MITIGATING INTERCONNECT AND END HOST CONGESTION IN MODERN NETWORKS

A Thesis
Presented to
The Academic Faculty

By

Yimeng Zhao

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

Georgia Institute of Technology

August 2020

MITIGATING INTERCONNECT AND END HOST CONGESTION IN MODERN NETWORKS

Approved by:

Date Approved: June 9, 2020

To my parents, for their endless love and support

ACKNOWLEDGEMENTS

I would like to express my greatest gratitude to my thesis advisors, Prof. Mostafa Ammar and Prof. Ellen Zegura for their support and guidance. Apart from their expertise in conducting high-quality research, I would especially appreciate their trust in my capability and the freedom they gave me in exploring my research topics. They are patient and supportive, always encouraging me even when I thought about my work being fruitless and my effort being futile. They motivated and trained me to grow into an independent researcher.

I am also grateful to the rest of my thesis committee members, Prof. Jim Xu, Prof. Ashutosh Dhekne, and Prof. Douglas M. Blough, for their constructive suggestions that improved many aspects of this dissertation. I would also want to thank Dr. Niky Riga, Sarah Edward, Dr. Vicraj Thomas, Dr. Chip Elliott from Raytheon BBN, Dr. Minsung Jang, Dr. Kaustubh Joshi from ATT Research Lab, Dr. Shiguang Wang, Dr. James Zeng from Facebook for hosting me as a summer intern where I got the chance to work on a set of interesting problems.

I would also like to acknowledge all my colleagues and friends in the Networking Research Group at Georgia Tech. I am thankful to Dr. Ahmed Saeed, for being supportive in several projects that we collaborated on. It was great working with Ahmed and the papers we co-authored laid a solid foundation for this dissertation. I am also fortunate to have worked with Dr. Samantha Lo. Although the paper that we co-authored is not part of this dissertation, a number of things that I learned from our collaborative work have helped me make progress in my thesis work. I would also like to thank Dr. AliReza Khoshgoftar Monfare, Dr. Karim Habak, Dr. Liang Liu, Dr. Sen Yang, Tarun Mangla, Long Gong, Yifeng Cao, Huayi Wang, Jingfan Meng for their everlasting friendship and help. The good time we spent together made my PhD journey much easier.

Last but not least, I am deeply indebted to my parents, Jun Zhao and Qifeng Ye, for their endless love and support. They encouraged me to start my research journey and always have faith in me. I cannot imagine being able to survive my PhD journey without their support. I would also like to express my appreciation to my boyfriend, Pingkai Liu, for providing constructive feedback on my research work and being a great support in my life.

TABLE OF CONTENTS

Acknowledgments v

List of Tables xi

List of Figures xii

Chapter 1: Introduction and Background 1

1.1 Primary Contributions 4

1.1.1 A Framework to Mitigate Inter-AS Congestion through CP/ISP Collaboration 4

1.1.2 A Backpressure Mechanism for Congested Queuing System at End Hosts 4

1.1.3 Analyzing the CPU Cost of Networking Stack 5

1.2 Thesis Outline 5

Chapter 2: Related Work 6

2.1 Traffic Engineering 6

2.1.1 Intra-domain TE for ISPs 6

2.1.2 Inter-domain TE from CPs' Perspective 7

2.1.3 Collaborative Inter-domain TE 8

2.1.4 Cooperative Content Distribution and TE 9

2.2 Mitigating Congestion in Data Center Networks 9

2.2.1 Congestion Control Algorithms 9

2.2.2 End Host Egress Path Congestion 10

2.2.3 End Host Ingress Path Congestion 11

2.3 End Host CPU Efficiency 11

Chapter 3: A Framework to Mitigate Inter-AS Congestion through CP/ISP Collaboration 13

3.1 Background 15

3.1.1 Network Context 15

3.1.2 The need for Content Provider-ISP Cooperation 17

3.1.3 Interdomain Congestion across ISP Entry Points 18

3.2 Unison Overview 19

3.2.1 Unison Design Goals 20

3.2.2 Architecture Overview and Operation 21

3.3 Unison vSwitch Mapper 24

3.4 Unison Throughput Optimizer 26

3.4.1 Traffic Matrix 27

3.4.2 Feasibility and Weighted Fairness 28

3.4.3 Bandwidth Allocation Algorithm 30

3.5 Evaluation 31

3.5.1 Experimental Setup 32

3.5.2 The Value of Unison 33

3.5.3 Impact of CP participation 36

3.5.4 Impact of environmental parameters 37

3.5.5 Summary 38

Chapter 4: A Backpressure Mechanism for Congested End Host 39

4.1 Background and Motivation 42

4.1.1 Packet Queuing at End Hosts 42

4.1.2 Types of Packet Drops 44

4.1.3 Cost of Long Queues 46

4.1.4 Cost of Packet Drops 46

4.2 zD Design Principles 50

4.3 zD Overview 52

4.3.1 Source Buffer Regulator 53

4.3.2 Backpressure Interface 55

4.3.3 Paused-Flows Queue 57

4.4 Implementation 58

4.4.1 TCP/IP Stack Implementation 58

4.4.2 Hypervisor Implementation 60

4.5 Evaluation 62

4.5.1 Experiments Setup 62

4.5.2 Overall Performance 65

4.5.3 Microbenchmark 69

4.6 Limitations of zD 73

4.7 Summary 73

Chapter 5: Analyzing the CPU Cost of Networking Stack 74

5.1 Measurement Setup 76

5.2 Overall Stack Performance 79

5.3 Admission Control to the Stack 80

5.3.1 Automatic Packet Sizing 81

5.3.2 Backpressure and Scheduling 83

5.3.3 Batching Ingress Packets 86

5.4 Per-packet Overhead 87

5.5 Summary 91

Chapter 6: Concluding Remarks 93

6.1 Future Directions 94

References 105

LIST OF TABLES

3.1 List of notation 28

4.1 Drops and retransmission in case of standard Linux kernel and hypervisor-only zD implementation. 71

5.1 Tuning parameters 78

LIST OF FIGURES

3.1 Network context. 15

3.2 Example of two ASes connected through two links (i.e., destination AS has two entry points). 16

3.3 CDF showing likelihood of congestion at one or more entry points. 18

3.4 Overview of Unison architecture. 22

3.5 Traffic matrix determination in the Mapping module 23

3.6 Influencing inbound traffic through hints. 24

3.7 Throughput gain created by Unison compared to the baseline over different ISP topologies. 33

3.8 Throughput gains comparing optimizing intra-domain only and Unison. . . 34

3.9 Comparison between WBA and MFC algorithm showing minor throughput impact with weighted fairness. 34

3.10 OptAll v.s. only for OptPartial showing that non-participating CPs enjoy a free ride of increased throughput in OptAll as participating CPs while only participating CPs achieve increased throughput in OptPartial. 35

3.11 Impact of window size used in Unison on total ISP throughput under dynamic traffic demand. 36

3.12 Optimality gap as function of aggregate flow. 37

4.1 Schematic of queue architecture at end hosts. 40

4.2 Architecture of queues in end hosts. 42

4.3 Bufferbloat, when pfifo queue size is 8k slots, leads to two orders of magnitude degradation in RTT. High contention and virtual packet drop rates, when pfifio queue size is 1k slots, leads to an order of magnitude degradation in tail latency compared to zD. 47

4.4 zD reduces CPU usage in both VM and the physical machine compared to standard kernel implementation for TSQ (pfifo). 48

4.5 CDF of frame size, showing the impact of tail RTT performance on the behavior of TSO autosizing algorithm. Larger tail latency yields smaller packets, causing higher CPU cost. 49

4.6 Schematic of zD architecture at end hosts. 51

4.7 Illustration of different backpressure steps. 56

4.8 Flow chart describing TCP/IP stack with zD 60

4.9 10Gbps network speed with a qdisc of 100 slots in the hypervisor 61

4.10 10Gbps network speed with a qdisc of 1000 slots in the hypervisor 63

4.11 1Gbps network speed with a qdisc of 1000 slots in the hypervisor 64

4.12 zD reduces the tail of RTT by 100x with both 10G network and 1G network 65

4.13 Compared with Carousel, zD achieves higher throughput, lower VM CPU usage, lower vHost CPU usage, and fewer TCP retransmissions 67

4.14 Compared with TSQ, zD achieves higher throughput and fewer TCP retransmissions when 1 vCPU is assigned to the VM. 67

4.15 CDF of flow RTT with hypervisor-only zD. 70

4.16 Throughput under different Qdiscs 70

4.17 CPU usage in VM under different Qdiscs 71

4.18 CPU usage in VM under light traffic load 72

5.1 Schematic of the packet transmission path with identified pain points marked in red. 77

5.2 Overall performance of the network stack as a function of the number of flows . . . 79

5.3 CUBIC v.s. BBR with 5% drop rate. . . . 82

5.4 CUBIC v.s. BBR with 0% drop rate. . . . 83

5.5 CDF of flow rate showing that FQ fails to ensure fairness among flows . . . 84

5.6 CPU usage as function of Qdisc queue length . . . 85

5.7 Average execution time of FQ enqueue operation . . . 86

5.8 Packet transmission function call graph . . . 87

5.9 Rates of RX interrupts and ACK per second . . . 88

5.10 Aggregate cache misses . . . 89

5.11 Time to acquire Qdisc lock . . . 90

5.12 Function profiling results for top functions . . . 92

SUMMARY

One of the most critical building blocks of the Internet is the mechanism to mitigate network congestion. While existing congestion control approaches have served their purpose well in the last decades, the last few years saw a significant increase in new applications and user demand, stressing the network infrastructure to the extent that new ways of handling congestion are required. This dissertation identifies the congestion problems caused by the increased scale of the network usage, both in inter-AS connects and on end hosts in data centers, and presents abstractions and frameworks that allow for improved solutions to mitigate congestion.

To mitigate inter-AS congestion, we develop Unison, a framework that allows an ISP to jointly optimize its intra-domain routes and inter-domain routes, in collaboration with content providers. The basic idea is to provide the ISP operator and the neighbors of the ISP with an abstraction of the ISP network in the form of a virtual switch (vSwitch). Unison allows the ISP to provide hints to its neighbors, suggesting alternative routes that can improve their performance. We investigate how the vSwitch abstraction can be used to maximize the throughput of the ISP.

To mitigate end-host congestion in data center networks, we develop a backpressure mechanism for queuing architecture in congested end hosts to cope with tens of thousands of flows. We show that current end-host mechanisms can lead to high CPU utilization, high tail latency, and low throughput in cases of congestion of egress traffic. We introduce the design, implementation, and evaluation of zero-drop networking (zD) stack, a new architecture for handling congestion of scheduled buffers.

Queue overflow is not the only cause of congestion on the egress path. Another cause of congestion is CPU resource exhaustion. The CPU cost of processing packets in networking stacks, however, has not been fully investigated in the literature. Much of the focus of the community has been on scaling servers in terms of aggregate traffic intensity (packets transmitted per second), but bottlenecks caused by the increasing number of concurrent flows have received little attention. We conduct a comprehensive analysis on the CPU cost of processing packets and identify the root cause that leads to high CPU overhead and degraded performance in terms of throughput and RTT. Our work highlights considerations beyond packets per second for the design of future stacks that scale to millions of flows.

CHAPTER 1
INTRODUCTION AND BACKGROUND

The Internet has established itself as a critical global infrastructure for information exchange. Originated from the ARPANET with a handful of nodes, the Internet has evolved continuously to adapt to new services and performance expectations. Today, the Internet is connecting millions of people and machines. It is now fully integrated in our society and has revolutionized not only the way we communicate but also the way we live our lives.

The huge success of the Internet is achieved with evolving designs and protocols to cope with new requirements. The network infrastructure and protocols may be well-suited to support the popular uses at the time they were designed, but as time goes on, the increase in the scale of traffic and the creation of new applications would always pose new challenges in managing networks. The mechanism to mitigate network congestion is a classic example of this kind. The importance of congestion control was first brought to attention in the 1980s. It is observed that the data transfer rate fell by orders of magnitude even though the network links stayed busy. To solve the problem, the research communities designed an algorithm that adjusts the data sending rate according to the congestion level in the network. The algorithms were developed in an iterative design, providing a solid foundation for more advanced mechanisms (i.e., TCP congestion control) that has been deployed widely in the global Internet today.

Mitigating network congestion is challenging, and there is never a once-and-for-all solution. Beyond the continuous evolution of the Internet, the last few years saw a significant increase in new applications and user demand. As more digital content has migrated to the Internet, it has become increasingly challenging for content providers and their proxies – in the form of content distribution networks (CDNs) - to reach end users with low latency. Another force driving change is the growing need for more services and applications at cloud data centers. According to the Cisco global cloud index, the traffic demand and the data storage in data center networks are experiencing a 20%- 30% compound annual growth rate now [1]. The increased scale of network usage stresses the network infrastructure, both in network and at data centers, requiring new mechanisms to mitigate congestion without significantly increasing the resources.

The central theme of this thesis is to identify the congestion problems caused by the increased scale of the network usage and explore new abstractions and frameworks that allow for improved solutions both in inter-AS connects and on end hosts in data centers to mitigate congestion. Specifically, we focus on the following challenges:

- **Inter-Autonomous System Congestion:** The Internet's goal of efficient global data connectivity led to its development as a sprawling hierarchical network of interconnected Autonomous Systems (ASes). BGP policies, initially created assuming by default that all ASes are equal, determine the role of an AS within the hierarchy. Mechanisms developed for BGP allow for fine-grain control over how traffic exits an AS. Each AS determines the next hop according to the policies agreed on with that next hop. This approach to inter-AS routing is feasible to manage under a hierarchical structure in which roles are clear, and in turn who pays whom for carrying the traffic is well established.

  However, with increased desire on low latency, the modern Internet is evolving from a hierarchy of Autonomous Systems (ASes) to a flatter structure [2]. Source ASes have the flexibility to choose how to reach their destination but it is not easy for destination ASes to control inbound traffic [3]. *This makes BGP highly asymmetrical.* The asymmetry is exacerbated by the advancements in Software Defined Interconnects [4, 5]. The asymmetry of Internet routing, along with the current flat topology of the network, leave routing decisions largely in the hand of content providers. This is not ideal; CPs have limited visibility into the ISP network and some of the routing decisions made by CPs can be erroneous, leading to congestion in ISP network [6]. This is especially problematic in the presence of congestion when an alternative entry point has to be selected that does not have to be close geographically to the user [7]. In addition, dynamic path selection by a content provider, independent from the ISP, complicates fault attribution. In particular, a user facing poor quality of experience of an online service will typically blame the ISP, despite that the problem can be caused by the CP selecting a longer or more congested path [8, 9, 10, 11, 12].

- **End Host Queuing Congestion:** Modern network stacks have to handle traffic from tens of thousands flows and hundreds of virtual machines per single host. This scale sparked interest in improved scheduling and prioritization between these applications through the introduction of efficient packet processing and scheduling mechanisms. However, congestion control of egress path at the end host has received little attention. In current settings, packets from different sources accumulate in the queue. Once a queue runs out of space packets are dropped. Drops at end host are particularly inefficient as recovery from such losses are typically handled the same way as losses in the network. This means that losses which can be recovered through simple signaling within the end host at nanosecond to microsecond timescales are handled through end to end signals which operate at microsecond to millisecond timescales. Drops can also induce severe reaction from congestion control which cuts its window in reaction to packet loss, leading to lowered throughput. A few proposals attempt to avoid packet drops of egress traffic at end hosts. However, their approaches have poor performance when handling a large number of senders [13, 14].

- **End Host CPU Inefficiency:** Queue overflow is not the only cause of congestion on the egress path. Another cause of congestion is exhausting CPU resources. Even if the server has enough networking capacity, traffic must wait before being handled by a CPU core. The CPU cost of processing packets in networking stacks, however, has not been fully investigated in the literature. Much of the focus of the community has been on scaling servers in terms of aggregate traffic intensity (packets transmitted per second)

[15, 16], but bottlenecks caused by the increasing number of concurrent clients, resulting in a large number of concurrent flows, have received little attention.

1.1 Primary Contributions

This thesis seeks to address the challenges to mitigate network congestion in the context discussed above. We present a brief description of our work in this section.

1.1.1 A Framework to Mitigate Inter-AS Congestion through CP/ISP Collaboration

We present the design of Unison, a framework that allows an ISP to help the CPs choose between the entry points, with the goal of jointly optimizing the intra-domain routes and inter-domain routes. The basic idea of Unison is to provide the ISP operator and the neighbors of the ISP with an abstraction of the ISP network in the form of a virtual switch (vSwitch). This abstraction allows the content providers to program the virtual switch with their requirements. In addition, Unison allows the ISP to provide hints to its neighbors, suggesting alternative routes that can improve their performance. We investigate how the vSwitch abstraction can be used to maximize the throughput of the ISP. We show through extensive simulations that Unison can improve ISP throughput by up to 30% through cooperation with content providers. We also show that cooperation of content providers only improves performance, even for non-cooperating content providers.

