number of buckets. The van Emde Boas tree is constructed on $O(C)$ space as a recursive data structure with each component having $O(\sqrt{c})$ elements, where c is the number of elements of its parent. A recursive data structure incurs higher memory overhead, compared to linear data structures, due to the pointer chasing problem [57]. Another example is Fusion trees which are B-tree-like with branching factor $w^{\frac{1}{5}}$ where w is the word size used to represent the priorities [58]. Fusion trees work in cases where the maximum value represented by a word

of width w is much larger than the word width needed to represent priority values. Hence, multiple priority values can be "encoded" and concatenated to fit in a single word allowing fast operations on multiple priority values in parallel. This encoding process converts a priority to a path in a binary tree to the location of the element. The decoding of the occupancy was shown to be worst case $O(\log_w C)$ while only n space is required. However, updating the tree is complex and costly as it requires reconfiguration of the encoded values. The result by Thorup shows that a data structure can be constructed that has $O(\log \log n)$ for both `Insert` and `ExtractMin` while requiring $O(n)$ space [59]. However, such theoretical data structures are complicated to implement.

2.4.2 Approximate priority queuing

Another way to trade off accuracy for efficiency relies on the use of approximate data structures that are then used for generic purposes. For instance, Soft-heap [60] is an approximate priority queue with a bounded error that is inversely proportional to the overhead of insertion. In particular, after n insertions in a soft-heap with an error bound $0 < \epsilon \leq 1/2$, the overhead of insertion is $O(\log(1/\epsilon))$. Hence, `ExtractMin` operation which can have a very large error under Soft-heap. Another example is the RIPQ which is was developed for caching [61]. RIPQ relies on a bucket-sort-like approach. However, the RIPQ implementation is suited for static caching, where elements are not moved once inserted, which makes it not very suitable for the dynamic nature of packet scheduling.

2.4.3 Customized Data Structures for Packet Scheduling

Recently, there has been some attempts to employ data structures specifically developed or re-purposed for efficiently implementing specific packet scheduling algorithms. For instance, Carousel [24], a system developed for rate limiting at scale, relies on Timing Wheel [39], a data structure that can support time-based operations in $O(1)$ and requires comparable memory to our proposed approach. However, Timing Wheel supports only

non-work conserving time-based schedules in $O(1)$. Timing Wheel is efficient as buckets are indexed based on time and elements are accessed when their deadline arrives. However, Timing Wheel does not support operations needed for non-work conserving schedules (i.e., `ExtractMin` or `ExtractMax`). Another example is efficient approximation of popular scheduling policies (e.g., Start-Time Fair Queueing [62] as an approximation of Weighted Fair Queuing [63], or the more recent Quick Fair Queuing (QFQ) [64]).

2.4.4 Hardware Priority Queues in Networks

Scheduling is supported in hardware switches using a short list of scheduling policies, including shaping, strict priority, and Weighted Round Robin [65, 66, 67, 29]. Another approach in hardware packet scheduling relies on pipelined-heaps [68, 69, 70]. These rely on configuring a small number of piplined-stages for enqueuing and dequeuing elements in a priority queue. This approach is useful for implementing scheduling algorithms that require keeping a sorted list of a large number of packets (e.g., earliest deadline first-based algorithms [71]). However, such approaches are not immediately applicable to software because it relies on hardware optimization that allow for pipelining operations of the queue in hardware. Furthermore, they typically assume operating over the whole range of possible priority values (e.g., p-heap in [68] supports 4 billion entries). However, in software it is typically preferable to operate over small moving ranges of priority values for CPU and memory efficiency.

CHAPTER 3

CAROUSEL: SCALABLE TRAFFIC SHAPING AT END HOSTS

Traffic shaping, including pacing and rate limiting, is fundamental to the correct and efficient operation of both datacenter and wide area networks. Sample use cases include policy-based bandwidth allocation to flow aggregates, rate-based congestion control algorithms, and packet pacing to avoid bursty transmissions that can overwhelm router buffers. Driven by the need to scale to millions of flows and to apply complex policies, traffic shaping is moving from network switches into the end hosts, typically implemented in software in the kernel networking stack.

While traffic shaping has historically targeted wide area networks, two recent trends bring it to the forefront for datacenter communications, which use end-host based shaping. The first trend is the use of fine grained pacing by rate-based congestion control algorithms such as BBR [33] and TIMELY [31]. BBR and TIMELY's use of rate control is motivated by studies that show pacing flows can reduce packet drops for video-serving traffic [72] and for incast communication patterns [31]. Incast arises from common datacenter applications that simultaneously communicate with thousands of servers. Even if receiver bandwidth is allocated perfectly among senders at coarse granularity, simultaneous bursting to NIC line rate from even a small subset of senders can overwhelm the receiver's network capacity.

Furthermore, traffic from end hosts is increasingly bursty, due to heavy batching and aggregation. Techniques such as NIC offloads optimize for server CPU efficiency — e.g., Transmission Segmentation Offload [73] and Generic Receive Offload [74]. Pacing reduces burstiness, which in turn reduces packet drops at shallow-buffered switches [75] and improves network utilization [42]. It is for these reasons that BBR and TIMELY rely on pacing packets as a key technique for precise rate control of thousands of flows per server.

The second trend is the need for network traffic isolation across competing applications

or Virtual Machines (VMs) in the Cloud. The emergence of Cloud Computing means that individual servers may host hundreds or perhaps even thousands of VMs. Each virtual endpoint can in turn communicate with thousands of remote VMs, internal services within and between datacenters, and the Internet at large, resulting in millions of flows with overlapping bottlenecks and bandwidth allocation policies. Providers use predictable and scalable bandwidth arbitration systems to assign rates to flow aggregates, which are then enforced at end systems. Such an arbitration can be performed by centralized entities, e.g., Bandwidth Enforcer [12] and SWAN [13], or distributed systems, e.g., EyeQ [76].

For example, consider 500 VMs on a host providing Cloud services, where each VM communicates on average with 50 other virtual endpoints. The provider, with no help from the guest operating system, must isolate these 25K VM-to-endpoint flows, with bandwidth allocated according to some policy. Otherwise, inadequate network isolation increases performance unpredictability and can make Cloud services unavailable [76, 77].

Traditionally, network switches and routers have implemented traffic shaping. However, inside a datacenter, shaping in middleboxes is not an easy option. It is expensive in buffering and latency, and middleboxes lack the necessary state to enforce the right rate. Moreover, shaping in the middle does not help when bottlenecks are at network edges, such as the host network stack, hypervisor, or NIC. Finally, when packet buffering is necessary, it is much cheaper and more scalable to buffer near the application.

Therefore, the need to scale to millions of flows *per server* while applying complex policy means that traffic shaping must largely be implemented in end hosts. The efficiency and effectiveness of this end-host traffic shaping is increasingly critical to the operation of both datacenter and wide area networks. Unfortunately, existing implementations were built for very different requirements, e.g., only for WAN flows, a small number of traffic classes, modest accuracy requirements, and simple policies. Through measurement of video traffic in a large-scale Cloud provider, we show that the performance of end-host traffic shaping is a primary impediment to scaling a virtualized network infrastructure. Ex-

isting end-host rate limiting consumes substantial CPU and memory; e.g., shaping in the Linux networking stack use 10% of server CPU to perform pacing (§3.2); shaping in the hypervisor unnecessarily drops packets, suffers from head of line blocking and inaccuracy, and does not provide backpressure up the stack (§3.2).

In this chapter, we present the design and implementation of Carousel, an improvement on existing, kernel-based traffic shaping mechanism. Carousel scales to tens of thousands of flows and traffic classes, and supports complex bandwidth-allocation mechanisms for both WAN and datacenter communications. We design Carousel around three core techniques: i) a single queue shaper using time as the basis for scheduling, specifically, Timing Wheel [39], ii) coupling actual packet transmission to the freeing of resources in higher layers, and iii) each shaper runs on a separate CPU core, which could be as few as one CPU. We use lock-free coordination across cores. Our production experience in serving video traffic at a Cloud service provider shows that Carousel shapes traffic effectively while improving the overall machine CPU utilization by 8% relative to state-of-art implementations, and with an improvement of 20% in the CPU utilization attributed to networking. It also conforms 10 times more accurately to target rates, and consumes two orders of magnitude less memory than existing approaches. While we implement Carousel in a software NIC, it is designed for hardware offload. By the novel combination of previously-known techniques, Carousel makes a significant advance on the state-of-the-art shapers in terms of efficiency, accuracy and scalability.

3.1 Traffic Shapers in Practice

In the previous section we established the need for at-scale shaping at end hosts: first, modern congestion control algorithms, such as BBR and TIMELY, use pacing to smooth bursty video traffic, and to handle large-scale incasts; second, traffic isolation in Cloud is critically dependent on efficient shapers. Before presenting details of Carousel, we first present an overview of the shapers prevalently used in practice.

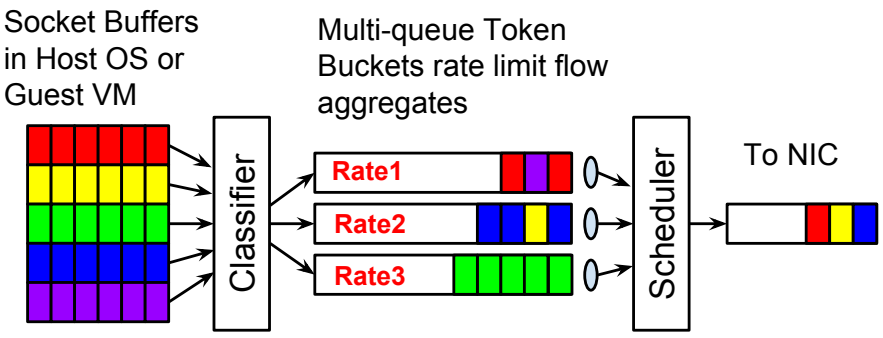


Figure 3.1: Token Bucket Architecture: pre-filtering with multiple token bucket queues.

Nearly all of rate limiting at end hosts is performed in software. Figure 3.1 shows the typical rate limiter architecture, which we broadly term as *pre-filtering with multiple token bucket queues*. It relies on a *classifier*, *multiple queues*, *token bucket shapers* and/or a *scheduler* processing packets from each queue. The classifier divides packets into different queues, each queue representing a different traffic aggregate.[1] A queue has an associated traffic shaper that paces packets from that queue as necessary. Traffic shapers are synonymous with token buckets or one of its variants. A scheduler services queues in round-robin order or per service-class priorities.

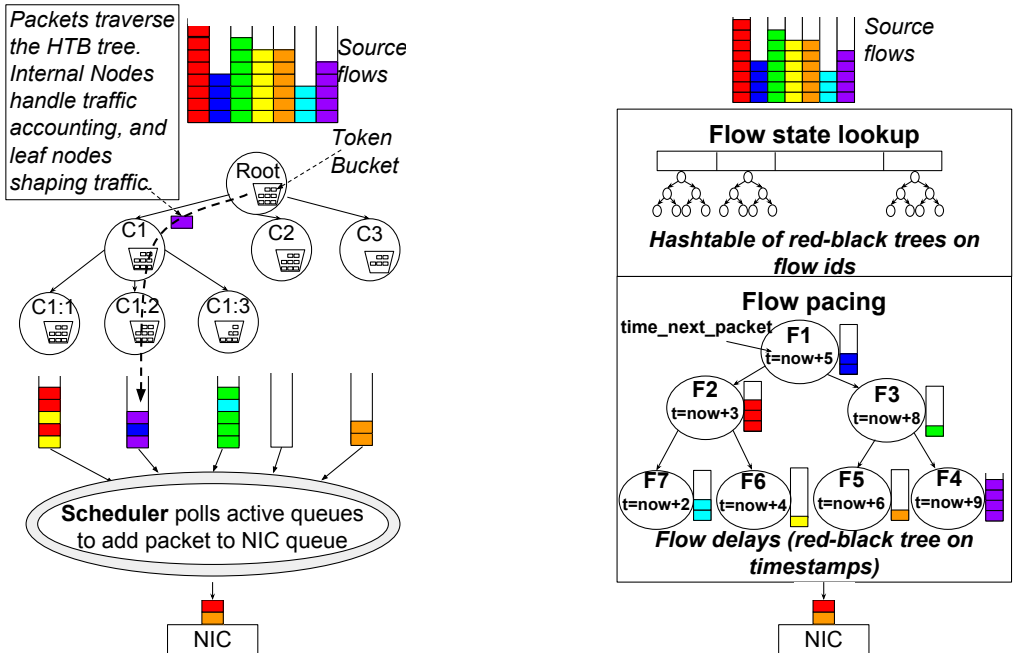
This design avoids head of line blocking, through a separate queue per traffic aggregate: when a token bucket delays a packet, all packets in the same queue will be delayed, but not packets in other queues. Other mechanisms, such as TCP Small Queues (TSQ) [78] in Linux, provide backpressure and reduce drops in the network stack.

Below, we describe three commonly used shapers in end-host stacks, hypervisors and NICs: 1) Policers; 2) Hierarchical Token Bucket; and 3) FQ/pacing. They are all based on per-traffic aggregate queues and a token bucket associated with each queue for rate limiting.

3.1.1 Policers

Policers are the simplest way to enforce rates. Policer implementations use token buckets with zero buffering per queue. A token bucket policer consists of a counter representing

[1]Traffic aggregate refers to a collection of individual TCP flows being grouped on a shaping policy.



(a) HTB (b) FQ/pacing

Figure 3.2: Architecture of Linux rate limiters.

the number of currently available tokens. Sending a packet requires checking the request against the available tokens in the queue. The policer drops the packet if too few tokens are available. Otherwise, the policer forwards the packet, reducing the available tokens accordingly. The counter is refilled according to a target *rate* and capped by a maximum *burst*. The rate represents the average target rate and the burst size represents the tolerance for jitter, or for short-term deviation from the target rate conformance.

3.1.2 Hierarchical Token Bucket (HTB)

Unlike Policers that have zero queueing, traffic shapers buffer packets waiting for tokens, rather than drop them. In practice, related token buckets are grouped in a complex structure to support advanced traffic management schemes. Hierarchical Token Bucket (HTB) [44] in the Linux kernel Queueing Discipline (Qdisc) uses a tree organization of shapers as shown in Figure 3.2a. HTB classifies incoming packets into one of several traffic classes, each associated with a token bucket shaper at a leaf in the tree. The hierarchy in HTB

is for borrowing between leaf nodes with a common parent to allow for work-conserving scheduling.

The *host enforcer* in the Bandwidth Enforcer (BwE) [12] system is a large-scale deployment of HTB. Each HTB leaf class has a one-to-one mapping to a specific BwE task flow group – typically an aggregate of multiple TCP flows. The number of HTB leaf classes is directly proportional to the number of QoS classes $\times$ destination clusters $\times$ number of tasks.

To avoid unbounded queue growth, shapers should be used with a mechanism to backpressure the traffic source. The simplest form of backpressure is to drop packets, however, drops are also a coarse signal to a transport like TCP. Linux employs more fine grained backpressure. For example, HTB Qdisc leverages TCP Small Queues (TSQ) [78] to limit two outstanding packets for a single TCP flow within the TCP/IP stack, waiting for actual NIC transmission before enqueuing further packets from the same flow. [2]. In the presence of flow aggregates, the number of enqueued packets equals the number of individual TCP flows $\times$ TSQ limit. If HTB queues are full, subsequent arriving packets will be dropped.

3.1.3 FQ/pacing

FQ/pacing [45] is a Linux Qdisc used for packet pacing along with fair queueing for egress TCP and UDP flows. Figure 3.2b shows the structure of FQ/pacing. The FQ scheduler tracks per-flow state in an array of Red-Black (RB) trees indexed on flow hash IDs. A deficit round robin (DRR) scheduler [79] fetches outgoing packets from the active flows. A garbage collector deletes inactive flows.

Maintaining per-flow state provides the opportunity to support egress pacing of the active flows. In particular, a TCP connection sends at a rate approximately equal to cwnd/RTT, where cwnd is the congestion window and RTT is the round-trip delay. Since cwnd and RTT are only best-effort estimates, the Linux TCP stack conservatively sets the flow pac-

[2]The limit is controlled via a knob whose default value of 128KB yields two full-sized 64KB TSO segments.

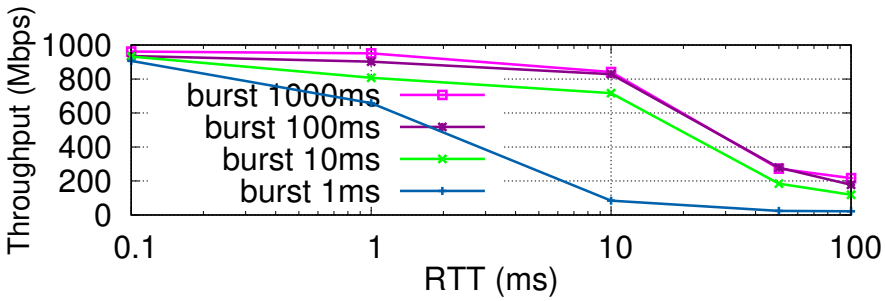


Figure 3.3: Policing performance for target throughput 1Gbps with different burst sizes. Policers exhibit poor rate conformance, especially at high RTT. The different lines in the plot correspond to the configured burst size in the token bucket.

ing rate to 2 $\times$ cwnd/RTT [80]. FQ/pacing enforces the pacing rate via a leaky bucket queue [81], where sending timestamps are computed from packet length and current pacing rate. Flows with next-packet timestamps far into the future are kept in a separate RB tree indexed on those timestamps. We note that pacing is enforced at the granularity of TCP Segmentation Offload (TSO) segments.

The Linux TCP stack further employs the pacing rate to automatically size packets meant for TCP Segmentation Offloading. The goal is to have at least one TSO packet every 1ms, to trigger more frequent packet sends, and hence better acknowledgment clocking and fewer microbursts for flows with low rates. TSO autosizing, FQ, and pacing together achieve traffic shaping in the Linux stack [82].

In practice, FQ/pacing and HTB are used either individually or in tandem, each for its specific purpose. HTB rate limits flow aggregates, but scales poorly with the number of rate limited aggregates and the packet rate (§3.2). FQ/pacing is used for pacing individual TCP connections, but this solution does not support flow aggregates. We use Policers in hypervisor switch deployments where HTB or FQ/pacing are too CPU intensive to be useful.

3.2 The Cost of Shaping

Traffic shaping is a requirement for the efficient and correct operation of production networks, but uses CPU cycles and memory on datacenter servers that can otherwise be used for applications. In this section, we quantify these tradeoffs at scale in a large Cloud service.

Policiers: Among the three shapers, Policers are the simplest and cheapest. They have low memory overhead since they are bufferless, and they have small CPU overhead because there is no need to schedule or manage queues. However, Policers have poor conformance to the target rate. Figure 3.3 shows that as round-trip time (RTT) increases, Policers deviate by 10x from the target rate for two reasons: first by dropping non-conformant packets, Policers lose the opportunity to schedule them at a future time, and resort to emulating a target rate with on/off behavior. Second, Policers trigger poor behavior from TCP, where even modest packet loss leads to low throughput [83] and wasted upstream work.

To operate over a large bandwidth-delay product, Policers require a large configured burst size, e.g., on the order of the flow round-trip time. However, large bursts are undesirable, due to poor rate conformance at small time scales, and tail dropping at downstream switches with shallow buffers. Even with a burst size as large as one second, Figure 3.3 shows that the achieved rate can be 5x below the target rate, for an RTT of 100ms using TCP CUBIC.

Pacing: FQ/pacing provides good rate conformance, deviating at most 6% from the target rate (§3.6), but at the expense of CPU cost. To understand the impact on packet loss and CPU usage, we ran an experiment on production video servers where we turn off FQ/pacing on 10 servers, each serving 37Gbps at peak across tens of thousands flows. We instead configured a simple PFIFO queueing discipline typically used as the default low overhead Qdisc [84]. We ensured that the experiment and baseline servers have the same machine configuration, are subject to similar traffic load, including being under the same

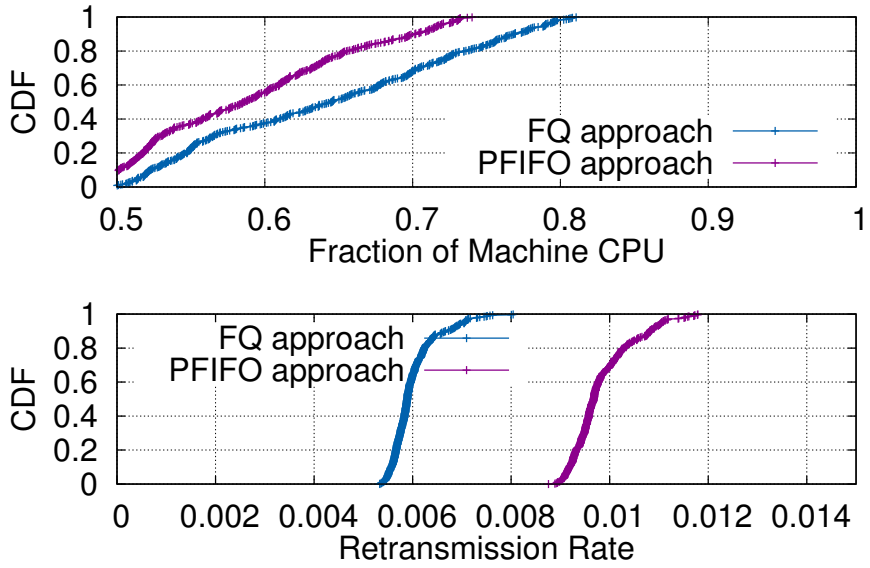


Figure 3.4: Impact of FQ/pacing on a popular video service. Retransmission rate on paced connections is 40% lower than that on non-paced connections (lower plot). Pacing consumes 10% of total machine CPU at the median and the tail (top plot). The retransmission rates and CPU usage are recorded over 30 second time intervals over the experiment period.

load balancers, and ran the experiments simultaneously for one week.

Figure 3.4 compares the retransmission rate and CPU utilization with and without FQ/-pacing (10 servers in each setting). The upside of pacing is that the retransmission rate on paced connections is 40% lower than those on non-paced connections. The loss-rate reduction comes from reducing self-inflicted losses, e.g., TSO or large congestion windows. The flip side is that pacing consumes 10% of total CPU on the machine at the median and the tail (90th and 99th percentiles); e.g., on a 64 core server, we could save up to 6 cores through more efficient rate limiting.

To demonstrate that the cost of pacing is not restricted to FQ/pacing Qdisc or its specific implementation, we conduct an experiment with QUIC [85] running over UDP traffic. QUIC is a transport protocol designed to improve performance for HTTPS traffic. It is globally deployed at Google on thousands of servers and is used to serve YouTube video traffic. QUIC runs in user space with a packet pacing implementation that performs MTU-sized pacing independent from the kernel's FQ/pacing. We experimented with QUIC's

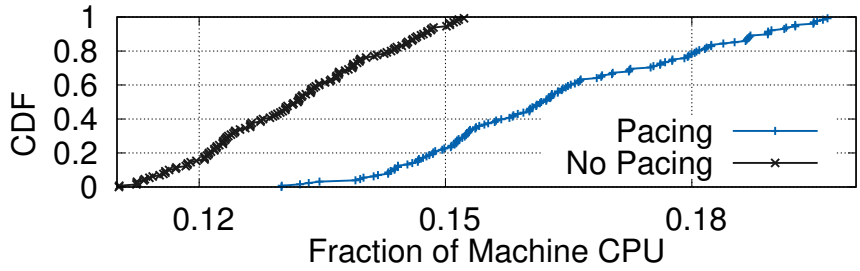


Figure 3.5: Pacing impact on a process generating QUIC traffic.

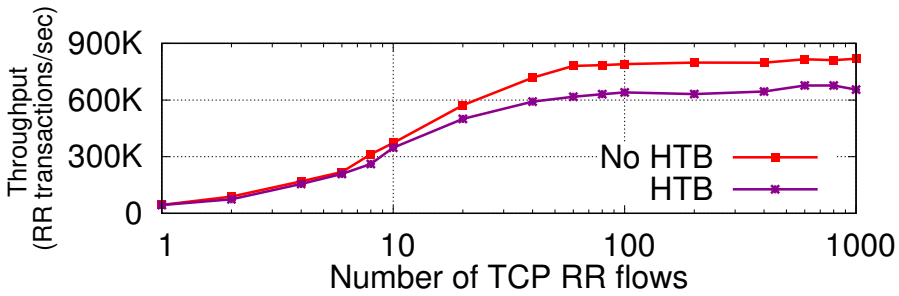
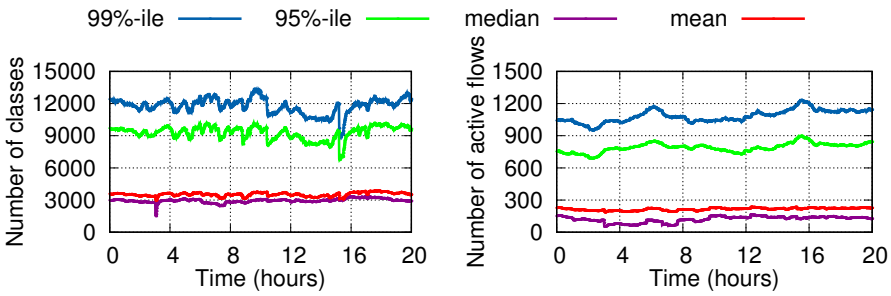


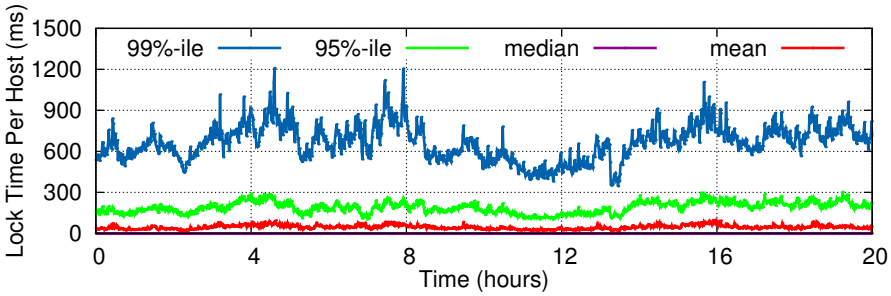
Figure 3.6: TCP-RR rate with and HTB. HTB is 30% lower due to locking overhead.

pacing turned ON and OFF. We obtained similar results with QUIC/UDP experiments as we obtained with the kernel's FQ/pacing. Figure 3.5 shows the CPU consumption of a process generating approximately 8Gbps QUIC traffic at peak. With pacing enabled, process CPU utilization at the 99th percentile jumped from 0.15 to 0.2 of machine CPU – a 30% increase, due to arming and serving timers for each QUIC session. Pacing lowered retransmission rates from 2.6% to 1.6%.

HTB: Like FQ/pacing, HTB can also provide good quality rate conformance, with a deviation of 5% from target rate (§3.6), albeit at the cost of high CPU consumption. CPU usage of HTB grows linearly with the packets per second (PPS) rate. Figure 3.6 shows an experiment with Netperf TCP-RR, request-response ping-pong traffic of 1 Byte in each direction, running with and without HTB. This experiment measures the overhead of HTB's shaping architecture. We chose the 1-byte TCP-RR because it mostly avoids overheads other than those specifically related to packets per second. In production workloads, it is desirable for shapers (and more generally networking stacks) to achieve high PPS rates



(a) Total number of HTB classes and active flows



(b) HTB lock acquire time

Figure 3.7: HTB statistics over one day from a busy production cluster.

while using minimal CPU. As the offered PPS increases in Figure 3.6 (via the increase in the number of TCP-RR connections on the x-axis), HTB saturates the machine CPU at 600K transactions/sec. Without HTB, the saturation rate is 33% higher at 800K transactions/sec.

The main reason for HTB's CPU overhead is the global Qdisc lock acquired on every packet enqueue. Contention for the lock grows with PPS. [86] details the locking overhead in Linux Qdisc, and the increasing problems posed as links scale to 100Gbps. Figure 3.7 shows HTB statistics from one of our busiest production clusters over a 24-hour period. Acquiring locks in HTB can take up to 1s at the 99^{th} percentile, directly impacting CPU usage, rate conformance, and tail latency. The number of HTB classes at any instant on a server is tens of thousands at the 90^{th} percentile, with 1000-2000 actively rate limited.