1.1.2 A Backpressure Mechanism for Congested Queuing System at End Hosts

We show that current end-host mechanisms can lead to high CPU utilization, high tail latency, and low throughput in cases of congestion of egress traffic within the end host when handling tens of thousands of flows. We introduce the design, implementation, and evaluation of zero-drop networking (zD) stack, a new architecture for handling congestion of scheduled buffers. It allows network operators to set a fixed queue size that is independent of the number of flows, eliminating bufferbloat issues at scale. The basic idea is to define a backpressure interface that triggers packet dispatch from senders only when the scheduled buffer has room for new packets. We implement zD in the Linux kernel to apply backpressure for two cases: 1) when the queues and traffic sources are within the kernel stack (i.e., in the same virtual or physical machine), and 2) when the traffic sources are in the virtual machine and the queues are in the hypervisor.

1.1.3 Analyzing the CPU Cost of Networking Stack

We conduct a measurement study to identify the bottlenecks caused by the increasing number of concurrent clients, resulting in a large number of concurrent flows. In particular, we define two broad categories of problems; namely, admitting more packets into the network stack than can be handled efficiently, and increasing per-packet overhead within the stack. We show that these problems contribute to high CPU usage and network performance degradation in terms of aggregate throughput and RTT. Our measurement and analysis are performed in the context of the Linux networking stack, the only fully-implemented, publicly available, most widely used network stack. However, because we are not overly reliant on Linux-specific features, our broad conclusions generalize well to other stacks. Our work highlights considerations required in the design of future networking stacks that need to be capable of handling large numbers of clients and flows.

1.2 Thesis Outline

The rest of this dissertation is organized as follows. Chapter 2 reviews the previous work related to this thesis. In chapter 3, we present the design of a system that allows an ISP to optimize its intra-domain routes and inter-domain routes, in collaboration with content providers. Chapter 4 investigates a scalable architecture at end hosts for applying backpressure from congested queues to traffic sources to reduce packet drop at end hosts. In chapter 5, we present our analysis on bottlenecks of the Linux networking stacks that lead to CPU inefficiency. We conclude this dissertation and provide future work in Chapter 6.

CHAPTER 2
RELATED WORK

Mitigating network congestion has been an active area of networking research for many years. In this chapter we review some of the prior works that are closely related to mitigating network congestion. We organize the related work by topic according to the challenges mentioned in section 1. We start by reviewing the traffic engineering (TE) approaches that aim at reducing network congestion in ISPs' and CPs' network. Then we present techniques and mechanisms to reduce congestion in the data center environment, including a set of congestion control algorithms as well as end-host approaches to mitigate egress-path and ingress-path congestion. Finally, we examine the state-of-art in improving the CPU efficiency of networking stacks at end hosts.

2.1 Traffic Engineering

Traffic engineering (TE) is an optimization method for efficiently allocating resources so that certain benefits are maximized under pre-defined constraints. A key goal of TE is to avoid congestion by optimizing routing policies based on network topology and expected traffic demand. TE approaches are applied in all kinds of networks, but we focus here on TE works that decide intra-domain and inter-domain routing for ISPs and content providers.

2.1.1 Intra-domain TE for ISPs

The routing protocols are clustered into two macin groups: intra-domain protocols to route traffic within an AS, and inter-domain protocols to route traffic between ASes. Intra-domain routing protocols, such as Open Shortest Path First (OSPF), Intermediate System-Intermediate System (IS-IS), and Routing Information Protocols (RIP), are widely used by ISPs. In these protocols, each link in the network is assigned a weight, and the shortest path between each source and destination node is calculated based on the assigned weights. However, these protocols fail to take capacity constraints and traffic characteristics into account in making routing decisions. In addition, arbitrary distribution of flows is not supported.

Alternative to shortest path routing, Multi-Protocol Label Switching (MPLS) was introduced as a more flexible intra-domain routing protocol. MPLS enables explicit routing, which allows a packet to follow a pre-determined path rather than a path calculated by shortest path routing protocols. The basic idea is to attach a fix-length label at the ingress, and the label rather than the IP header will be used to make forwarding decisions [17].

Some other work uses SDN-based mechanisms to implement network management frameworks to achieve varying traffic engineering goals [18, 19]. Merlin [18] is a programming language for provisioning network resources within a single domain. SOL [19] presents a framework that allows network operators to express a range of constraints and objectives in high level languages from which SOL generates and solves the optimization problems. These frameworks demonstrate the potential of SDN techniques for efficiently solving complex network optimization problems.

2.1.2 Inter-domain TE from CPs' Perspective

Large content providers have already taken initiatives to improve inter-domain routing aimed at delivering high-volume traffic while improving user-perceived performance [4, 5]. To tackle the limitations of BGP, Facebook designed Edge Fabric, a system for optimizing routing at the edge [5]. Edge Fabric monitors capacities and demand for outgoing traffic, and enforces better route selection by overriding the router's normal BGP selection for outbound traffic in Points of Presence (PoPs). Google takes a similar approach, designing an edge architecture that delivers high-demand traffic with low latency [4]. While Facebook only optimizes routing in PoPs, Google's architecture has a global traffic engineering system that enables application-aware routing at Internet scale. Both systems use their already deployed SDN infrastructure to dynamically change BGP entries.

2.1.3 Collaborative Inter-domain TE

A few research studies have explored the benefits of allowing neighboring domains to collaboratively manage traffic [20, 21]. In these inter-domain architectures, neighboring ISPs exchange information about their traffic volume and preferred routes, and participate in negotiations until they reach mutually acceptable routes. Mahajan et al. [20] propose a negotiation-based routing framework where neighboring ISPs exchange their preference for inter-domain paths. Shrimali et al. [21] use the idea of multi-criteria optimization and Nash bargaining to approach the inter-domain routing problem. The negotiation-based approach requires clean-slate architectures and protocols, which suffer from deployment challenges.

Another research direction proposes a centralized inter-domain routing broker to provide end-to-end guaranteed paths [22, 23, 24]. In some of these proposals, ISPs provide QoS-enabled pathlets [22], which are stitched together by a centralized mediator called a service broker. The design requires that users submit their requirements and service providers submit their topology information to a service broker, who chooses the proper path in each domain and stitches the paths together to form an end-to-end path based on a global view of all participating networks. While a centralized network provisioning approach may optimize the inter-domain routing in an efficient way, the system is difficult to scale. The global network has a large number of independently operated networks and a large number of BGP-speaking routers. It remains to be seen whether ISPs will agree to participate and whether they would willing to share information with the service broker.

Other proposals show that Internet exchange points (IXPs), the physical locations where multiple networks connect to exchange traffic, provide an ideal location to improve the existing routing system [25, 26]. Those approaches build on recent technology trends of Software Defined Networking (SDN) to utilize traffic-management capabilities and explore various use cases ranging from inbound route selection to application-specific peering. In the SDX approach [25], participants exchange BGP update messages with the IXP route server, and the SDN controller combines the SDN policy with the BGP routing information to compute forwarding table entries in the IXP fabric. However, such proposals do not provide control over all possible ingress paths to an ISP as IXPs do not represent all possible connection points between ISPs and CPs [4].

2.1.4 Cooperative Content Distribution and TE

In order to improve the content delivery efficiency, the collaboration of ISP and CDN has been proposed. Jiang et al. [27] focus on the joint optimization of TE in ISP and server selection in CDN. Poese et al. [28] shows that server selection alone, without TE in ISP, is sufficient enough to improve the content delivery. Another line of work focuses on mapping clients' requests to the closest CDN clusters using DNS-based approach or SDN-based approach [29, 30].

2.2 Mitigating Congestion in Data Center Networks

2.2.1 Congestion Control Algorithms

Congestion control protocols are an integral part of current data center networks. They seek to maximize utilization and achieve a desired allocation of network resources without oversubscribing any link. Currently, TCP and its variants dominate the data center networks. DCTCP [31], HULL [32], and TIMELY [33] relies on explicit signals that come in a variety of forms, ranging from packet drop to Explicit Congestion Notification (ECN) to RTT, to adjust its window size or explicit rate in response to different congestion levels.

Protocols like XCP [34], RCP [35], pFabric [36], and Fastpass [37], shift congestion control to intermediate switches within the network. Those protocols are effective in reducing TCP flow latency with fast convergence time because more information, collected from all flows passing through switches, is used to make decisions. However, they require significant modification on current hardware architecture, thus are not deployed in large scale in data center networks.

2.2.2 End Host Egress Path Congestion

TCP Small Queue (TSQ) [14] is the most prominent mechanism in practice today that aims at mitigating the congestion at the queuing systems in Linux kernel. TSQ limits the number of packets of any TCP flow that can be queued below the transport layer. It prevents further packets from queuing into the IP layer if there are already two outstanding packets waiting to be transmitted by NIC. TSQ has been deployed in the Linux TCP stack and works well in reducing the buffering in the network stack, but it does not solve the problem fundamentally. With thousands of TCP flows, limiting each flow to two outstanding packets cannot prevent packet drop from happening.

While TSQ relies on signaling within the kernel stack to maintain the per-flow limit, more recent proposals extend TSQ signalling to enforce the per-flow limit to queues beyond the kernel stack. Carousel employs delayed delivery of completion signals from the NIC to the TCP stack to apply backpressure from a user-space network processor to the kernel TCP stack [38]. While traditional completion is implemented as a signal from driver to transport stack in the same order of packets arriving at the NIC, asking the transport stack to send more packets, Carousel implements out-of-order completions and relies on TSQ to limit the number of packets per flow. PicNIC [39] extends TSQ signalling to allow backpressure from a hypervisor to traffic sources inside a VM. It also proposes a per VM budget of packets, for cases when a VM doesn't support the backpressure signal. Note that Carousel and PicNIC exhibit the inherent TSQ issues discussed earlier as queues have to accommodate $O(N)$ packets for N flows. End host queue buildup can be handed in a similar manner to in-network queue buildup through congestion control algorithms [40]. This approach does not eliminate packet drops but helps improve tail latency.

Queue overflow is not the only cause of congestion on the egress path. Another cause of congestion is exhausting CPU resources. Several systems proposed improve the CPU efficiency of queuing in the network stack, thus allowing it to handle more packets and flows. SENIC [41] improves rate limiting scalability by allowing for software queues to make use of hardware to improve rate limiting performance. Carousel [38] employs a time-based marking of packets and the timing wheel data structure to improve the performance of software-only rate limiting. Eiffel [42] presents a software only solution for general purpose packet scheduling. Several proposals explore improving efficiency of scheduling algorithms by offloading them to hardware [43, 44].

2.2.3 End Host Ingress Path Congestion

Recently several proposals have looked at congestion control of the ingress path, implementing scalable networking stacks [45, 46] and enforcing isolation between receiving flows [47, 39]. Ingress path congestion at the end host occurs when one receiver (e.g., VM or socket) receives packets at a high rate so that it overwhelms the CPU at the receiver. Congestion control of ingress traffic typically requires fine grain CPU scheduling to allocate enough resources to process incoming packets for all receivers. Congestion can also happen due to incast scenarios when ingress traffic demand exceeds the NIC capacity at receiver. Resolving incast issues in data center networks has been an active area of congestion control research [31, 33, 48, 49, 50]. DCTCP [31] depends on an explicit feedback mechanism to rate limit data sources for reducing congestion. D_3 [50] is a deadline-aware congestion control that utilizes flow deadline information to allocate bandwidth. Timely [33] is an RTT-based congestion control scheme running over NICs with OS-bypass capabilities.

2.3 End Host CPU Efficiency

A number of approaches are proposed to improve the CPU efficiency of networking stacks at end hosts. Much of the focus of these work has been on scaling servers in terms of aggregate traffic intensity in terms of packets transmitted per second, while maintaining low latency [51, 52, 53, 54, 45]. Another line of work focus on improving a specific stack such as the scheduling stack [55, 38, 42, 56] or the TCP stack [15, 16], without understanding the interaction between different protocols and layers.

Some recent proposals address scaling the whole stack to handle a large number of flows [57, 58, 59, 60]. TAS steers tasks into fast and slow paths and moves the fast path to a dedicated core to reduce the system call, cache coherence and locking overheads [59]. Fast-path handles common-case TCP packet processing and resource enforcement while flow-path handles heavy-weight tasks such as connection setup/teardown, congestion control, and timeouts. TAS dynamically allocates the appropriate amount of CPUs to accommodate the fast-path, depending on the traffic load. With changing traffic load, AS dynamically allocates the appropriate amount of CPUs to the fast-path to improve CPU efficiency. Other works demonstrate the benefit of running the entire networking stack in the user-space to improve CPU efficiency [51, 57]. mTCP [57] provides a scalable TCP stack by batching both I/Os and functions calls and is evaluated at a maximum of 16k flows. Other systems are evaluated at a few thousands flows [59] and up to twenty thousand flows [58, 60]. These work typically focus on specific functionality (e.g., RPC performance or transport layer performance), with emphasis on short lived flows. None of the existing optimized components was tested with a load larger than 50k flows. Our work in this thesis fills the gap by looking at the complicated interaction between different components when the server scales up to handle hundreds of thousands of long-lived flows.

CHAPTER 3
A FRAMEWORK TO MITIGATE INTER-AS CONGESTION THROUGH CP/ISP COLLABORATION

The asymmetry of Internet routing, along with the current flat topology of the network, leave routing decisions largely in the hand of content providers. For example, systems like Egde Fabric [4] and Espresso [5], employed by Facebook and Google, respectively, improve reaction time of content providers (CPs) to congestion. On the other hand, access ISPs still have to rely on typical, ineffective, standard BGP tools that take tens of minutes to converge. This is not ideal for two main reasons. First, content providers can only make decisions based on their view of the network which is typically based on estimates of capacity from the ISP entry point to the user (e.g., relying on CDNs or Points of Presence physically closest to the user). This is especially problematic in the presence of congestion when an alternative entry point has to be selected that does not have to be close geographically to the user [7]. Second, dynamic path selection by a content provider, independent from the ISP, complicates fault attribution. This problem can be alleviated with better coordination between ISPs and content providers. There has been attempts to allow such exchange of information through brokers or at Internet Exchange Points (IXPs) [22, 23, 24]. Broker-based solutions are not scalable as they represent a centralized Internet. IXP-based solutions (e.g., SDX [25]) provide a good first step, however, their impact is limited to entry points connected to a single IXP and do not specify how to operate at the full scale of an ISP network.

In this chapter, we focus on the congestion problem at inter-AS connects. We present the design of Unison, a system that allows an ISP to jointly optimize its intra-domain routes and inter-domain routes, in collaboration with content providers (§3.2). Unison's design is based on the argument that deciding which entry point traffic should take to reach a user is a decision that should be performed jointly by both the ISP and the content provider. Our work is motivated by two observations: 1) measurements of interconnect congestion show that while some entry points between a content provider and an ISP can be congested several other entry points are uncongested across geographic regions (§3.1.3), and 2) the availability of software defined interconnect systems at content providers makes it feasible to coordinate between multiple networks and control inter-domain traffic.

The basic idea of Unison is to provide the ISP operator and the neighbors of the ISP with an abstraction of the ISP network in the form of a virtual switch. This abstraction allows the content providers to program the virtual switch with their requirements. It also allows the ISP to use that information to optimize the performance of its network. In addition, Unison allows the ISP to provide hints to its neighbors, suggesting alternative routes that can improve their performance. Unison leverages recent advancements in SDN. In particular, Unison makes use of SDN infrastructure at most modern ISPs [61, 62] as well as the programmable Interconnects at content providers [4, 5]. It also leverages SDX as a means to convert a vSwitch configuration into OpenFlow and BGP rules. This enables Unison to be a programmable platform that can be used for multiple Inter-domain routing applications (e.g. load balancing, or redirection through middleboxes).

We focus on the objective of maximizing throughput of content provider traffic going through the ISP. In particular, we are interested in the creation of a vSwitch abstraction from an ISP topology (§3.3). Then, we investigate how this abstraction can be used to maximize the throughput of the ISP (§3.4). We formulate the problem as an integer program. Through that formulation, we investigate the value of Unison in terms of improving ISP throughput in case of congestion. We also show the impact of non-cooperating content providers. Our evaluation of Unison is conducted through simulations (§4.5). We show that Unison can improve ISP throughput by up to 30% through cooperation with content providers. We also show that cooperation of content providers only improves performance, even for non-cooperating content providers (e.g., a single cooperating neighbour can improve ISP

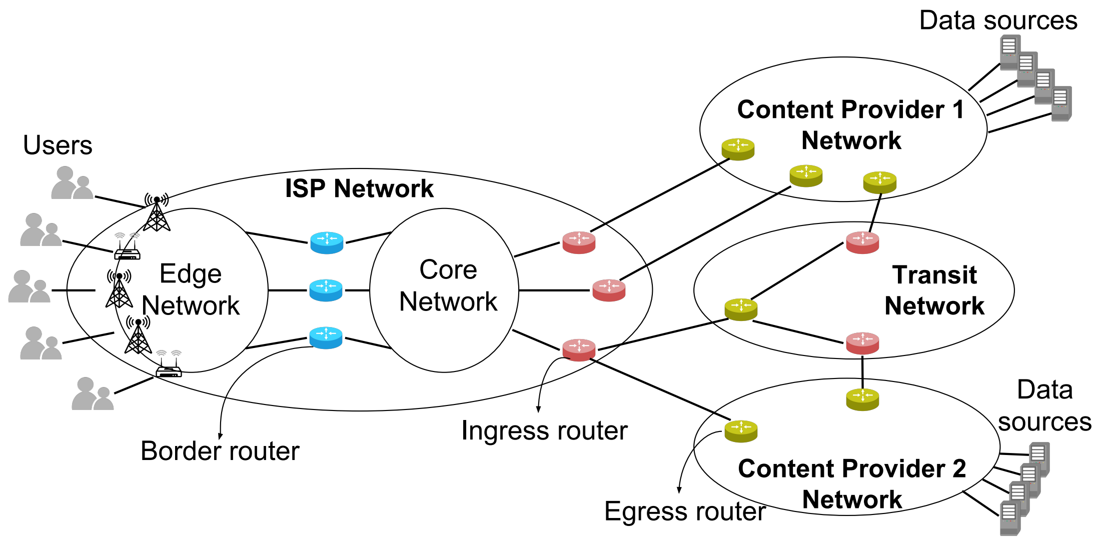


Figure 3.1: Network context.

throughput by up to 6%).

3.1 Background

3.1.1 Network Context

It has become increasingly important for content providers (CPs) to reach consumers with low latency. One way this has been achieved is through direct peering between CP and ISP networks. While this has helped, we believe it is necessary in today's demanding environment to also coordinate traffic routing across this peering connection. Recent work provides evidence that large CPs use peering links to carry the majority of the traffic to access ISPs [4], making this coordination essential for the CP to achieve its reduced latency objective.

We consider a network similar to the schematic in Figure 3.1. Multiple CPs are connected to an ISP, either directly at their points of presence, or through transit autonomous systems. We focus on the prevalent scenario where access ISPs connect directly with CPs. The ISP network is composed of *Ingress Routers* that receive traffic intended for users. Traffic is routed through the ISP's *Core Network* to *Edge Networks* that users connect to

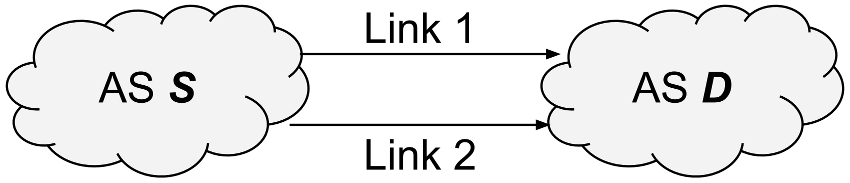


Figure 3.2: Example of two ASes connected through two links (i.e., destination AS has two entry points).

directly (e.g., cellular edge). The Core Network and Edge Network are connected through *Border Routers*. These Border Routers deliver traffic to a large number of users. In this paper, we are concerned with the problem of routing data from Ingress Routers to Border Routers, as multiple such routes can exist [5]. However, we assume that once traffic reaches a Border Router, its path to the user is deterministic.

We assume that some or all participating networks rely on programmable infrastructure to determine and configure routes. These assumptions are increasingly becoming the reality in the modern Internet as announced by ISPs [61, 62] and CPs [4, 5]. We note that CP networks without programmable infrastructure are able to handle routing suggestions from the ISP using existing APIs, motivated by the promise of higher throughput. Moreover, Unison does not require all CPs to cooperate with the ISP. Our results show that Unison can remain beneficial for the majority of CPs, even if only a subset of them cooperate.

We do not make any assumptions about data placement, as none is needed for our context. This is because our interest is in cases where network congestion, rather than physical distance, is the main bottleneck. Although ISP-CDN collaboration allows for strategic placement of data and routing optimization, which improves data delivery [27, 28, 29, 30], we are interested in reducing congestion where some links between the CP and the ISP are congested. Circumventing this congestion requires using a different entry point, that can be in a different physical location. This scenario is typical as we show later.

3.1.2 The need for Content Provider-ISP Cooperation

Internet Routing Asymmetry: Current Internet routing is asymmetric because it gives traffic sources much more control over route selection compared to traffic destinations. This asymmetry is necessary to ensure traffic is routable in case of conflicting preferences. For instance, consider the case in Figure 3.2. Suppose the source AS prefers to send traffic over Link 1. An irresolvable conflict would arise if the destination AS prefers to receive the traffic over Link 2.

Current BGP mechanisms such as path prepending and selective announcement are very limited in terms of their expression of preference. In particular, an ISP can stop announcing certain prefixes through certain entry points, which is an extreme approach and typically not preferred for redundancy. The other available approach is path prepending which does not provide clear preference between paths and does not necessarily differentiate between CPs. Furthermore, these approaches rely on BGP convergence which is known to be slow, especially compared to Software Defined Interconnects. The asymmetry problem can be mitigated through the use of BGP communities that depend on cooperation between peering partners, but BGP comminities tend to leak critical information such as network topology hence is not an ideal solution [63]. We consider our solution as an argument against using BGP communities.

Determining Best Path to End Users: Typically, CPs try to route traffic to end users through the geographically closest point of presence (i.e., entry point to the ISP). However, if that entry point is congested, CPs can only guess which alternative entry point to use. CPs do not have visibility into the ISP's network. This means that by selecting another entry point, CPs cannot guarantee enough capacity from that entry point to the end user. Selecting the best entry point can only be achieved if the CPs cooperate with the ISP.

Attribution of Bad QoE: When end users face bad quality of experience (QoE), it is natural for users to blame the ISP [12]. Blaming the ISP implies that the ISP did not allocate enough capacity for traffic to reach the user. However, this does not necessarily

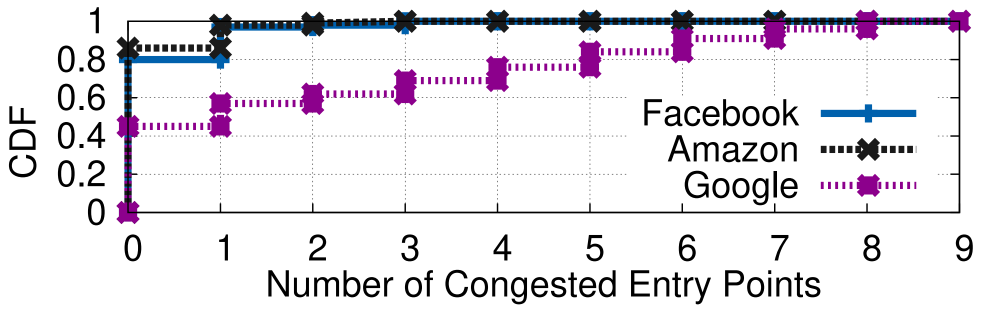


Figure 3.3: CDF showing likelihood of congestion at one or more entry points.

have to be the case. It can be that the entry point used by the CP is congested due to large traffic volume from that CP. It can also be due to the CP choosing an entry point that does not have the proper capacity in its connection to the targeted users, while other entry points have that needed capacity. It can also be the case that the CP's network is congested. This attribution is very hard to achieve accurately and can be costly to the ISP if the CP unilaterally moves traffic between entry points. For example, this unilateral behavior can force the ISP to upgrade and increase the capacity of parts of its network while the same outcome could have been achieved by simply asking the CP to use a different entry point.

3.1.3 Interdomain Congestion across ISP Entry Points

Our main hypothesis in developing Unison is that when one point of entry to an ISP is congested, several other entry points are not congested. This hypothesis is critical as it implies the existence of the option to move traffic from the congested entry point to another. Unison allows this decision to be made by the ISP rather than the ISP's neighbouring AS because the ISP knows the best, or second best, entry point to reach its customers. We validate our hypothesis by examining interdomain congestion data between a single ISP and three CPs over a period of two years [64]. The data provides measurements of latency over multiple links connecting the ASes (i.e., egress-ingress router links in Figure 3.1). Links are grouped based on location where every location captures a single point of entry in our analysis. Each data point represents the congestion status inferred from the latency over a period of 24 hours. The congestion measurement method is based on an intuition that if the latency to the far end of the link is elevated but that to the near end is not, then it is highly likely that the interdomain link is congested [65].

Figure 3.3 shows the CDF of simultaneously congested entry points between Comcast and three CPs: Facebook, Google, and Amazon. Each content provider AS is connected to the ISP through at least twelve points of entry. The results validate our hypothesis. It is not uncommon to have multiple links congested at the same time while there are still links that have available capacity. In particular, we find that in the worst cases of congestion there are 20.1%, 14.6%, 55.3% of the links are congested at a certain data point for Facebook, Amazon, and Google respectively. This means that there are at least seven alternative entry points available even in the worst cases of congestion. The objective of Unison is to allow the ISP to help the CPs choose between the entry points.

3.2 Unison Overview

Unison allows an ISP to expose a programmable interface to other autonomous systems connected to it. Unison limits the amount of information exchanged by only providing the vSwitch abstraction which does not reveal information about exact ISP topology, but provides some hints about capacity in exchange for improving performance. In particular, an AS can specify its preferred routing policies, which without Unison it would enforce regardless of the state of the ISP. Furthermore, the ISP can take into account the preferences of ASes connected to it, as well as its own capacity, to send hints back to neighbouring ASes suggesting better routes if any. It is left up to the neighbouring ASes to use these hints, thus preserving the distributed nature of the current Internet. Unison performs these functions by configuring ISP inter-domain and intra-domain routing simultaneously. Inter-domain routes are configured based on the state of the ISP as well as the routing preferences of its neighbouring ASes by providing neighbouring ASes with hints on entry points for aggregates of traffic that would optimize network performance. Intra-domain routing is optimized by configuring capacity within the ISP to accommodate demand from peer ASes. We leave further anonymization of the hints to future research.

The insight we build on is that, given proper controller infrastructure, BGP routers can be programmed dynamically based on centrally made decisions to control inter-domain routing at scale. This was demonstrated by software defined Internet routing systems developed and deployed by CPs such as Espresso by Google [5] and Edge Fabric by Facebook [4]. Unison also builds on systems that allow the realization of a single policy from preferences set by multiple autonomous systems developed for Internet Exchange Points (e.g., SDX [25]). Unison is developed for an ISP setting which requires interaction with peer ASes, as well as consolidating ISP objectives and peer ASes objectives. The design of Unison has two major components:

1. Overlay software defined control over BGP infrastructure, à la Espresso, designed to control the Interconnect.

2. Cross-controller coordination and consolidation, à la SDX, designed to handle coordination between the ISP and ASes connected to it in order to reach a feasible resource allocation.

We find that despite progress made in such systems from the perspective of CPs and Internet exchange points, the ISP perspective poses a set of new challenges and constraints. For the rest of this section, we elaborate on these challenges as well as give an overview of Unison.

3.2.1 Unison Design Goals

An access ISP can be connected with multiple CPs at potentially multiple *ingress* points for each CP. Our goal is to provide a way for ISPs to control the network taking into account considerations from all CPs and users in addition to its own network and business considerations. Although Unison can be used to achieve a wide range of objectives, we focus on the simple and natural objective of maximizing ISP network throughput subject to weighted differential treatment of different CPs. Hence, all CPs observe a less congested ISP network, which is the main goal of CP-based solutions [5, 4]. Moreover, the ISP achieves higher utilization of its network in addition to achieving its business obligations by providing paying CPs more bandwidth. This approach is challenging and has to be handled under a very strict set of constraints:

- Benefits both CPs and ISPs. ISPs have to balance many CPs. Unison should improve ISP throughput while ensuring weighted differential treatment of CPs. Our system should also improve throughput of CPs within an ISP.
- Does not require CPs to cooperate. The benefits of Unison should be achieved even if only a subset of CPs connecting to an ISP agree to participate, without penalizing non-participating ones.
- Limits information disclosure: ISPs will not be willing to disclose detailed information such as network topology, traffic load, and customer information, often considered proprietary by ISPs, to third parties such as CPs.

3.2.2 Architecture Overview and Operation

Figure 3.4 shows the details of connectivity between an ISP and a Content Provider (CP). Note that we focus on the prevalent scenario where access ISPs connect directly with CPs. However, our technique will work as long as we can construct a traffic matrix that can be translated into a virtual switch. Our approach works on a (src ip prefix, dst ip prefix) granularity. Hence, it should tailor different negotiations to different clients of the transit network. Unison operates in the control plane of the ISP and provides an interface to CPs. Unison has two main components: (i) a vSwitch Synthesizer that creates and maintains the mapping between the vSwitch abstraction and actual network equipment at the ISP, and (ii) a vSwitch Controller that combines programs from CPs as well as the ISP to generate

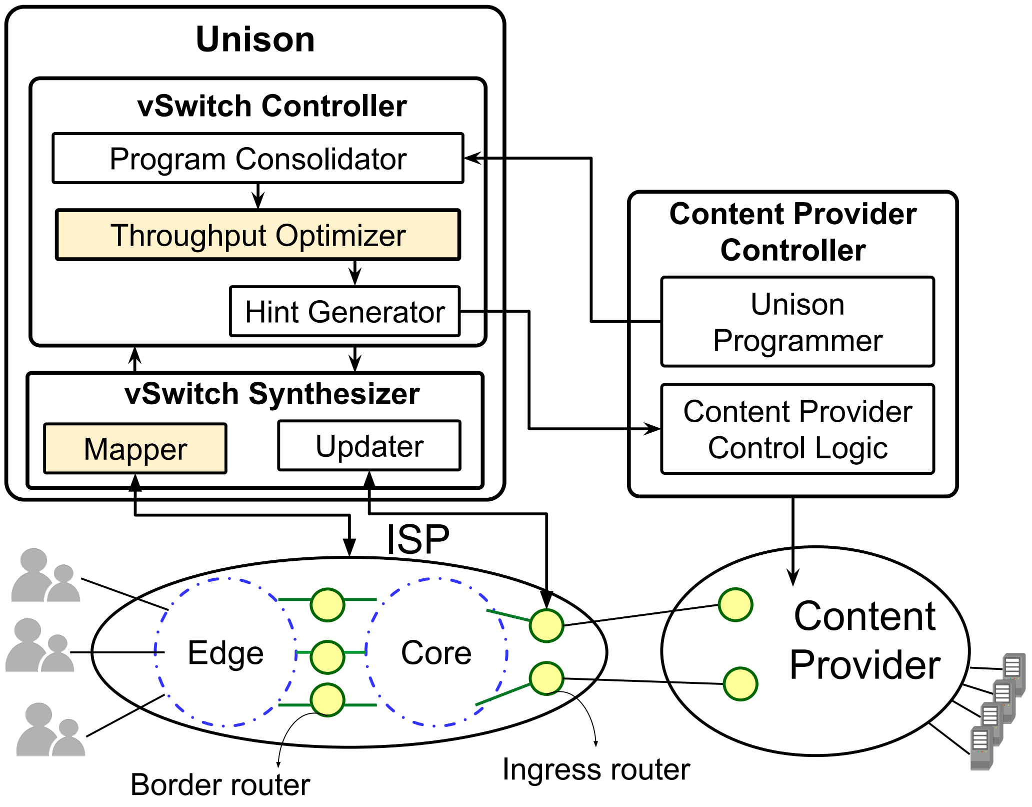
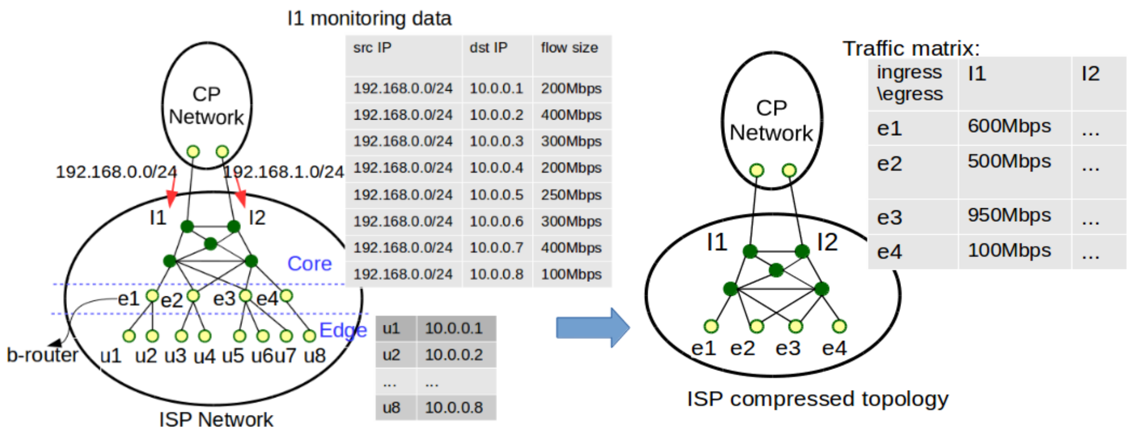


Figure 3.4: Overview of Unison architecture.

vSwitch configurations. Unison also requires minor changes in the controller of the CP network to specify its policies as well as receive and take into account hints from the ISP.

vSwitch Synthesizer: The main function of this component is to convert the complex topology of an ISP's network to a simple vSwitch with well-defined input and output ports. It also realizes high level programs of the vSwitch into actual route configurations in network elements. These two functions are the responsibility of the Mapper and Updater modules, respectively. The vSwitch Synthesizer is inspired by recent work in programmable inter-domain controllers introduced by content providers [5, 4]. In particular, these recent advancements show that a central controller can make routing decisions that reconfigure BGP routers either on per packet basis [5] or per point-of-presence basis [4].

vSwitch Controller: This component is responsible for programming the vSwitch as well as providing hints to neighbouring ASes. A vSwitch program is created by combin-



src IP	dst IP	flow size
192.168.0.0/24	10.0.0.1	200Mbps
192.168.0.0/24	10.0.0.2	400Mbps
192.168.0.0/24	10.0.0.3	300Mbps
192.168.0.0/24	10.0.0.4	200Mbps
192.168.0.0/24	10.0.0.5	250Mbps
192.168.0.0/24	10.0.0.6	300Mbps
192.168.0.0/24	10.0.0.7	400Mbps
192.168.0.0/24	10.0.0.8	100Mbps

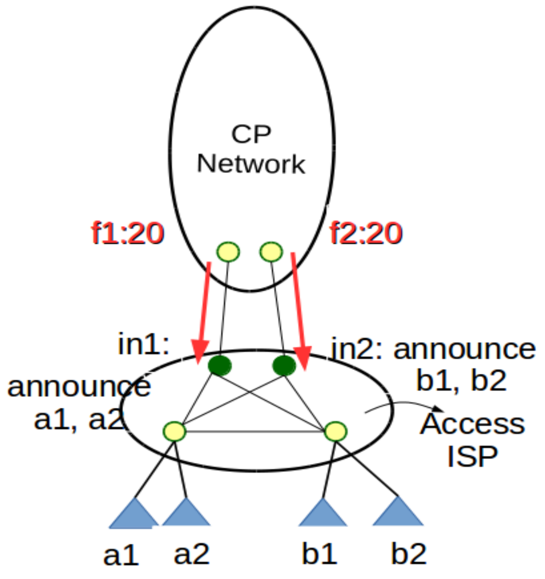
u1	10.0.0.1
u2	10.0.0.2
...	...
u8	10.0.0.8

ingress \egress	I1	I2
e1	600Mbps	...
e2	500Mbps	...
e3	950Mbps	...
e4	100Mbps	...