Memory: There is also a memory cost related to traffic shaping: the memory required to queue packets at the shaper, and the memory to maintain the data structures implementing the shaper. Without backpressure to higher level transports like TCP, the memory re-

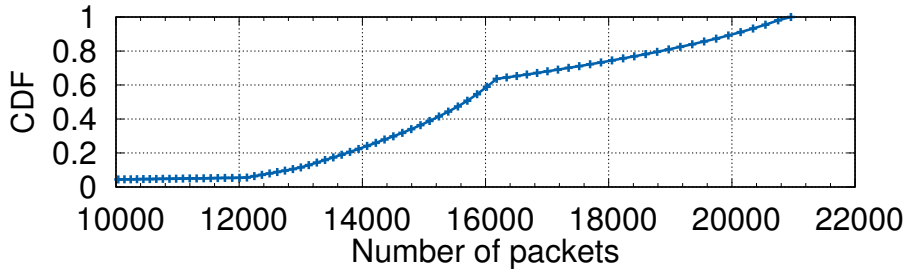


Figure 3.8: Buffered packets of a VM at a shaper in hypervisor. The bump in the curve is because there are two internal limits on the number of buffered packets: 16K is a local per-VM limit, and 21K is at the global level for all VMs. There are packet drops when the 16K limit is exceeded, which impacts the slope of the curve.

quired to queue packets grows with the number of flows $\times$ congestion window, and reaches a point where packets need to be dropped. Figure 3.8 shows the number of packets at a rate limiter in the absence of backpressure for a Cloud VM that has a single flow being shaped at 2Gbps over a 50ms RTT path. In the 99^{th} percentile, the queue of the shaper has 21 thousand MTU size packets (or 32 MB of memory) because of CUBIC TCP's large congestion window, adding 120ms of latency from the additional buffering. In the presence of backpressure (§3.4), there will be at most two TSO size worth of packets or 85 MTU sized packets[3] (or 128KB of memory), i.e., two orders of magnitude less memory.

A second memory cost results from maintaining the shaper's data structures. The challenge is partitioning and constraining resources such that there are no drops during normal operation; e.g., queues in HTB classes can be configured to grow up to 16K packets, and FQ/pacing has a global limit of 10K packets and a per queue limit of 1000 packets, after which the packets are dropped even if there is plenty of global memory available. With poorly tuned configurations, packets within a queue may be dropped even while there is plenty of room in the global pool.

Summary: The problem with HTB and FQ/pacing is more than just accuracy or timer granularities. The CPU inefficiency of HTB and FQ/pacing is intrinsic to these mechanisms, and not the result of poor implementations. The architecture of these shapers is

[3]We assume maximum TSO size of 64KB and MTU size of 1.5KB.

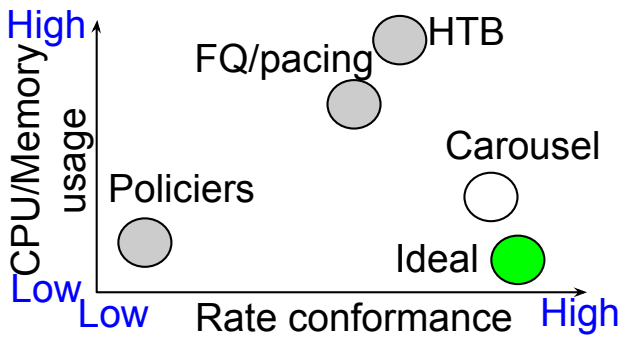


Figure 3.9: Policers have low CPU usage but poor rate conformance. HTB and FQ/pacing have good shaping properties but a high CPU cost.

based on synchronization across multiple queues and cores, which increases the CPU cost:

1) The token bucket architecture of FQ/pacing and HTB necessitates multiple queues, with one queue per rate limit to avoid head of line blocking. The cost of maintaining these queues grows, sometimes super-linearly, with the number of such queues. CPU costs stem from multiple factors, especially the need to poll queues for packets to process. Commonly, schedulers either use simple round-robin algorithms, where the cost of polling is linear in the number of queues, or more complex algorithms tracking active queues.

2) Synchronization across multiple cores: Additionally, the cost on multi-CPU systems is dominated by locking and/or contention overhead when sharing queues and associated rate limiters between CPUs. HTB and FQ/pacing acquire a global lock on a per-packet basis, whose contention gets worse as the number of CPUs increases.

Solving these problems requires a fundamentally different approach, such as the one we describe in the following sections. Today, practitioners have a difficult choice amongst rate limiters: Policers have high CPU/memory efficiency but unacceptable shaping performance. HTB and FQ/pacing have good shaping properties, but fairly high CPU and memory costs. Figure 3.9 summarizes this tradeoff. 10% CPU overhead from accurate shaping can be worth the additional network efficiency, especially for WAN transfers. Now, the question becomes whether we can get the same capability with substantially less CPU overhead.

3.3 Carousel Design Principles

Our work begins with the observation that a unified, accurate, and CPU-efficient traffic shaping mechanism can be used in a variety of important settings. Our design principles follow naturally from the following requirements:

1) Work compatibly with higher level congestion control mechanisms such as TCP.

a)**Pace packets correctly**: avoid bursts and unnecessary delays.

b) **Provide backpressure and avoid packet drops**: delaying packets because of rate limiting should quickly result in slowing down the end application. Each additional packet generated by the application will otherwise need to be buffered or dropped in the shaper, wasting memory and CPU to either queue or regenerate the packet. Additionally, loss-based congestion control algorithms often react to packet loss by significantly slowing down the corresponding flows, requiring more time to reach the original bandwidth.

c) **Avoid head of line blocking**: shaping packets belonging to a traffic aggregate should not delay packets from other aggregates.

2) Use CPU and memory efficiently: Rate limiting should consume a very small portion of the server CPU and memory. As implied by §3.2, this means avoiding the processing overhead of multiple queues, as well as the associated synchronization costs across CPUs in multi-core systems.

We find that these requirements are satisfied by a shaper based on three simple tenets:

1. Single Queue Shaping (§3.4.1): Relying on a single queue to shape packets alleviates the inefficiency and overhead with multi-queue systems which use token buckets, because these require a queue per rate limit. We employ a single queue indexed by time, e.g., a Calendar Queue [38] or Timing Wheel [39]. We insert all packets from different flows into the same queue and extract them based on their transmission schedule. We set a send-time timestamp for each packet, based on a combination of its flow's rate limit, pacing rate, and bandwidth sharing policy.

2. Deferred Completions (§3.4.2): The sender must limit the number of packets in flight per flow, potentially using existing mechanisms such as TCP Small Queues or the congestion window. Consequently, there must be a mechanism for the network stack to ask the sender to queue more packets for a specific flow, e.g., completions or acknowledgments. With appropriate backpressure such as *Deferred Completions* described in §3.4.2, we reduce both HoL blocking and memory pressure. Deferred Completions is a generalization of the layer 4 mechanism of TCP Small Queues to be a more general layer 3 software switch mechanism in a hypervisor/container software switch.

3. Silos of one shaper per-core (§3.4.3): Siloing the single queue shaper to a core alleviates the CPU inefficiency that results from locking and synchronization. We scale to multi-core systems by using independent single-queue shapers, one shaper per CPU. Note that a system can have as few as one shaper assigned to a specific CPU, i.e. not every CPU requires a shaper. To implement shared rates across cores we use a re-balancing algorithm that periodically redistributes the rates across CPUs.

Figure 3.10 illustrates the architecture using Timing Wheel and Deferred Completions. Deferred Completions addresses the first of the requirements on working compatibly with congestion control. Deferred Completions allows the shaper to slow down the source, and hence avoids unnecessary drops and mitigates head of line blocking. Additionally, Deferred Completions also allows the shaper to buffer fewer packets per flow, reducing memory requirements. The use of a single Timing Wheel per core makes the system CPU-efficient, thus addressing the second requirement.

Carousel is a more accurate and CPU-efficient than state-of-art token-bucket shapers. Carousel's architecture of a single queue per core, coupled with Deferred Completions can be implemented anywhere below the transport, in either software or NIC hardware.

To demonstrate the benefits of our approach, we explore shaping egress traffic, in §3.4 using Deferred Completions for backpressure, and shaping ingress incast traffic at the receiver, in §3.5 using ACKs deferral for backpressure.

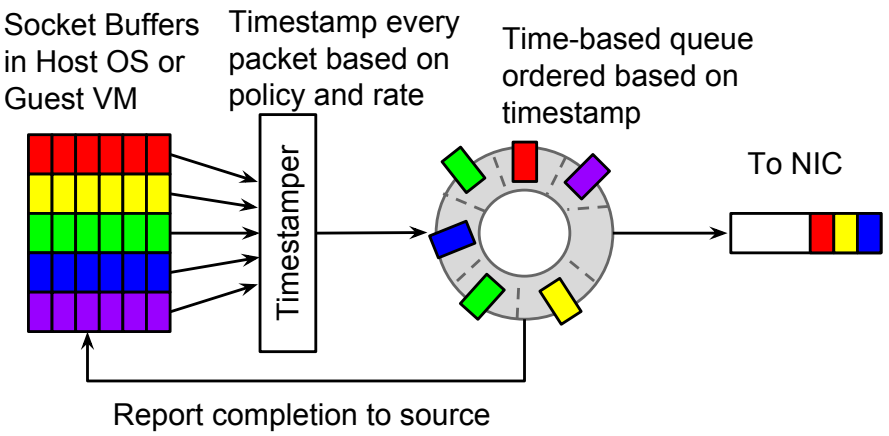


Figure 3.10: Carousel architecture.

3.4 The Carousel System

Carousel shaping consists of three sequential steps for each packet. First, the network stack computes a timestamp for the packet. Second, the stack enqueues the packet in a queue indexed by these timestamps. Finally, when the packet's transmission deadline passes, the stack dequeues the packet and delivers a completion event, which can then transmit more packets.

We implement Carousel in a software NIC in our servers. Software NICs dedicate one or more CPU cores for packet processing, as inSoftNIC and FlexNIC [19, 87]. Our software NIC operates in busy-wait polling mode. Polling allows us to check packet timestamps at high frequency, enabling accurate conformance with the packets' schedules. Polling also assists in CPU-efficient shaping, by alleviating the need for locking, and for timers for releasing packets.

We note that Software NICs are not a requirement for Carousel, and it is possible to implement Carousel in the kernel. However, an implementation in the kernel would not be a simple modification of one of the existing Queuing Disciplines [88], and would require re-engineering the kernel's Qdisc path. This is because the kernel is structured around parallelism at the packet level, with per-packet lock acquisitions, which does not scale efficiently to high packet rates. Our approach is to replicate the shaping-related data structure

per core, and to achieve parallelism at the level of flows, not at the level of a shared shaper across cores. Since Software NICs already uses dedicated cores, it was easy to rapidly develop and test a new shaper like Carousel, which is a testament to their value.

3.4.1 Single Queue Shaping

We examine the two steps of emulating the behavior of multiqueue architecture using a single time-indexed queue while improving CPU efficiency.

Timestamp Calculation

Timestamp calculation is the first step of traffic shaping in Carousel. Timestamp of a packet determines its *release time*, i.e., the time at which packet should be transmitted to the wire. A single packet can be controlled by multiple shaping policies, e.g., transport protocol pacing, per-flow rate limit, and aggregate limit for the flow's destination. In typical settings, these policies are enforced independently, each at the point where the policy is implemented in the networking stack. Carousel requires that at each point a packet is timestamped with transmission time according to the rate at that point rather than enforcing the rate there. Hence, these policies are applied sequentially as the packet trickles down the layers of the networking stack. For example, the TCP stack first timestamps each packet with a release time to introduce inter-packet gaps within a connection (pacing); subsequently, different rate limiters in the packet's path modify the packet's timestamp to conform to one or more rates corresponding to different traffic aggregates.

The timestamping algorithm operates in two steps: 1) timestamp calculation based on individual policies, such as pacing and rate limiting, and 2) consolidation of the different timestamps, so that the packet's final transmission time adheres to the overall desired behavior. The packet's final timestamp can be viewed as the *Earliest Release Time* (ERT) of the packet to the wire.

Setting Timestamps: Both pacing and rate limiting use the same technique for cal-

culating timestamps. Timestamps can be set as either relative timestamps, i.e., transmit after a certain interval relative to now, or the absolute wall time timestamps, i.e., transmit at a certain time. In our implementation, we use the latter. Both pacing and rate limiting force a packet to adhere to a certain rate: $2\times$cwnd/RTT for pacing, and a target rate for rate limiting. In our implementation, the timestamp for pacing is set by sender's TCP, and that for target rate is set by a module in the Software NIC, which receives external rates from centralized entities. Policy i enforces a rate R_i and keeps track of the latest timestamp it creates, LTS_i. The timestamp for the j^{th} packet going through the i^{th} policy, P_j^i is then calculated as: $LTS_i' = LTS_i + len(P_j^i)/R_i$. Timestamps are thus advanced on every packet. Note that packets through a policy can belong to the same flow, e.g., in the case of pacing, or multiple flows, e.g., in case of rate limiting flow aggregates. Packets that are neither paced nor rate limited by any policy carry a timestamp of zero.

Consolidating Timestamps: Consolidation is straightforward, since larger timestamps represent smaller target rates. In the course of a packet's traversal through multiple policies, we choose the largest of timestamps, which avoids violating any of the policies associated with the packet. Note that the effective final calculation of a flow's rate is equivalent to a series of token buckets, as the slowest token bucket ends up dominating and slowing a flow to its rate. We also note that Carousel is not a generic scheduler. Hence, this consolidation approach will not work for all scheduling policies.

Single Time-indexed Queue

One of the main components of Carousel is Timing Wheel [39], an efficient queue slotted by time and a special case of the more generic Calendar Queue [38]. The key difference is that Calendar Queue has O(1) amortized insertion and extraction but can be O(N) in skewed cases. The constant for O(1) in Timing Wheel is smaller than that of Calendar Queue. A Calendar Queue would provide exact packet ordering based on timestamps even within a time slot, a property that we don't strictly require for shaping purposes.

Algorithm 1 Timing Wheel implementation.

```
1:  procedure INSERT(PACKET, TS)
2:      Ts = Ts/Granularity
3:      if Ts <= FrontTimestamp then
4:          Ts = FrontTimestamp
5:      else if Ts > FrontTimestamp + NumberOfSlots − 1 then
6:          Ts = FrontTimestamp + NumberOfSlots − 1
7:      end if
8:      TW[Ts%NumberOfSlots].append(Packet)
9:  end procedure
10: procedure EXTRACT(now)
11:     now = now/Granularity
12:     while now >= FrontTimestamp do
13:         if TW[now%NumberOfSlots].empty() then
14:             FrontTimestamp+ = Granularity
15:         else
16:             return TW[now%NumberOfSlots].PopFront()
17:         end if
18:     end while
19:     return Null
20: end procedure
```

A Timing Wheel is an array of lists where each entry is a timestamped packet. The array represents the time slots from now till the preconfigured time $horizon$, where each slot represents a certain time range of size g_{min} within the overall horizon, i.e., the minimum time granularity. The array is a circular representation of time, as once a slot becomes older than now and all its elements are dequeued, the slot is updated to represent $now + horizon$. The number of slots in the array is calculated as $\frac{horizon}{g_{min}}$. Furthermore, we allow extracting packets every g_{min}. Within the g_{min} period, packets are extracted in FIFO order. The Timing Wheel is suitable as a single time-indexed queue operated by Carousel. We seek to achieve line rate for the aggregate of tens of thousands of flows. Carousel relies on a single queue to maintain line rate i.e. supporting tens of thousands of entries with minimal enqueue and dequeue overhead. The Timing Wheel's O(1) operations makes it a perfect design choice.

With the Timing Wheel, Carousel keeps track of a variable now representing current

time. Packets with timestamps older than *now* do not need to be queued and should be sent immediately. Furthermore, Carousel allows for configuring $horizon$, the maximum between *now* and the furthermost time that a packet can be queued upto. In particular, if r_{min} represents the minimum supported rate, l_{max} is the maximum number of packets in the queue belonging to the same aggregate that is limited to r_{min}, then the following relation holds: $horizon = \frac{l_{max}}{r_{min}}$ seconds. As Carousel is implemented within a busy polling system, the Software NIC can visit Carousel with a maximum frequency of f_{max}, due to having to execute other functions associated with a NIC. This means that it is unnecessary to have time slots in the queue representing a range of time smaller than $\frac{1}{f_{max}}$, as this granularity will not be supported. Hence, the minimum granularity of time slots is $g_{min} = \frac{1}{f_{max}}$.

An example configuration of the Timing Wheel is for a minimum rate of 1.5Mbps for 1500B packet size, slot granularity of 8us, and a time horizon of 4 seconds. This configuration yields 500K slots, calculated as $\frac{horizon}{g_{min}}$. The minimum rate limit or pacing rate supported is one packet sent per time horizon, which is 1.5Mbps for 1500B packet size. The maximum supported per-flow rate is 1.5Gbps. This cap on maximum rate is due to the minimum supported gap between two packets of 8μs and the packet size of 1500B. Higher rates can be supported by either increasing the size of packets scheduled within a slot (e.g., scheduling a 4KB packet per slot increases the maximum supported rate to 4Gbps), or by decreasing the slot granularity.

We note that Token Bucket rate limiters have an explicit configuration of $burst$ size that determines the largest line rate burst for a flow. Such a burst setting can be realized in Carousel via timestamping such that no more than a $burst$ of packets are transmitted at line rate. For example, a train of consecutive packets of a total size of $burst$ can be configured to be transmitted at the same time or within the same timeslot. This will create a burst of that size.

Algorithm 1 presents a concrete implementation of a Timing Wheel. There are only two operations: `Insert` places a packet at a slot determined by the timestamp and `ExtractNext`

retrieves the first enqueued packet with a timestamp older than the current time, now. This allows the Timing Wheel to act as a FIFO if all packets have a timestamp of zero. All packets with timestamp older than now are inserted in the slot with the smallest time.

We have two options for packets with a timestamp beyond the horizon. In the first option, the packets are inserted in the end slot that represents the timing wheel horizon. This approach is desirable when only flow pacing is the goal as it doesn't drop packets, but instead results in a temporary rate overshoot because of bunched packets at the horizon. The second approach is to drop packets beyond the horizon, which is desired when imposing a hard rate limit on flow, and an overshoot is undesirable.

A straightforward Timing Wheel implementation supports O(1) insertion and extraction where a list representing a slot is a dynamic list, e.g., `std::list` in C++. Memory is allocated and deallocated on every insertion and deletion within the dynamic list. Such memory management introduces significant cost per packet. Hence, we introduce a *Global Pool* of memory: a free list of nodes to hold packet references. Timing Wheel allocates from the free list when packets are enqueued. To further increase efficiency, each free list node can hold references to n packets. Preallocated nodes are assigned to different time slots based on need. This eliminates the cost of allocation and deallocation of memory per packet, and amortizes the cost of moving nodes around over n packets. Freed nodes return to the global pool for reuse by other slots based on need. §3.6 quantifies the Timing Wheel performance.

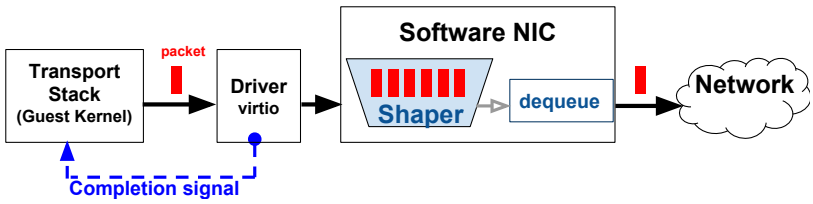
3.4.2 Deferred Completions

Carousel provides backpressure to higher layer transport without drops, by bounding the number of enqueued packets per flow. Otherwise, the queue can grow substantially, causing packet drops and head of line blocking. Thus, we need mechanisms in the software NIC or the hypervisor to properly signal between Carousel and the source. We study two such approaches in detail: *Deferred Completions* and delay-based congestion control.

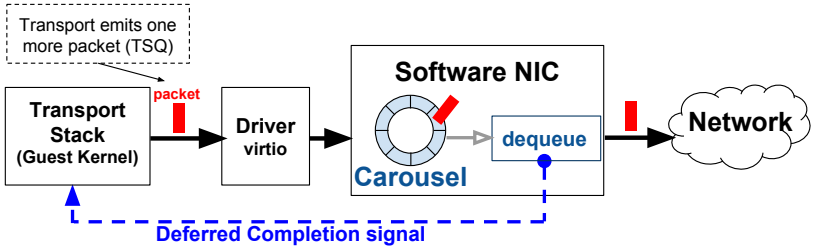
Completion is a signal reported from the driver to the transport layer in the kernel signaling that a packet left the networking stack (i.e. *completed*). A completion allows the kernel stack to send more packets to the NIC. The number of packets sent by the stack to the NIC is transport dependent, e.g., in the case of TCP, TSQ [78] ensures that the number of packets outstanding between the stack and the generation of a completion is at most two segments. Unreliable protocols like UDP could in theory generate an arbitrary number of packets. In practice, however, UDP sockets are limited to 128KB of outstanding send data.

The idea behind Deferred Completions, as shown in Figure 3.11, is *holding* the completion signal until Carousel transmits the packet to the wire, preventing the stack from enqueueing more packets to the NIC or the hypervisor. In Figure 3.11a, the driver in the guest stack generates completion signals at enqueue time, thus overwhelming the Timing Wheel with as much as a congestion window's worth of packets per TCP flow, leading to drops past the Timing Wheel horizon. In Figure 3.11b, Carousel in Software NIC returns the completion signal to the guest stack only after packet is dequeued from the Timing Wheel, and thus strictly bounds the number of packets in the shaper, since the transport stack relies on completion signals to enqueue more packets to Qdiscs.

Implementing Deferred Completions requires modifying the way completions are currently implemented in the driver. Completions are currently delivered by the driver in the order in which packets arrive at the NIC, regardless of their application or flow. Carousel can transmit packets in an order different from their arrival because of shaping. This means that a flow could have already transmitted packets to the wire, but be blocked because the transport layer has not received its completion yet. Consider a scenario of two competing flows, where the rate limit of one flow is half of the other. Both flows deliver packets to the shaper at the same arrival rate. However, packets of the faster flow will be transmitted at a higher rate. This means that packets from the faster flow can arrive at the shaper later than the packets from the slow flow and yet be transmitted first. The delivery rate of completion signals will need to match the packet transmission rate, or else the fast flow will be unnec-



(a) Immediate Completions



(b) Deferred Completions

Figure 3.11: Illustration of Completion signaling. In (a), a large backlog can build up to substantially overwhelm the buffer, leading to drops and head-of-line blocking. In (b), only two packets are allowed in the shaper and once a packet is released another is added. Indexing packets based on time allows for interleaving packets.

essarily backpressured and slowed down. Ordered completion delivery can cause head of line blocking, hence we implemented out-of-order completions in the driver of the guest kernel.

We implement out-of-order completions by tracking of all the packets still held by Carousel, in a hashmap in the guest kernel driver. On reception of a packet from guest transport stack, the Software NIC enqueues the reference to that packet in the hashmap. When Carousel releases the packet to the network, the Software NIC generates a completion signal, which is delivered to the guest transport stack and allows removal of the packet from the hashmap. The removal of the packet from the hashmap is propagated up in the stack, allowing TSQ to enqueue another packet [89]. This requires changing both the driver and its interface in the guest kernel, i.e., virtio [90].

Carousel is implemented within a Software NIC that relies on zero-copy to reduce memory overhead within the NIC. This becomes especially important in the presence of Carousel, because when the Software NIC shapes traffic, it handles more packets as it would without any shaping. Hence, the Timing Wheel carries references (pointers) to

packets and not the actual packets themselves. The memory cost incurred is thus roughly the cost of the number of references held in Carousel, while the packet memory is fully attributed to the sending application until the packet is sent to the wire (completed). In practice, this means that for every one million of packets outstanding in the Timing Wheel our software NIC needs to allocate only in the order of 8MB of RAM. The combination of Deferred Completions and holding packet references provides backpressure while using only a small memory footprint.

Delay-based Congestion Control (CC) is an alternative form of backpresssure to Deferred Completions. Delay-based CC senses the delay introduced in the shaper, and modulates TCP's congestion window to a smaller value as the RTT increases. The congestion window serves as a limit on the number of packets in Timing Wheel. We note that delay-based CC should discount the pacing component of the delay when computing the RTT. Operating with an RTT that includes pacing delay introduces undesirable positive feedback loops that reduce throughput. On the other hand, for Retransmission Timeout (RTO) computations, it is crucial that pacing delay be included, so as to avoid premature firing of the RTO timer. §3.6 provides a comparison across backpressure mechanisms.

3.4.3 Scaling Carousel with multiple cores

So far we have described the use of a single Timing Wheel per CPU core to shape traffic. Using a single shaper in multi-core systems incurs contention costs that would make the solution impractical. It is easy to scale Carousel by using independent Timing Wheels, one per CPU. For our approach, each TCP connection is hashed to a single Timing Wheel. Using one shaper per core suffice for pacing or rate limiting individual flows, and allows lock-free queuing of packets to a NIC instance on a specific core.

However, this simple approach does not work for the case when we want to shape multiple TCP connections by a single aggregate rate, because the individual TCP connections land on different cores (through hashing) and hence are shaped by different Timing Wheels.

We resolve this with two components:

1) NBA: Our implementation uses a *NIC-level bandwidth allocator* (NBA) that accepts a total rate limit for flow-aggregates and re-distributes each rate limit periodically to individual per-core Timestampers. NBA uses a simple water-filling algorithm to achieve work-conservation and max-min fairness. The individual Timestampers provide NBA with up-to-date flow usage data as a sequence of (byte count, timestamp of the byte count measurement) pairs; rate assignments from NBA to Timestampers are in bytes per second. The water-filling algorithm ensures that if flows on one core are using less than their fair share, then the extra is given to flows on the remaining cores. If flows on any core have zero demand, then the central algorithm still reserves 1% of the aggregate rate, to provide headroom for ramping up.

2) Communication between NBA and per-core Timestampers: Usage updates by Timestampers and the dissemination of rates by NBA are performed periodically in a lazy manner. We chose lazy updates to avoid the overhead of locking the data path with every update. The frequency of updates is a tradeoff between stability and fast convergence of computed rates; we currently use $\approx$100ms.

3.5 Carousel at the Receiver

We apply the central tenets discussed in §3.3 to show how a receiver can shape flows on the ingress side. Our goal is to allow overwhelmed receivers to handle incast traffic through a fair division of bandwidth amongst flows [91]. The direct way of shaping ingress traffic to queue incoming data packets, which can consume considerable memory. Instead, we shape the acknowledgements reported to the sender. Controlling the rate of acknowledged data allows for fine grained control over the sender's rate.

The ingress shaping algorithm modifies the acknowledgment sequence numbers in packets' headers, originally calculated by TCP, to acknowledge packets at the configured target rate. We apply Carousel as an extension of DCTCP without modifying its behavior.

The algorithm keeps track of a per flow Last Acked Sequence Number (SN_a), Latest Ack Time (T_a), and Last Received Sequence Number (SN_r). The first two variables keep track of the sequence number of last-acknowledged bytes and the time that acknowledgement was sent. For an outgoing packet from the receiver to the sender, we check the acknowledgement number in the packet. We use that to update SN_r. Then, we change that number based on the following formula: $NewAckSeqNumber = \min((now - T_a) \times Rate + SN_a, SN_r)$. We update T_a to now and SN_a to $NewAckSeqNumber$, if SN_a is smaller than $NewAckSeqNumber$. Acknowledgments are generated by Carousel when no packet has been generated by the upper layer for the time $\frac{MTU}{Rate}$. The variable $Rate$ is set either through a bandwidth allocation system, or it is set as the total available bandwidth divided by the number of flows.

3.6 Evaluation

We evaluate Carousel in small-scale microbenchmarks where all shaped traffic is between machines within the same rack, and in production experiments on machines serving video content to tens of thousands of flows per server. We compare the overhead of Carousel to HTB and FQ/Pacing in similar settings and demonstrate that Carousel is 8% more efficient in overall machine CPU utilization (20% more efficient in CPU utilization attributable to networking), and 6% better in terms of rate conformance, while maintaining similar or lower levels of TCP retransmission rates. Note that the video serving system is just for the convenience of production experiments in this section; the same shaping mechanisms are also deployed in our network operations such as enforcing Bandwidth Enforcer (BwE) rates.

3.6.1 Microbenchmark

Experiments setup: We conduct experiments for egress traffic shaping between two servers sharing the same top of rack switch. The servers are equipped with software NIC on which

we implemented Carousel to shape traffic. For baseline, servers run Linux with HTB configured with 16K packets per queue; FQ/Pacing is configured with a global limit of 10K packets and a flow limit of 1000 packets. These settings follow from best practices. We generate traffic with `neper` [92], a network performance measurement tool that allows for generating large volumes of traffic with up to thousands of flows per server. We vary RTT using `netem`. All reported results are for an emulated RTT of 35ms. We found the impact of RTT on Carousel to be negligible. Experiments are run for 30 seconds each and enough number of runs to make the standard deviation 1% or smaller. Unless otherwise mentioned, the Timing Wheel granularity is two microseconds with a horizon of two seconds.