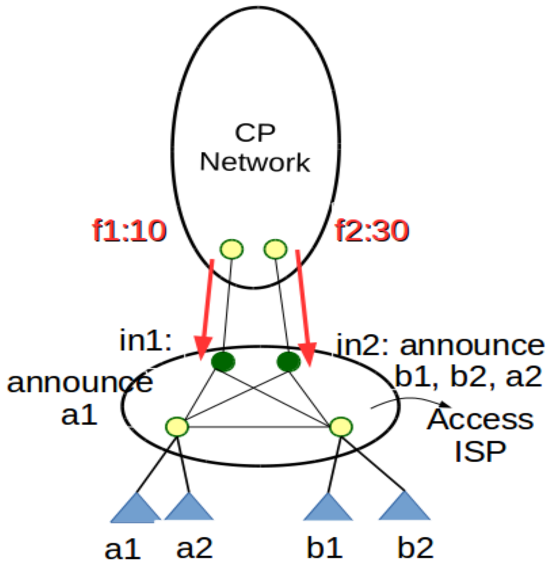
Figure 3.5: Traffic matrix determination in the Mapping module

ing programs from different neighbours of the ISP as well as the objective from ISP. Each program from each neighbour AS specifies its traffic demand as well as its routing preferences. Programs are combined in the *Program Consolidator*. The combined program is fed to the *Throughput Optimizer* which generates network configuration as well as hints to the neighbours of the ISP. This component is inspired by recent advancements in interconnect abstractions (e.g., SDX [25]). We leverage these advancements to allow the neighbours of an ISP to indicate to the ISP how they prefer their work to be routed. Similar to SDX virtual switch abstraction, our proposed approach allows for dynamic allocation of IPs within the virtual switch, allowing for capturing of the complex dynamic topology.

This architecture captures a generic Unison that can be programmed to perform a wide variety of functions, depending on the programs provided by the CPs. However, in this paper we focus on the case of maximizing ISP throughput where CP programs only provide demand as a function of the input and output ports of the vSwitch. We note that challenges in building components such as the Program Consolidator and the Updater have been addressed in SDX. In particular, SDX combines and joins policies from multiple participating ASes to program a virtual switch abstraction of an IXP. Then, it converts such combined program into BGP and OpenFlow rules. In the paper, we focus on the two components highlighted in Figure 3.4: the Mapper and the Throughput Optimizer.



(a) Before update



(b) After update

Figure 3.6: Influencing inbound traffic through hints.

3.3 Unison vSwitch Mapper

The function of the Mapper is to convert the complicated topology of the ISP into a vSwitch. In particular, the Mapper aims at identifying the input and output ports of the vSwitch. This is particularly challenging when taking scalability into account. In particular, a vSwitch defined by individual ports on individual routers in the ISP topology will lead to an intractably large vSwitch. To handle the TE problem that considers millions of egress flows destined to hundreds thousands of external IP prefixes, recent work [66] proposes a hierarchical framework for ISP network. The framework divides a global optimization problem into sub-problems, each of which is assigned to a child worker so the computation can be accelerated through parallelism. Our mapper provides an alternative for ISPs who are not capable or not willing to build such a hierarchy framework. To insure scalability, the ISP simplifies its network by representing it as a traffic matrix where each element is an aggregate flow. An aggregate flow is defined by an entry point to the ISP from a specific CP to a group of users. The entry point for an aggregate flow pair is easy to define, and is fixed. The function of the Mapper is mostly concerned with grouping users which are typically represented by an IP-prefix. We take a greedy approach, starting to de-aggregate flows from the entry points with a predefined value for the maximum number of flows we can handle. We consider boundary routers the same as the entry points so we have one aggregate flow for each entry point. Then we look at the routers that are directly connected with the boundary routers and consider them as new boundary routers. We keep doing this until the number of aggregated flows exceeds the threshold. To that end, we divide the ISP's network into "core" and an "edge" networks. Boundary Routers (*b-routers*) separate the ISP's core from the edge. Unison is concerned with routing CP data within the core network only, representing the core network by vSwitch. End users are aggregated such that data flow to and from the users is routed to a single b-router. The division of core from edge network is determined by the ISP. The closer the b-routers are to the users, the more effective Unison will be in controlling individual user performance but the larger the scale of the problem Unison solves.

Realization of the aggregate flow can be achieved through existing tunneling techniques, such as MPLS, GRE, and VPNs, or emerging SDN approaches based on flow space allocation [67]. An example of the mapping function is shown Figure 3.5. In the figure $I1$ and $I2$ are ISP ingress routers connected directly with the CP network. $e1$ to $e4$ are b-routers connecting the core and edge networks. Traffic demands from each ingress router to each user are shown in the left side of the figure. The aggregate flows are shown in one column of the traffic matrix resulting from the mapping process.

For Unison efficiency, the traffic matrix, which is the output of the mapper, has to be relatively fixed. This means that traffic from a specific CP to a group of users has to go through the same entry point. This is achieved by the Hint Generator which communicates to CPs to fix traffic going to a specific user IP prefix to a specific entry point. This feature is already supported by SDX for inbound traffic engineering. For ISPs that are not connected with SDX, we discuss other alternatives. Figure 3.6 demonstrates an example of such process, which announces nonoverlaping prefixes to different interconnection links, to inform CPs of the suggested inter-domain traffic metrics. Some configuration is required between CP and ISP (e.g., disabling route damping). Figure 3.6a shows the situation before the update takes place. There are four flows, each with a size of 10 units, destined to $a1$, $a2$, $b1$, $b2$ respectively. Ingress router in_1 announces IP address of $a1$ and $a2$, and ingress router in_2 announces IP address of $b1$ and $b2$. To shift traffic from the left peering link to the right peering link, the ISP could announce IP address $a2$ at ingress router in_2 instead of at in_1. Although this approach is easy to deploy, it reduces the network resilience and may lead to routing table explosion. An alternative approach is to use AS path prepending or MEDs. For the example shown in Figure 3.6, the ISP could announce IP address of a2 at both ingress router in_1 and in_2 but with a shorter AS path or a smaller MEDs value in the announcement from in_2. To generate the router-level BGP configuration from high-level BGP policies, ISPs may use a BGP synthesizer [68], which takes as input the routing policies and generates Quagga router configurations.

One possible concern of dynamically changing BGP entries is that unstable routes may cause unexpected interactions among multiple nodes in a large network [69, 70]. One possible solution is to use Root Cause Notification (RCN), shown in recent work [70] to effectively eliminate false suppression and undesirable timer interactions. Further, although our design expects the ISP to trigger the monitoring and optimizing periodically (e.g., every few minutes), the ISPs are not obliged to change the routing every time when the Optimizer module generates a new inter-domain routing. The ISP may change the inter-domain routing only when the new routing can significantly improve the throughput.

3.4 Unison Throughput Optimizer

Unison provides the neighbor ASes of an ISP with a virtual switch abstraction connecting the neighbour AS to customers of the ISP. This abstraction allows the ISP as well as its neighbours to program the virtual switch to implement different inter-domain applications. We focus on the application of maximizing the total throughput of the ISP (i.e., the number of bits per second delivered from the ISP entry points to the ISP customers). With arguments still raging aroung Net Neutrality in the US [71], we propose a framework that can be tuned to provide a neutral or biased ISP. In particular, we look at throughput optimization with weighted fairness constraints, assigning different weights in the lack of net neutrality regulations and equal weights otherwise.

The optimization problem that runs at the Optimizer is critical to the performance of the system. The Optimizer module takes as input elements a network topology (ingress routers, core network and b-routers as shown in the previous section), a traffic matrix that represents the demand for each CP from each ingress location to/from each of the b-routers, the pre-defined CPs' behavior (i.e., participating or non-participating) stated in the agreement between CPs and ISPs, and the link capacity constraints. These constraints are translated into decision variables for an optimization solver [72, 73] . Table 3.1 summarizes our notation.

3.4.1 Traffic Matrix

For the *intra-domain traffic matrix*, we define $intraTM_{i,i'}$ as the volume of traffic that enters the ISP network at ingress point i and exits at b-router point i'. We use this intra-domain traffic matrix for traffic from non-participating CPs. For participating CPs, an *inter-domain traffic matrix* is constructed by summing the intra-domain traffic matrices for each b-router point. For a network with two ingress points i and j, and two b-router points i' and j', given the intra-domain matrix $intraTM_{i,i'}$, $intraTM_{i,j'}$, $intraTM_{j,i'}$, $intraTM_{j,j'}$, the elements of the inter-domain matrices are constructed as $interTM_{k,i'} = intraTM_{i,i'} + intraTM_{j,i'}$ and $interTM_{k,j'} = intraTM_{i,j'} + intraTM_{j,j'}$.

Non-cooperating Neighbours: Unison can also handle cases when neighbouring ASes do not provide their traffic demand or accept the hints provided by the Hint Generator. In particular, the traffic matrix can be inferred through monitoring. Recent work [74] shows that it is possible to monitor network traffic for any prefix within an ISP network within

Table 3.1: List of notation

Variable	Description
$G(V, E)$	network with V routers and E links
c_e	capacity of edge e in E
I	a set of CPs
M	a set of aggregate flows
J	a set of paths
T	a set of aggregate flows that cannot receive a higher allocated bandwidth
$t_{i,m}$	allocated bandwidth to aggregate flow m for CP i in T
$b_{j,e}$	is edge e contained in path j; binary
$d_{i,m}$	bandwidth demand from CP i on aggregate flow m
$r_{i,m,j}$	bandwidth allocated to flows from CP i on aggregate flow m over path j
$r_{i,m}$	bandwidth allocated to flows from CP i on aggregate flow m
n_i	number of aggregate flows for CP i
$w_{i,m}$	weight of an aggregate flow m for CP i
w_i	weight of CP i
$CPSat_i$	satisfaction of CP i
b_l	lower bound of allocated bandwidth
b_h	upper bound of allocated bandwidth

milliseconds. The proposed approach does not require modification on current vendor hardware and is easy to deploy. Furthermore, to fix the aggregate flow pairs, the ISP can leverage the existing BGP mechanisms like selective announcements or prepending. We show the impact of non-cooperating neighbours on performance in Section 4.5.

3.4.2 Feasibility and Weighted Fairness

We model the ISP network (ingress routers, core network and b-routers) as a directed graph $G = (V, E)$, where V is the set of routers and E is the set of links that connect the routers. Assume there are CPs competing for resources, each CP requesting resources for n_i aggregate flows (e.g., in Figure 3.5 the CP is requesting resources for 8 aggregate flows). We define I as the set of CPs and M as the set of aggregate flows among all CPs. We use $d_{i,m}$ to represent the bandwidth demand from the i^{th} CP for aggregate flow m and use $r_{i,\,m}$ to express the rate allocated to aggregate flow m for the i^{th} CP. We use $r_{i,m,\,j}$ to express the rate allocated to flows from i^{th} CP on aggregate flow m over path j. The ISP takes as input the CP's bandwidth requests for aggregate flows (i.e., $d_{i,m}$) as well as the topology capacity, and generates allocations for each aggregate flow (i.e., $r_{i,m}$).

Feasibility: An allocation policy is feasible if no link capacity is exceeded. The upper limit of a feasible solution can be found by solving the following optimization:

$$\begin{aligned} \text{maximize} \quad & \sum_i \sum_m \sum_j r_{i,m,j} \\ \text{subject to} \quad & \sum_j r_{i,m,j} \leq d_{i,m}, \forall m \in M, \forall i \in I \\ & \sum_i \sum_m \sum_j r_{i,m,j} \times b_{j,e} \leq c_e, \forall e \in E \\ & b_{j,e} \in \{0,1\}, r_{i,m,j} \succeq 0, \forall j \in J \end{aligned} \tag{3.1}$$

$b_{j,e}$ is the binary variable on whether path j contains edge e and c_e is capacity of edge e. There is no fairness constraint on this optimization, so the result may assign high bandwidth to aggregate flows from to a few CPs and completely starve the others in an effort to maximize the total throughput.

Weighted Fairness In addition to being feasible, a bandwidth allocation policy should also be fair. Fair bandwidth allocation to flows has been extensively studied in the past [75]. Demirci et al. [76] studied how to extend these fairness definitions to multiple overlay networks instantiated on one substrate. Kleinberg et al. [77] take routing into consideration and prove the problem is NP-hard. In this section, we present a definition for fair allocation among multiple CPs.

We define a weighted fairness index (WFI) to evaluate the fairness of a bandwidth allocation policy in a multi-CP-setting. We define normalized weight of an aggregate flows as follows:

$$w_{i,m} = \frac{d_{i,m}}{\frac{\sum_i \sum_m d_{i,m}}{\sum_i n_i}} \tag{3.2}$$

Algorithm 1 WBA: Weighted Bandwidth Allocation

```
Input: Traffic metrics $d_{i,m}$, a set of paths $b_{j,e}$ in
Output: Allocated rate $t_{i,m}$ out
1: $w_{i,m} \leftarrow \frac{\sum_i n_i \times d_{i,m}}{\sum_i \sum_m d_{i,m}}$, $k \leftarrow \lceil \log_a [\frac{\max_{d_{i,m} \times w_{i,m}}}{u}] \rceil$
2: $T \leftarrow \emptyset$
3: for $n = 1 ... k$ do
4:     for $r_{i,m} \in BMCF(a^{n-1}u, a^n u)$ do
5:         if $(i, m) \notin T$ and $r_{i,m} \leq \min(d_{i,m}, a^n u \times w_{i,m})$ then
6:             $T \leftarrow T + (i, m)$, $t_{i,m} \leftarrow r_{i,m}$
   return $t_{i,m} : (i, m) \in T$
```

The weight of an aggregate flow is proportional to its demand and is normalized by the average aggregate flow demand for all CPs. This insures that bandwidth allocations are positively correlated with aggregate flow demands. As will be described in the next section, we use these weights to insure the weighted fairness of the allocation algorithm. The weight of a CP is defined as the sum of the CP aggregate flow weights: $w_i = \sum_m w_{i,m}$. We define CP satisfaction (*CPSat*) in the same way as the network satisfaction metric (*NetSat*) in [76] with *CPSat*$_i$ denoting the satisfaction of CP i. The *CPSat* describes how close the CP aggregate flow bandwidth allocation in the presence of other CPs is to the allocation it would receive had it been without competition. WFI is defined as the weighted standard deviation of the CP satisfaction metrics across all CPs sharing the resources of the ISP.

3.4.3 Bandwidth Allocation Algorithm

The brute force method to find the optimal routing is to (i) enumerate all paths between every ingress node and b-router node pair, and then (ii) apply max-min fair bandwidth allocation algorithm to all possible path selections to find the optimal selection that achieves the highest total rate. To make the computation faster, we limit the possible paths to k shortest paths instead of enumerating all paths between ingress and b-router node pair. To further reduce the computation time, the path generation process is performed offline. We expect valid paths to change infrequently.

The max-min fairness bandwidth allocation algorithm computes the allocation for each flow iteratively: maximizing the minimal flow rate, freezing the minimal flows and then repeating the steps for the second minimal flow. The computation quickly becomes infeasible as the number and size of a network grows. Inspired by SWAN approximate max-min fairness heuristic [78], we use the Weighted Bandwidth Allocation (WBA) algorithm shown in Algorithm 1. The algorithm achieves weighted fairness between CPs by solving an optimization problem which we call Bounded MCF (BMCF) in k steps. In every iteration, BMCF solves a multi-commodity problem (MCF) problem that aims at maximizing $\sum_i \sum_m \sum_j r_{i,m,j}$, which is similar with optimization problem (3.1). The difference is that BMCF tries to achieve weighted fairness among CPs, so in each iteration it puts a lower bound and upper bound on rate allocated to each aggregate flow:

$$b_l w_{i,m} \leq \sum_j r_{i,m,j} \leq \min(d_{i,m}, b_h w_{i,m}), \forall (i,m) \notin T \tag{3.3}$$

$b_l = a^{n-1}u$ and $b_h = a^n u$, which is passed by WBA in step n (line 4). Aggregate flows with lower demands have smaller weights, ending with fewer allocated rates. If an aggregate flow is allocated with its full demand or it cannot receive a higher allocation because of the link capacity constraints, the aggregate flow is frozen and is removed from the next round of computation. If every aggregate flow has the same demand, this allocation is identical to max-min fair allocation. Note that any changes in the traffic matrix require rerunning the optimization problems. This overhead can be mitigated by only recalculating routes for the affected parts of the network. We leave such enhancements for future work.

3.5 Evaluation

In this section, we focus our evaluation efforts on exploring how useful Unison can be under the limitations discussed in §3.2.1. We show that Unison can provide improved throughput and differential treatment between CPs while not harming the performance of any CPs, even with a limited number of cooperating CPs. We also evaluate the impact of various parameters and settings on the system's performance. To evaluate the performance of our design, we implement a proof-of-concept Optimizer that calls the CPLEX solver through its python API to solve the optimization problem described earlier. We conduct simulations to study the performance of our algorithm in realistic settings.

3.5.1 Experimental Setup

Topologies: We conduct experiments with a setting of one ISP and twenty CPs. Each CP is connected with the ISP in multiple interconnection nodes (i.e, ingress nodes) and the number of inter-domain links ranges from 1 to 5. Our simulation uses a variety of topologies from topology zoo [79] for the ISP and CP network.

Traffic demand: We assume that there is one flow from each CP source node to each ISP egress node. We consider all nodes excluding egress points in CP topology as source nodes and all nodes excluding ingress points in ISP topology as egress nodes. We simulate the traffic demand using a gravity model [80], which predicts that the traffic demand of a CP is proportional to the corresponding node population.

Link capacity: In our simulation, inter-domain link capacities are drawn from distribution of congested interconnections in recent work [4]. We generate the inter-domain link capacity by multiplying the traffic demand with the fraction of congestion shown in [4]. For the intra-domain link capacities, we assume that all links in the ISP have the same capacity and the link weights are assumed to be one. The value of this capacity is calculated through the following steps. First, we compute the routing with the default routing (i.e, OSPF) for the ISP network and identify the link with the most traffic demand. Then we compute a capacity by multiplying a congestion parameter with the demand carried in the most heavily loaded link. The goal of this approach is guarantee that a few links are congested. We also experimented with other link capacity distributions (i.e., uniform random distribution) and we observe that the results remain qualitatively similar.

Baseline: We use the early-exit policy for the default routing. The chosen interconnec-

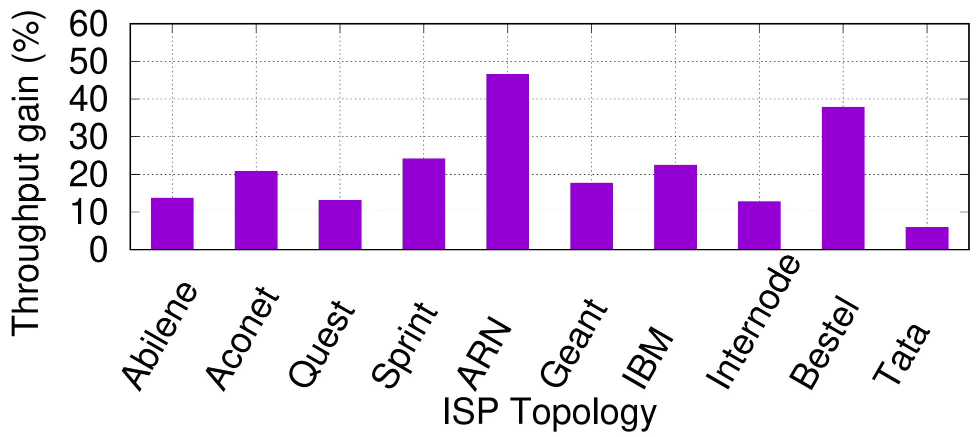


Figure 3.7: Throughput gain created by Unison compared to the baseline over different ISP topologies.

tion is the one that is the closest to the source. We assume that flows belong to the same aggregate flow will be routed in the same way and will not be split between multiple paths.

3.5.2 The Value of Unison

Value to ISP: We compare Unison to the baseline in terms of the amount of traffic they can deliver from CPs to end users. Figure 3.7 shows the throughput gain of Unison. We assume all CPs are participating, i.e., agree to use the new inter-domain routing as suggested by the ISP. It is clear that Unison's approach to jointly optimize inter-intra-domain routing improves ISP throughput. We also note that topologies with smaller average node degree improve more with Unison. Compared to complex topologies, simple topologies have less candidate paths between each ingress and egress node pair and it is more likely that a few links are heavily used by a large portion of paths. Therefore, changing the inter-domain routing is effective at diminishing unbalanced link usage.

Value to CPs: To better understand the value of Unison, we look at how increase in ISP throughput is viewed from the CP. Figure 3.8 shows the performance gain in percentage. We compare Unison to a baseline that attempts to optimize network utilization through optimizing OSPF parameters only (i.e., Optimizing intra-domain only). We observe that

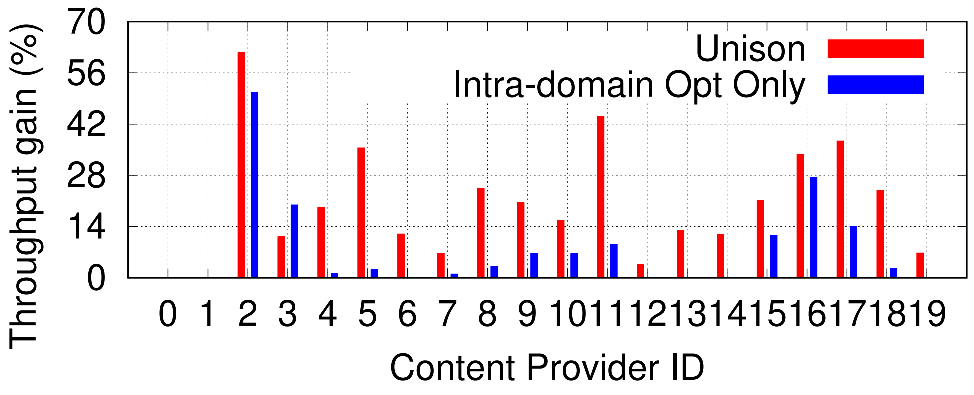
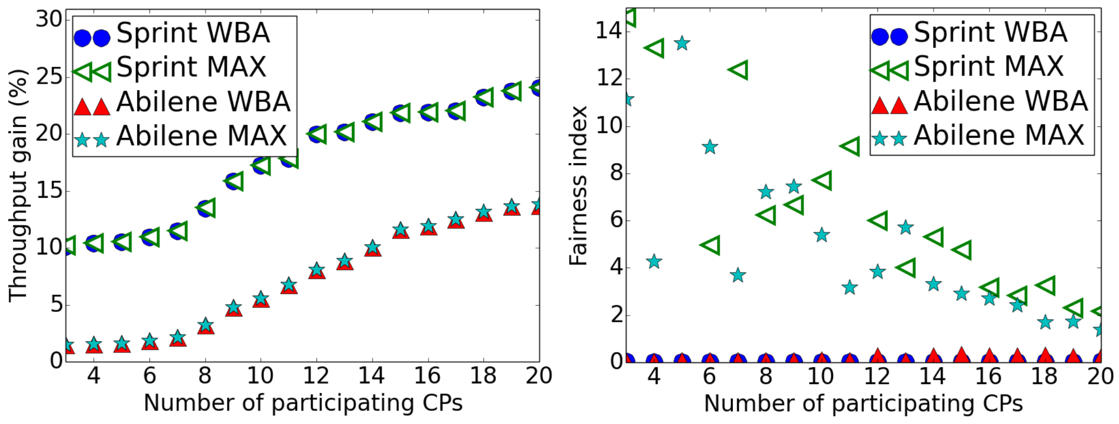


Figure 3.8: Throughput gains comparing optimizing intra-domain only and Unison.



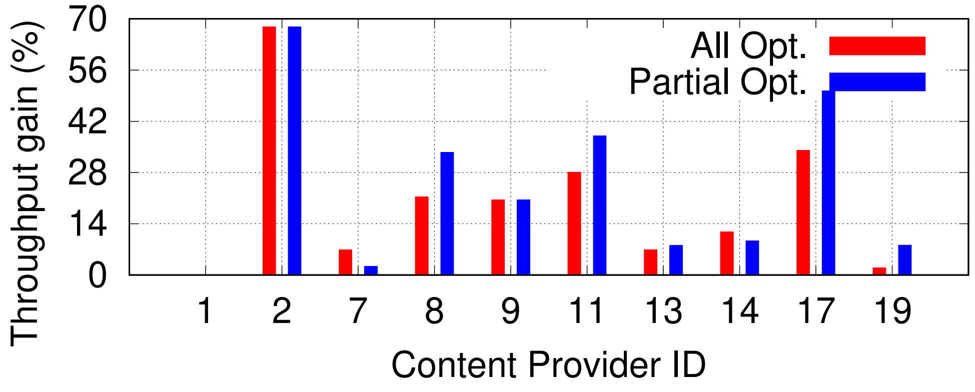
(a) Throughput gain

(b) Weighted Fairness Index

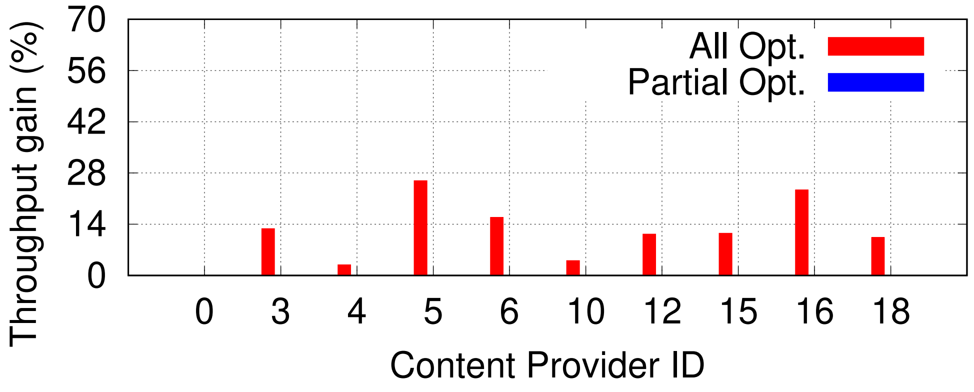
Figure 3.9: Comparison between WBA and MFC algorithm showing minor throughput impact with weighted fairness.

most CPs achieve a much higher throughput gain when the ISP relies on Unison compared to only optimizing intra-domain routing. This shows the value in the joint optimization of inter- and intra-domain routing even from the perspective of CPs.

Value of jointly optimizing for weighted fairness and throughput: To show the value of our proposed algorithm WBA, we compare it to the multi-commodity flow (MCF) algorithm. We compare the two algorithms on Sprint and Abilene topology. Each ISP is connected to 20 CPs and we change the number of participating CPs from 3 to 20. Figures 3.9a and 3.9b show the total allocated bandwidth and the weighted fairness index (WFI)



(a) Participating CPs



(b) Non-participating CPs

Figure 3.10: OptAll v.s. only for OptPartial showing that non-participating CPs enjoy a free ride of increased throughput in OptAll as participating CPs while only participating CPs achieve increased throughput in OptPartial.

respectively. For both of the topologies, the WFI for the bandwidth allocation generated by the WBA algorithm is lower (better) than that of the MCF algorithm. The throughput gain for both algorithms almost match with MCF performing negligibly better in some cases. We also observe that the WFI achieved by the MCF algorithm shows a decreasing trend as the number of participating CPs increases. The main reason is that the MCF algorithm does not enforce any constraints on the bandwidth assigned to each aggregate flow and participating CPs have advantages over non-participating CPs by adjusting the inter-domain routing. As the number of participating CPs increases, the effect of favorable treatment on a few CPs starts to diminish.

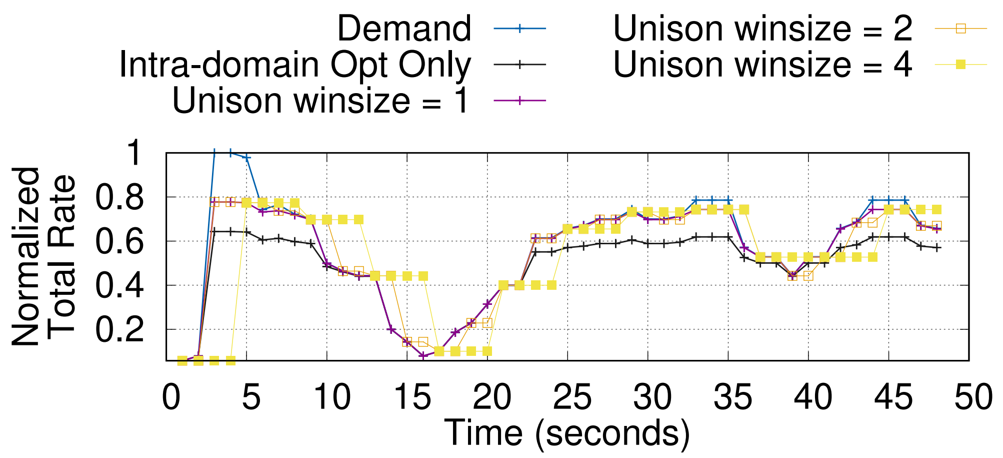


Figure 3.11: Impact of window size used in Unison on total ISP throughput under dynamic traffic demand.

3.5.3 Impact of CP participation

In the previous experiments, our algorithm optimizes the inter-domain routing only for participating CPs and optimizes the intra-domain traffic for both participating and non-participating CPs. We call this approach Optimizing for All (OptAll). An alternative approach is to not change any routing, including intra-domain routing and the allocated rate, for non-participating CPs while optimizing both intra- and inter-domain routing for participating CPs. We now compare the performance of optimizing for all (OptAll) against optimizing only for participating CPs (OptPartial). We conduct four sets of experiments with different participating CPs and non-participating CPs selection, but due to space limits we only show results (Figure 3.10) for the experiment with 10 participating CPs and 10 non-participating CPs. Our result shows that OptAll achieves a slightly higher ISP throughput gain than OptPartial does. Compared with OptAll, OptPartial achieves similar or higher throughput gain for participating CPs. Non-participating CPs enjoy a free ride of increased throughput in OptAll as participating CPs. This effect may decrease CP's motivation to participate if there are changes to have throughput gain without participating.

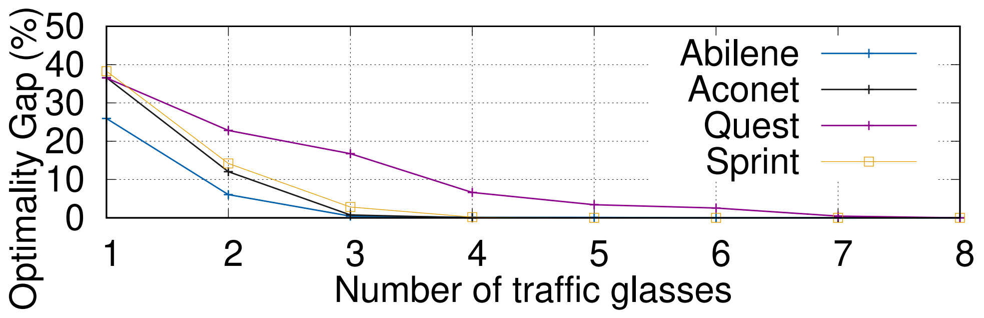


Figure 3.12: Optimality gap as function of aggregate flow.

3.5.4 Impact of environmental parameters

Impact of inaccurate prediction of dynamic traffic: In our design, the Estimation module collects the current traffic demand and uses it to estimate demand for a future window. This estimation is the main source of error. This error in demand prediction can cause under- or over-provisioning of bandwidth to some aggregate flows. The value of the error is a function of the estimation window size. To measure the impact of errors in estimating the traffic demand, we conduct an experiment with dynamic traffic. The traffic data is drawn from recent measurement study on YouTube network traffic at a campus network [81]. Figure 3.11 compares Unison with different window sizes to the baseline routing scheme. Unison adapts to changes when a small window size is used leading to better throughput than the baseline. When a large window size is used, Unison causes under-and over-provisioning frequently, which makes the default routing more preferable. Note that the window size depends on the frequency monitoring, mapping, optimization, and update can be run.