Rate Conformance

We measure rate conformance as the absolute deviation from the target rate. This metric represents the deviation in bytes per second due to shaping errors. We measure absolute deviation from target rate by collecting achieved rate samples once every 100 milliseconds. We collect samples at the output of the shaper, i.e., on the sender, as we want to focus on the shaper behavior and avoid factoring in network impact on traffic. Samples are collected by using `tcpdump` where throughput is calculated by measuring the amount of bytes sent every interval of 100 milliseconds.

We compare Carousel's performance to HTB and FQ/pacing: 1) pacing is performed by FQ/pacing on a per-flow basis where a maximum pacing rate is configured in TCP using `SO_MAX_PACING_RATE` [45], and 2) rate limiting where HTB enforces rate on a flow aggregate.

Change target rate. We investigate rate conformance for a single TCP connection. As the target rate is varied, we find that HTB and FQ have a consistent 5% and 6% deviation between the target and achieved rates (Figure 3.12). This is due to their reliance on timers which fire at lower rates than needed for accuracy at high rates. For instance, HTB's timer fires once every 10ms. Carousel relies on accurate scheduling of packets in busy-wait

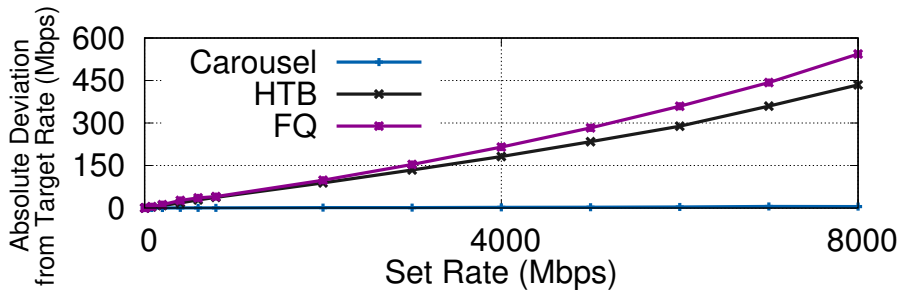


Figure 3.12: Comparison between Carousel, HTB, and FQ in their rate conformance for a single flow.

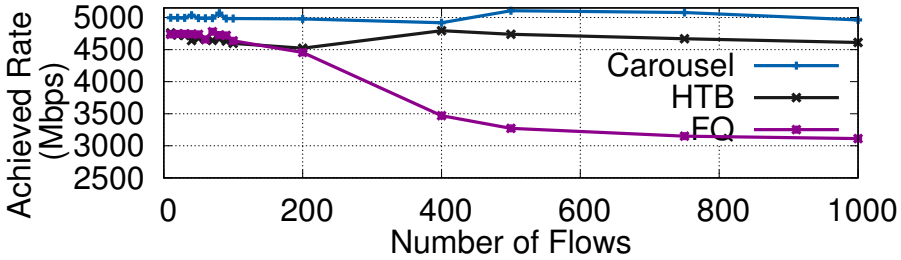


Figure 3.13: Comparison between Carousel, HTB, and HTB/FQ showing the effect of the number of flows load on their rate conformance to a target rate of 5 Gbps.

polling Software NIC, by placing them in a Timing Wheel slot that transmits a packet within a slot size of its scheduled time.

Vary the number of flows. We vary the number of flows in our experiments between one and 10K, typical for user facing servers. Figure 3.13 shows the effect of varying the number of flows until 1000 flows. Carousel is not impacted as it relies on the Timing Wheel with O(1) insertion and extraction times. HTB maintains a constant deviation of 5% from the target rate. However, FQ/Pacing conformity significantly drops beyond 200 flows. This is due to the limited number of per-queue and global packet limits. However, increasing this number significantly increases its CPU overhead due to its reliance on RB-trees for maintaining flows and packets schedules [45].

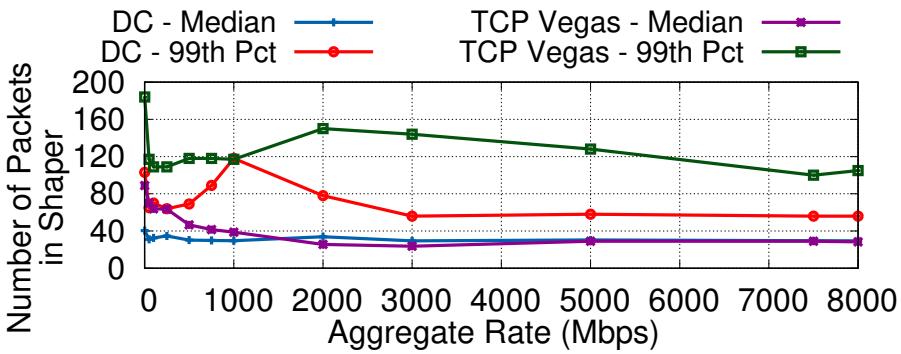


Figure 3.14: A comparison of Deferred Completion (DC) and delay-based congestion control (TCP Vegas) backpressure, for fifteen flows when varying the aggregate rate limits.

Memory Efficiency

We compare memory efficiency of Deferred Completions and delay-based CC as backpressure schemes (§3.4.2). We use the metric of number of packets held in the shaper, which directly reflects the memory allocation expected from the shaper along with the possibility of packet drops as memory demand exceeds allocation. We collect samples of the number of packets every time a new packet is extracted from the shaper.

Change target rate. Figure 3.14 shows that with Deferred Completions, the average of number of packets in the Carousel is not affected by changes in the target rate. While delay-based CC maintains similar average number of packets like in Deferred Completions, it has twice the number of packets in the 99th percentile. This is because delay variations can grow the congestion window in Vegas to large values, while Deferred Completions maintains a strict bound regardless of the variations introduced by congestion control. The bump in 99th percentile at 1Gbps rate for Deferred Completions results from an interaction of TSQ and TSO autosizing. Deferred Completions behavior is consistent regardless of the congestion control algorithm used in TCP. Our experiments for Deferred Completions used TCP Cubic in Linux.

Vary number of flows. In Figure 3.15, we compare the number of packets held in the shaper for a fixed target rate while varying the number of flows. In the absence of Deferred

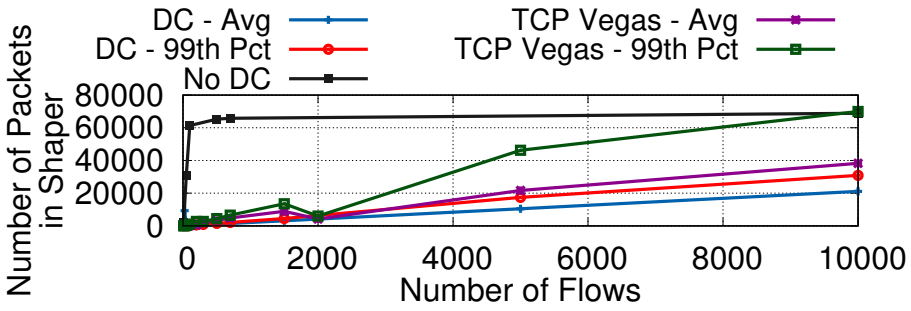


Figure 3.15: A comparison of Deferred Completion (DC) and delay-based congestion control (TCP Vegas) backpressure, while varying the number of flows with an aggregate rate limit of 5Gbps.

Completions, the flows continually send until all of the 70K memory in the shaper is filled. For Deferred Completions, the number of packets held in the shaper is deterministic as approximately twice the number of active flows. This is also reflected in the 99th percentile of the number of packets held. For delay-based CC, the median number of packets is higher than the 99th percentile behavior of Deferred Completion. Furthermore, the 99th percentile of delay-based CC is less predictable than that of the 99th percentile of Deferred Completions because it can be affected by variations in RTT. We note that we didn't observe any significant difference in rate conformance between both approaches. We find that Deferred Completion is the better backpressure approach because of its predictable behavior.

There is no hard limit on the number of individual connections or flow aggregates that Carousel can handle. Carousel operates on the references (pointers) to packets, which are small. Surely, we have to allocate memory to hold the references, but in practice this memory is a small fraction of memory available on the machine.

Impact of Timing Wheel Parameters

In §3.4, we choose the Timing Wheel due to it is high efficiency and predictable behavior. We study the effect of its different parameters on the behavior of Carousel.

Slot Granularity. Slot granularity controls the burst size and also determines the smallest supported gap between packets which affects the maximum rate that Carousel can pace

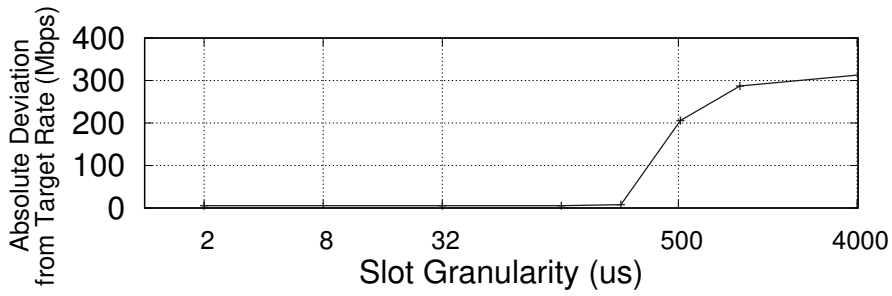


Figure 3.16: Rate conformance of Carousel for target rate of 5Gbps for different slot granularities.

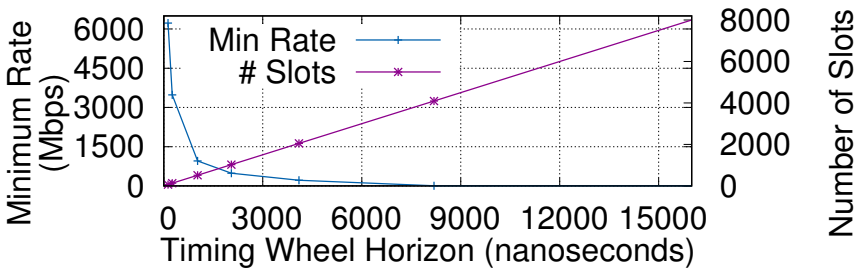


Figure 3.17: An analytical behavior of the minimum supported rate for a specific Timing Wheel horizon along with the required number of slots for a slot size of two microseconds.

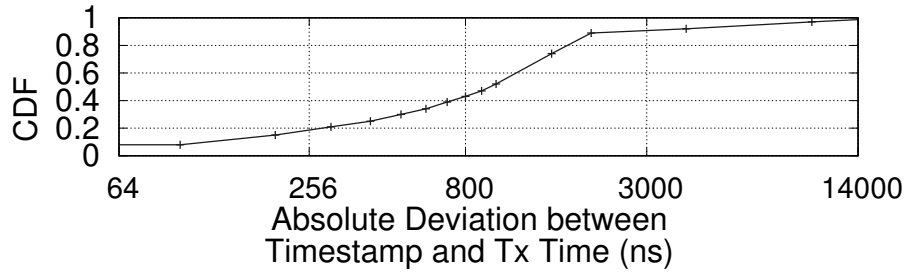


Figure 3.18: A CDF of deviation between a packet timestamp and transmission time.

at. Furthermore, for a certain granularity to be supported, the Timing Wheel has to be dequeued at least that rate. We investigate the effect of different granularities on rate conformance. Figure 3.16 shows that a granularity of 8-16μs suffices to maintain a small deviation from the target rate. We also find that rate conformity at larger slot granularities such as 4ms deviates by 300 Mbps which is comparable to the deviation of HTB and FQ at the same target rate as shown in Figure 3.12.

Timing Wheel Horizon. The horizon represents the maximum delay a packet can

encounter in shaper (hence minimum supported rate), and is the product of slot granularity and number of slots. Figure 3.17, shows the minimum rate and the the number of slots needed for different horizon values, all for a granularity of two microseconds. Note that results here assume that only one flow is shaped by Carousel which means that the horizon needs to be large enough to accommodate the time gap between only two packets.

Deviation between Timestamp and Transmission Time. Figure 3.18 shows a CDF of the difference between the scheduled transmission time of the packet and the actual transmission time of the packet in nanoseconds. This metric represents the accuracy of the Timing Wheel in how much it adheres to the schedules. Over 90% of packets have deviations smaller than the slot granularity of two microseconds, which occurs due to the rounding down error when mapping packets with schedules calculated in nanoseconds to slots of 2μs. Deviations that exceed a slot size due to excessive queuing delay between the timestamper and the shaper are rare.

CPU Overhead of Timing Wheel. For a system attempting to achieve line rate, every nanosecond counts for processing each packet. The Timing Wheel implementation is the most resource consuming component in the system as the rest of the operations merely change parts of the packet metadata. We explore two parameters that can affect the delay per packet in the Timing Wheel implementation: 1) the impact of the data structures used for representing a slot, and 2) the number of packets held in the Timing Wheel. Table 3.1 shows the impact of both parameters. It's clear that because of the O(1) insertion and extraction overhead, the Timing Wheel is not affected by the number of packets or flows. The factor affecting the delay per packet is the overhead of insertion and extraction of packets from each slot. We find that implementing a slot data structure that avoids allocating memory for every insertion, i.e., Global Pool (described in §3.4.1), reduces the overhead of C++ `std::list` by 50%.

Table 3.1: Overhead per packet of different Timing Wheel implementations.

Number of Packets in Shaper	1000	4000	32000	256000	20000000
STDList (ns per packet)	22	21	21	21	22
Global Pool (ns per packet)	12	11	11	11	11

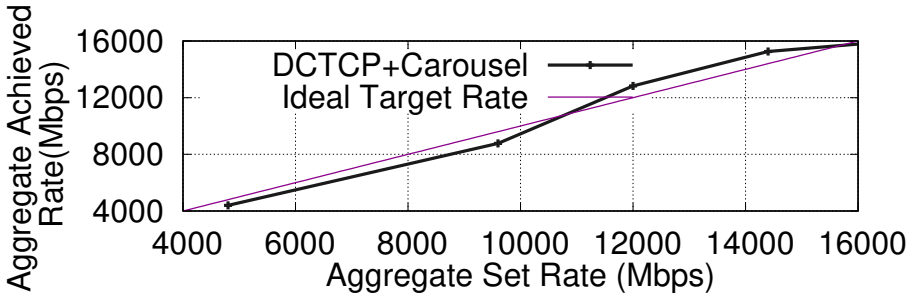


Figure 3.19: Comparing receiver side rate limiting with target rate.

Carousel Impact at the Receiver

We evaluate the receiver-based shaper in a scenario of 100:1 incast: 10 machines with 10 connections each send incast traffic to one machine; all machines are located in the same rack. Unloaded network RTT is ≈10us, packet size is 1500 Bytes, and experiment is run for 100 seconds. Senders use DCTCP and Carousel to limit the throughput of individual incast flows. Figure 3.19 shows the rate conformance of Carousel for varying aggregate rates enforced at the receiver. The deviation of aggregate achieved rate is within 1% of the target rate.

3.6.2 Production Experience

To evaluate Carousel with real production loads, we choose 25 servers in five geographically diverse locations. The servers are deployed in two US locations (East and West Coast), and three European locations (Western and Eastern Europe). They are serving streaming video traffic from a service provider to end clients. Each server generates up to 38Gbps of traffic serving as many as 50,000 concurrently active sessions. For comparison, the servers support both conventional FQ/pacing in Linux kernel and Carousel implemented

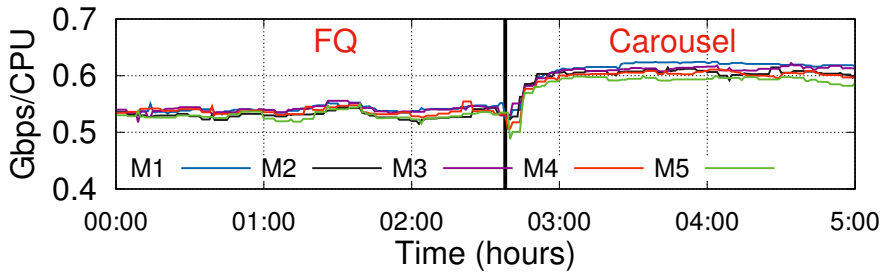


Figure 3.20: Immediate improvement in CPU efficiency after switching to Carousel in all five machines in a US west coast site.

on software NIC like SoftNIC [19].

The metrics we use are: 1) retransmission rates, 2) server CPU efficiency, and 3) software NIC efficiency. To measure server CPU efficiency, we use Gbps/CPU metric which is the total amount of egress traffic divided by the total number of CPUs on a server and then divided by their utilization. For example, if a server sends 18 Gbps with 72 CPUs that are 50% utilized on average, it is said to have $\frac{18}{72*0.5}$ = 0.5Gbps/CPU efficiency. We present only the data for periods of use when server CPUs are more than 50% utilized (approximately 6 hours every day). Peak periods is when efficiency is the most important because server capacity must be provisioned for such peaks. Gbps/CPU is better than direct CPU load for measuring CPU efficiency. This is because our production system routes new client requests to the least loaded machines to maintain high CPU utilization at all machines. This load-balancing behavior means that more efficient machines will receive more client requests.

CPU efficiency. In the interest of space, we do not show results from all sites. Figure 3.20 shows the impact of Carousel on Gbps/CPU metric on five machines in the West Coast site. Figure 3.21 compares two machines one using Carousel and the other using FQ/pacing. We pick these two machines specifically as they have similar CPU utilization during our measurement period. This provides a fair comparison for the Gbps/CPU by unifying the denominator which focuses on how efficiently the CPU is used to push data. At 50^{th} and 90^{th} percentiles, Carousel is 6.4% and 8.2% more efficient (Gbps/CPU) in serving

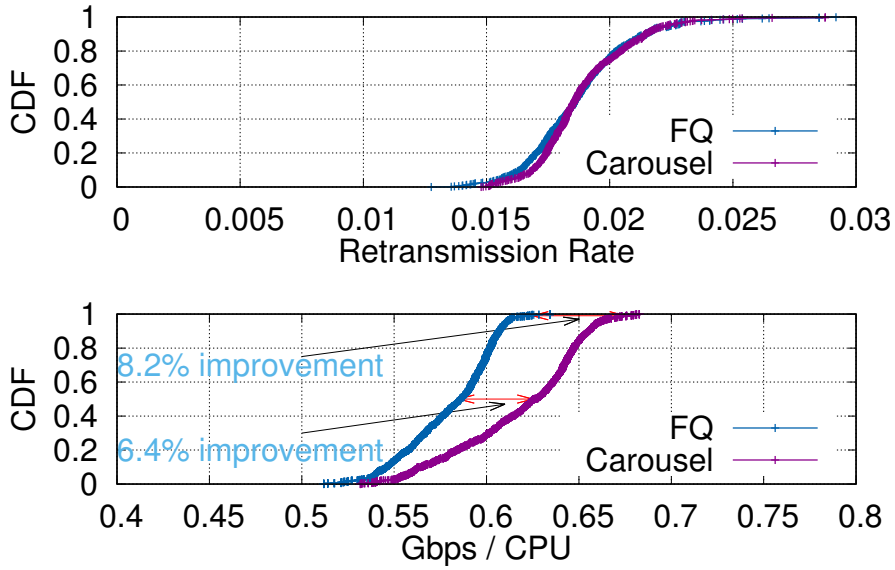


Figure 3.21: Comparison between FQ and Carousel for two machines serving large volumes of video traffic exhibiting similar CPU loads showing that Carousel can push more traffic for the same CPU utilization.

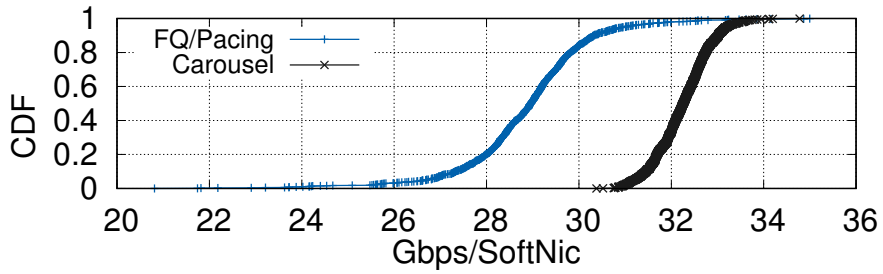


Figure 3.22: Software NIC efficiency comparison with pacing enforced in kernel vs pacing enforced in software NIC. Bandwidth here is normalized to a unit of software NIC self-reported utilization.

traffic. On 72-CPU machines this translates to saving an average of 4.6 CPUs per machine. Considering that networking accounts for ≈40% of the machine CPU, the savings are 16% and 20% in the median and tail for CPU attributable to networking. The two machines exhibit similar retransmission rate which shows that using Carousel maintains pacing value while significantly reducing its cost.

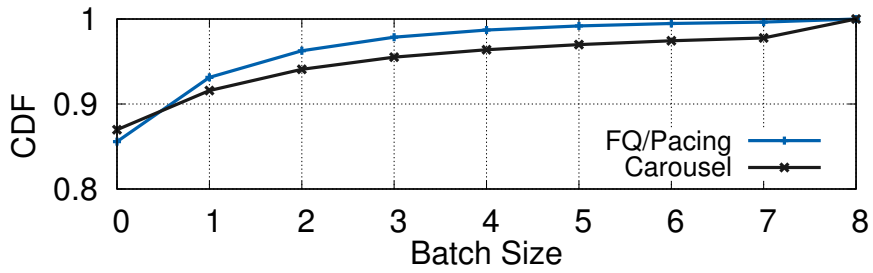


Figure 3.23: Cumulative batch sizes (in TSO packets) that software NIC receives from kernel when pacing is enforced in kernel vs when pacing is enforced in software NIC.

3.7 Discussion

Carousel on Software NIC. We examine the overhead of Carousel on the software NIC. Surprisingly, we find that moving pacing to software NIC in fact *improves* its performance. The software NIC operates in spin-polling mode, consuming 100% of fixed number of CPUs assigned to it. The NIC reports the CPU cycles it spent in a spin loop and whether it did any useful work in that particular loop (i.e. received a batch that has at least one packet, or there was a packet emitted by timing wheel); the CPU cycles spent in useful loops divided by the total number of CPU cycles in all loops is the Software NIC utilization level. To measure software NIC efficiency, we normalize observed throughput by software NIC self-reported utilization, thus deriving Gbps/SoftNIC metric. Figure 3.22 shows Gbps/-SoftNIC metric for pacing with FQ/pacing and Carousel. We see that moving pacing to the software NIC improves its efficiency by 12% (32.2 Gbps vs 28.8 Gbps per software NIC). The improved efficiency is from larger batching from kernel TCP stack to software NIC, because the pacing is moved from kernel to the NIC. Figure 3.23 shows cumulative batch sizes, including empty batches, for machines serving production traffic with pacing enforcement in kernel and pacing enforcement in software NIC. On average, non-empty packet batch sizes increased by 57% (from an average batch size of 2.1 to an average batch size of 3.3).

Application of Carousel in Hardware NICs: Some hardware devices, such as Intel

NICs [93, 94], provide support for rate limiting. The common approach requires one hardware queue per rate limiter, where the hardware queue is bound to one or more transmit queues in the host OS. This approach heads in the direction of increasing transmit queues in hardware which does not scale because of silicon constraints. Hence, hardware rate limiters are restricted for use in custom applications, rather than as a normal offload that the OS can use.

With Carousel, the single queue approach is a sharp departure from the trend of needing more number of hardware queues, and can be implemented follows: 1) hardware is configured with one transmit queue per CPU to avoid contention between the OS and the NIC; 2) Classification and rates on the hardware are configured via an API, 3) hardware is able to consume input packets, generate per-packet timestamps based on configured rates, and enqueue them to Timing Wheel, 4) dequeue packets that are due, 5) return the completion event relative to the packet sent.

Limitation The main limitation of Carousel stems from the use of a *single* queue shaper with timestamping of packets. The challenge is how to make such timestamping consistent when the timestamps are not set in one centralized place. An approach is to standardize all sources to set timestamps in nanoseconds (or microseconds) from the Unix epoch time. Other challenges include handling timestamps from unsynchronized CPU clocks, and timestamps that may be of different granularities. The second limitation is that it implements only a single scheduling policy, which is the challenge we address in the next chapter.

3.8 Summary

Even though traffic shaping is fundamental to the correct and efficient operation of datacenters and WANs, deployment at scale has been difficult due to prohibitive CPU and memory overheads. We showed in this chapter that two techniques can help overcome scaling and efficiency concerns: a single time-indexed queue per CPU core for packets, and manage-

ment of higher-layer resources with Deferred Completions. Scaling is further helped by a multicore-aware design. Carousel is able to individually shape tens of thousands of flows on a single server with modest resource consumption.

We believe the most exciting consequence of our work will be the creation of novel policies for pacing, bandwidth allocation, handling incast, and mitigation against DoS attacks. No longer are we restricted to cherry-picking handful of flows to shape. We are confident that our results with Carousel make a strong technical case for networking stacks and NIC vendors to invest in the basic abstractions of Carousel.

CHAPTER 4

EIFFEL: EFFICIENT AND FLEXIBLE SOFTWARE PACKET SCHEDULING

Our focus in this chapter is on the design and deployment of packet scheduling in software. Software schedulers have several advantages over hardware including shorter development cycle and flexibility in functionality and deployment location. We substantially improve current software packet scheduling performance, while maintaining flexibility, by exploiting underlying features of packet ranking; namely, packet ranks are integers and, at any point in time, fall within a limited range of values. At a fundamental level, a scheduling policy that has m ranking functions associated with a packet (e.g., pacing rate, policy-based rate limit, weight-based share, and deadline-based ordering) typically requires m priority queues in which this packet needs to be enqueued and dequeued [27], which translates roughly to $O(m \log n)$ operations per packet for a scheduler with n packets enqueued. We show how to reduce this overhead to $O(m)$ for any scheduling policy (i.e., constant overhead per ranking function).

We introduce Eiffel, an efficient and flexible software scheduler that instantiates our proposed approach. Eiffel is a software packet scheduler that can be deployed on end-hosts and software switches to implement any scheduling algorithm. To demonstrate this we implement Eiffel (§4.3) in: 1) the kernel as a Queuing Discipline (qdisc) and compare it to Carousel and FQ/Pacing [45] and 2) the Berkeley Extensible Software Switch (BESS) [95, 19] using Eiffel-based implementations of pFabric [11] and hClock [43]. We evaluate Eiffel in both settings (§4.4). Eiffel outperforms Carousel by 3x and FQ/Pacing by 14x in terms of CPU overhead when deployed on Amazon EC2 machines with line rate of 20 Gbps. We also find that an Eiffel-based implementation of pFabric and hClock outperforms an implementation using comparison-based priority queues by 5x and 40x respectively in terms of maximum number of flows given fixed processing capacity and target rate.

4.1 Eiffel Design Objectives

In modern networks, packet scheduling can easily become the system bottleneck. This is because schedulers are burdened with the overhead of maintaining a large number of buffered packets sorted according to scheduling policies. Despite the growing capacity of modern CPUs, packet processing overhead remains a concern. Dedicating CPU power to networking takes from CPU capacity that can be dedicated to VM customers especially in cloud settings [96]. One approach to address this overhead is to optimize the scheduler for a specific scheduling policy [45, 44, 43, 24, 64]. However, with specialization two problems linger. First, in most cases inefficiencies remain because of the typical reliance on generic default priority queues in modern libraries (e.g., RB-trees in kernel and Binary Heaps in C++). Second, even if efficiency is achieved, through the use of highly efficient specialized data structures (e.g., Carousel [24] and QFQ [64]) or hybrid hardware/software systems (e.g. SENIC [21]), this efficiency is achieved at the expense of programmability. The Eiffel system we develop in this chapter is designed to be both efficient and programmable. In this section we examine these two objectives, show how existing solutions fall short of achieving them and highlight our approach to successfully combine efficiency with flexibility.

Efficient Priority Queuing:

Packet queues have the following characteristics that can be exploited to significantly lower the overhead of packet insertion and extraction:

- *Integer packet ranks:* Whether it is deadlines, transmission time, slack time, or priority, the calculated rank of a packet can always be represented as an integer.

- *Packet ranks have specific ranges:* At any point in time, the ranks of packets in a queue will typically fall within a limited range of values (i.e., with well known maximum and minimum values). This range is policy and load dependent and can be determined in advance by operators (e.g., transmission time where packets can be

scheduled a maximum of a few seconds ahead, flow size, or known ranges of strict priority values). Ranges of priority values are diverse, ranging from just eight levels [97], to 50k for a queue implementing per flow weighted fairness which requires a number of priorities corresponding to the number of flows (i.e., 50k flows on a video server [24]), and up to 1 million priorities for a time indexed priority queue [24].