Impact of the traffic granularity: Traffic granularity refers to the level of traffic aggregation. In our baseline experiment, we consider aggregating all traffic entering at an ISP ingress router and routed to a b- router as an aggregate flow, and flows in the same aggregate flow are not splittable. We expect increasing the number aggregate flows per each ingress and b-router pair should increase the total rate until the highest rate has been achieved. Fig

3.12 shows the gap between the achieved total rate and the optimal allocation. Unsurprisingly, the computation time increases linearly as the number of aggregate flows increases. However, this is not a big concern. In most cases, we find that the throughput gain reaches its largest value with aggregate flows numbers as low as 5.

3.5.5 Summary

In this chapter we propose a framework to be deployed in an access ISP network for joint inter-intra-domain routing. We consider practical deployment issues and evaluate different design choices. We develop a resource allocation strategy that can be deployed by ISPs that maximizes the allocation to the CPs within the ISP capacity constraints while insuring fairness among CP allocations. Our evaluation shows that such framework is beneficial to both CPs and ISPs, improving total throughput of CPs within an ISP and improving ISP throughput. We also show that the benefits of Unison can be achieved even if only a subset of CPs connecting to an ISP agree to participate.

CHAPTER 4
A BACKPRESSURE MECHANISM FOR CONGESTED END HOST

In this chapter, we focus on the congestion problem at the queuing system within end hosts. For years, improved chips added more cores than more capacity per core [82]. Rather than relying on improved performance through increased per-core performance, parallel execution became the only way of making use of the new chips [83, 84]. 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 with requirements in the tens of thousands of connections per machine (e.g., reports mention servers handling up to 50k flows per end host [38]). This is further enabled by advancements in virtualization and containerization that allows applications belonging to different users to coexist and share network resources on the same end host (e.g., reports mention 120 VMs per end host [85]). This scale sparked interest in improved scheduling and prioritization between these applications through the introduction of efficient packet processing and scheduling mechanisms [38, 42, 86, 87, 85]. Processing of egress traffic in such stacks relies on holding packets in a cascade of queues pending their processing and eventual scheduling to be transmitted on the wire.

Packet queues at an end host serve as buffers between producers and consumers with different speeds. There are two types of buffers we are interested in: *source buffers* and *scheduled buffers*. *Source buffers* hold packets prepared by traffic sources while awaiting consumption by the underlying layer in the stack. *Scheduled buffers* consume packets from multiple traffic sources and then determine the order of their transmission according to their configured scheduling policy. While most components of the networking stack have evolved to cope with the growing scale of applications, handling of overflowing scheduled buffers which can lead to packet drops has received little attention.

Figure 4.1 shows how packets flow in a typical system. Packets are first sent from

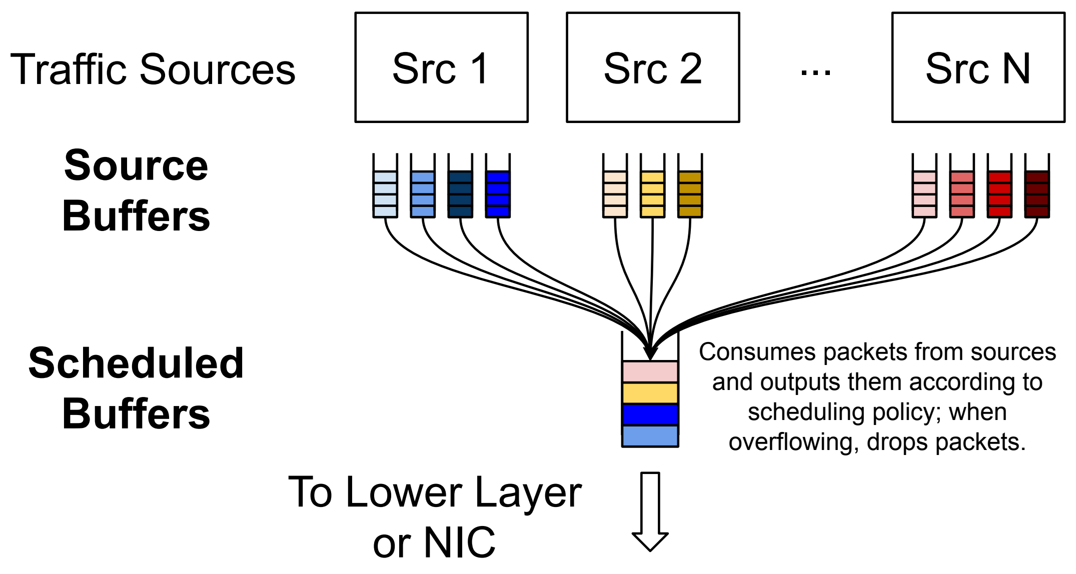

Traffic Sources
Src 1
Src 2
...
Src N
Source Buffers
Scheduled Buffers
Consumes packets from sources and outputs them according to scheduling policy; when overflowing, drops packets.
To Lower Layer or NIC

Figure 4.1: Schematic of queue architecture at end hosts.

source buffers (e.g., TCP socket buffers) to a scheduled buffer (e.g., Qdisc [88]). Packets from different sources accumulate in the scheduled buffer. If a scheduled buffer runs out of space, packets are dropped. Examples of such end-host congestion exist in large scale public clouds where a single end host is shared between multiple applications [47].

In general, packet drops are inefficient. Resources used on processing the dropped packet (e.g., CPU) have to be used again to send the retransmissions. Moreover, drops increase latency by adding more processing time to attempt retransmissions. Finally, drops can also induce severe reaction from congestion control which cuts its window in reaction to packet loss, leading to lowered throughput. When packet drops occur inside the network, then this type of inefficiency is unavoidable because of the need for end-to-end signaling. However, if packets are dropped inside the source host in the manner described in the scenario of Figure 4.1, then they can be handled through signaling within the host. For this latter type of loss we find that they are responsible for an up to 14% increase in CPU utilization and an order of magnitude increase in tail latency (§4.1). Our goal is to consider how signaling within the host can recover from these packet drops faster (in nanoseconds to microseconds as opposed to microseconds to milliseconds) while avoiding the CPU overhead.

In this chapter, we introduce the design, implementation, and evaluation of zD, a new architecture for handling congestion of scheduled buffers. zD has three components (§4.2): 1) a source buffer regulator that allows a congested scheduled buffer to pause and resume a traffic source, ii) a CPU efficient backpressure interface to define the interaction between the congested scheduled buffer and the traffic sources, and iii) a scheduler for paused flows to make sure that zD does not interfere with the scheduling policy implemented in the scheduled buffer. zD allows network operators to set a fixed queue size that is independent of the number of flows, eliminating bufferbloat issues at scale. zD maintains CPU efficiency by defining a backpressure interface that triggers packet dispatch from senders only when the scheduled buffer has room for new packets[1]. The task performed by zD can be viewed as controlling access to the scheduled buffer rather than leaving it the CPU scheduler. Thus, zD also reduces contention in accessing the scheduled buffer, further saving CPU resources. zD avoids interfering with the scheduling policy implemented in the packet queue (e.g., Qdisc policy) by scheduling flows in a way that is consistent with the underlying packet scheduling policy. To achieve CPU efficient scheduling, zD leverages recent developments in software schedulers introduced by the Eiffel system [42].

We implement zD[2] (§4.4) in the Linux kernel to handle backpressure for two cases: 1) when the queues and traffic sources are within the kernel stack (i.e., in the same virtual or physical machine), and 2) when the traffic sources are in the virtual machine and the queues are in the hypervisor. We find that zD can significantly improve network performance at high loads (§4.5). In particular, zD improves throughput by up to 60%, reduces retansmission by up to 1000x, and improves tail RTT by at least 10x at high loads. Furthermore, zD improves CPU utilization spent on the networking stack by up to 2x at the end host by reducing the effort spent on resending packets that have been dropped. We also find that

[1]Note that drops due to packet corruption can still happen.

[2]zD Code and a tutorial for using it are available at `https://zd-linux.github.io/`

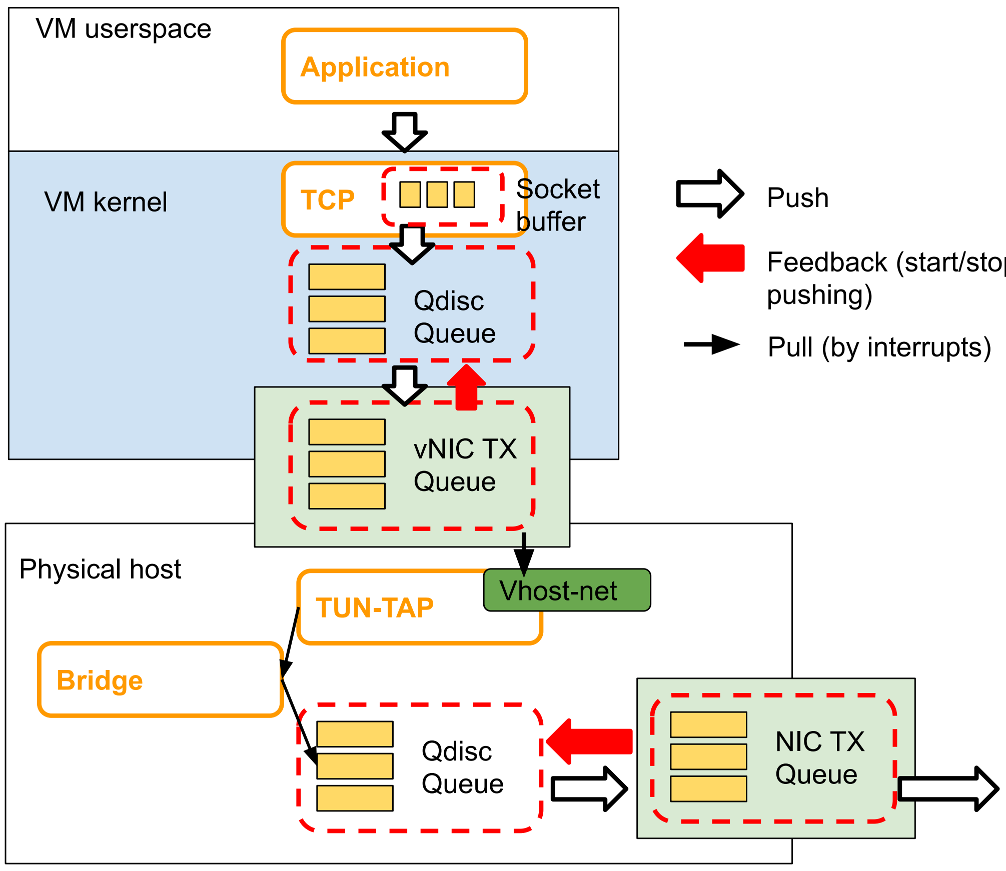


Figure 4.2: Architecture of queues in end hosts.

zD is lightweight as it does not incur extra overhead when the system is operating at low utilization. The only downside to zD is that in some scenarios it can increase the CPU overhead inside the hypervisor.

4.1 Background and Motivation

4.1.1 Packet Queuing at End Hosts

We start by giving an overview of the packet queuing architecture at end hosts. We focus on a common architecture used in modern data centers. In particular, we focus on the case where the end host is running a Linux virtualized environment, where the IO driver interface between the guest and host is handled by `virtio` [89] and `vhost` [90]. `virtio` is an I/O para-virtualized (PV) standard used for connecting the guest and host. To avoid context switching in the host, `vhost` allows the dataplane of the guest to be mapped directly into the kernel space of the host. The queuing architecture is shown in Figure 4.2. We focus on queues in the packet path and differentiate between queues where it is possible to have packet drops and those that already have a form of backpressure.

The user space application in the VM generates a packet and copies it into the kernel space socket buffer. The return value of the socket system call indicates whether the socket buffer is full. This operation is lossless (i.e., zero drop). Packets from the socket buffer are then queued into a Queuing Discipline (Qdisc). Packet drops can happen if the Qdisc is full. This happens when sockets push packets faster than the Qdisc transmission speed. Next, the Qdisc sends the packet to the vNIC TX queue. The vNIC TX queue does not drop packets. In particular, when there is no available space in the vNIC TX queue, the Qdisc will be paused until the queue length drops below the threshold, making the Qdisc the primary location for drops in the VM.

The hypervisor processes packets generated by the VM through `vhost` which starts a kernel thread that performs busy polling on the queue between the hypervisor and the VM. There can be multiple such queues, called `vring`s with a different `vhost` thread assigned for each `vring`. The `vhost` process polls the packet and sends it through the TAP device then to a Bridge device. Packets received in the virtual bridge will be forwarded to the Qdisc in the physical machine and then transmitted to the NIC TX queue. Note that in this setting, the TAP and Bridge devices do not hold or drop packets, delivering all packets they process in order to the Qdisc in the hypervisor. The Qdisc, or its counterpart in a more complicated architecture (e.g., OpenVSwitch [91]), is the main place where packets can be dropped due to congestion in the hypvervisor. We mark the existing backpressure mechanism (i.e., zero drop) with solid red arrows in Figure 4.2.

We choose this setting because it is a stripped-down, yet general-purpose, virtualized network stack. This architecture shares the same queuing components with more complicated architectures. For instance, consider Andromeda [47], Google's virtual network stack. Andromeda relies on a similar basic architecture and augments it with an efficient fast path. Note that packet drops can only happen at Andromeda itself which corresponds to the Qdisc in the above architecture. Furthermore, the architecture we consider here, unlike DPDK-based stacks, does not require a spinning core dedicated for network processing. This allows us to perform fine grain measurements of CPU efficiency (e.g., experiments where the VM runs on a single core). This architecture also captures the major characteristics of other stacks in terms of potential for packet drops at the end host. For instance, `vhost` used in our architecture has an analogous `vhost-user` used in DPDK-based stacks where packet queues will be in the userspace network processing system. In cases where OpenVSwitch [91] is used, the TAP and Bridge devices are replaced by OpenVSwitch. Hence, we find that the conceptual building blocks we develop in this paper for solving the congestion problem apply to other settings.

Ingress traffic: Most packet drops of egress traffic can be handled by coordination within the sender. However, drops of ingress traffic can require end-to-end coordination [92, 93] or careful allocation of CPU resources [45]. We focus on packet drops that occur due to congestion that can be handled through signaling within the end host, which are mostly egress traffic packet drops.

4.1.2 Types of Packet Drops

In-network packet drops are easily defined as packets being discarded by a network element (e.g., switch). This singular definition typically has some well defined reaction from the source associated with it (e.g., retransmission of the lost packet and congestion control reacting by adjusting its window). However, at end hosts we find that there are two types of packet drops. Both types of drops are expensive because a packet is processed for transmission, destroyed, and a replacement packet has to be generated which leads to higher CPU cost as well as higher latency. However, the two types differ in the reaction of the traffic source.

Virtual Packet Drops: In such cases the traffic source is aware that the packet was dropped at the end host. This type of drop is only feasible when transmission through the stack is performed through a series of nested function calls. The return value of these functions indicates whether the packet was successfully transmitted or dropped by one of these functions. If a packet is virtually dropped, the caller becomes aware of the location of the drop, allowing it to react appropriately. For instance, the reaction of TCP to a detected virtual packet drop is to simply attempt to resend the dropped packet without triggering its retransmission mechanisms and the congestion control algorithms.

Physical Packet Drops: In such cases the traffic source is unaware that the drop happened at the end host and consequently reacts as if the packet was dropped in the network. For example, the reaction of TCP to a physical packet drop will include triggering retransmission and congestion control algorithms. This type of drop is more expensive as it can lead to reduced network utilization, due to congestion control reaction (i.e., forcing flows to operate at a low rate), in addition to the higher CPU cost and latency.

In the stack described in Figure 4.2, virtual packet drops happen inside the VM where the TCP stack is aware of Qdisc packet drops. In current implementations, TCP reacts to virtual packet drops by immediately attempting to resend the dropped packet without consideration to contention at the Qdisc. This is a CPU intensive approach as we discuss in the next section. Physical packet drops occur in the hypervisor Qdisc which does not explicitly report drops to the guest kernel. Another important distinction between the two types of drops is that physical packet drops can be completely avoided. However, virtual packet drops are necessary in some cases. For example, a new flow cannot know whether the queue is full or not until it probes the queue with a packet that can be virtually dropped. Hence, the goal of a backpressure mechanism is to minimize virtual packet drops and eliminate physical packet drops.

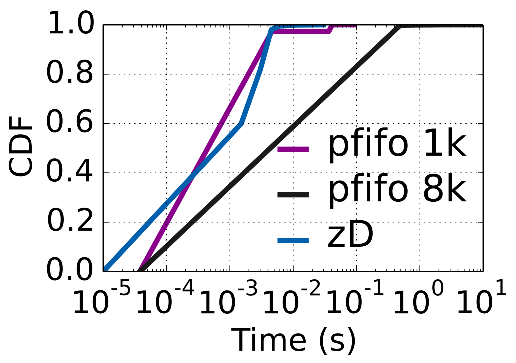
4.1.3 Cost of Long Queues

A naive approach to avoid loss in queues is to increase the queue size. Increasing the queue size exhibits fundamental limitation in accommodating the increasing number of concurrent connections, despite TCP Small Queue which attempts to combat bufferbloat [14]. To highlight these limitations, we conduct a simple experiment within a VM, running a large number of TCP connections using different lengths for the queue used in the VM Qdisc. In particular, we use `neper` [94] to generate 4000 TCP flows. The flows run in a VM. Queue accumulation only happens in the guest by setting a large rate for the VM in a queue not contended by any other VMs. We use the `pfifo` Qdisc [95] in the guest kernel with different queue lengths, aiming at examining behavior in two cases: 1) TSQ operation point where no packets are dropped (i.e., 2 packets are enqueued per flow), leading to a queue length of 8k slots, and 2) a queue length of 1k slots representing queue sizes that avoid bufferbloat. We also compare using the two cases to zD to highlight potential improvements. More details about our experimental setup is presented in Section 4.5.1.