- *Large numbers of packets share the same rank:* Modern line rates are in the ranges of 10s to 100s of Gbps. Hence, multiple packets are bound to be transmitted with nanosecond time gaps. This means that packets with small differences in their ranks can be grouped and said to have the same rank with minimal or no effect on the accurate implementation of the scheduling policy. For instance, consider a busy-polling-based packet pacer that can dequeue packet at fixed intervals (e.g., order of 10s of nanoseconds). In that scenario, packets with gaps smaller than 10 nanoseconds can be considered to have the same rank.

These characteristics make the design of a packet priority queue effectively the design of bucketed integer priority queues over a finite range of rank values $[0, C]$ with number of buckets N, each covering C/N interval of the range. The number of buckets, and consequently the range covered by each bucket, depend on the required ranking granularity which is a characteristic of the scheduling policy. The number of buckets is typically in the range of a few thousands to hundreds of thousands. Elements falling within a range of a bucket are ordered in FIFO fashion. Theoretical complexity results for such bucketed integer priority queues are reported in [56, 58, 59].

Integer priority queues do not come for free. Efficient implementation of integer priority queues requires pre-allocation of buckets and meta data to access those buckets. In a packet scheduling setting the number of buckets is fixed, making the overhead per packet a constant whose value is logarithmic in the number of buckets, because searching is performed on the bucket list not the list of elements. Hence, bucketed integer priority queues achieve CPU efficiency at the expense of maintaining elements unsorted within a single

bucket and pre-allocation of memory for all buckets. Note that maintaining elements unsorted within a bucket is inconsequential because packets within a single bucket effectively have equivalent rank. Moreover, the memory required for buckets, in most cases, is minimal (e.g., tens to hundreds of kilobytes), which is consistent with earlier work on bucketed queues [24]. Another advantage of bucketed integer priority queues is that elements can be (re)moved with $O(1)$ overhead. This operation is used heavily in several scheduling algorithms (e.g., hClock [43] and pFabric [11]). This efficiency allows for performing different scheduling operations efficiently, unlike earlier approaches that attempt to develop customized data structures for specific scheduling operations as summzrized in Chapter 2. This leads us to our first objective for Eiffel:

Objective 1: Develop data structures that can be employed for any scheduling algorithm providing $O(1)$ processing overhead per packet leveraging integer priority queues (§4.2.1).

Flexibility of Programmable Packet Schedulers: Despite advancements made in programmable schedulers discussed in Chapter 2. The design of a flexible and efficient packet scheduler remains an open research challenge. It is important to note here that the efficiency of programmable schedulers is different from the efficiency of policies that they implement. An efficient programmable platform aims to reduce the overhead of its building blocks (i.e., Objective 1) which makes the overhead primarily a function of the complexity of the policy itself. Thus, the efficiency of a scheduling policy becomes a function of only the number of building blocks required to implement it. Furthermore, an efficient programmable platform should allow the operator to choose policies based on their requirements and available resources by allowing the platform to capture a wide variety of policies. To address this challenge, we choose to extend the PIFO model due to its existing efficient building blocks. In particular, we introduce flows as a unit of scheduling in the PIFO model. We also allow modifications to packet ranking and relative ordering both on enqueue and dequeue.

Objective 2: Provide a fully expressive scheduler programming abstraction by extending the PIFO model (§4.2.2).

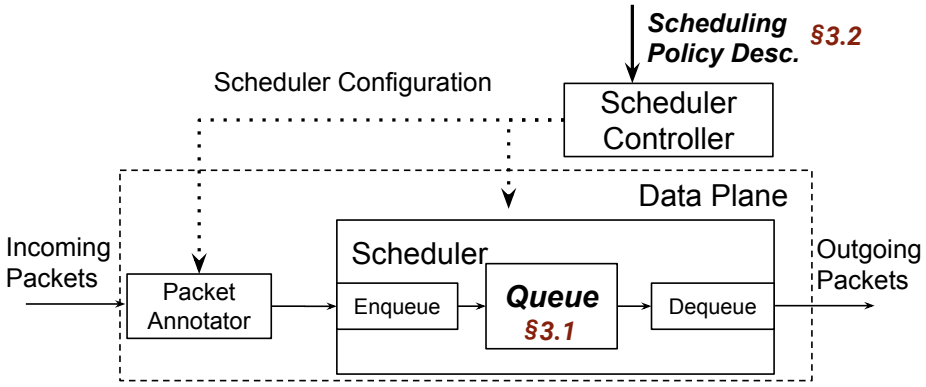


Figure 4.1: Eiffel programmable scheduler architecture highlighting Eiffel's extensions.

4.2 Eiffel Design

Figure 4.1 shows the architecture of Eiffel with four main components: 1) a packet annotator to set the input to the enqueue component (e.g., packet priority), 2) an enqueue component that calculates a rank for incoming packets, 3) a queue that holds packets sorted based on their rank, and 4) a dequeue component which is triggered to re-rank elements in the queue, for some scheduling algorithms. Eiffel leverages and extends the PIFO scheduler programming model to describe scheduling policies [29, 98]. The functions of the packet annotator, the enqueue module, and the dequeue module are derived in a straightforward manner from the scheduling policy. The only complexity in the Scheduler Controller, namely converting scheduling policy description to code, has been addressed in earlier work on the PIFO model [98]. The two complicated components in this architecture, therefore, correspond with the two objectives discussed in the previous section: the Queue *(Objective 1)* and The Scheduling policy Description *(Objective 2)*. For the rest of this section, we explain our efficient queue data structures along with our extensions to the programming model used to configure the scheduler.

4.2.1 Priority Queueing in Eiffel

A priority queue maintains a list of elements, each tagged with a priority value. A priority queue supports one of two operations efficiently: `ExtractMin` or `ExtractMax` to get the element with minimum or maximum priority respectively. Our goal, as stated in Objective 1 in the previous section, is to enable these operations with $O(1)$ overhead. To this end we first develop a circular extension of efficient priority queues that rely on the FindFirstSet (FFS) operation, found in all modern CPUs [99, 100]. Our extensions allow FFS-based queues to operate over large moving ranges while maintaining CPU efficiency. We then improve on the FFS-based priority queue by introducing the approximate gradient queue, which can perform priority queuing in $O(1)$ under some conditions. The approximate priority queue can outperform the FFS-based queue by up to 9% for scenarios of a highly occupied bucketed priority queue (§4.4.2). Note that for all Integer Priority Queues discussed in this section, enqueue operation is trivial as buckets are identified by the priority value of their elements. This makes the enqueue operation a simple bucket lookup based on the priority value of the enqueued element.

Circular FFS-based Queue (cFFS)

FFS-based queues are bucketed priority queues with a bitmap representation of queue occupancy. Zero represents an empty bucket, and one represents a non-empty bucket. FFS produces the index of the leftmost set bit in a machine word in constant time. All modern CPUs support a version of Find First Set at a very low overhead (e.g., Bit-Scan-Forward (`BSR`) takes three cycles to complete [99]). Hence, a priority queue, with a number of buckets equal to or smaller than the width of the word supported by the FFS operation can obtain the smallest set bit, and hence the element with the smallest priority, in $O(1)$ (e.g., Figure 4.2). In the case that a queue has more buckets than the width of the word supported by a single FFS operation, a set of words can be processed sequentially to represent the queue with every bit representing a bucket. This results in an $O(M)$ algorithm that is very

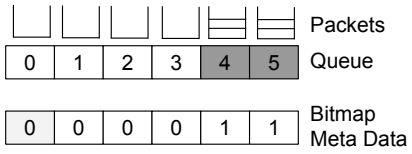


Figure 4.2: FFS-based queue where FFS of a bit-map of six bits can be processed in O(1).

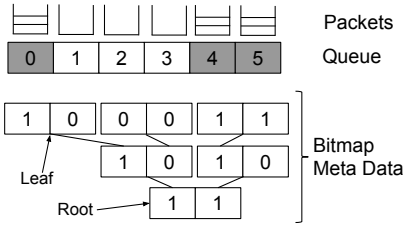


Figure 4.3: Hierarchical FFS-based queue where FFS of bit-map of two bits can be processed in O(1) using a 3-level hierarchy

efficient for very small M, where M is the number of words. For instance, realtime process scheduling in the linux kernel has a hundred priority levels. An FFS-based priority queue is used where FFS is applied sequentially on two words, in case of 64-bit words, or four words in case of 32-bit words [101]. This algorithm is not efficient for large values of M as it requires scanning all words, in the worst case, to find the index of the highest priority element. FFS instruction is also used in QFQ to sort groups of flows based on the eligibility for transmission where the number of groups is limited to a number smaller than 64 [64]. QFQ is an efficient implementation of fair queuing which uses FFS efficiently over a small number of elements. However, QFQ does not provide any clear direction towards implementing other policies efficiently.

To handle an even larger numbers of priority levels, hierarchical bitmaps may be used. One example is Priority Index Queue (PIQ) [70], a hardware implementation of FFS-based queues, which introduces a hierarchical structure where each node represents the occupancy of its children, and the children of leaf nodes are buckets. The minimum element can be found by recursively navigating the tree using FFS operation (e.g., Figure 4.3 for a word width of two). Hierarchical FFS-based queues have an overhead of $O(\log_w N)$ where w is the width of the word that FFS can process in $O(1)$ and N is the number of

buckets. It is important to realize that, for a given scheduling policy, the value of N is a given fixed value that doesn't change once the scheduling policy is configured. Hence, a specific instance of a Hierarchical FFS-based queue has a constant overhead independent of the number of enqueued elements. In other words, once an implementation is created N does not change.

Hierarchical FFS-based queues only work for a fixed range of priority values. However, as discussed earlier, typical priority values for packets span a moving range. PIQ avoids this problem by assuming support for the universe of possible values of priorities. This is an inefficient approach because it requires generating and maintaining a large number buckets, with relatively few of them in use at any given time.

Typical approaches to operating over a large moving range while maintaining a small memory footprint rely on *circular queues*. Such queues rely on the mod operation to map the moving range to a smaller range. However, the typical approach to circular queuing does not work in this case as it results in an incorrect bitmap. For example, if we add a packet with priority value six to the queue in Figure 4.2 selecting the bucket with a mod operation, the packet will be added in slot zero and consequently mark the bit map at slot zero. Hence, once the range of an FFS-based queue is set, all elements enqueued in that range have to be dequeued before the queue can be assigned a new range so as to avoid unnecessary resetting of elements. In that scenario, enqueued elements that are out of range are enqueued at the last bucket, and thus losing their proper ordering. Otherwise, the bitmap meta data will have to be reset in case any changes are made to the range of the queue.

A natural solution to this problem is to introduce an overflow queue where packets with priority values outside the current range are stored. Once all packets in the current range are dequeued, packets from that "secondary" queue are inserted using the new range. However, this introduces a significant overhead as we have to go through all packets in the buffer every time the range advances. We solve this problem by making the secondary

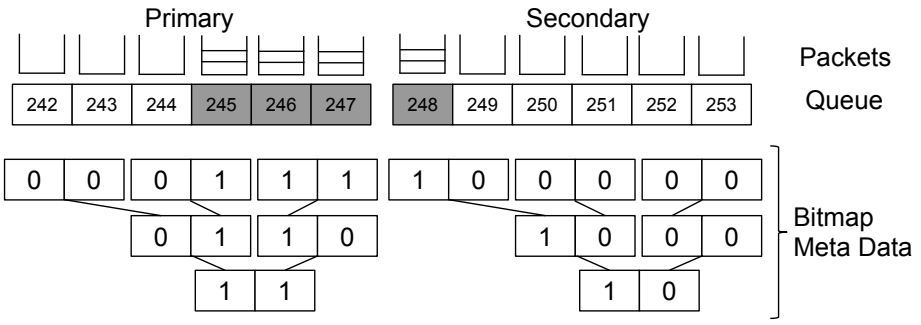


Figure 4.4: Circular Hierarchical FFS-based queue is composed of two Hierarchical FFS-based queues, one acting as the main queue and the other as a buffer.

queue an FFS-based queue, covering the range that is immediately after the range of the queue (Figure 4.4). Elements outside the range of the secondary queue are enqueued at the last bucket in the secondary queue and their values are not sorted properly. However, we find that to not be a problem as ranges for the queues are typically easy to figure out given a specific scheduling policy.

A *Circular Hierarchical FFS-based queue*, referred to hereafter simply as a cFFS, maintains the minimum priority value supported by the primary queue (`h_index`), the number of buckets (`q_size`) per queue, two pointers to the two sets of buckets, and two pointers to the two sets of bitmaps. Hence, the queue "circulates" by switching the pointers of the two queues from the buffer range to the primary range and back based on the location of the minimum element along with their corresponding bitmaps.

Note that work on efficient priority queues has a very long history in computer science with examples including van Emde Boas tree [56] and Fusion trees [58]. However, such theoretical data structures are complicated to implement and require complex operations. cFFS is highly efficient both in terms of complexity and the required bit operations. Moreover, it is relatively easy to implement.

Approximate Priority Queuing

cFFS queues still require more than one step to find the minimum element. We explore a tradeoff between accuracy and efficiency by developing a gradient queue, a data structure

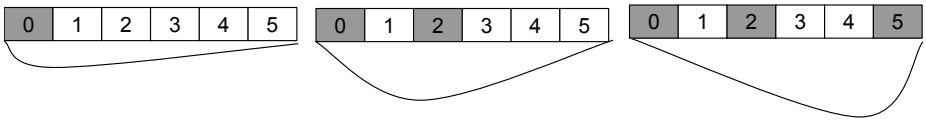


Figure 4.5: A sketch of a curvature function for three states of a **maximum** priority queue. As the maximum index of nonempty buckets increases, the critical point shifts closer to that index.

that can find a *near* minimum element in one step.

Basic Idea The Gradient Queue (GQ) relies on an algebraic approach to calculating FFS. In other words, it attempts to find the index of the most significant bit using algebraic calculations. *This makes it amenable to approximation*. The intuition behind GQ is that the contribution of the most significant set bit to the value of a word is larger than the sum of the contributions of the rest of the set bits. We consider the weight of a non-empty bucket to be proportional to its index. Hence, Gradient Queue occupancy is represented by its curvature function. The curvature function of the queue is the sum of the weight functions of all nonempty buckets in the queue. More specifically, a specific curvature shape corresponds to a specific occupancy pattern. *A proper weight function* ensures the uniqueness of the curvature function per occupancy pattern. It also makes finding the non-empty bucket with the maximum index equivalent to finding the critical point of the queue's curvature (i.e., the point where the derivative of the curvature function of the queue is zero). A sample sketch of a curvature function is illustrated in Figure 4.5.

Exact Gradient Queue On a bucket becoming nonempty, we add its weight function to the queue's curvature function, and we subtract its function when it becomes empty. We define a desirable weight function as one that is: 1) easy to differentiate to find the critical point, and 2) easy to maintain when bucket state changes between empty and non-empty. We use weight function, $2^i(x-i)^2$ where i is the index of the bucket and x is the variable in the space of the curvature function.

This weight function results in queue curvature of the form of $ax^2 - bx + c$, where the critical point is located at $x = b/2a$. Hence, we only care about a and b where $a = \sum_i 2^i$

and $b = \sum_i i2^i$ for all non-empty buckets i. The maintenance of the curvature function of the queue becomes as simple as incrementing and decrementing a and b when a bucket becomes non-empty or empty respectively. Theorem 1, in Appendix A, shows that determining the highest priority non-empty queue can be calculated using $ceil(b/a)$.

A Gradient Queue with a single curvature function is limited by the the range of values a and b can take, which is analogous to the limitation of FFS-based queues by the size of words for which FFS can be calculated in $O(1)$. A natural solution is to develop a hierarchical Gradient Queue. This makes Gradient Queue an equivalent of FFS-based queue with more expensive operations (i.e., division is more expensive than bit operations). However, due to its algebraic nature, Gradient Queue allows for approximation that is not feasible using bit operations.

Approximate Gradient Queue Like FFS-based queues, gradient queue has a complexity of $O(\log_w N)$ where w is the width of the representation of a and b and N is the number of buckets. Our goal is reduce the number of steps even further for each lookup. We are particularly interested in having lookups that can be made in one operation, which can be achieved through approximation. The advantage of the curvature representation of the Gradient Queue compared to FSS-based approaches is that it lends itself naturally to approximation.

A simple approximation is to make the value of a and b corresponding to a certain queue curvature smaller which will allow them to represent a larger number of priority values. In particular, we change the weight function to $2^{f(i)}(x-i)^2$ which results in $a = \sum_i 2^{f(i)}$ and $b = \sum_i i2^{f(i)}$ where $f(i) = i/\alpha$ and α is a positive integer. This approach leads to two natural results: 1) the biggest gain of the approximation is that a and b can now represent a much larger range of values for i which eliminates the need for hierarchical Gradient Queue and allows for finding the minimum element with one step, and 2) the employed weight function is no longer proper. While `BSR` instruction is 8-32x faster than `DIV` [99], the performance gained from the reduced memory lookups required per `BSR` operation.

This approximation stems from using an "improper" weight function. This leads to breaking the two guarantees of a proper weight function, namely: 1) the curvature shape is no longer unique per queue occupancy pattern, and 2) the index of the maximum non-empty bucket no longer corresponds to the critical point of the curvature *in all cases*. In other words, the index of the maximum non-empty bucket, M, is no longer $ceil(b/a)$ due the fact that the weight of the maximum element no longer dominates the curvature function as the growth is sub-exponential. However, this ambiguity does not exist for all curvatures (i.e., queue occupancy patterns).

We characterize the conditions under which ambiguity occurs causing error in identifying the highest priority non-empty bucket. Hence, we identify scenarios where using the approximate queue is acceptable. The effect of $f(i) = i/\alpha$ can be described as introducing ambiguity to the value of $ceil(b/a)$. This is because exponential growth in a and b occurs not between consecutive indices but every α indices. In particular, we find solving the geometric and arithmetic-geometric sums of a and b that $\frac{b}{a} = \frac{M}{1-g(\alpha,M)} + u(\alpha)$ where $g(\alpha, M) = (2^{1/\alpha})^{-M-1}$ is a logarithmically decaying function of M and α. $u(\alpha) = 1/(1-2^{1/\alpha})$ is non-linear but slowly growing function of α. Hence, an approximate GQ can operate as a bucketed-queue where indices start from I_0 where $g(\alpha, M_0) \approx 0$ and end at I_{max} where $2^{f(I_{max})}$ can be precisely represented in the CPU word used to represent a and b. In this case, there is a constant shift in the value $ceil(b/a)$ that is calculated by $u(\alpha)$. For instance, consider an approximate queue with an α of 16. The function $g(\alpha, M)$ decays to near zero at $M = 124$ making the shift $u(\alpha) = 22$. Hence, $I_0 = 124$ and $I_{max} = 647$ which allows for the creation of an approximate queue that can handle 523 buckets. Note that this configuration results in an exact queue only when all buckets between I_0 and I_{max} are nonempty. However, error is introduced when some elements are missing. In Section 4.4.2, we show the effect of this error through extensive experiments; more examples are shown in Appendix B.

Typical scheduling policies (e.g., timestamp-based shaping, Least Slack Time First,

and Earliest Deadline First) will generate priority values for packets that are uniformly distributed over priority levels. For such scenarios, the approximate gradient queue will have zero error and extract the minimum element in one step. This is clearly not true for *all* scheduling policies (e.g., strict priority will probably have more traffic for medium and low level priorities compared to high priority). For cases where the index suggested by the function is of an empty bucket, we perform linear search until we find a nonempty bucket. Moreover, for a cases of a moving range, a circular approximate queue can be implemented as with cFFS.

Approximate queues have been used before for different use cases. For instance, Soft-heap [60] is an approximate priority queue with a bounded error that is inversely proportional to the overhead of insertion. In particular, after n insertions in a soft-heap with an error bound $0 < \epsilon \leq 1/2$, the overhead of insertion is $O(\log(1/\epsilon))$. Hence, `ExtractMin` operation which can have a large error under Soft-heap. Another example is the RIPQ which is was developed for caching [61]. RIPQ relies on a bucket-sort-like approach. However, the RIPQ implementation is suited for static caching, where elements are not moved once inserted, which makes it not very suitable for the dynamic nature of packet scheduling.

4.2.2 Flexibility in Eiffel

Our second objective is to deploy flexible schedulers that have full expressive power to implement a wide range of scheduling policies. Our goal is to provide the network operator with a compiler that takes as input policy description and produces an initial implementation of the scheduler using the building blocks provided in the previous section. Our starting point is the work in PIFO which develops a model for programmable packet scheduling [29]. PIFO, however, suffers from several drawbacks, namely: 1) it doesn't support reordering packets already enqueued based on changes in their flow ranking, 2) it does not support ranking of elements on packet dequeue, and 3) it does not support shap-

ing the output of the scheduling policy. In this section, we show our augmentation of the PIFO model to enable a completely flexible programming model in Eiffel. We address the first two issues by adding programming abstractions to the PIFO model and we address the third problem by enabling arbitrary shaping with Eiffel by changing how shaping is handled within the PIFO model. We discuss the implementation of an initial version of the compiler in Section 4.

PIFO Model Extensions

Before we present our new abstractions, we review briefly the PIFO programming model [29]. The model relies on the Push-In-First-Out (PIFO) conceptual queue as its main building block. In programming the scheduler, the PIFO blocks are arranged to implement different scheduling algorithms.

The PIFO programming model has three abstractions: 1) scheduling transactions, 2) scheduling trees, and 3) shaping transactions. A scheduling transaction represents a single ranking function with a single priority queue. Scheduling trees are formed by connecting scheduling transactions, where each node's priority queue contains an ordering of its children. The tree structure allows incoming packets to change the relative ordering of packets belonging to different policies. Finally, a shaping transaction can be attached to any non-root node in the tree to enforce a rate limit on it. There are several examples of the PIFO programming model in action presented in the original paper [29]. The primitives presented in the original PIFO model capture scheduling policies that have one of the following features: 1) distinct packet rank enumerations, over a small range of values (e.g., strict priority), 2) per-packet ranking over a large range of priority values (e.g., Earliest Deadline First [71]), and 3) hierarchical policy-based scheduling (e.g., Hierarchical Packet Fair Queuing [102]).

Eiffel augments the PIFO model by adding two additional scheduler primitives. The first primitive is *per-flow ranking and scheduling* where the rank of all packets of a flow

```
#On enqueue of packet p of flow f:
f.rank = f.len
#On dequeue of packet p of flow f:
f.rank = f.len
```

Figure 4.6: Example of implementation of Longest Queue First (LQF)

depend on a ranking that is a function of the ranks of all packets enqueued for that specific flow. We assume that a sequence of packets that belong to a single flow should not be reordered by the scheduler. Existing PIFO primitives keep per-flow state but use them to rank each packet individually where an incoming packet for a certain flow does not change the ranking of packets already enqueued that belong to the same flow. The per-flow ranking extension keeps track of that information along with a queue per flow for all packets belonging to that flow. A single PIFO block orders flows, rather than packets, based on their rank. The second primitive is *on-dequeue scheduling* where incoming and outgoing packets belonging to a certain flow can change the rank of all packets belonging to that flow on enqueue and dequeue.

The two primitives can be integrated in the PIFO model. All flows belonging to a *per-flow* transaction are treated as a single flow by scheduling transactions higher in the hierarchical policy. Also note that every individual flow in the flow-rank policy can be composed of multiple flows that are scheduled according to per packet scheduling transactions. Figure 4.6 is an example of how both primitives are expressed in a PIFO programming model. The example implements Longest Queue First (LQF) which requires flows to be ranked based on their length which changes on enqueue and dequeue. The example allows for setting a rank for flows, not just packets (i.e., `f.rank`). It also allows for programming actions on both enqueue and dequeue. We realize that this specification requires tedious work to describe a complex policy that handles thousands of different flows or priorities. However, this specification provides a direct mapping to the underlying priority queues. We believe that defining higher level programming languages describing packet schedulers as well as formal description of the expressiveness of the language to be topics for future

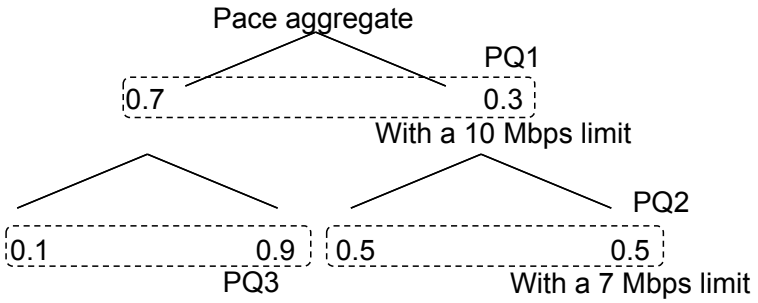


Figure 4.7: Example of a policy that imposes two limits on packets the belong to the rightmost leaf.

research.

Arbitrary Shaping

A flexible packet scheduler should support any scheme of bandwidth division between incoming flows. Earlier work on flexible schedulers either didn't support shaping at all (e.g., OpenQueue) or supported it with severe limitations (e.g., PIFO). We allow for arbitrary shaping by decoupling work conserving scheduling from shaping. A natural approach to this decoupling is to allow any flow or group of flows to have a shaper associated with them. This can be achieved by assigning a separate queue to the shaped aggregate whose output is then enqueued into its proper location in the scheduling hierarchy. However, this approach is extremely inefficient as it requires a queue per rate limit, which can lead to increased CPU and memory overhead. We improve the efficiency of this approach by leveraging recent results that show that any rate limit can be translated to a timestamp per packet, which yields even better adherence to the set rate than token buckets [24]. Hence, we use only one shaper for the whole hierarchy which is implemented using a single priority queue.

As an example, consider the hierarchical policy in Figure 4.7. Each node represents a policy-defined flow with the root representing the aggregate traffic. Each node has a share of its parent's bandwidth, defined by the fraction in the figure. Each node can also have a policy-defined rate limit. In this example, we have a rate limit at a non-leaf node and a leaf node. Furthermore, we require the aggregate traffic to be paced. We map the hierarchical

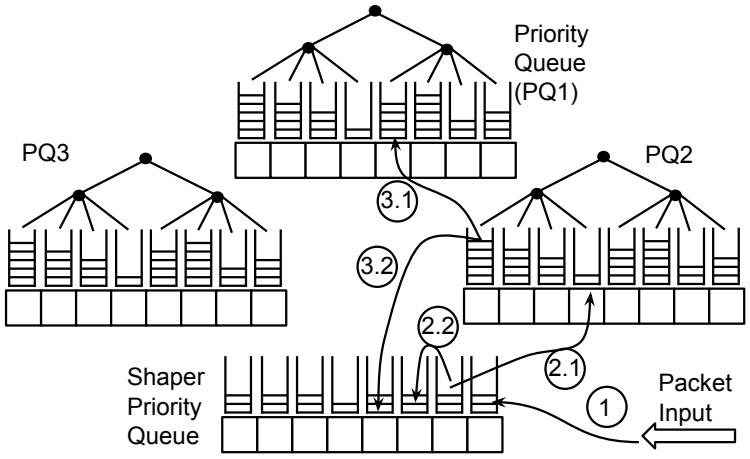


Figure 4.8: A diagram of the implementation of the example in Figure 4.7.

policy in Figure 4.7 to its priority-queue-based realization in Figure 4.8. Per the PIFO model, each non-leaf node is represented by a priority queue. Per our proposal, a single shaper is added to rate limit all packets according to all policy-defined rate limits.

To illustrate how this single shaper works, consider packets belonging to the rightmost leaf policy. We explore the journey of packets belonging to that leaf policy through the different queues, as shown in Figure 4.8. These packets will be enqueued to the shaper with timestamps set based on a 7 Mbps rate to enforce the rate on their node (step 1). Once dequeued from the shaper, each packet will be enqueued to PQ2 (step 2.1) and the shaper according to the 10 Mbps rate limit (step 2.1). After the transmission time of a packet belonging to PQ2 is reached, which is defined by the shaper, the packet is inserted in both the root's (PQ1) priority queue (3.1) and the shaper according to the pacing rate (3.2). When the transmission time, calculated based on the pacing rate, is reached the packet is transmitted. To achieve this functionality, each packet holds a pointer to the priority queue they should be enqueued to. This pointer avoids searching for the queue a packet should be enqueued to. Note that having the separate shaper allows for specifying rate limits on any node in the hierarchical policy (e.g., the root and leaves) which was not possible in the PIFO model, where shaping transactions are tightly coupled with scheduling transactions.

4.3 Eiffel Implementation

Packet scheduling is implemented in two places in the network: 1) hardware or software switches, and 2) end-host kernel. We focus on the software placements (kernel and userspace switches) and show that Eiffel can outperform the state of the art in both settings. We find that userspace and kernel implementations of packet scheduling face significantly different challenges as the kernel operates in an event-based setting while userspace operates in a busy polling setting. We explain here the differences between both implementations and our approach to each. We start with our approach to policy creation.

Policy Creation: We extend the existing PIFO open source model to configure the scheduling algorithm [29, 98]. The existing implementation represents the policy as a graph using the DOT description language and translates the graph into C++ code. We rely on the cFFS for our implementation, unless otherwise stated. This provides an initial implementation which we tune according to whether the code is going to be used in kernel or userspace. We believe automating this process can be further refined, but the goal of this work is to evaluate the performance of Eiffel algorithms and data structures.

Kernel Implementation We implement Eiffel as a qdisc [103] kernel module that implements enqueue and dequeue functions and keeps track of the number of enqueued packets. The module can also set a timer to trigger dequeue. Access to qdiscs is serialized through a global qdisc lock. In our design, we focus on two sources of overhead in a qdisc: 1) the overhead of the queuing data structure, and 2) the overhead of properly setting the timer. Eiffel reduces the first overhead by utilizing one of the proposed data structures to reduce the cost of both enqueue and dequeue operations. The second overhead can be mitigated by improving the efficiency of finding the smallest deadline of an enqueued packet. This operation of `SoonestDeadline()` is required to efficiently set the timer to wake up at the deadline of the next packet. Either of our supported data structures can support this operation efficiently as well.

Userspace Implementation We implement Eiffel in the Berkeley Extensible Software Switch (BESS, formerly SoftNIC [19]). BESS represents network processing elements as a pipeline of modules. BESS is busy polling-based where a set of connected modules form a unit of execution called a task. A scheduler tracks all tasks and runs them according to assigned policies. Tasks are scheduled based on the amount of resources (CPU cycles or bits) they consume. Our implementation of Eiffel in BESS is done in self-contained modules.

We find that two main parameters determine the efficiency of Eiffel in BESS: 1) batch size and 2) queue size. Batching is already well supported in BESS as each module receives packets in batches and passes packets to its subsequent module in a batch. However, we find that batching per flow has an intricate impact on the performance of Eiffel. For instance, with small packet sizes, if no batching is performed per flow, then every incoming batch of packets will activate a large number of queues without any of the packets being actually queued (due to small packet size) which increases the overhead per packet (i.e., queue lookup of multiple queues rather than one). This is not the case for large packet sizes where the lookup cost is amortized over the larger size of the packet improving performance compared to batching of large packets. Batching large packets results in large queues for flows (i.e., large number of flows with large number of enqueued packets). We find that batching should be applied based on expected traffic pattern. For that purpose, we setup `Buffer` modules per traffic class before Eiffel's module in the pipeline when needed. We also perform output batching per flow in units of 10KB worth of payload which was suggested as a good threshold that does not affect fairness at a macroscale between flows [43]. We also find that limiting the number of packets enqueued in Eiffel can significantly affect the performance of Eiffel in BESS. This is something we did not have to deal with in the kernel implementation because of TCP Small Queue (TSQ) [78] which limits number of packets per TCP flow in kernel stack. We limit the number of packets per flow to 32 packets which we find, empirically, to maintain performance.

4.4 Evaluation

We evaluate Eiffel through three use cases. Then, we perform microbenchmarks to evaluate different data structures and measure the effect of approximation on network wide objectives. Finally, we present a guide for picking a data structure.

4.4.1 Eiffel Use Cases

Methodology: We evaluate our kernel and userspace implementation through a set of use cases each with its corresponding baseline. We implement three common use cases, one in kernel and two in userspace. In each use case, we evaluated Eiffel's scheduling behavior as well as its CPU performance as compared to the baseline. The comparison of scheduling behavior was done by comparing aggregate rates achieved as well as order of released packets. However, we only report CPU efficiency results as we find that Eiffel matches the scheduling behavior of the baselines.

A key aspect of our evaluation is determining the metrics of comparisons in kernel and userspace settings. The main difference is that a kernel module can support line rate by using more CPU. This requires us to fix the packet rate we are evaluating at and look at the CPU utilization of different scheduler implementations. On the other hand, a userspace implementation relies on busy polling on one or more CPU cores to support different packet rates. Hence, in the case of userspace, we fix the number of cores used, to one core unless otherwise is stated, and compare the different scheduler implementations based on the maximum achievable rate. The following use cases cover the two settings, presenting the proper evaluation metrics for each setting. We find that for all use cases Eiffel reduces the CPU overhead per packet which leads to lower CPU utilization in kernel settings and higher maximum achievable rate in userspace settings.

Use Case 1: Shaping in Kernel

Traffic shaping (i.e., rate limiting and pacing) is an essential operation for efficient utilization [42] and correct operation of modern protocols (e.g., both TIMELY [31] and BBR

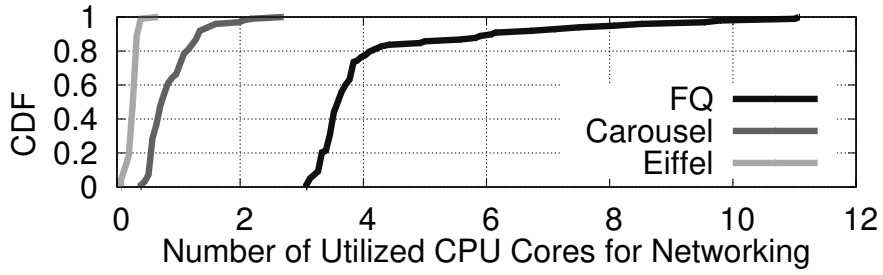


Figure 4.9: A comparison between the CPU overhead of the networking stack using FQ/-pacing, Carousel, and Eiffel.

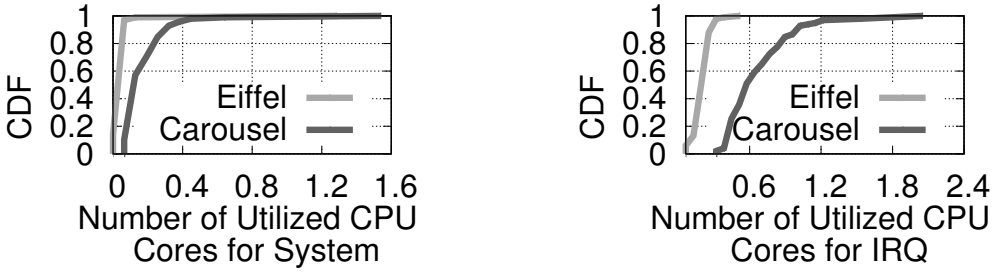


Figure 4.10: A Comparison between detailed CPU utilization of Carousel and Eiffel in terms of system processes (left) and soft interrupt servicing (right).

[33] require per flow pacing). Recently, it has been shown that the canonical kernel shapers (i.e., FQ/pacing [45] and HTB qdiscs [44]) are inefficient due to reliance on inefficient data structures; they are outperformed by the userspace-based implementation in Carousel [24]. To offer a fair comparison we implement all systems in the kernel.

We implement a rate limiting qdisc whose functionality matches the rate limiting features of the existing FQ/pacing qdisc [45]. In particular, we allow a flow to set a `SO_MAX_PACING_RATE` parameter which is a rate limit for the flow. Furthermore, we also calculate a pacing rate limit for each TCP flow in case of unspecified maximum rate. We also compare Eiffel to a Carousel-based qdisc. We implement a qdisc where all packets are queued in a timing wheel. A timer fires every time instant (according to the granularity of the timing wheel) and checks whether it has packets that should be sent.

We implemented Eiffel as a qdisc. The queue is configured with 20k buckets with a maximum horizon of 2 seconds and only the shaper is used. We implemented the qdisc in

kernel v4.10. We modified only `sock.h` to keep the state of each socket allowing us to avoid having to keep track of each flow in the qdisc. We conduct experiments for egress traffic shaping between two servers within the same cluster in Amazon EC2. We use two `m4.16xlarge` instances equipped with 64 cores and capable of sustaining 25 Gbps. We use `neper` [92] to generate traffic with a large number of TCP flows. In particular, we generate traffic from 20k flows and use `SO_MAX_PACING_RATE` to rate limit individual flows to achieve a maximum aggregate rate of 24 Gbps. This configuration constitutes a worst case in terms of load for all evaluated qdiscs as it requires the maximum amount of calculations. We measure overhead in terms of the number of cores used for network processing which we calculate based on the observed fraction of CPU utilization. Without `neper` operating, CPU utilization is zero, hence, we attribute any CPU utilization during our experiments to the networking stack, except for the CPU portion attributed to userspace processes. We track CPU utilization using `dstat`. We run our experiments for 100 seconds and record the CPU utilization every second. This continuous behavior emulates the behavior handled by content servers which were used to evaluate Carousel [24].

Figure 4.9 shows the overhead of all three systems. It is clear that Eiffel is superior, outperforming FQ by a median 14x and Carousel by 3x. We find the overhead of FQ to be consistent with earlier results [24]. This is due to its complicated data structure which keeps track internally of active and inactive flows and requires continuous garbage collection to remove old inactive flows. Furthermore, it relies on RB-trees which increases the overhead of reordering flows on every enqueue and dequeue. To better understand the comparison with Carousel, we look at the breakdown of the main components of CPU overhead, namely overhead spent on *system processes* and *servicing software interrupts*. Figure 4.10 details the comparison. We find that the main difference is in the overhead introduced by Carousel in firing timers at constant intervals while Eiffel can trigger timers exactly when needed (Figure 4.10 right). The overhead of the data structures in both cases introduces minimal overhead in system processes (Figure 4.10 left).

```
#On enqueue of packet p of flow f:
f = flow(p)
# Shaping transaction ranking
f.r_rank = f.r_rank + \
    p.size / f.reservation
f.l_rank = f.l_rank + p.size / f.limit
# Scheduling transaction ranking
f.s_rank = f.s_rank + p.size / f.share
```

Figure 4.11: Implementation of hClock in Eiffel.

Use Case 2: Flow Ranking Policy in Userspace

We demonstrate the efficiency of Eiffel's Per-Flow Rank programming abstraction by implementing hClock [43], a hierarchical packet scheduler that supports limits, reservations, and proportional shares. hClock is the main implementation of scheduling in NetIOC in VMware's vSphere [25] and relies on min heaps the incur $O(\log n)$ overhead per packet batch [43]. We implement hClock based on its original specs and compare that with hClock using Eiffel. Both versions are implemented in BESS; each in a self-contained single module. We also attempt to replicate hClock's behavior using the traffic control (*tc*) mechanisms in BESS. However, this requires instantiating a module corresponding to every flow which incurs a large overhead for a large number of flows.

Figure 4.11 shows the implementation of hClock using PIFO model along with Eiffel extensions. hClock relies on three ranks to determine order of packets: `r_rank` corresponds to the reservation of the flow which is the ranking used to order flows in the arbitrary shaper, `s_rank` is calculated based on the share of a flow compared to its siblings in the tree and this rank is used in the scheduling transaction, and `l_rank` is the rank of a packet based on the rate limit of its flow. The packet dequeued from the shaper belongs to the flow with the smallest `r_rank` smaller than the current time whose `l_rank` is also smaller than current time. Allowing each flow to have multiple ranks as such creates a complication not capture by the Eiffel syntax. Hence, this complication requires adding some code manually. However, it does not hurt efficiency.

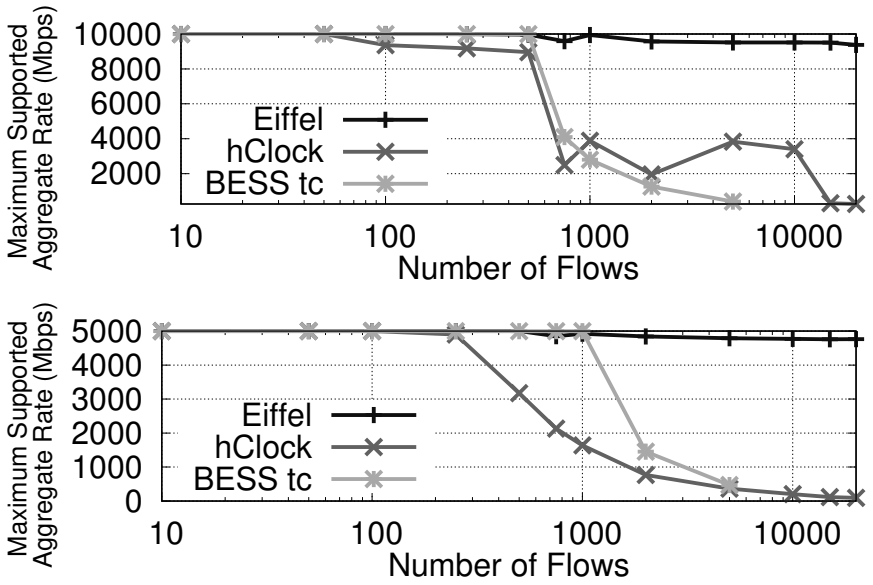


Figure 4.12: Comparison between maximum supported aggregate rate limit (top) and behavior at a rate limit of 5 Gbps (bottom) for hClock, Eiffel's implementation of hClock, and BESS tc on a single core with no batching.

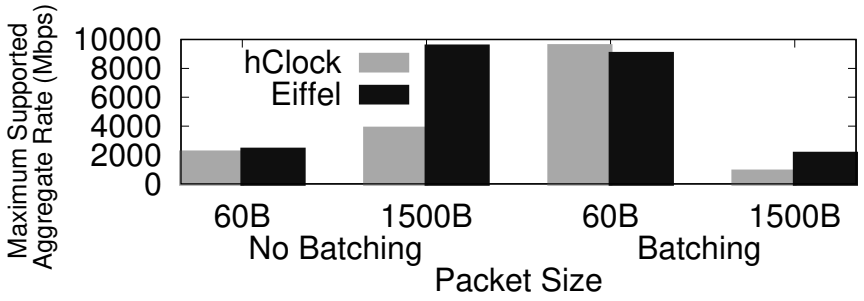


Figure 4.13: Effect of batching and packet size on throughput for both Eiffel and hClock for 5k flows.

We conduct our experiments on lab machines equipped with Intel Xeon CPU ES-1620 with 8 cores, 64GB of memory, and Intel X520-SR2 dual port NICs capable of an aggregate of 20 Gbps. We use a simple packet generator implemented in BESS and a simple round robin annotator to distribute packets over traffic classes. We change the number of traffic classes (i.e., flows) and measure the maximum aggregate rate. To demonstrate the efficiency of Eiffel, we conduct all experiments on a single core. We run each experiment for 10 seconds and plot the average observed throughput at the port of the sender which matches the rate at the receiver.

```
#On enqueue of packet p of flow f:
f.rank = min(p.rank, f.rank)
#On dequeue of packet p of flow f:
f.rank = min(p.rank, f.front().rank)
```

Figure 4.14: Implementation of pFabric in Eiffel.

Figure 4.12 shows the results of our experiments for varying number of flows. Packets are 1500B which is MTU size. We run experiments at a line rate of 10 Gbps and with a rate limit of 5 Gbps. It is clear that Eiffel can support line rate at up to 40x the number of flows compared to hClock at both aggregate rate limits and even larger advantage compared to tc in BESS. We find that all three implementations scale well with large number of flows when operating on two cores. However, it is clear that using Eiffel results in significant savings.

We also investigate the combined effects of packet size and per-flow batching (Figure 4.13). As discussed earlier, performance for small packet sizes requires batching to improve performance due to memory efficiency (i.e., all packets within the same batch are queued in the same place). However, for large packet sizes the value of batching is offset by the overhead of large queue lengths. Our results validate this hypothesis showing that when per-flow batching is applied with small packet sizes both hClock and Eiffel can maintain performance close to line rate with Eiffel performing worse than hClock by 5-10%. For per-flow batching with large packet sizes, per-flow queues are larger causing performance degradation. We note that the typical behavior of a network will avoid per-flow batching, as it might cause unnecessary latency. In that case, Eiffel outperforms hClock and can sustain line rate for large packet sizes, which should also be the common case.

Use Case 3: Least/Largest X First in Userspace

One of the most widely used patterns for packet scheduling is ordering packets such that the flow or packet with the least or most of some feature exits the queue first. Many examples of

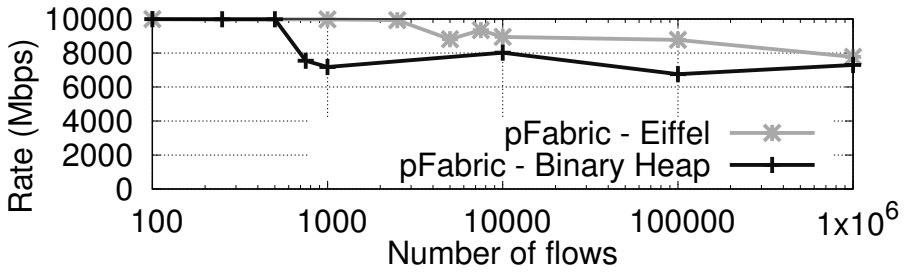


Figure 4.15: Performance of pFabric implementation using cFFS and a binary heap showing Eiffel sustaining line rate at 5x number of flows.

such policies have been promoted including Least Slack Time First (LSTF) [104], Largest Queue First (LQF), and Shortest/Least Remaining Time First (SRTF). We refer to this class of algorithms as L(X)F. This class of algorithms is interesting as some of them were shown to provide theoretically proven desirable behavior. For instance, LSTF was shown be a universal packet scheduler that can emulate the behavior of any scheduling algorithm [104]. Furthermore, SRTF was shown to schedule flows close to optimally within the pFabric architecture [11]. We show that Eiffel can improve the performance of this class of scheduling algorithms.

We implement pFabric as an instance of such class of algorithms where flows are ranked based on their remaining number of packets. Every incoming and outgoing packet changes the rank of all other packets belonging to the same flow, requiring on dequeue ranking. Figure 4.14 shows the representation of pFabric using the PIFO model with per-flow ranking and on dequeue ranking provided by Eiffel. We also implemented pFabric using $O(\log n)$ priority queue based on a Binary Heap to provide a baseline. Both designs were implemented as queue modules in BESS. We used packets of size 1500B. Operations are all done on a single core with a simple flow generator. All results are the average of ten experiments each lasting for 20 seconds. Figure 4.15 shows the impact of increasing the number of flows on the performance of both designs. It is clear that Eiffel has better performance. The overhead of pFabric stems from the need to continuously move flows between buckets which has $O(1)$ using bucketed queues while it has an overhead of $O(n)$ as it requires re-

heapifying the heap every time. The figure also shows that as the number of flows increases the value of Eiffel starts to decrease as Eiffel reaches its capacity.

4.4.2 Eiffel Microbenchmark

Our goal in this section is evaluate the impact of different parameters on the performance of different data structures. We also evaluate the effect of approximation in switches on network-wide objectives. Finally, we provide guidance on how one should choose among the different queuing data structures within Eiffel, given specific scheduler user-case characteristics. To inform this decision we run a number of microbenchmark experiments. We start by evaluating the performance of the proposed data structures compared to a basic bucketed priority queue implementation. Then, we explore the impact of approximation using the gradient queue both on a single queue and at a large network scale through ns2-simulation. Finally, we present our guide for choosing a priority queue implementation.

Experiment setup: We perform benchmarks using Google's benchmark tool [105]. We develop a baseline for bucketed priority queues by keeping track of non-empty buckets in a binary heap, we refer to this as BH. We ignore comparison-based priority queues (e.g., Binary Heaps and RB-trees) as we find that bucketed priority queues performs 6x better in most cases. We compare cFFS, approximate gradient queue (Approx), and BH. In all our experiments, the queue is initially filled with elements according to queue occupancy rate or average number of packet per bucket parameters. Then, packets are dequeued from the queue. Reported results (i.e., y-axis of figures 4.16 and 4.17) are in terms of million packets per seconds.

Effect of number of packet per bucket: The number of buckets configured in a queue is the main determining factor for the overhead of a bucketed queue. Note that this parameter controls queue granularity which is the priority interval covered by a bucket. High granularity (i.e., large number of buckets) implies a smaller number of packets per bucket for the same workload. Hence, the number of packets per bucket is a good proxy to the

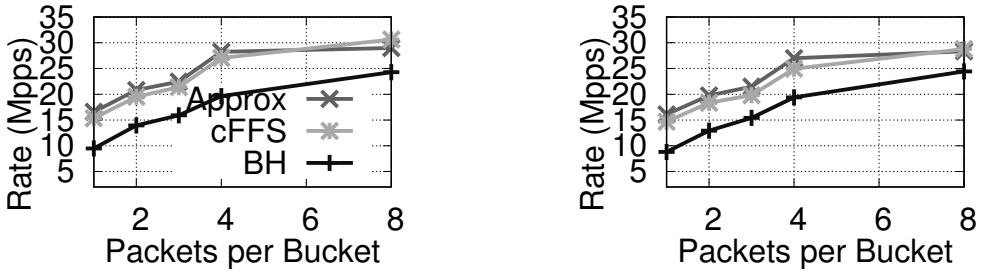


Figure 4.16: Effect of number of packets per bucket on queue performance for 5k (left) and 10k (right) buckets.

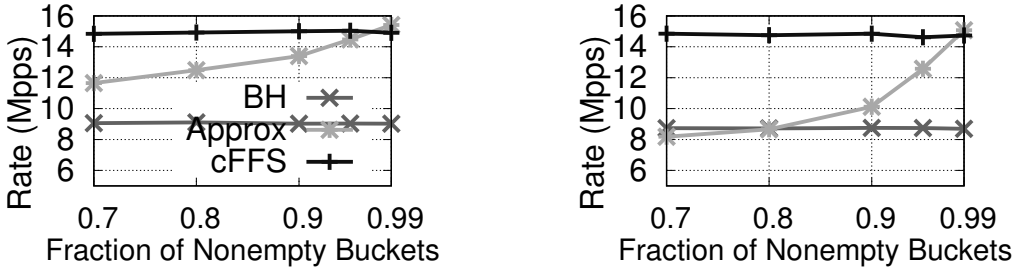


Figure 4.17: Effect of queue occupancy on performance of Approximate Queue for 5k (left) and 10k (right) buckets.

configured number of buckets. For instance, if we choose a large number of buckets with high granularity, the chance of empty buckets increases. On the other hand, if we choose a small number of buckets with coarser granularity, we get higher number of elements per bucket. This proxy is important because in the case of the approximate queue, the main factor affecting its performance is the number of empty buckets.

Figure 4.16 shows the effect of increasing the average number of packets per bucket for all three queues for 5k and 10k buckets. For a small number of packets per bucket, which also reflects choosing a fine grain granularity, the approximate queue introduces up to 9% improvement in performance in the case of 10k buckets. In such cases, the approximate queue function has zero error which makes it significantly better. As the number of the packets per bucket increases, the overhead of finding the smallest indexed bucket is amortized over the total number of elements in the bucket which makes FFS-based and approximate queues similar in performance.

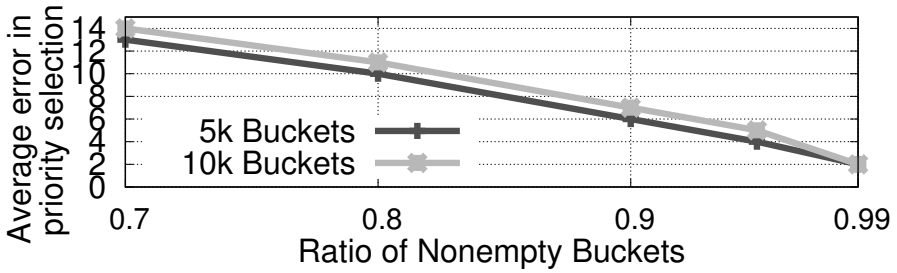


Figure 4.18: Effect of having empty buckets on the error of fetching the minimum element for the approximate queue.

We also explore the effect of having empty buckets on the performance of the approximate queue. Empty buckets cause errors in the curvature function of the approximate queue which in turn trigger linear search for non-empty buckets. Figure 4.16 shows throughput of the queue for different ratios of non-empty buckets. As expected, as the ratio increases the overhead decreases which improves the throughput of the approximate queue. Figure 4.18 shows the error in the approximate queue's fetching of elements. As the number of empty buckets increases the error in the approximate queue is larger and the overhead of linear search grows. We suggest that cases where the queue is more than 30% empty should trigger changes in the queue's granularity based on the queue's CPU performance and to avoid allocating memory to buckets that are not used.

The granularity of the queue determines the representation capacity of the queue. It is clear for our results that picking low granularity (i.e., high number of packets per bucket) yields better performance in terms of packets per second. On the other hand, from a networking perspective, high granularity yields exact ordering of packets. For instance, a queue with a granularity of 100 microseconds cannot insert gaps between packets that are smaller than 100 microseconds. Hence, we recommend configuring the queue's granularity such that each bucket has at least one packet. This can be determined by observing the long term behavior of the queue. We also note that this problem can be solved by having non-uniform bucket granularity which is dynamically set to achieve the result of at least one packet per bucket. We leave this problem for future work.

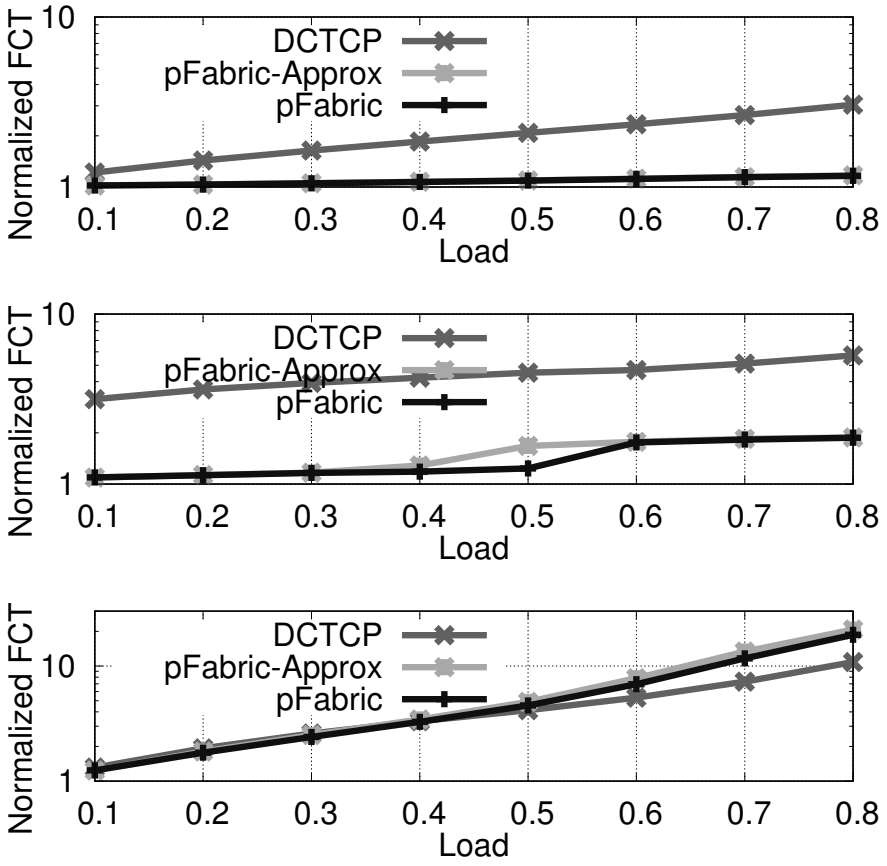


Figure 4.19: Effect of using an Approximate Queue on the performance of pFabric in terms of normalized flow completion times under different load characteristics: Average FCT for (0, 100kB] flow sizes, 99th percentile FCT for (0, 100kB] for sizes, and Average FCT for (10MB, inf) flow sizes.