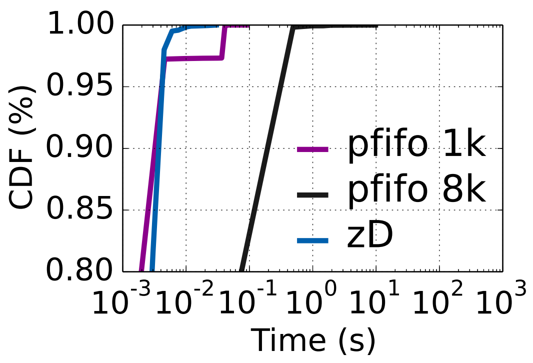
We find that longer queue length leads to longer RTT, implying that relying on TSQ leads to performance degradation as the number of flows grows. Figure 4.3a compares the RTT of TCP flows with a pfifo queue with two queue length values. The result shows that excess buffering in a long queue increases latency as well as causing packet delay variation (long tail in Figure 4.3b). We also repeat the experiment with Fair Queue (FQ) Qdisc [96] and observes that FQ has similar RTT as pfifo for a queue size of 8k slots. This behavior occurs despite FQ attempting to reduce the variance in RTT using round robin scheduling of active flows.

4.1.4 Cost of Packet Drops

We characterize the cost of packet drops in terms of both CPU utilization as well as tail latency. We find that both metrics are interconnected with a negative feedback loop where high CPU cost leads to high tail latency, which in turns increases the CPU cost further.



(a) CDF of RTT

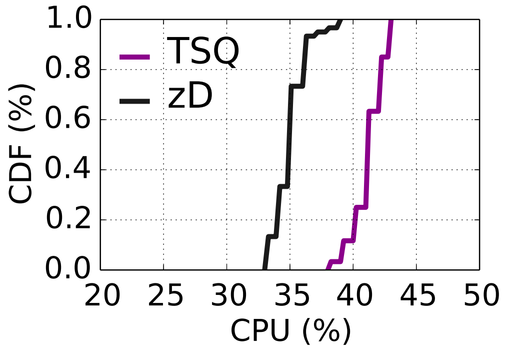


(b) Zoomed in CDF of RTT

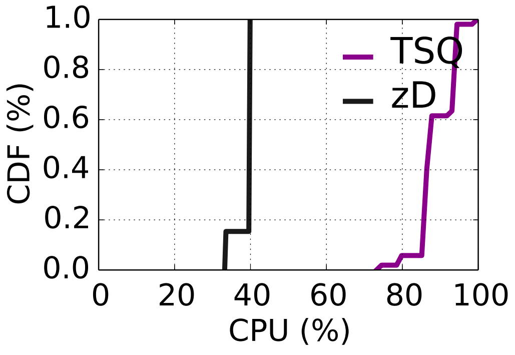
Figure 4.3: Bufferbloat, when pfifo queue size is 8k slots, leads to two orders of magnitude degradation in RTT. High contention and virtual packet drop rates, when pfifio queue size is 1k slots, leads to an order of magnitude degradation in tail latency compared to zD.

In this section, we explain in detail the causes of this peculiar behavior. We examine packet queues in the same setting as the previous section (i.e., flows started inside a VM). This allows us to examine CPU cost inside the stacks of both guest and host kernels. To illustrate these costs, we contrast the performance of the standard Linux implementation to our proposed system zD, which does not suffer from the same issues. We use zD simply to illustrate the inefficiency of the current approach used in the Linux kernel, explaining its details in subsequent sections.

CPU Cost: The CPU cost of packet queuing in the guest kernel is caused by the contention between TCP flows competing to acquire Qdisc lock and fill its limited space. This CPU overhead is a well documented issue [38, 41]. This overhead is exacerbated in cases where virtual or physical packet drops occur. In particular, a flow competes to acquire a lock to the Qdisc only to have it dropped, forcing the flow to try to acquire the lock again for the same packet. This overhead is shown in Figure 4.4a. The CPU cost in the host kernel is similar to that of the guest kernel in terms of contention to acquire Qdisc lock between multiple VMs. Furthermore, the hypervisor runs a `vhost` thread per `vring` to process traffic generated by the VM. The CPU utilization of `vhost-net` threads grows as the number of packets generated by a VM grows. In our experiments, we have a sin-



(a) CPU usage in VM



(b) CPU usage of vhost

Figure 4.4: zD reduces CPU usage in both VM and the physical machine compared to standard kernel implementation for TSQ (pfifo).

gle `vring` per VM. We find that avoiding packet drops and contention also reduces CPU cost of the `vhost-net` thread (Figure 4.4b), recorded by pinning the thread to a specific core and measuring the utilization of that core. In order to explain this behavior, we first examine the cost of packet drops on tail latency.

Latency Cost: Packet drops, in addition to time wasted on lock contention, cause delays to packet transmission. In particular, a packet has to successfully acquire the lock to the queue, and find room in the queue, in order to be transmitted. Otherwise, the packet is dropped, either physically or virtually, and forced to reattempt the process. This is clear in comparing zD, which avoids the mentioned overhead, and standard kernel implementation with 1k slots, shown in Figure 4.3. In particular, the impact of bufferbloat explains the behavior of the case when a queue size of 8k is used. However, the improvements in tail latency provided by zD compared to standard kernel implementation with 1k slots are explained by reducing contention as well as avoiding packet drops.

Impact of RTT tail performance on `vhost-net` CPU: This strange interaction is an artifact of years of optimization of the TCP stack yielding unexpected scenarios. These optimizations are summarized in [97]. We note that all optimizations we mention here are enabled by default in the Linux kernel stack. They start with TCP Segmentation Offload (TSO), a mechanism to achieve low CPU utilization at high networking speed by offload-

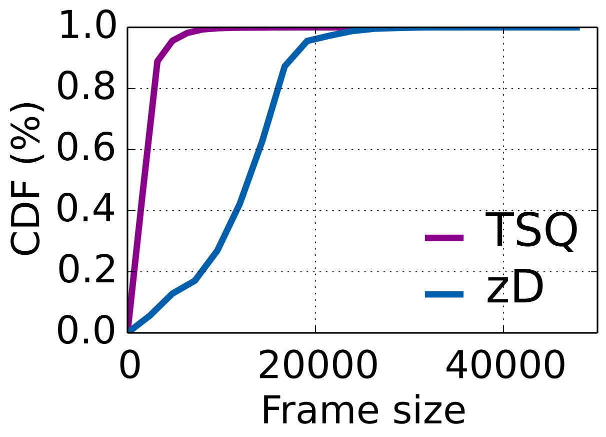


Figure 4.5: CDF of frame size, showing the impact of tail RTT performance on the behavior of TSO autosizing algorithm. Larger tail latency yields smaller packets, causing higher CPU cost.

ing TCP segmentation to hardware. However, TSO, with fixed segment size, may lead to microbursts for flows with low rate, which is not desirable in networks relying on merchant silicon switches with short buffers. Here lies a tradeoff between CPU and network performance; relying on large fixed segment size saves host CPU but results in a bursty network and using small segment sizes increases CPU cost, through processing of more packets, but yields better network performance. The current approach used in Linux attempts a compromise by automatically determining the size of TSO segments based on the transmission rate.

TSO autosizing was introduced to decide the size of data in a burst [96]. The goal of TSO autosizing is regulating the number of packets transmitted by any single TCP flow by changing the TSO size, and consequently reducing the burst size of the TCP flow. In particular, TSO autosizing aims at making TCP flows send a packet every millisecond rather than a hundred packets every 100 milliseconds. The algorithm calculating TSO size relies on an estimate of the rate of the flow calculated as $2 \times cwnd/RTT$, where $cwnd$ is the congestion window size and RTT is a moving average of the measured RTT value in the kernel. This means that a long tailed RTT distribution leads to a smaller pacing rate, which means the data will be chunked into smaller sizes. This leads to higher CPU cost at the `vhost-net` thread, as shown in Figure 4.4b. A CDF of packet sizes under standard kernel implementation and zD is shown in Figure 4.5, where the difference between packet sizes can be explained by the difference in tail RTT shown in Figure 4.3. Note that when the CPU utilization of the core handling the `vhost-net` reaches 100%, the latency faced by packets can increase, further impacting packet sizes, leading to a negative feedback loop of bad performance.

4.2 zD Design Principles

Packet drops are caused by demand exceeding capacity. This means that traffic sources will get less bandwidth than their demand. The only solution to this problem is to change capacity or demand. However, congestion control aims at optimizing reaction to such scenarios. Hence, the overarching goal of zD is to change the indicator of congestion at end hosts from packet drops, and to consequently achieve less throughput than demand, lowering throughput without drops. This avoids sending an ambiguous signal that does not differentiate between end-host drops and in-network drops. It also allows for better CPU and network performance as discussed earlier. This high level goal has to be achieved in tandem with the following objectives:

- **Prevent drops due to scheduled buffer overflows:** This is the main objective of zD. As discussed in the previous section, packet drops lead to poor network and CPU performance. zD allows overflowing queues to apply backpressure to traffic sources to prevent them from enqueuing more packets.
- **Maintain CPU efficiency:** Preventing drops can lead to cases where the traffic sources are constantly busy polling on available slots in the queue. This behavior trades CPU efficiency for network efficiency. zD avoids this type of behavior.
- **Maintain consistency with packet scheduling policies:** Backpressure is a form of controlling access to a congested scheduled buffer. zD should avoid scenarios where its

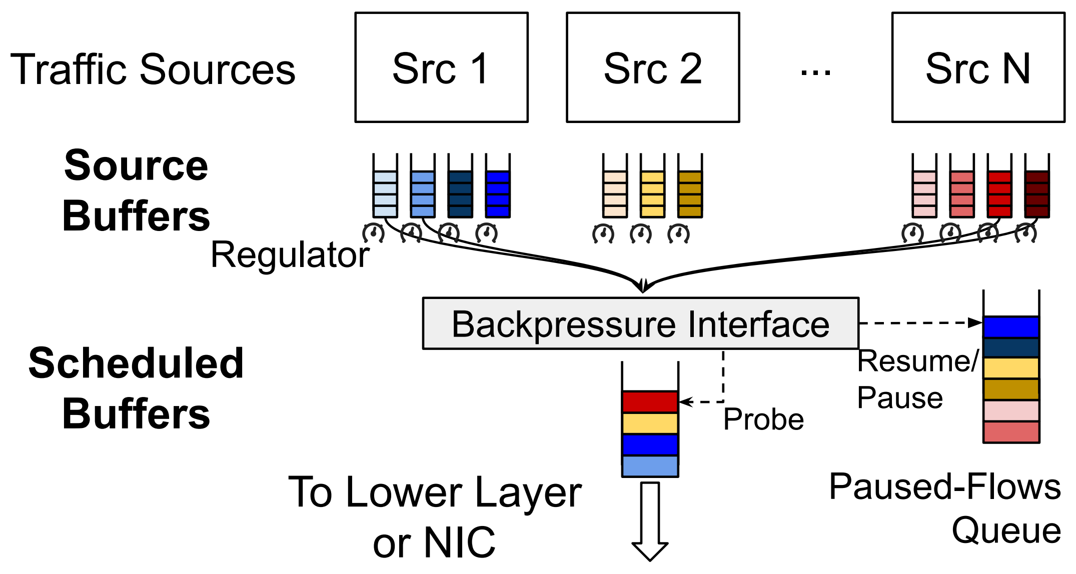


Figure 4.6: Schematic of zD architecture at end hosts.

coordination of access to the queue conflicts with the scheduling algorithm performed by the queue itself. An example of such conflict is an overflowing queue that has room for low priority traffic and no room for high priority traffic. zD ensures that only high priority packets get enqueued by applying backpressure to flows in a way corresponding to the scheduling algorithm of the queue which can be configured when the scheduled buffer is configured.

We find that these objectives can be achieved through a structuring of the queuing architecture at end hosts that implements the following mechanisms (Figure 4.6):

1. Source Buffer Regulator (§4.3.1): The source buffer should keep a copy of packets still being processed by the networking stack until it is fully transmitted. The source buffer should also support an interface that allows the underlying stack to pause and resume transmission of packets from that buffer (e.g., TSQ). This augmentation of source buffers allows for avoiding physical packet drops by always retaining a copy of dispatched packets till they are actually consumed. It also provides an interface for the backpressure mechanism to pause and resume packet dispatch.

2. Backpressure Interface (§4.3.2): To eliminate physical drops, packet queues should be able to pause senders when they are full. Furthermore, senders should be able to probe for room in the queues. Such interaction between the packet queues and senders should be well defined through a backpressure interface. Furthermore, it should be CPU efficient, to avoid CPU being the bottleneck of the networking stack.

3. Paused-Flows Queue (§4.3.3): Backpressure should be applied in a way that does not change the intended scheduling behavior of the packet queue. Hence, zD schedules access to the packet queue by keeping paused flows in a queue that is sorted in a way consistent with that of the underlying packet scheduling policy.

The design of zD and TSQ share the regulator. Both zD and TSQ employ a mechanism that pauses and resumes source buffers. The difference between zD and TSQ lies in how the pause and resume decisions are made. In the case of TSQ, pause and resume decisions are made by the source buffer. TSQ forces source buffers to maintain a maximum of two dispatched packets per flow, leading to queue occupancy that grows as the number of active flows grow. This limits the effectiveness of TSQ backpressure in handling bufferbloat as it ignores the occupancy of the scheduled buffer. Furthermore, access to the scheduled buffer becomes dependent on CPU scheduling of sender buffers and their ability to gain the lock to the scheduled buffer. In the previous section, we show that these limitations in TSQ lead to significant performance degradation. zD mitigates these problems by extending the regulator as well as providing a Backpressure Interface and a Paused-Flows Queue.

4.3 zD Overview

zD applies backpressure from packet queues, which can overflow and drop packets, to source buffers from which packets are dispatched. It provides a layer between the sender buffer and the scheduled buffer. Instead of continuously pushing packets into the scheduled buffer only to drop them when the queue is full, zD adds a set of additional steps in the path of a packet. First, a copy of the packet is created to avoid physical packet drops. Then, the packet copy is used to probe the packet queue to check if it has room, and proceeds normally if the packet queue has empty slots. However, if the packet queue has no empty slots, the packet copy is dropped, causing only a virtual packet drop. This is used as a backpressure signal to the source buffer of that packet. The backpressure signal pauses the backpressued flow and registers it with zD so that it can be resumed when there is room for its packets. Figure 4.6 summarizes modifications to the current queue architecture. The zD logic is summarized in Algorithm 2. For the rest of this section, we elaborate on each step described in this algorithm.

zD mechanisms can be applied to multiple settings where there are source buffers and scheduled buffers. In this paper, we focus on two such settings: 1) the TCP/IP kernel stack, where TCP buffers are the source buffers and Qdisc is the scheduled buffer, and 2) the hypervisor networking stack in the kernel, where the vrings of the VM are the source buffer and the Qdisc is the scheduled buffer. The details of our implementation of zD in these two settings are presented in Section 4.4.

Memory overhead: zD has no data-plane memory overhead except for the packets copies used to probe scheduled queue occupancy. Such packet copies are only copies of packet descriptors which are commonly used for different purposes in networking stacks. In our Linux implementation, we use one of the copies already created by the kernel's stack, incurring no exta memory overhead. Backpressure keeps data in the application buffer thus preventing the creation of new packets. The control plane overhead of zD is limited to the Paused-Flows Queue that keeps a per-flow descriptor. In our Linux implementation in a 64 bit machine, with 20k flows, the memory overhead is less than 160KB.

4.3.1 Source Buffer Regulator

This module has two functions: 1) define pause/resume operations, and 2) keep a copy of the dispatched packet until its transmission to the wire is confirmed. A flow can have two states "Active" and "Paused". The reaction of the stack to each state, and consequently the implementation of pause/resume functions depend on whether the stack is push-based or

Algorithm 2 zD Flow Algorithm

```
1: procedure ProcessFlow(Flow F, Packet p, Queue q)
2:     if $F.pause$ then return $//Regulator$
3:     if $!q.probe()$ then $//Backpressure\ Interface$
4:         $F.pause \leftarrow true$ $//Regulate$
5:         $PausedFlowsQ.append(F)$
6:     else
7:         if $!F.sendTwo()$ or $PausedFlowsQ.empty()$ then
8:             enqueue(F, p)
9:         else
10:            $F.pause \leftarrow true$
11:            $PausedFlowsQ.append(F)$
12: procedure ResumeFlow(Flow F)
13:     $F \leftarrow GQ.popFront()$
14:     $F.pause \leftarrow false$
15:     $F.resume()$
```

pull-based. In cases of a push-based stack (e.g., TCP/IP kernel stack), marking a flow as "Paused" implies that no further packets are pushed by that flow. New packets generated by the application are queued in the source buffer. Once the flow is resumed (i.e., marked as "Active"), packets residing in the source buffer are pushed to the lower layer. On the other hand, a pull-based stack already has to sleep when it has no packet to process. We follow a similar approach by forcing the pull-based stack to sleep when it has no active flows. Note that a busy-polling stack on a dedicated core (e.g., DPDK) does not need to sleep, making the implementation of these functions a simple marking operation.