Impact of Approximation on Network-wide Objectives: A natural question is: how does approximate prioritization, at *every* switch in a network, affect network-wide objectives? To answer that question, we perform simulations of pFabric, which requires prioritization at every switch. Our simulation are based on `ns2` simulations provided by the authors of pFabric [11] and the plotting tools provided by the authors of QJump [16]. We change only the priority queuing implementation from a linear search-based priority queue to our Approximate priority queue and increase queue size to handle 1000k elements. We use DCTCP [30] as a baseline to put the result in context. Figure 4.19 shows a snapshot of results of the simulations of a 144 node leaf-spine topology. Due to space limitations, We show results for only web-search workload simulations which are based on clusters in Mi-

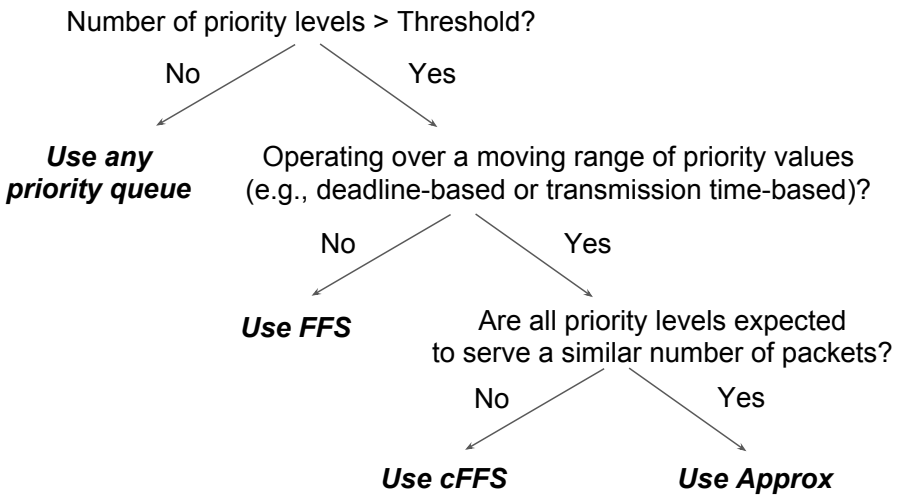


Figure 4.20: Decision tree for selecting a priority queue based on the characteristics of the scheduling algorithm.

crosoft datacenters [30]. The load is varied between 10% to 80% of the load observed. We note that the setting of the simulations is not relevant for the scope of this work, however, what is relevant is comparing the performance of pFabric using its original implementation to pFabric using our approximate queue. We find that approximation has minimal effect on overall network behavior which makes performance on a mircorscale the only concern in selecting a queue for a specific scheduler.

A Guide for Choosing a Priority Queue for Packet Scheduling Figure 4.20 summarizes our takeaways from working with the proposed queues. For a small number of priority levels, we find that the choice of priority queue has little impact and for most scenarios a bucket-based queue might be overkill due to its memory overhead. However, when the number of priority levels or buckets is larger than a threshold the choice of queues makes a significant difference. We found in our experiments that this threshold is 1k and that the difference in performance is not significant around the threshold. We find that if the priority levels are over a fixed range (e.g., job remaining time [11]) then an FFS-based priority queue is sufficient. When the priority levels are over a moving range, where the number of levels are not all equally likely (e.g., rate limiting with a wide range of limits [24]), it is better to use cFFS priority queue. However, for priority levels over a moving range with

highly occupied priority levels (e.g., Least Slack Time-based [104] or hierarchical-based schedules [43]) approximate queue can be beneficial.

Another important aspect is choosing the number of buckets to assign to a queue. This parameter should be chosen based on both the desired granularity and efficiency which form a clear trade-off. Proposed queues have minimal CPU overhead (e.g., a queue with a billion buckets will require six bit operations to find the minimum non-empty bucket using a cFFS). Hence, the main source of efficiency overhead is the memory overhead which has two components: 1) memory footprint, and 2) cache freshness. However, we find that most scheduling policies require thousands to tens of thousands of elements which require small memory allocation for our proposed queues.

4.5 Discussion

Impact of Eiffel on Next Generation Hardware Packet Schedulers: Scheduling is widely supported in hardware switches using a fixed short list of scheduling policies, including shaping, strict priority, and Weighted Round Robin [65, 66, 67, 29]. Recently, programmable networking hardware has been moving steadily towards wide adoption and deployment over the past few years. For example, P4 [106] provides a data plane programming language following match action tables. Furthermore, P4 allows for setting meta data per packet which provide a way to schedule packets that relies on packet tagging to determine their relative ordering. However, P4, so far, still does not provide a programming model for packet scheduling [107]. PIFO [29] proposed a hardware scheduler programming abstraction that can be integrated in a programmable data plane model. Other efforts for programmable network hardware include SmartNICs by Microsoft that leverages FPGAs to implement network logic in programmable NICs [96].

We believe that the biggest impact Eiffel will have is making the case for a reconsideration of the basic building blocks of the packet schedulers in hardware. Current proposals for packet scheduling in hardware (e.g., PIFO model [29] and SmartNICs [96]), rely on

parallel comparisons of elements in a single queue. This approach limits the size of the queue. Earlier proposals that rely on pipelined-heaps [68, 69, 70] required a priority queue that can capture the whole universe of possible packet rank values, which requires significant hardware overhead. We see Eiffel as a step on the road of improving hardware packet schedulers by reducing the number of parallel comparisons through an FFS-based queue meta data or through an approximate queue metadata. For instance, Eiffel can be employed in a hierarchical structure with parallel comparisons to increase the capacity of individual queues in a PIFO-like setting. Future programmable schedulers can implement a hardware version of cFFS or the approximate queue and provide an interface that allows for connecting them according to programmable policies. While the implementation is definitely not straight forward, we believe this to be the natural next step in the development of scalable packet schedulers.

Impact of Eiffel on Network Scheduling Algorithms and Systems: Performance and capacity of packet schedulers are key factors in designing a network-wide scheduling algorithm. It is not rare that algorithms are significantly modified to map a large number of priority values to the eight priority levels offered in IEEE 802.1Q switches [97] (e.g., Qjump [16]). Otherwise, algorithms are deemed unrealistic when they require priority queues with large number of priorities. We believe Eiffel can enable a reexamination of the approach to the design of such algorithms taking into account that complex scheduling can be performed using Eiffel in software in Virtual Network Functions (VNF). Hence, in settings that tolerates a bit of latency (e.g., systems already employing latency), network scheduling can be moved from the hardware to software. On the other hand, if the network is not congested, such scheduling can be performed at endhosts [104].

4.6 Summary

Efficient packet scheduling is a crucial mechanism for the correct operation of networks. Flexible packet scheduling is a necessary component of the current ecosystem of pro-

grammable networks. In this chapter, we showed how Eiffel can introduce both efficiency and flexibility for packet scheduling in software relying on integer priority queuing concepts and novel packet scheduling programming abstractions. We showed that Eiffel can achieve orders of magnitude improvements in performance compared to the state of the art while enabling packet scheduling at scale in terms of both number of flows or rules and line rate. We believe that our work should enable network operators to have more freedom in implementing complex policies that correspond to current networks needs where isolation and strict sharing policies are needed. Eiffel also makes the case for a reconsideration of the basic building blocks of packet schedulers which should motivate future work on schedulers implemented in hardware in NICs and switches.

CHAPTER 5

ANNULUS: A DUAL CONGESTION CONTROL LOOP FOR DATACENTER AND WAN TRAFFIC AGGREGATES

Datacenters host a large variety of applications including web search, social networks, recommendation systems, and database storage. These applications demand stringent performance requirements while relying heavily on communication among servers that can be within the same datacenter or in different datacenters. The number of servers involved in simultaneous *intra- and inter-datacenter* communication can be in the tens to hundreds or even in the thousands. The communication can also be quite intensive. Intra-datacenter traffic for distributed applications often features heavy fan-in traffic patterns and traffic microbursts. Inter-datacenter traffic travels over high-end premium peering links at high average throughputs, insuring cost-effectiveness as datacenter operators typically have to pay for their peak (95 percentile) utilization.

Inter-datacenter (WAN) and intra-datacenter (LAN) traffic will typically share network bottlenecks. Yet, they are generally different both in the nature of the applications using them and the network equipment carrying their traffic. WAN traffic generally faces large RTTs and jitter in latency, while having small per flow throughput. Furthermore, WAN equipment, used in access networks, is typically expensive with deep buffers. On the other hand, LAN equipment is typically more cost effective with shallow buffers, where traffic is expected to have very small RTTs. This divergence between WAN and LAN traffic motivated the development of customized congestion control protocols tailored for the requirements of each type of traffic. Intra-datacenter protocols, including DCTCP [30], TIMELY [31], and DCQCN [32], aim at estimating queue lengths and maintaining short queues. While WAN congestion control protocols (e.g., BBR [33], and Copa [34]) are typically more aggressive, or have a "competitive mode", to provide robustness to loss and avoid

long ramp up times caused by long RTTs. This aggressive behavior is typically motivated by the need to compete with buffer filling protocols (e.g., CUBIC [35] and NewReno [36]) and is enabled by the deeper buffers supported by WAN equipment. However, WAN and LAN traffic share their first few hops which, by design, allows WAN traffic to get more bandwidth compared to LAN traffic due to its aggressive nature.

In this chapter, we first aim to understand how WAN and LAN traffic interact in the presence of congestion control and, in particular, whether one type of traffic negatively impacts the other. To this end, we study data from two production clusters belonging to a large datacenter operator over the period of a month (§5.1). We find that WAN traffic can negatively impact LAN traffic and, in particular, the tail latency of LAN traffic increases with heavier WAN traffic. We also observe that congestion due to the interaction between WAN and LAN traffic typically occurs at the ToR uplinks, the first point of oversubscription in the studied network topologies. We call this type of congestion event *near-source congestion* because such links are normally close to the source. Another type of congestion event, *deep path congestion*, can still be present at any point on the rest of the path of the packet. It is typically caused by homogeneous traffic (i.e., pure WAN or LAN traffic).

Using insights from our study of the production clusters above, we then consider the question of how congestion control can be designed to handle the mix of WAN and LAN traffic in a manner where WAN traffic is prevented from negatively affecting LAN traffic. More generally, we are interested in an approach where the impact of one type of traffic on the other is configurable. Handling the mixture of WAN and LAN traffic presents a significant challenge. One approach that can be deployed at near-source bottlenecks is to separate WAN and LAN traffic into different sets of QoS queues. Queues within each set are then further allocated to different application priorities. However, commodity switches have only eight QoS levels, which means that relying on this approach wastes an already scarce resource. Further, these queues typically share memory which can still allow one type of traffic to impact the behavior of the other. Another approach is to rely on central allocation

of bandwidth for different types of traffic, as done for WAN traffic scheduling [12]. However, scheduling LAN traffic requires a faster control loop and a significant communication overhead making such solutions challenging. A solution for near-source congestion should provide isolation between WAN and LAN traffic while relying on network feedback within the timescale of LAN RTT (§5.2).

In this chapter we propose, design and evaluate *Annulus,* a scheme designed to handle the mixture of WAN and LAN traffic. Annulus (§5.3) relies on a dual congestion control loop protocol where each loop targets one of the types of congestion events: near-source and deep path congestion. Annulus relies on typical WAN and LAN congestion control protocols to handle deep path congestion for their corresponding cases. Annulus employs a *second* control loop common to LAN and WAN traffic that relies on network feedback from near-source bottlenecks to react within a LAN RTT. For this second loop, Annulus relies on per flow congestion feedback supported by commodity datacenter switches. Such fast feedback signal is ideal for fast detection of congestion events near the traffic source [108]. Our design addresses two challenging aspects of Annulus design; namely, the design of the near-source control loop (§5.3.1) and how the two control loops should interact (§5.3.2).

We implement Annulus in a userspace network processing stack and in simulations (§3.4). We evaluate the Annulus implementation on a testbed of three racks, two in one cluster, and one in separate cluster, connected by a private WAN. In the testbed, we compare Annulus to a setup where DCTCP is used for LAN congestion control and BBR is used for WAN congestion control. We find that Annulus improves local traffic tail latency by 43.2% at medium loads, and by up to 56x in cases where WAN traffic dominates the network. Annulus improves fairness between WAN and LAN while allowing for configurable weighted fairness. We also find that Annulus improves bottleneck utilization by 10%. In simulations, we compare Annulus to TCP CUBIC, DCTCP, and DCQCN under various workloads. We find that Annulus reduces LAN flow completion time by up to 3.5x and 2x compared to DCTCP and DCQCN, respectively. It also improves WAN flow completion

time by around 10% compared to both DCTCP and DCQCN.

5.1 A Closer Look at WAN and LAN Traffic Interaction

Early deployments of congestion control in networks have typically adopted a one-size-fits-all approach across different types of networks such as Internet (WAN) or datacenters (LAN) with the goal of (i) maximizing utilization, (ii) fairness, and (iii) minimizing queuing or losses. Recently, we see congestion control algorithms becoming more specialized for the target environment such as DCTCP for datacenters [30], and BBR for the Internet [33]. While they still share the same goals, the specialization addresses the difference in the network and traffic characteristics in order to provide better performance for its respective targets. Specialization, however, presents coexistence challenges in situations when the two types of traffic, controlled by the different algorithms, share part of the network and compete with each other. This exacerbates existing coexistence challenges stemming from the difference in nature between the two types of traffic even when using the same congestion control algorithm.

In datacenter LANs, network feedback is typically fast (i.e., RTT values are very small), with relatively small bandwidth-delay product. This allows for the use of fast reacting congestion control algorithms that can keep queues short relying on fast feedback from the network, while keeping utilization high by sending a small number of packets per flow. This approach is also facilitated by having the whole datacenter network under one federation, making network feedback and network measurements reliable and deployment of customized packet marking algorithms feasible. This design philosophy can be seen in several algorithms that are designed to react proportional to network congestion while keeping queues short (e.g., DCTCP[30], TIMELY[31], and DCQCN[32]).

WAN paths, on the other hand, are much longer compared to LAN paths, causing orders of magnitude larger RTT values. Hence, WAN congestion control algorithms tend to be conservative in the face of congestion to avoid large latency, while achieving high

utilization through network probing that relies on filling the buffer of the bottleneck. Behavior of WAN congestion control has been improved by making the initial window larger to avoid slow down by slow start [109]. Another improvement is to avoid severe reaction to drops caused by transient queue buildup, and only react to drops caused by network congestion, improving network utilization [33]. More recent innovations attempt to estimate queueing delay and use it to calculate a target transmission rate, with the goal of keeping queues short to keep the estimates accurate [34]. These approaches bring WAN congestion control closer to maintaining the exact number of packets in flight needed to maintain high utilization without causing many drops [33]. However, WAN protocols have to keep a large number of packets in flight compared to LAN protocols due to their larger bandwidth-delay product, attempting to achieve high utilization by filling network buffers.

Considering the above differences one can conclude that WAN congestion control can be more aggressive in grabbing bandwidth and introducing delays when competing with LAN traffic. For the rest of this section, we validate this hypothesis by through two approaches: 1) small scale simulation of such interactions, and 2) measurements of two production clusters at Google. Our main finding is that the difference in congestion control schemes and traffic characteristics cause the WAN traffic to negatively impact the performance of the LAN traffic.

5.1.1 Simulating WAN and LAN Interaction

We perform simple NS2 [110] simulations to carefully examine the interaction between DCTCP [30] and TCP NewReno [36] traffic. We pick the two protocols as representatives of their respective classes of congestion control protocols. In particular, DCTCP attempts to maintain a short queue

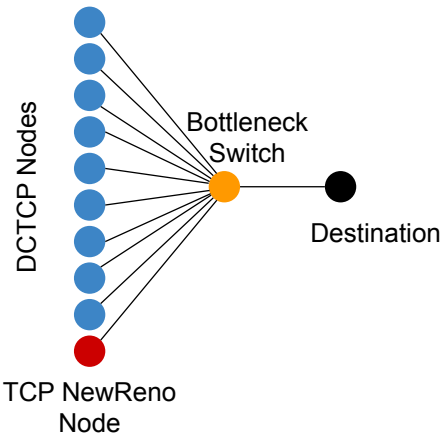
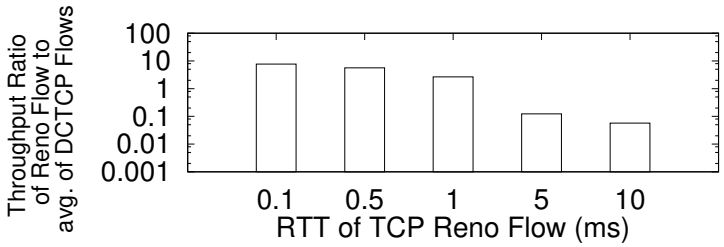
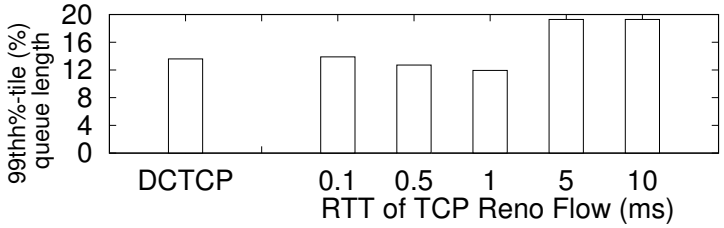


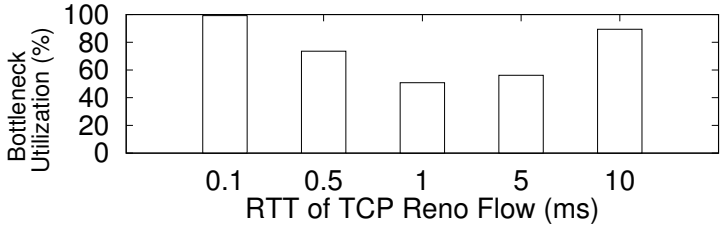
Figure 5.1: Illustration of the simulated network.



(a) The ratio between the throughput of the TCP NewReno flow and the average throughput of all DCTCP flows (y-axis in logarithmic scale).



(b) 99th percentile queue length at the bottleneck switch.



(c) Utilization of the bottleneck link

Figure 5.2: Interaction between TCP NewReno and DCTCP flows during the simulation of the network in Figure 5.1.

while still being robust to wide range of pa-

rameter choices [111]. On the other hand, TCP Reno represents buffer filling algorithms that react only to packet loss. We select LAN to WAN traffic ratio of 9:1 which is consistent with earlier measurements [112, 113] of traffic mixes in production datacenters as well as our own measurements presented in the next section.

We simulate the network shown in Figure 5.1. The network simulates an incast scenario where 9 DCTCP flows compete with a single TCP NewReno flow. All links have a capacity of 1 Gbps. All links have a default latency of 50 μs. We vary the latency of the link connecting the NewReno to examine the impact of RTT on the behavior of the two

competing congestion control algorithms. We look at three metrics: 1) the ratio between the throughput of the TCP NewReno flow and the average throughput of all DCTCP flows (Figure 5.2a), 2) the 99th percentile queue length for the duration of the experiment as a proxy of tail latency (Figure 5.2b), and 3) the bottleneck utilization (Figure 5.2c).

The obtained results are consistent with the hypothesis discussed earlier. The results can be viewed as have two modes of operations: 1) small WAN RTT values lower than 20x LAN RTT (e.g., up to 1ms in our simulations), and 2) large WAN RTT values. For short WAN RTT values, the difference in the objectives of the two congestion control protocols determine their interaction. In particular, TCP NewReno is aggressive and attempts to fill the buffer of the bottleneck switch, capturing between 2.6x and 7.7x the throughput achieved by the average DCTCP flow. Within that range of WAN RTT values, the longer the RTT value the slower the reaction of the TCP NewReno algorithm. This leads to lower utilization due to fluctuation of throughput caused by the "sawtooth" pattern of TCP congestion control, which becomes of lower frequency as RTT increases.

As WAN RTT increases, RTT unfairness [114] starts to be the dominant factor controlling the interaction. In particular, the fraction of bandwidth captured by the TCP NewReno flow starts to shrink compared to that of the average DCTCP flow, capturing throughput as low as 0.05x of the average DCTCP flow at 10ms WAN RTT. The reaction of the TCP NewReno also gets slower. This means that it takes longer to back-off from a large window size, leading to a 40% increase in 99th percentile queue length. On the other hand, as the share of the TCP NewReno of the overall throughput shrinks utilization improves as utilization is dominated by DCTCP flows.

The main conclusion here is that under the conflicting objectives of the two congestion control algorithms, there is no mode of operation where both types of traffic achieve good performance. At small WAN RTT, LAN flows achieve low throughput. On the other hand, at large WAN RTT, LAN flows have high latency and WAN flows get almost no throughput. Hence, a new scheme is needed to achieve a mode of operation where no type of traffic has

to be negatively affected by the behavior of the other type.

5.1.2 WAN and LAN in the Wild

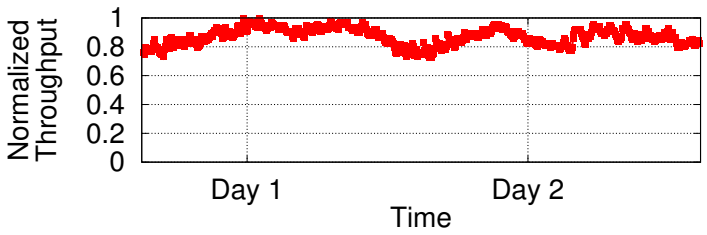
We validate and augment the observations we made in the previous section by look at the interaction between WAN and LAN traffic in a production environment. We examine the interaction in the wild between BBR, as a WAN protocol, and DCTCP, as a LAN protocol[1]. We study the interaction between WAN and LAN in the wild by studying traffic from two production clusters. In the studied clusters, intra-cluster congestion control uses DCTCP[2], while WAN traffic uses BBR. Note that the switches deployed in the WAN used in our experiments are shallow buffered [54], where we use a patched, less aggressive, experimental version of BBR. Hence, the results we report here make a stronger argument for competition between WAN and LAN traffic, as WAN in our experiments is less aggressive.

We collect throughput measurements, averaged over periods of five minutes, and end-to-end Remote Procedure Call (RPC) latency measurements, averaged over periods of twenty minutes. Both measurements are collected at end hosts. End-host measurements are classified into WAN (i.e., traffic exiting the cluster) and LAN (i.e., traffic that remains in the cluster). Furthermore, we collect drop rates at switches, averaged over five minutes, to determine the bottleneck location and severity. We study flows behavior aggregated over all machines in the clusters over the period of a month, and correlate end-host measurements with switch measurements.

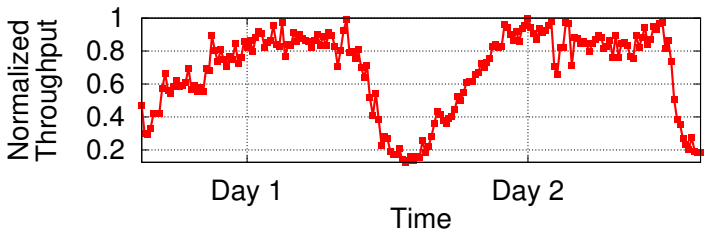
We start by focusing on Cluster 1 over a period of two days. Figure 5.3a shows the normalized total traffic sent by machines in the cluster. The figure shows the total load normalized by the peak load. The figure shows that total load is relatively stable over the period of two days, with average load being 87% of the maximum load. However, the WAN load changes significantly as shown in Figure 5.3b, dropping to 20% of maximum WAN

[1] We study the interaction between DCTCP, DCQCN, and TCP CUBIC in simulation in Section 5.4.

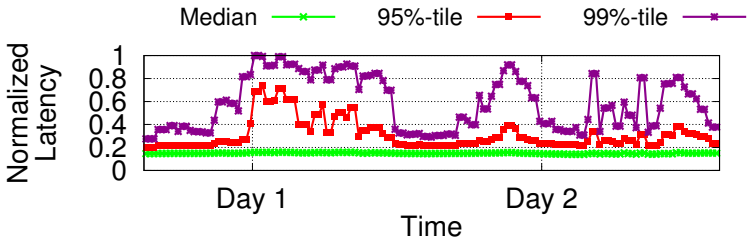
[2] We rely on a variant of DCTCP that still reacts in proportion to the fraction of ECN bytes seen within a round trip.



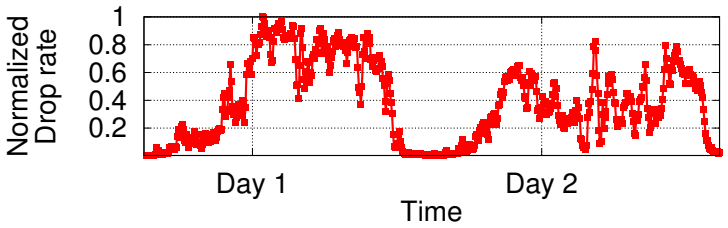
(a) Normalized aggregate throughput of all traffic initiated at studied cluster



(b) Normalized aggregate throughput of WAN traffic initiated from studied cluster

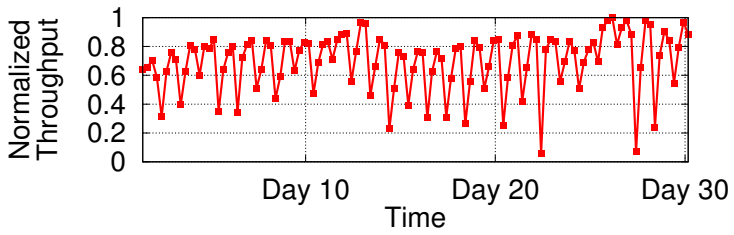


(c) Aggregate latency for intra-cluster traffic

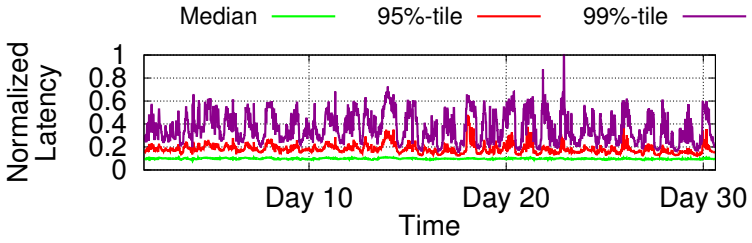


(d) Aggregate drop rate at ToR uplinks

Figure 5.3: Analysis of impact of WAN traffic on Local traffic from Cluster 1 over two days. All figures capture the same period.



(a) Normalized aggregate throughput of WAN traffic initiated from studied cluster



(b) Aggregate latency for intra-cluster traffic

Figure 5.4: Analysis of the impact of WAN traffic on Local traffic from cluster 1 over 30 days.

load. The change in WAN traffic impacts the 99th percentile of RPC latency of local traffic which has a correlation coefficient of 0.82 with WAN traffic volume. The relationship is clear in Figure 5.3c. During that period, we look at aggregate drops in stages of the topology of cluster. We find that drop rates are insignificant in all topology stages except at ToR uplinks (i.e., links connecting ToRs to superblocks) which is the first oversubscription point on the traffic path. Figure 5.3d shows the aggregate drop rate at all ToRs in the studied cluster which has a correlation coefficient of 0.79 with changes in WAN traffic. We also found similar trends on the second cluster where the correlation coefficients between WAN traffic and tail latency of LAN RPC and drops at the ToR are 0.8 and 0.79, respectively. These results validate our hypothesis that WAN traffic grabs more bandwidth and creates longer queues at near-source bottlenecks which negatively affecting LAN traffic. This behavior leads to our first two observations.

Observation 1: *Even at relatively stable aggregate traffic load, surges in WAN traffic affect local traffic negatively.*

Observation 2: *Interaction between the two types of traffic can be concentrated at the first*

point of oversubscription (i.e., very close to the source).

We study the behavior of the clusters over a period of a month. Figure 5.4a shows WAN traffic over the period of interest. Figure 5.4b shows surges of tail latency of local traffic tracking surges in WAN traffic. This behavior was also observed in the second cluster. This leads to our final observation.

Observation 3: *The identified issues are not transient.*

LAN and WAN congestion control protocols are developed with separate objectives in mind. LAN schemes are sensitive to queue build up with fast reaction and low loss as a core requirement. WAN schemes are aggressive and robust in the presence of loss to avoid the slow ramp up due to large RTT. The nature of both types of protocols is well motivated and suitable for their respective deployments. However, cases where both types of traffic are mixed are typically overlooked. In such cases, the presence of aggressive WAN traffic causes high latency, and even loss, at bottlenecks where both types of traffic compete. Furthermore, such bottlenecks are typically very close to the source.

5.2 Scheduling Near-Source Bottlenecks

As mentioned earlier, our objective is to develop a way that prevents WAN traffic from negatively affecting LAN traffic due to near-source contention, while maintaining their overall behavior for the rest of the path. In this section, we discuss the design space to reach this objective. Then, we move to discuss the design requirements of Annulus.

5.2.1 Design Space

Handling the identified issues that arise from WAN and LAN traffic mixing at near-source switches is challenging. Figure 5.5 shows the design space for a near-source bottleneck coordination between WAN and LAN traffic. The solution space varies in terms of how effectively it is isolating WAN and LAN traffic (vertical axis). On the other hand, they vary in terms of their effectiveness in sharing network resources and thus their cost effectiveness

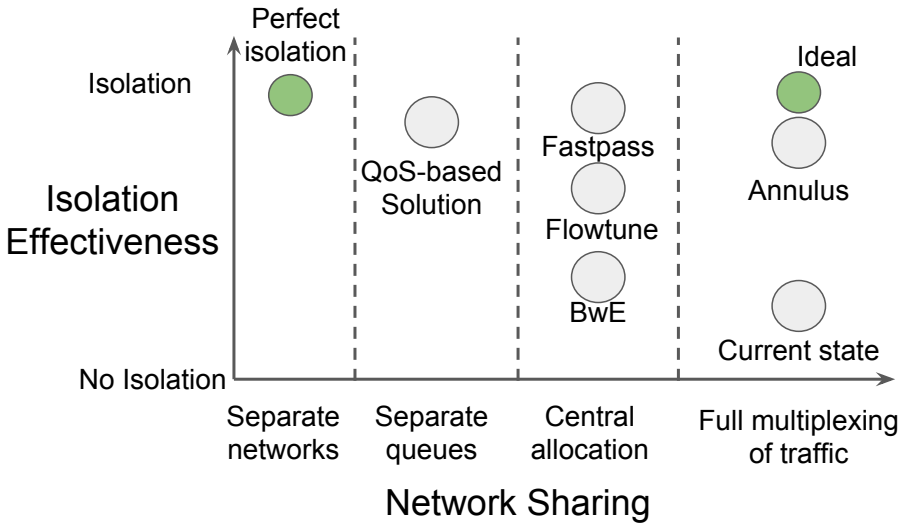


Figure 5.5: Design space for near-source bottleneck coordination.

(horizontal axis). An extreme approach to solve the problem is to completely separate WAN and LAN traffic into separate networks or separate parts of the cluster which leads to inefficient use of cluster resources. On the other hand, the ideal solution can be achieved by allowing switches to divide link capacity fairly, using Weighted Fair Queuing (WFQ), between competing flows. Hence, avoiding one type of traffic dominating the bandwidth. Then, each switch can report the sending rate of individual flows, using direct explicit messages, to the traffic sources. Thus, allowing traffic sources to adjust their sending rate for both WAN and LAN traffic at the same time scale and also allow sources to set their sending rate according to the capacity of the bottleneck link. However, this ideal solution suffers from several problems. First, the solution has a prohibitive message complexity as switches need to communicate with all sources sending traffic through them. Second, implementing WFQ per flow requires programmable switches [29] which are still under development with almost no deployments.

A natural approach is to rely on the eight QoS classes supported by IEEE 802.1Q [97]. This approach has several drawbacks. First, the number of available classes is small and is typically used to differentiate classes of applications. Leveraging QoS classes will further reduce the number of available classes for application-based classification. Furthermore,

QoS classes typically share memory which does not provide perfect isolation between the classes, which leaves room for WAN and LAN traffic to continue to impact each other. Statically allocating memory per QoS class can provide perfect isolation by wasting precious buffer space in shallow-buffered switches due to dedicating memory space for specific type of traffic. Another approach relies on centrally allocated bandwidth [12, 52, 115]. In such solutions, each flow reports its demand to a central entity which is also aware of network capacity. The central entity in return determines the rate at which each flow can operate, and in some proposals, the timing of sending individual packets. Centralized solutions suffer from several issues including control loop latency [12] and processing and messaging complexity [52, 115] which makes the overhead prohibitive for realistically large clusters. Furthermore, centralized solutions depend heavily on the accuracy of demand estimation, where inaccurate estimation can lead to inefficient network utilization. Note that solutions that require small queues at the switches (e.g., NUMFabric [116]) are not suitable in this case as they require specific congestion control algorithms that do not meet the requirements discussed earlier.

5.2.2 Design Requirements

1) Uniform reaction at the shared bottleneck: The main source of the identified problem is the difference in nature between LAN and WAN congestion control. A natural requirement is providing a homogeneous reaction for both types of traffic when they share a bottleneck. In particular, LAN and WAN backoff behavior should be similar in terms of its magnitude and timing to ensure that neither type of traffic is too cautious or too aggressive.
2) Reaction within LAN RTT: WAN RTT can be orders of magnitude larger than LAN RTT. This large difference can impact the interaction between the two types of traffic, even when their reaction algorithms are identical. In particular, WAN traffic becomes much slower to react which harms short-lived LAN flows. On the other hand, if both types of traffic are long lived, RTT unfairness can be an issue. This requires the reaction time of

both types of traffic to be close, if not the same.

3) Tunable reaction to near-source congestion: Each type of traffic has different priority classes within it (e.g., user facing WAN traffic and file copy WAN traffic). This requires that when sharing a bottleneck, WAN and LAN traffic competition should be based on policies defined by the operator that are only concerned with how these types of traffic compete and independent of how they compete with other flows of the same type at congestion points far from the source.

We find that these requirements can be satisfied by introducing a second control loop that works in tandem with the existing control loops for WAN and LAN traffic. The purpose of the second control loop is to determine reaction to congestion at near-source bottlenecks. Several potential solutions exist that can provide such control.

5.3 Annulus Congestion Control

We approximate the ideal solution discussed in the previous section by using feedback from near-source congestion points, while leaving deep path congestion to be handled by their typical WAN or LAN congestion control schemes. In particular, we propose Annulus which, as suggested by its name, maintains two control loops:

- End-to-end control loop: This control loop relies on existing congestion control algorithms to handle congestion that occur deep in the traffic path (i.e., not close to traffic sources where WAN and local traffic mix).

- Near-source control loop: This control loop relies on congestion notification from switches near the source to adjust its sending rate. This allows WAN and local traffic to adjust their rate at the same time granularity, using the same rate adjustment scheme which improves fairness and reaction time. *For this loop, we choose QCN [108] as it provides a practical approach to gain explicit feedback for near-source congestion points, as QCN is supported by the majority of datacenter switches.*

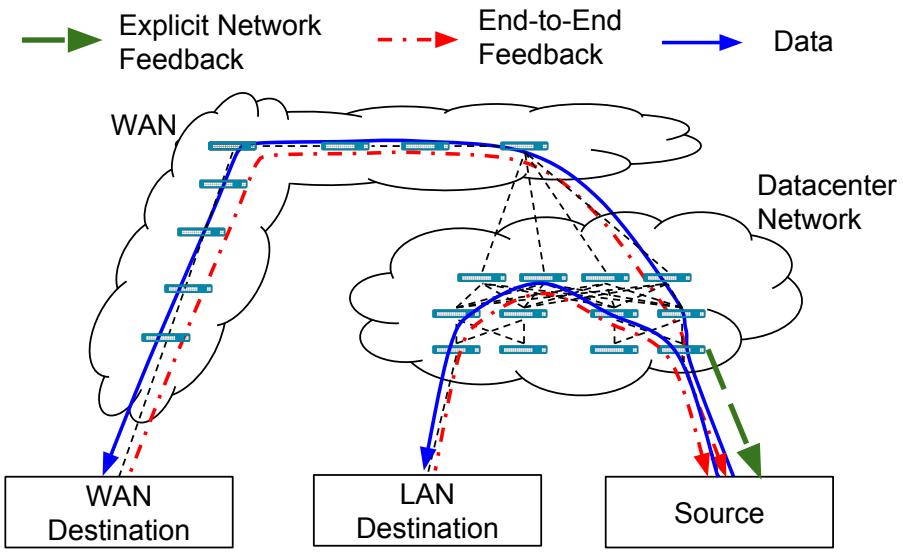
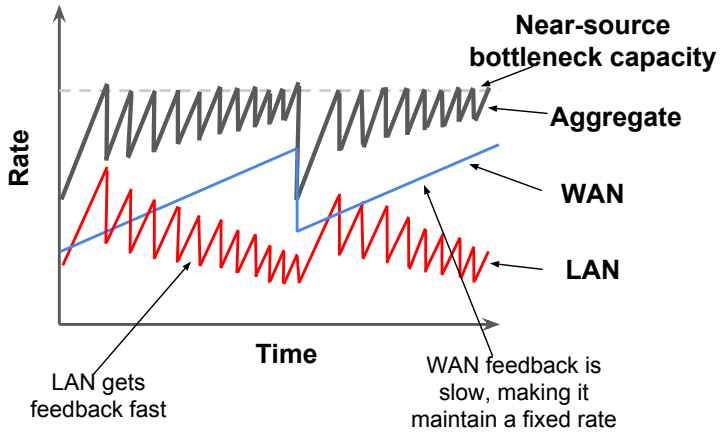
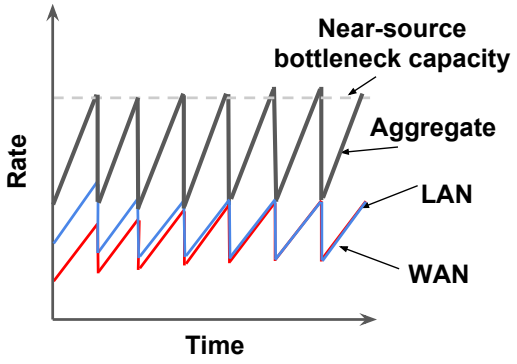


Figure 5.6: Network Overview.

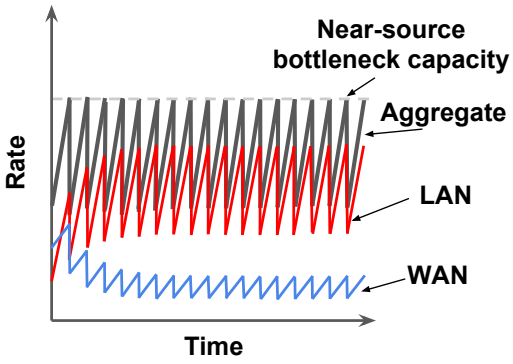
Why Annulus works Annulus is implemented at end-hosts in software by augmenting existing congestion control protocols with an additional control loop that adapts quickly to near-source bottlenecks. This is achieved by relying on fast explicit feedback from near-source bottlenecks as depicted in Figure 5.6. Figure 5.7 compares the behavior of competing WAN and LAN traffic using the current congestion control schemes and Annulus. Under current schemes, WAN traffic can react orders of magnitude slower than LAN traffic (e.g., 300x slow for 30ms RTT for WAN compared to 100us RTT for LAN). This is illustrated in the figure where the rate of WAN traffic does not change while LAN traffic keeps backing off. On the other hand, Annulus allows both types of traffic to react equally fast to near source congestion. This allows for better bandwidth division at near-source bottlenecks, including weighted prioritization of different types of traffic depending on operator set policy (e.g., figures 5.7b and 5.7c). Note that this is achieved by configuring the near-source control loop with a weight corresponding to the relative importance of the traffic. This design, by definition, essentially satisfies the requirements presented in the previous section. Our validation of Annulus is through experiments. Intuitively, if a flow is facing a single bottleneck, Annulus inherits the characteristics proven for the underlying



(a) Current state.



(b) Annulus with equal priority of WAN and LAN



(c) Annulus for high priority LAN

Figure 5.7: Comparison of traffic behavior under current schemes and using Annulus. Currently, LAN traffic reacts much faster than WAN which degrades the behavior of LAN traffic. Annulus allows both types of traffic to react equally fast, allowing for relative prioritization at near-source bottlenecks.

protocol [117, 111].

One of our main goals is to keep Annulus practical which largely depends on the practicality of QCN. The Achilles' heel of QCN's practicality is that it is an L2 congestion control protocol. In particular, QCN identifies flows based on their source and destination MAC address, and implements its logic in switches and NICs. This makes network-scale deployment (i.e., at every congestion point) of QCN near impossible in modern datacenter networks which are IP-routed networks with NICs that don't support QCN [32]. However, our goal is not to deploy QCN in the whole network. We are interested in QCN deployment at the congestion points close to traffic sources. This allows us to make use of the L2

Learning feature in current switches that allows a switch to learn the source MAC address of incoming packets in a dedicated memory, even in an IP-routed network.

Annulus for Edge Caching: We find that the dual control loop of Annulus is not only useful for when WAN and Local traffic are mixed, but it is useful when a cluster, or a part thereof, is used as a CDN. Annulus allows for the improvement of WAN traffic by handling congestion in the cluster orders of magnitude faster than end-to-end-only solutions. We show the value of Annulus in such scenarios in Section 5.4.

5.3.1 Leveraging QCN in Software

Background

IEEE 802.1Qau [118], based on the QCN algorithm, was designed to provide congestion management for data center bridging traffic. We start with a brief overview of the QCN algorithm, highlighting the more relevant aspects to our design. We refer to [118] and [108] for more details. The QCN algorithm has two components:

Congestion Point (CP): This is the component in the switch whose goal is to maintain the queue at the desired buffer occupancy Q_{eq}. The CP samples the incoming packets, measures the extent of the congestion by looking at the queue length, and conveys that information back to the source of the packet. The sampling rate is a configurable parameter that attempts to balance the number of feedback messages generated by the switch and the responsiveness of the algorithm to congestion. The feedback is a multi-bit congestion score feedback $F_b = (Q - Q_{eq}) + w \cdot (Q - Q_{old})$, where Q is the instantaneous queue-size, Q_{old} is the queue-size when the last packet was sampled, and w is a non-negative constant which is typically set to 2. If $F_b > 0$, CP sends a feedback message containing the quantized value of F_b to the source of the sampled packet. The intuition is that F_b captures the queue-size excess $(Q - Q_{eq})$ as well as the derivative of the queue size $(Q - Q_{old})$. Hence, positive F_b means that either the buffer or the link are oversubscribed. The CP generates feedback messages that contains the F_b value as well as 64 bytes of the sampled packet. This is

enough to identify the flow that sent the packet at the end host by looking up its source and destination IP and ports.

Reaction Point (RP): This is a component at the sender side with a Rate Limiter (RL), which decreases the rate based on the feedback it receives from CP and actively probes for available bandwidth in the absence of congestion. The RP algorithm maintains two rates: (a) *current rate (R_C)*, which is the sending rate at any point in time, and (b) *target rate (R_T)*, which is current rate just before receiving the last congestion feedback. RP goes through three main phases: (i) *Rate decrease:* When RP receives a congestion notification, it updates the target rate by setting $R_T = R_C$ and cuts R_C in proportion to the received F_b by setting $R_C = R_C(1 - G_d \cdot F_b)$, where G_d is set such that the sending rate can be decreased by at most 50%. (ii) *Fast Recovery (FR):* After cutting its rate, RP enters the FR phase to gradually recover the lost bandwidth and get back to the target rate. The recovery cycles are based on timer expirations or byte counter resets. At the end of each cycle, R_C is updated as follows: $R_C = \frac{1}{2}(R_C + R_T)$. (iii) *Active Increase (AI):* RP enters active increase phase after completing five cycles of FR to probe for extra bandwidth. During this phase, RP increases its sending rate by a configurable R_{AI} value in each stage (also based on byte-counting and a timer).

QCN as an L4 Protocol

Designing the end-host logic for QCN-based congestion control requires addressing several challenges as QCN messages are L2 messages. These challenges include: (i) The need for NIC support, which is currently not widely supported. (ii) Implementing QCN as a rate-based congestion control in hardware requires implementing a large number of queues to support a large range of rate limits. (iii) Lack of ACKs, due to the L2 implementation, required developing timers for recovering from sever rate drops which can be expensive to implement.

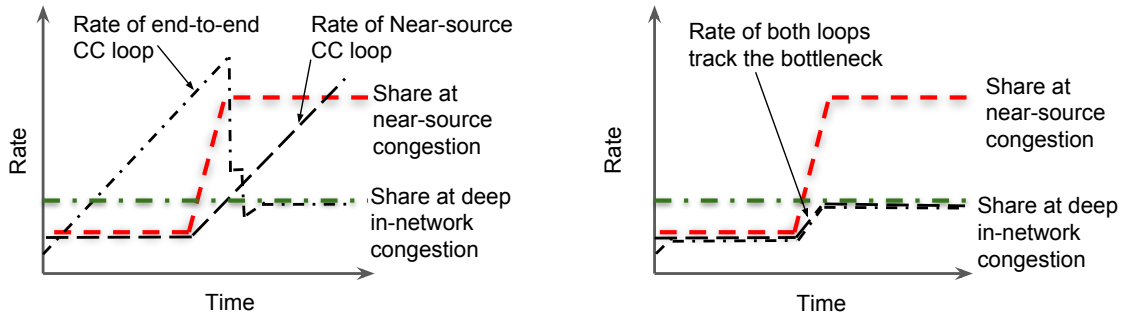
Annulus implements QCN as an L4 protocol, where QCN messages delivered to the

NIC are passed to a userspace network processing engine. The engine implements the congestion control logic of Annulus which includes reaction to the QCN messages. This implementation allows for overcoming the challenges faced when implementing an L2 algorithm. For instance, NIC support is no longer needed. Furthermore, using efficient rate limiting mechanisms (e.g., [24]) allows for efficient and accurate rate-based congestion control implementation. Finally, Annulus relies on WAN and LAN congestion control protocols which rely on ACKs for reliable transport. The availability of such ACKs is used to clock the recovery of for the near-source congestion control loop, eliminating the need for timers.

5.3.2 Integrating the Two Control Loops

For correct behavior, Annulus coordinates rate updates and window updates of both control loops. To understand the need for this coordination, consider the case where the two loops are left uncoordinated. Each control loop sets its transmission rate independently of one another. Annulus picks the smaller rate R_1 of control loop C_1 to accommodate the smaller botteleck. The other control loop C_2 sets a larger rate R_2. Because the two control loops are uncoordinated, C_2 does not get any feedback operating at R_2 and keeps increasing R_2. This increase has no effect on the actual transmission rate of the flow, hence, C_2's view of the network is distorted by the lack of feedback. Hence, as R_1 starts to increase, due to increased capacity for C_1 bottleneck, C_2 bursts at a very large R_2 which can take multiple RTTs to reduce to a reasonable level. Figure 5.8a shows the behavior of the rates produced by the two control loops when left uncoordinated.

We address this problem by allowing for signaling between the two control loops. The control loop setting the current smaller rate signals the other control loop to pause the growth of its window or rate. For example, in the previous scenario, C_1 signals C_2 to pause the growth of R_2. Furthermore, drops in the smaller rate are signalled to trigger proportional drops in the rate set by the other control loop. Figure 5.8b shows an illustration of the



(a) Behavior of the two loops if uncoordinated, which can result in loss.

(b) Behavior when both loops are coordinated tracks the minimum bandwidth on the path.

Figure 5.8: Illustration of the impact of control loop coordination on Annulus rate behavior.

approach. The rate always tracks the minimum available bandwidth, avoiding exceeding the rate at the bottleneck. This avoids creating bursts when bottlenecks shift between near-source and deep path. We realize that this approach can lead to lower utilization in case the congestion does not change in the larger control loop. However, our goal is to avoid burts or drops caused by the introduction of the additional control loop.

Another scenario is that two control loops can react to the same near-source bottleneck. However, we find this case to be very rare as QCN's multi-bit feedback outperforms single bit ECN-based algorithms in all our simulations and testbed evaluation. Hence, it reacts and stabilizes queue length at near-source bottlenecks before the end-to-end control loop detects queue build up. This can also be further guaranteed by setting the Q_{eq} of QCN to a smaller value than the ECN threshold.

5.3.3 Annulus Implementation

Figure 5.9 shows the architecture of Annulus, where feedback is obtained from both near-source switches and typical end-to-end signals. While the end-to-end control loop for WAN and LAN traffic can be different (e.g., DCTCP, BBR, CUBIC, or DCQCN), the near-source control loop is the same (described in §5.3.1). Each control loop uses the feedback it gets to configure a pacing rate for each flow. Pacing rate is translated into a timestamp allowing a timestamp-based rate limiter [24] to apply the smallest rate. The two control

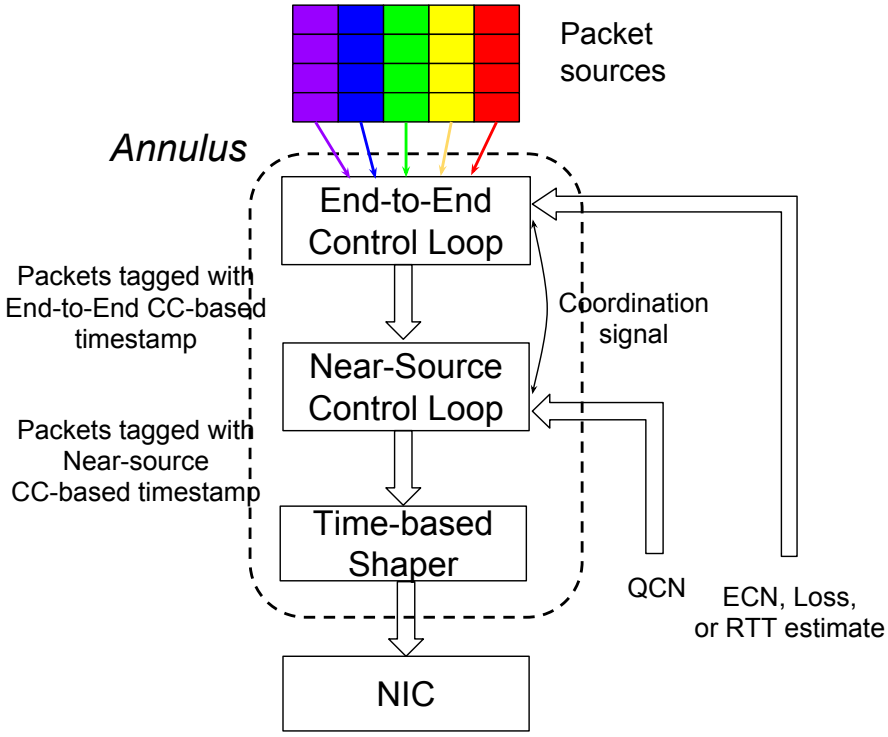


Figure 5.9: Architecture of Annulus end host.

loops coordinate their rate growth (described in §5.3.2).

The main challenge of deploying QCN is doing so at scale in an L3 routed network. To overcome this issue, we leverage some standard, but limited switch features, which makes deploying QCN feasible by focusing on congestion that happen only close to the source (i.e., ToRs and Superblocks). In particular, we leverage the standard L2 learning feature available on any L3 commodity switch today. L2 learning is a hardware feature in which a switch caches the source MAC address of a packet (data packet in QCN's case) along with the corresponding input switch port number. The switch can then properly forward an L2 packet (e.g. QCN notification frame) to a MAC address that has been already cached through the corresponding cached port number. Today's data center switches typically have dedicated L2 tables, that cannot be used for L3 entries. These tables can accommodate at the order of 100,000 MAC entries. This is more than sufficient for maintaining the cached MAC address of the sampled packet long enough before its corresponding congestion con-

trol frame traverses back in the reverse direction using the cached information. [3]

Hence, all that is needed for QCN's notification packets to get routed back to their L2 sources is preserving their L2 source MAC address throughout the fabric (i.e. don't overwrite that value end-to-end) and turning on L2 learning. Furthermore, we focus on deploying QCN close to traffic sources (i.e., ToR and Superblocks). This reduces the pressure on the memory. Thus, switches can continue to simultaneously routed IP packets based on the IP table information while forwarding non-IP L2 frames based on the L2 table information that gets populated based on the L2 header of IP packets.

5.4 Evaluation

We evaluate the performance of Annulus on a testbed as well as using packet-level simulations in ns2 [110]. Annulus implements the near-source control loop as discussed earlier while relying on DCTCP and BBR for LAN and WAN end-to-end control loop, respectively. We compare the performance of Annulus to DCTCP and BBR in the testbed and to TCP CUBIC, DCTCP, and DCQCN in simulations. We evaluate Annulus in scenarios of traffic mixes and pure WAN and LAN traffic. The testbed evaluation demonstrates the feasibility and value of Annulus, while the simulations allow us to evaluate Annulus under conditions that are hard to replicate in the testbed. Our main evaluation metric is RPC latency. We focus on RPC tail latency for local traffic and transfer latency and throughput for WAN traffic. This focus is because of the typical drive to operate WAN traffic at near capacity [119, 120] while local traffic typically is more concerned with latency [30, 31].

[3] The worst case here is when as many packets as the capacity of the L2 table arrive with a different MAC address each at the highest speed (i.e. from all ports). Conservatively assuming 100,000 L2 entries and a 16-port 40Gbps switch, with a 0.5KB average packet size, it is impossible that an L2 cached entry could be evicted in less than $600\mu s$ from the time it was added, which is much more than the round-trip time in any reasonably designed data center today.

5.4.1 Testbed Evaluation

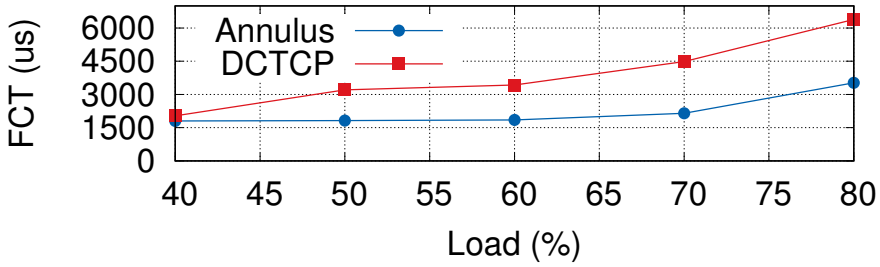
Setup: We conduct experiments on three racks, two co-located in the same cluster with 20 machines each while the third is in a different cluster with 10 machines, where RTT between the two clusters is 8ms. All machines equipped with dual port 20 Gbps NICs. We maintain control over path congestion by forcing congestion to be at the ToR by setting a high oversubcription ratio of 5:1 from ToR to middleblock, and operating in non-peak hours. We configured QCN only at ToR switches. We set Q_{eq} to 100KB per port and a W of 8, where packets are samples every 15KB. Note that while some figures label baselines as only DCTCP or BBR, all testbed evaluation scenarios use DCTCP exclusively for LAN and TCP BBR exclusively for WAN.

Workload: We generate synthetic RPC loads. RPC sizes are set to 127KB unless otherwise stated. We use a cross-rack all to all communication pattern, where all machines in one rack send traffic to all machines in another rack. This communication pattern generates a large number of flows and allows for stress testing the behavior of the congestion control algorithms. We generate three types of workloads: 1) LAN/WAN mixture with various mixture ratios, 2) edge caching traffic where the traffic is only WAN traffic, and 3) purely LAN traffic that reflects the more common case in cluster traffic. WAN traffic is generated by sending traffic from 10 machines in one rack in one cluster to 10 machines in the rack in the other cluster, while LAN traffic is generated by sending traffic from 20 machines in one rack to 20 machines in another rack.

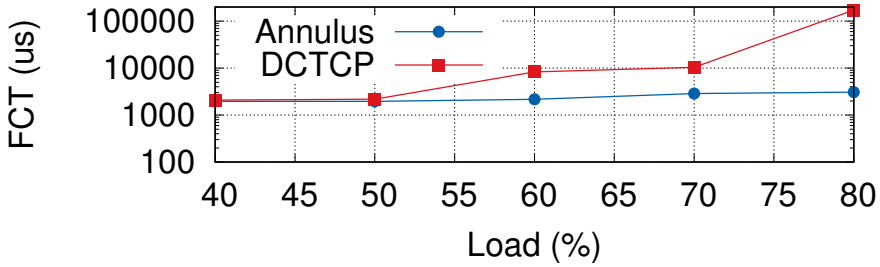
WAN/LAN Mix

We start by looking at overall performance when WAN and LAN traffic are mixed. In this scenario, we care about improving LAN traffic isolation from WAN traffic, while retaining WAN traffic performance.

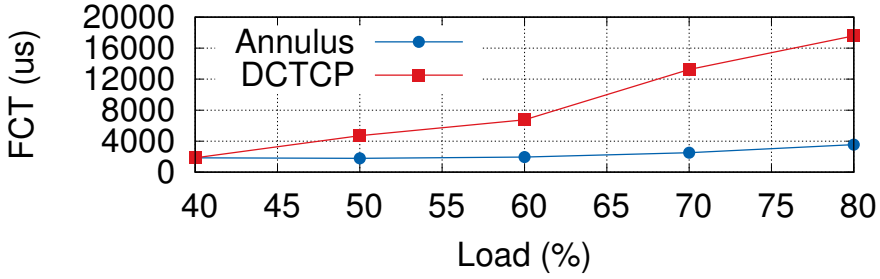
LAN Latency: Figure 5.10 shows the measured performance of 99th percentile RPC latency over LAN. We find that at common mixes of 5:1 LAN to WAN, even at small



(a) 1:5.



(b) 5:1 (logarithmic y-axis).



(c) 1:1.

Figure 5.10: RPC latency of local traffic for WAN/LAN traffic mix for different WAN:LAN ratios.

loads (i.e., 50%), latency is reduced by 43.2%. This is achieved without observing any changes in WAN traffic performance. On the other hand, in scenarios where traffic is more dominated by WAN traffic, where the ratio is reversed (i.e., 1:5 in Figure 5.10b), Annulus produces LAN RPC latency that is 56x better at high loads. This is also combined by 40% reduction in average WAN traffic latency. Even in cases where traffic mix is even, we find that Annulus produces 2x improvements at 60% of bottleneck capacity. This shows that, as intended, Annulus reduces the impact of WAN traffic on LAN traffic even at large shares of WAN traffic.

Table 5.1: Ratio of LAN to WAN traffic for different schemes.

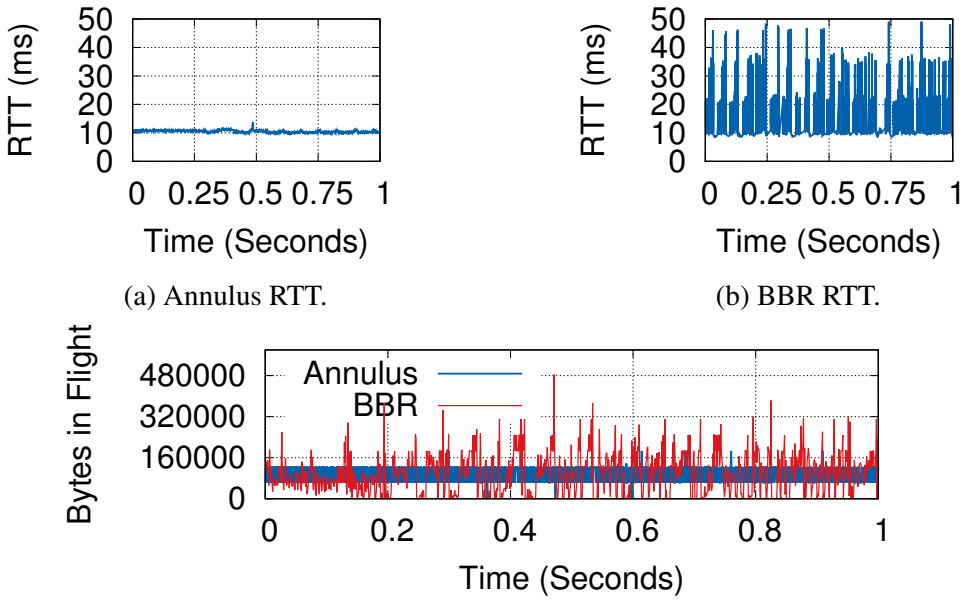
	Annulus	DCTCP/BBR	Max/Min Fair
LAN:WAN Ratio	5.8:1	1:7.6	4:1

Table 5.2: Bottleneck utilization under different traffic mixes.

Setting	Annulus	DCTCP/BBR
LAN/WAN mix	91.51%	86.63%
LAN	90.42%	82.13%
WAN	92.1%	83.28%

Fairness: We also perform experiments without forcing the traffic ratio between LAN and WAN to observe the result sharing scheme when aggressive WAN congestion control interacts with LAN traffic. Table 5.1 shows the results of such experiments. Annulus allows for a sharing policy that is proportional to the number of flows (400 flows for LAN and 100 flows for WAN). This can be further tuned by changing the G_d value in the QCN algorithm, reducing or increasing the aggressiveness of the QCN control loop. Such tuning is much harder using a single control loop as it will affect the behavior of the congestion control protocol at other bottlenecks, while Annulus allows for tuning reaction at only near-source congestion.

Bottleneck utilization: Experiments without forcing the traffic ratio allows us to also observe the maximum achievable goodput, summarized in Table 5.2. Annulus improves maximum achievable goodput by 5-10%. To better understand this impact, we look at the tcpdump for a single WAN flow over the period of a second. Figure 5.11 shows RTT and bytes in flights produced by both protocols. We find that Annulus produces more stable behavior, maintaining two segments in flight over the whole period while sustaining stable RTT between 10-13 ms by keeping the queues stable. On the other hand, BBR produces a fluctuating RTT ranging from 9-48ms which is caused by fluctuations in queue length. These fluctuations cause drops which leads BBR to frequently backoff, reducing its goodput.



(a) Annulus RTT.

(b) BBR RTT.

(c) Bytes in flight.

Figure 5.11: Behavior of a single flow over the period of one second showing bytes in flight and RTT for Annulus and BBR.

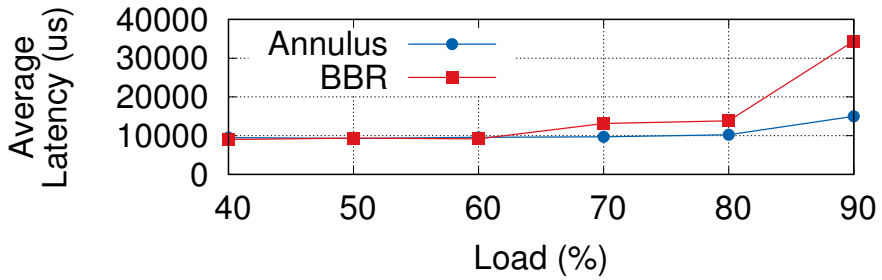
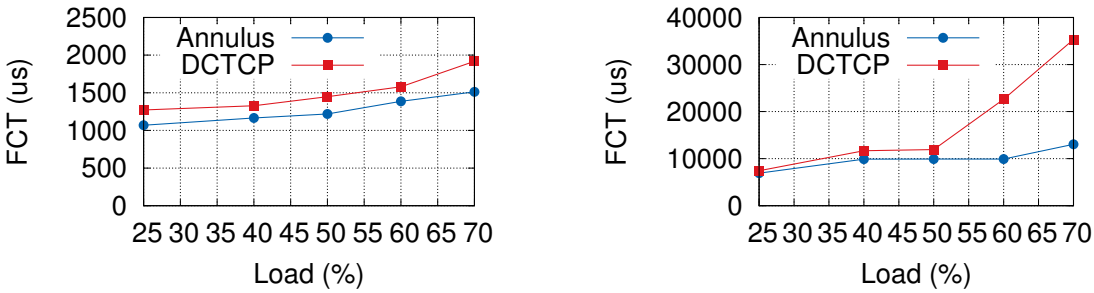


Figure 5.12: Transfer latency for pure WAN traffic.

Edge Caching (purely WAN traffic)

In this case, we only look at the average latency of message transfer between ends as it better reflects the requirements of edge caching applications. Figure 5.12 shows the results. Annulus reduces average latency by 26% at 70% load and 80% at 90% load. It also improves latency by 2.2x at near bottleneck capacity while improving goodput by 10% (Table 5.2).



(a) Small flows FCT.

(b) Large flows FCT.

Figure 5.13: 99th percentile FCT for pure LAN traffic.

Purely LAN traffic

We validate that Annulus does not cause regression in performance of LAN traffic by comparing it to DCTCP under the scenario where traffic is 100% LAN. In this experiment, we mix RPC load of small sized messages of 10KB and large sized messages of 1MB. The goal of this experiment is to show that Annulus does not introduce regression in the more common case of purely LAN traffic. Figure 5.13 shows the tail FCT for both jobs where Annulus improves performance of both jobs by up to 2.7x for the large RPC job at 70% utilization.

5.5 Discussion

Further improving Edge Caching: We focus on developing the near-source control loop to handle congestion at the first two hops. However, another major point of congestion for edge caching applications is at the border switches connecting a cluster to the WAN. Deploying QCN at such points of congestion can be beneficial as latency between traffic source and the border switch is still at the microseconds scale, which will allow the benefits of Annulus to remain in effect. However, such deployment carries the challenges of fully deploying L2 and L3 networks. We hope that the results we present here rekindle this discussion and encourage innovation that will allow for fast reaction to near-source congestion.

Impact on Applications: Although our experiments focus on synthetic workloads, we envision that Annulus will have significant impact on applications. In particular, Annulus provides a unique improvement of WAN traffic latency which can help with modern gaming and livestreaming web applications. These improvements are in addition to more typical improvements in tail latency of LAN traffic which is a known requirements for distributed applications operating in datacenters and average goodput which is typically improved by WAN congestion control protocols.

Limitations We view our work as an initial step in the direction of multi-loop congestion control. Hence, it is natural for our work on Annulus to suffer from several limitations leaving a lot of room for future innovation. In particular, our validation of Annulus is based on experiments. Constructing models for dual loop congestion control protocols remains an open and challenging problem. Furthermore, our implementation focuses only on ToR and middleblock congestion. Extending the reach of the near-source congestion control loop remains a challenge. Finally, exploring the impact of different congestion signals for the end-to-end control loop for LAN traffic remains unclear.

5.6 Summary

WAN and LAN traffic have significantly different requirements and path characteristics. These differences have caused independent development of congestion control protocols for the different types of traffic. We showed in this chapter that this approach of independent design and deployment causes significant problems, as the intentionally-aggressive nature of WAN traffic causes deterioration in the performance of the cautious LAN traffic. To solve this problem, we introduce Annulus, a dual congestion control loop that relies on QCN to implement the near-source congestion control loop. This approach significantly improves LAN latency, utilization of bottlenecks, and fairness between WAN and LAN traffic. We envision that this work will motivate further investigation of multi-control loop congestion control where different types of bottlenecks are handled differently to provide

accurate estimation of bandwidth at the bottleneck as well as isolation between different types of traffic.

CHAPTER 6
CONCLUSION

We consider the work presented in this dissertation as building blocks towards a network that can intelligently schedule and prioritize traffic based on higher-level application semantics, rather than drop or delay packets when it is congested. Such networks will require efficient and flexible queue-scheduling building blocks as well as congestion control schemes that keep such queues short.

Through the work presented in this dissertation, we have shown that using specialized data structures like the Timing Wheel, Integer Priority Queues, and the Gradient Queue leads to significant improvement in network performance. Furthermore, we find that employing such efficient data structures allows for the practical implementation of complex scheduling algorithms at end hosts by balancing CPU efficiency and full utilization of NIC capacity. We also maintaining efficient utilization of the network requires avoid congestion in the network as well as at the end host. Thus, we show the value of using the completion signal to control congestion at the end host. The scheduler can delay delivering the completion signal, and thus prevent traffic source from enqueuing more packets. We also show that when LAN and WAN traffic compete a single signal helps better control their interaction while not changing the behavior of the original congestion control algorithms developed for their respective cases. This enforces the thesis behind this work, which is that it is possible to implement such networks, if the underlying data structures are optimized and the proper network signals are used.

6.1 Future Work

The work done in this thesis can be extended in several directions. We describe some potential future work below.

Scalable coordinated scheduling across end hosts The work in this dissertation explores efficient implementation of scheduling policies where the decision is made based on locally available information at the end host. Such policies are useful in prioritizing between traffic within the same machine. However, practical scheduling policies can go beyond the boundary of a single end host. Access to the network can be coordinated and prioritized between flows originating in different end hosts. Such distributed access to the network is currently achieved through congestion control and centralized network schedulers. These approaches operate at fixed granularities in terms of scheduling unit or time scale (i.e., per flow coordination at microsecond scale in case of congestion control and miliseconds to seconds time scale in case of centralized schemes). Efficient and flexible scheduling schemes are needed for distributed schedulers where the scheduler can operate at the RTT time scale coordinating flows according to a programmable scheduling policy.

Application-Network Interaction Interaction between applications and the network currently happen through the standard Berkeley Socket API, now 36 years old. Under such interaction, the application is oblivious to the scheduling decisions made by the network. This typically leads application developers to optimize their code for the network behavior they observe in practice, leading to applications encoding certain network logic (e.g., handling failure and pacing). A better interface between applications and the network can simplify application code by exposing network state as well as network function to the applications, allowing application to simply react and utilize such information rather than reimplementing network functions inside the application.

Scalable backpressure at end hosts The backpressure mechanism we develop for Carousel has two main components: TCP Small Queue (TSQ) that limits number of packets in the stack and the delayed completion signal which prevents TSQ from enqueuing more packets until it receives the signal. This approach, while practical for current settings, suffers from one major pitfall. In particular, the number of packets handled by the stack depends on the number of flows handled by the stack. This characteristic is inher-

ited from TSQ which limits the number of packets in the stack to two packets per flow. This limit becomes impractical as the number of flows keeps growing, where long queues can lead to significant problems including bufferbloat. Future backpressure mechanisms should the number of packets handled by the networking stack regardless of the number of flows it is serving. Such approaches will keep complex scheduling policies feasible (i.e., by limiting the number of packets to be scheduled) as well as improve network performance in general (e.g., reducing network latency by reducing queuing delay).

Modeling dual control loop congestion control algorithms Our work on Annulus demonstrates the practical value of a dual congestion control algorithm. However, there are no models yet for such congestion control algorithms. Modeling such algorithms requires improved network models to capture multiple types of bottlenecks. Furthermore, it requires more complex models to capture the multi-delay nature of dual congestion control loops. Such modeling can lead to better algorithm design by taking into account theoretical limitations on performance as well as interaction requirements between the two control loops.

Appendices

APPENDIX A

GRADIENT QUEUE CORRECTNESS

Theorem 1. *The index of the maximum non-empty bucket,* N*, is* $ceil(b/a)$.

Proof. We encode the occupancy of buckets by a bit string of length N where zeros represent empty buckets and ones represent nonempty buckets. The value of the bit string is the value of the critical point $x = \frac{b}{a}$ for queue represented by the bit of strings. We prove the theorem by showing an ordering between all bit strings, where the maximum value is N and the minimum value is larger than $N - 1$. The minimum value is when all buckets are nonempty (i.e., all ones). In that case, $a = \sum_{i=1}^{N} 2^i$ and $b = \sum_{i=1}^{N} i2^i$. Note that b is an Arithmetic-Geometric Progression that can be simplified to $N2^{N+1} - (2^{N+1} - 2)$ and a is a Geometric Progression that can be simplified to $2^{N+1} - 2$. Hence, the critical point $x = \frac{N2^{N+1}}{2^{N+1}-2} - 1 = \frac{N}{1-2^{-N}} - 1$ where $\frac{N}{1-2^{-N}} < N + 1$ and $ceil(x) = N$. The maximum value occurs when only bucket N is nonempty (i.e., all zeros). It is straightforward to show that the critical point is exactly $x = N$. Now, consider any N-bit string, where the Nth bit is 1, if we flip one bit from 1 to zero, the value of the critical point increases. It is straight forward to show that $\frac{b-j2^j}{a-2^j} - \frac{b}{a} > 0$, where j is the index of the flipped bit. □

APPENDIX B

EXAMPLES OF ERRORS IN APPROXIMATE GRADIENT QUEUE

To better understand the effect of missing elements on the accuracy of the approximate queue, consider the following cases of elements distribution for a maximum priority queue with N buckets:

- Elements are evenly distributed over the queue with frequency $1/\alpha$, which is equivalent to an Exact Gradient Queue with N/α elements,
- $N/2$ elements are present in buckets from 0 to $N/2$ and then a single element is present in bucket indexed $3N/4$, where the concentration of the elements at the beginning of the queue will create an error on the estimation of the index of the maximum element $\epsilon = ceil(b/a) + u(\alpha) - 3N/4$. We note that in this case $\epsilon < 0$ because the estimation of $ceil(b/a)$ will be closer to the concentration of elements that is pulling the curvature away from $3N/4$. The error in such cases grows proportional to size of the concentration and inversely proportional to the distance between the low concentration and the high concentration.
- All elements are present, which allows the value $\epsilon = ceil(b/a) + u(\alpha)$ to be exactly where the maximum element is.

REFERENCES

[1] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 28, 2000, pp. 248–259.

[2] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] M. D. McCool, "Scalable programming models for massively multicore processors," *Proceedings of the IEEE*, vol. 96, no. 5, 2008.

[4] R. S. Williams, "What's next?[the end of moore's law]," *Computing in Science & Engineering*, vol. 19, no. 2, 2017.

[5] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, "Performance issues in parallelized network protocols," in *Presented as part of the USENIX conference on Operating Systems Design and Implementation*, ser. OSDI '94, USENIX, 1994.

[6] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving network connection locality on multicore systems," in *Proceedings of the ACM European Conference on Computer Systems*, ser. EuroSys '12, ACM, 2012, pp. 337–350.

[7] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer," in *Presented as part of the ACM Workshop on Hot Topics in Networks*, ser. HotNets-VIII, 2009.

[8] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Ozcan, "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics," *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB '15)*, vol. 8, no. 13, 2015.

[9] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proceedings of the ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII, ACM, 2013, 10:1–10:7.

[10] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, 2012.

[11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-optimal Datacenter Transport," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '13, ACM, 2013, pp. 435–446.

[12] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat, "BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing," in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM '15)*, 2015.

[13] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-driven WAN," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '13, ACM, 2013, pp. 15–26.

[14] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, Not Foes: Synthesizing Existing Transport Strategies for Data Center Networks," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '14, ACM, 2014, pp. 491–502.

[15] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic Flow Scheduling for Commodity Data Centers," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '15, USENIX Association, 2015, pp. 455–468.

[16] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can JUMP them!" In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '15, USENIX Association, 2015, pp. 1–14.

[17] W. Wang, C. Feng, B. Li, and B. Liang, "On the Fairness-Efficiency Tradeoff for Packet Processing with Multiple Resources," in *Proceedings of the ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, ACM, 2014, pp. 235–248.

[18] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource Fair Queueing for Packet Processing," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12, ACM, 2012, pp. 1–12.

[19] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, 2015.

[20] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '18, USENIX Association, 2018, pp. 373–387.

[21] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: scalable NIC for end-host rate limiting," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '14, USENIX Association, 2014, pp. 475–488.

[22] *Intel 82599 10gbe controller*, `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`, 2016.

[23] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '15, USENIX Association, 2015, pp. 117–130.

[24] A. Saeed, N. Dukkipati, V. Valancius, T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End-Hosts," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '17, ACM, 2017, pp. 404–417.

[25] V. Inc., *Performance Evaluation of Network I/O Control in VMware vSphere 6*, `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/network-ioc-vsphere6-performance-evaluation-white-paper.pdf`, 2015.

[26] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. Nikolenko, A. V. Sirotkin, and P. Eugster, "A Programmable Buffer Management Platform," in *Proceedings of the IEEE International Conference on Network Protocols*, ser. ICNP '17, IEEE, 2017, pp. 1–10.

[27] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan, "Towards Programmable Packet Scheduling," in *Proceedings of the ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV, ACM, 2015, 23:1–23:7.

[28] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," *arXiv preprint arXiv:1710.11583*, 2017.

[29] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable Packet Scheduling at Line Rate," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '16, ACM, 2016, pp. 44–57.

[30] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '10, ACM, 2010, pp. 63–74.

[31] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "TIMELY: RTT-based Congestion Control for the Datacenter," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '15, ACM, 2015, pp. 537–550.

[32] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale RDMA deployments," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '15, ACM, 2015, pp. 523–536.

[33] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, 2016.

[34] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'18, USENIX, 2018, pp. 329–342.

[35] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-friendly High-speed TCP Variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, 2008.

[36] T. Henderson, S. Floyd, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 6582, 2012, `https://tools.ietf.org/html/rfc6582`.

[37] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '19, USENIX Association, 2019, pp. 17–32.

[38] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, no. 10, 1988.

[39] G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP '87, ACM, 1987, pp. 25–38.

[40] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switching networks," in *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, ser. SIGCOMM '90, ACM, 1990, pp. 19–29.

[41] G. Kumar, S. Kandula, P. Bodik, and I. Menache, "Virtualizing traffic shapers for practical resource allocation," in *Presented as part of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*, 2013.

[42] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '08, ACM, 2008, pp. 197–210.

[43] J.-P. Billaud and A. Gulati, "hClock: Hierarchical QoS for packet scheduling in a hypervisor," in *Proceedings of the ACM European Conference on Computer Systems*, ser. EuroSys '13, ACM, 2013, pp. 309–322.

[44] M. Devera, *Linux Hierarchical Token Bucket*, `http://luxik.cdi.cz/~devik/qos/htb/`, 2003.

[45] E. Dumazet and J. Corbet, *Tso sizing and the fq scheduler*, `https://lwn.net/Articles/564978/`, 2013.

[46] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[47] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware Datacenter TCP (D2TCP)," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12, ACM, 2012, pp. 115–126.

[48] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter, "Practical DCB for improved data center networks," in *Proceedings of the IEEE Conference on Computer Communications*, ser. INFOCOM '14, IEEE, 2014, pp. 1824–1832.

[49] J. Postel, "Internet Control Message Protocol," STD 5, 1981, `https://tools.ietf.org/html/rfc792`.

[50] D. Zats, A. P. Iyer, G. Ananthanarayanan, R. Agarwal, R. Katz, I. Stoica, and A. Vahdat, "FastLane: Making short flows shorter with agile drop notification," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15, ACM, 2015, pp. 84–96.

[51] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12, ACM, 2012, pp. 139–150.

[52] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '14, ACM, 2014, pp. 307–318.

[53] N. Dukkipati, "Rate control protocol (RCP): Congestion control to make flows complete quickly," PhD thesis, Stanford University, 2007.

[54] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '13, ACM, 2013, pp. 3–14.

[55] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP fast open," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '11, ACM, 2011, 21:1–21:12.

[56] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time," in *Proceedings of the Annual Symposium on Foundations of Computer Science*, ser. FOCS' 75, IEEE, 1975, pp. 75–84.

[57] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII, ACM, 1998, pp. 115–126.

[58] M. L. Fredman and D. E. Willard, "Blasting through the information theoretic barrier with fusion trees," in *Proceedings of the Annual ACM Symposium on Theory of Computing*, ser. STOC '90, ACM, 1990, pp. 1–7.

[59] M. Thorup, "Equivalence between priority queues and sorting," *Journal of the ACM (JACM)*, vol. 54, no. 6, 2007.

[60] B. Chazelle, "The soft heap: An approximate priority queue with optimal error rate," *Journal of the ACM (JACM)*, vol. 47, no. 6, 2000.

[61] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, "RIPQ: Advanced Photo Caching on Flash for Facebook," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '15, USENIX Association, 2015, pp. 373–386.

[62] P. Goyal, H. M. Vin, and H. Chen, "Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '96, ACM, 1996, pp. 157–168.

[63] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, ser. SIGCOMM '89, ACM, 1989, pp. 1–12.

[64] F. Checconi, L. Rizzo, and P. Valente, "QFQ: Efficient packet scheduling with tight guarantees," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 3, 2013.

[65] *Arista 7500 series data center switch*, `https://www.arista.com/assets/data/pdf/Datasheets/7500_Datasheet.pdf`, 2017.

[66] *Arista 7010T Gigabit Ethernet Data Center Switches*, `https://www.arista.com/assets/data/pdf/Datasheets/7010T-48_Datasheet.pdf`, 2017.

[67] *Cisco: Understanding Quality of Service on the Catalyst 6500 Switch*, `https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11_538840.html`, 2017.

[68] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings of the IEEE Conference on Computer Communications*, ser. INFOCOM '00, IEEE, 2000, pp. 538–547.

[69] A. Ioannou and M. G. H. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 15, no. 2, 2007.

[70] H. Wang and B. Lin, "Per-flow queue management with succinct priority indexing structures for high speed packet scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, 2013.

[71] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, 1973.

[72] M. Ghobadi, Y. Cheng, A. Jain, and M. Mathis, "Trickle: Rate Limiting YouTube Video Streaming," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC '12, USENIX, 2012, pp. 191–196.

[73] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder, *Offload of tcp segmentation to a smart adapter*, US Patent 5,937,169, 1999.

[74] *net: Generic receive offload*, https://lwn.net/Articles/358910/, 2008.

[75] Y. Ganjali and N. McKeown, "Update on buffer sizing in internet routers," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 5, 2006.

[76] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "EyeQ: Practical Network Performance Isolation at the Edge," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '13, USENIX, 2013, pp. 297–311.

[77] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, 2010.

[78] E. Dumazet and J. Corbet, *TCP small queues*, `https://lwn.net/Articles/507065/`, 2012.

[79] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," in *SIGCOMM 95*.

[80] *tcp: TSO packets automatic sizing*, https://lwn.net/Articles/564979/.

[81] *Leaky bucket as a queue*, https://en.wikipedia.org/wiki/Leaky_bucket.

[82] Y. Cheng and N. Cardwell, "Making Linux TCP Fast," in *Netdev Conference, 2016*.

[83] T. Flach, P. Papageorge, A. Terzis, L. Pedrosa, Y. Cheng, T. Karim, E. Katz-Bassett, and R. Govindan, "An internet-wide analysis of traffic policing," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '16, ACM, 2016, pp. 468–482.

[84] *pfifo-tc: PFIFO Qdisc*, https://linux.die.net/man/8/tc-pfifo/.

[85] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, ACM, 2017, pp. 183–196.

[86] J. D. Brouer, *Network stack challenges at increasing speeds: The 100gbit/s challenge*, LinuxCon North America, 2015.

[87] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Proceedings of the International*

Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '16, ACM, 2016, pp. 67–81.

[88] W. Almesberger, J. H. Salim, and A. Kuznetsov, "Differentiated services on linux," in *Proceedings of the IEEE Global Telecommunications Conference*, ser. GLOBECOM '99, IEEE, 1999, 831–836 vol. 1b.

[89] D. D. Clark, "The structuring of systems using upcalls," in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP '85, ACM, 1985, pp. 171–180.

[90] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, 2008.

[91] W. Bai, K. Chen, H. Wu, W. Lan, and Y. Zhao, "PAC: taming TCP Incast congestion using proactive ACK control," in *Proceedings of the IEEE International Conference on Network Protocols*, ser. ICNP '14, IEEE, 2014, pp. 385–396.

[92] *neper: a Linux networking performance tool*, `https://github.com/google/neper`, 2016.

[93] Intel Networking Division, *Ethernet Switch FM10000*, 2016.

[94] ——, *82599 10 GbE Controller Datasheet*, 2016.

[95] *Bess: Berkeley extensible software switch*, `https://github.com/NetSys/bess/wiki`, 2017.

[96] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '18, USENIX, 2018, pp. 51–66.

[97] "IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks," *IEEE Std 802.1Q-2005 (Incorporates IEEE Std 802.1Q1998, IEEE Std 802.1u-2001, IEEE Std 802.1v-2001, and IEEE Std 802.1s-2002)*, pp. 1–300, 2006.

[98] *C++ reference implementation for Push-In First-Out Queue*, `https://github.com/programmable-scheduling/pifo-machine`, 2016.

[99] *Intel 64 and ia-32 architectures optimization reference manual*, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.

[100] *MD64 Architecture Programmers Manual Volume 3: General-Purpose and System Instructions*, https://support.amd.com/TechDocs/24594.pdf, 2017.

[101] *Real-Time Scheduling Class (mapped to the SCHED_FIFO and SCHED_RR policies)*, https://elixir.bootlin.com/linux/latest/source/kernel/sched/rt.c#L1494, 2017.

[102] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '96, ACM, 1996, pp. 143–156.

[103] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, P Schroeder, J. Spaans, and P. Larroy, "Linux advanced routing & traffic control," in *Ottawa Linux Symposium*, 2002, p. 213.

[104] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal Packet Scheduling," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '16, USENIX, 2016, pp. 501–521.

[105] *Benchmark Tools*, https://github.com/google/benchmark, 2018.

[106] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[107] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.P4: Programming the Forwarding Plane of a Data-center Switch," in *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, ACM, 2015, 2:1–2:8.

[108] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms: Congestion control theory and IEEE standardization," in *Proceedings of the IEEE Annual Allerton Conference on Communication, Control, and Computing*, ser. Allerton'08, IEEE, 2008, pp. 1270–1277.

[109] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, "Increasing TCP's Initial Window," RFC 6928, 2013, https://tools.ietf.org/html/rfc6928.

[110] *NS-2 network simulator*, http://nsnam.sourceforge.net/wiki/index.php/Main_Page, 2011.

[111] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '11, ACM, 2011, pp. 73–84.

[112] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10, ACM, 2010, pp. 267–280.

[113] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the ACM Special Interest Group on Data Communication Conference*, ser. SIGCOMM '15, ACM, 2015, pp. 123–137.

[114] T. V. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 3, 1997.

[115] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet Control for Datacenter Networks," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'17, USENIX Association, 2017, pp. 421–435.

[116] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '16, ACM, 2016, pp. 188–201.

[117] M. Alizadeh, A. Kabbani, B. Atikoglu, and B. Prabhakar, "Stability analysis of QCN: The averaging principle," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '11, ACM, 2011, pp. 49–60.

[118] "Ieee standard for local and metropolitan area networks– virtual bridged local area networks amendment 13: Congestion notification," *IEEE Std 802.1Qau-2010 (Amendment to IEEE Std 802.1Q-2005)*, pp. c1–119, 2010.

[119] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering Egress with Edge Fabric: Steering Oceans of Content to the World," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, ACM, 2017, pp. 418–431.

[120] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, ACM, 2017, pp. 432–445.

VITA

Ahmed Saeed is a PhD student at Georgia Tech, where he is advised by Prof. Mostafa Ammar and Prof. Ellen Zegura. His research interests include scalable packet processing in software, and wireless and mobile computing. Before joining Georgia Tech, Ahmed worked as research fellow on a joint appointment at Carnegie Mellon University and Qatar University. Ahmed received the Google PhD Fellowship in Systems and Networking in 2017. He received his bachelor's degree from Alexandria University in 2010.