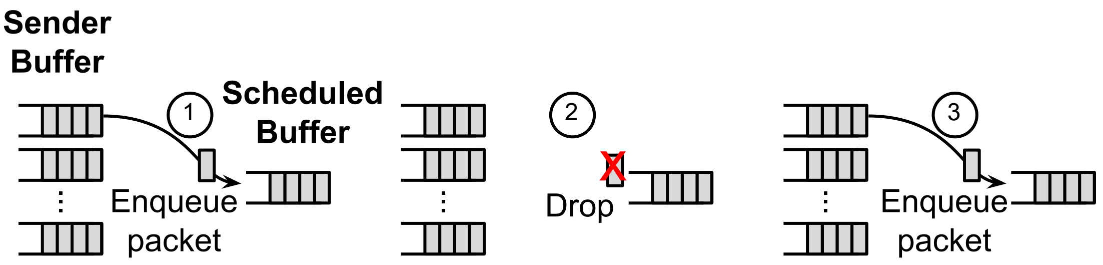
Like TSQ, zD keeps the number of packets enqueued by a single flow to a maximum of two packets. Limiting the number of packets per flow is necessary to avoid head of line blocking, where a single flow enqueues a large number of packets in the queue, slowing other flows. We found that further limiting to a single packet per flow causes performance degradation. In particular, there can be a delay between a flow becoming active and the processing of its packet. In the case of a push-based model, this delay is caused by the multi-threaded nature of a push-based stack, where marking a flow as "Active" does lead to the immediate dispatch of a packet by that flow. Typically, once a flow is marked as active a thread is started to kick-start packet dispatch for that flow. This approach has a processing delay associated with delaying the dispatch of packets. In the case of a pull-based stack, marking a flow as “Active” might happen during a sleep cycle. zD amortizes this delay over multiple packets by making sure that an active flow has two packets pushed to the scheduled buffer before it is paused again. Note that when a flow becomes active, it has to check the number of its packets still in the scheduled queue and make sure that it never exceeds two packets. We found that this approach, and specifically limiting the number of packets to only two, provides a good compromise between amortizing the cost of pause/resume operations and unfairness (i.e., less than two packets leads to under utilization and more than two pakcets leads to head of line blocking and unfairness).

Unlike TSQ, the sender buffer regulator can pause a flow that does not have less than two packets in the scheduled buffer. This is critical in order to decouple the queue length from the number of flows, avoiding bufferbloat scenarios in cases where there is a large number of flows.

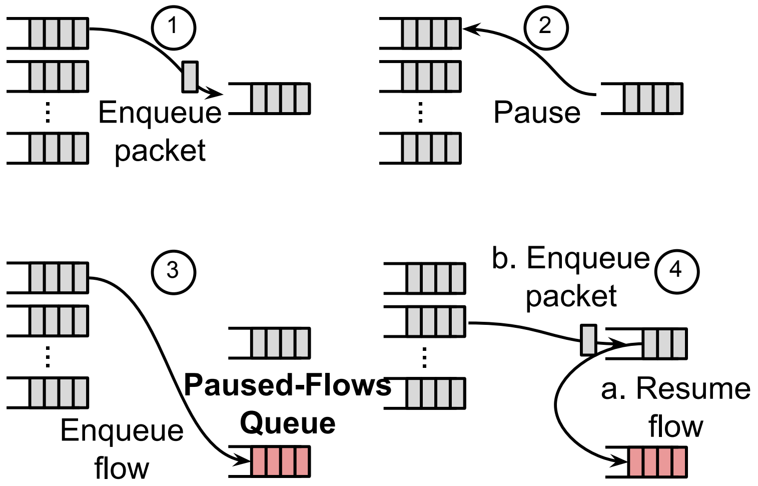
4.3.2 Backpressure Interface

This interface defines the interaction between source buffers and the scheduled buffer. In particular, it defines three operations: `probe`, `pause_flow`, and `resume_flow`. `probe` informs the sender buffer on whether it can push packets to the scheduled buffer. Scheduled buffers with different scheduling policy should have different implementation of `probe` function. For example, with the simplest First-in-first-out (FIFO) queue, the `probe` function returns false when the number of packets in the queue is equal to or larger than the queue capacity and returns true otherwise. For more complicated scheduling policies such as fair queue, the `probe` function needs to classify the flow first and then checks whether the flow exceeds its assigned share of the scheduled buffer.

If `probe` returns false, implying no room for that flow in the scheduled queue, `pause_flow` is invoked. `pause_flow` marks the flow as “Paused” triggering the logic of the sender buffer regulator. It also adds the flow to the Paused-flows queue. When a scheduled



(a) Steps of backpressure in TCP/IP Stack



(b) Steps of backpressure in zD

Figure 4.7: Illustration of different backpressure steps.

buffer has room (i.e., a packet is transmitted), `resume_flow` is invoked. `resume_flow` fetches the highest ranked flow in the Paused-flows queue. Then, mark it as "Active", trigger the resume logic of the sender buffer regulator. Note that this logic is deadlock-free.

The advantage of this interface is that a flow is only active if either the scheduled buffer has room for the packet or it is the first attempt of that flow to access a congested scheduled buffer. This is unlike existing attempts where flows are always active causing either physical or virtual packet drops by continuously attempting to enqueue packets to the scheduled buffer. Hence, the backpressure interface improves both CPU and network performance by avoiding drops as well as only doing work when useful. The difference between the two approaches is summarized in Figure 4.7.

It should be noted that the granularity of the scheduled buffer decides the granularity of Backpressure Interface. For example, in our implementation of backpressure in the hypervisor, the backpressure is performed per VM because packets lose flow-level information when it passes through from the VM to the hypervisor. The Qdisc in the physical machine treats all traffic from a VM as an aggregate flow and `probe` API provides information at the granularity of VMs.

4.3.3 Paused-Flows Queue

The aforementioned building blocks rely on the ability of zD to track paused flows. This tracking function is performed by the Paused-flows queue. The paused-flows queue is a global queue accessible to all stack threads through a global lock. The order in which flows are sorted within this global queue determines the overall scheduling policy for traffic going through the stack. zD implements a library of Paused-Flows Queuing Disciplines that correspond to the queuing disciplines implemented in the scheduled queue. The network operator has to install a Paused-Flows Queuing Discipline that corresponds to their chosen queuing discipline in the scheduled queue. This operation can be simplified by a simple network utility application. We note that the focus of our work on zD is managing congestion due to queue overflow of packets. Hence, in this work we implement only a small library of Paused-Flows Queuing Disciplines (i.e., FIFO and rate limiting disciplines). Complex queuing disciplines can be implemented to extend the functionality of zD. Efficient implementation of such disciplines is critical to avoid congestion due to high CPU utilization. Such efficient implementation is feasible relying on building blocks proposed in our earlier work on efficient per-flow scheduling [42].

4.4 Implementation

We implement backpressure in two places: (1) the Linux TCP/IP stack, and (2) the vHost stack in the Linux hypervisor stack. Our implementation is based on Linux kernel 4.14.67, however it is not restricted to that specific version. While the zD design described in Section 4.3 can be generally applied to both cases, we focus on these two settings as discussed earlier.

4.4.1 TCP/IP Stack Implementation

Implementing zD requires modifying the way that the TCP stack interacts with the IP stack. We start by giving an overview of the transmission path (Tx path) of the standard TCP/IP stack implementation in the kernel. In the TCP Tx path, data from the userspace application is pushed into the Socket Buffer (`skb`) and all paths of function calls end up calling `tcp_write_xmit` function regardless of whether the TCP socket is sending a packet for the first time or is retransmitting a packet. In the `tcp_transmit_skb` function, each `skb` is cloned so that TCP can always keep a copy of the original data until the packet is ACKed by the receiver. The `tcp_transmit_skb` function calls `dev_xmit_skb`, which tries to queue the packet into the corresponding Qdisc (i.e., the scheduled buffer). If the Qdisc queue is full, the `skb` will be virtually dropped. In particular, the pointer to the next packet to send, `sk_send_head`, will not be advanced.

Under the standard kernel implementation, when an `skb` is virtually dropped, TCP will attempt to resend it immediately unless the socket is throttled by TSQ. TSQ reduces the number of TCP packets in the Tx path by limiting the amount of memory allocated to the socket, forcing `sk->sk_wmem_alloc` to not grow above a given limit. By default, if a socket already has two TSO packets in flight, the socket will be throttled until at least one of the packets is freed. Note that TSQ can be viewed as the sender buffer regulator. A socket paused by TSQ will be resumed by a callback function when a skb is free (i.e., when skb_free function is executed), with the assumption that if an skb is destroyed, an extra space in the queue is available. This approach means that when a slot in the queue is freed, its replacement is notified. This implies that the approach of reattempting to send a dropped skb immediately can only make congestion at the Qdisc worse. Our implementation is shown in Figure 4.8, where the yellow blocks show function calls we modified.

Probe: Before dev_xmit_skb function pushes the packet into the queue according to the queueing discipline, it checks whether the packet should be passed to the next scheduled buffer through our extended probe API. We implement probe for the three most basic scheduling algorithms: pfifo_fast as the default qdisc for Linux interfaces, classful multiqueue (mq) for multiqueue devices, and token bucket filter (TBF) as a traffic shaper.

Pause: If the probe returns false, instead of resuming the socket, we mark the socket as stopped and place a pointer of the socket into a global queue shared by all sockets. Access to the global list is serialized through a global lock.

Resume: After an skb is consumed by the driver, the global list dequeues a socket and marks the socket as nonstop. To ensure the socket is resumed immediately, we use a tasklet to schedule the retransmission operation as soon as the CPU allows. We use a tasklet as a per-CPU variable for performance considerations. As indicated earlier, the existing TSQ interface for handling flow pause and resume is not very helpful for zD. In particular, TSQ relies on the sk_wmem_alloc field of struct sock to make decisions on throttling the socket. However, our implementation keeps increasing the value sk_wmem_alloc until has_room returns true. Hence, TSQ cannot properly decide whether the flow should be throttled. Therefore, we disable TSQ and implement our flow activation algorithm discussed in the previous section.

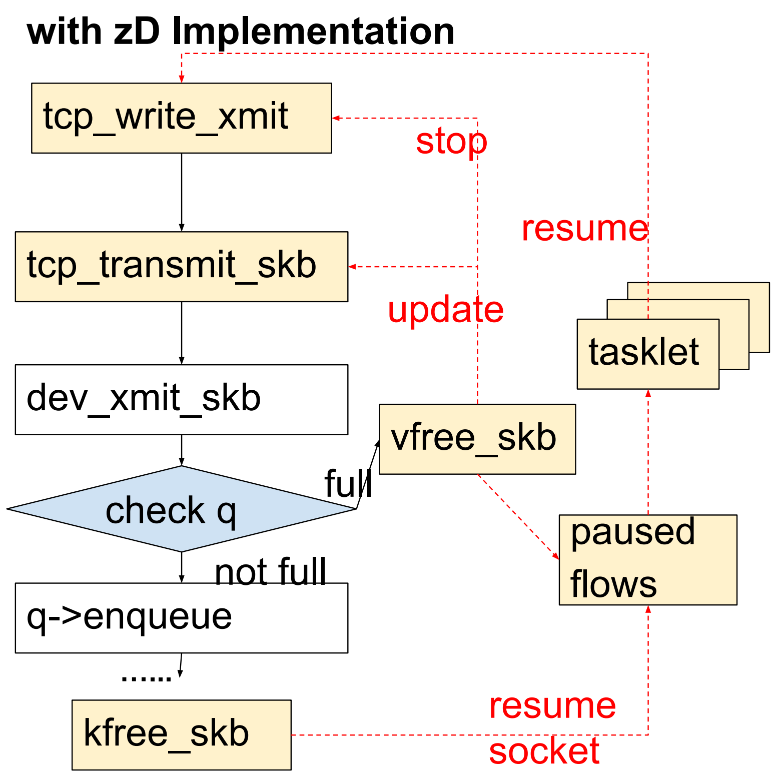
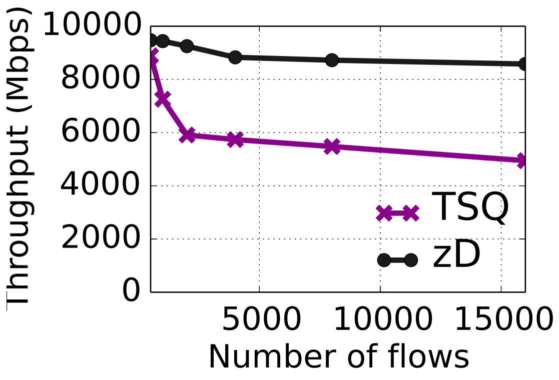


Figure 4.8: Flow chart describing TCP/IP stack with zD

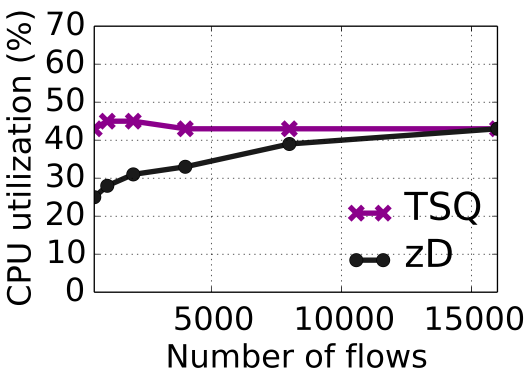
4.4.2 Hypervisor Implementation

We implement zD in the hypervisor based on the zero-copy `virtio` Tx path. Zero-copy transmit is effective in transmitting large packets between a guest VM to an external network without affecting throughput, consuming lower CPU and introducing less latency [98]. The `vring`, where `virtio` buffers packets, is a set of single-producer, single-consumer ring structures that share scatter-gather I/O between the physical machine and the guest VM. `vring` keeps track of two indexes: `upend_idx` and `done_idx`. The indexes represent the last used index for outstanding DMA zerocopy buffers in the `vring` and the first used index for DMA done zerocopy buffers, respectively. The `vhost` thread pulls packets from the `vring` and attempts to enqueue them to the Qdisc.

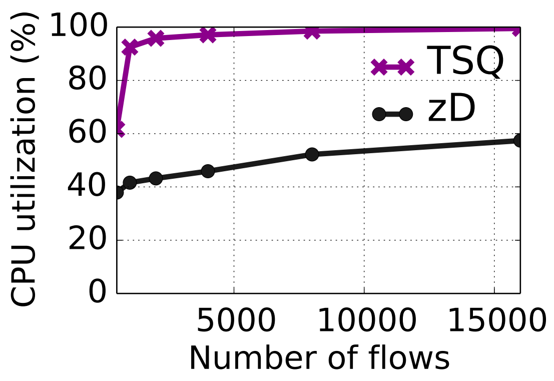
When a process transmits data, the kernel must format the data into kernel space buffers.



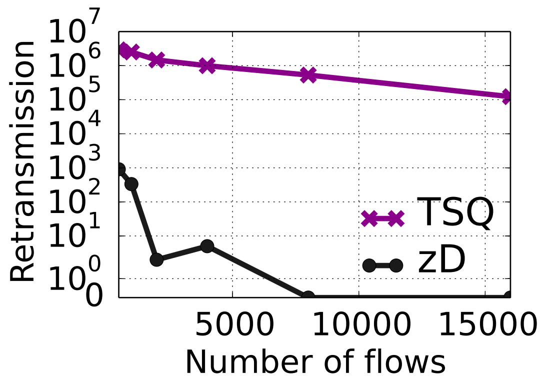
(a) Throughput



(b) VM CPU



(c) vHost CPU



(d) Retransmission

Figure 4.9: 10Gbps network speed with a qdisc of 100 slots in the hypervisor

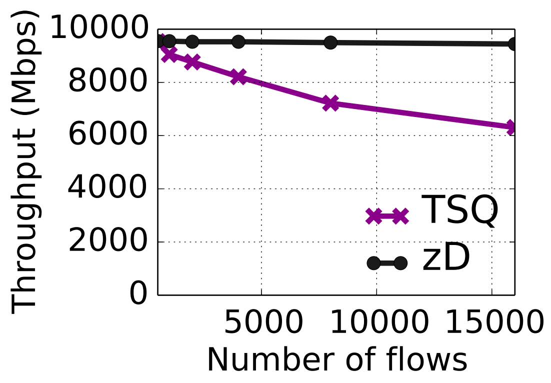
Zero-copy mode allows the physical driver to get the external buffer to directly access memory from the guest `virtio-net` driver, hence reducing the number of data copies that require CPU involvement. In the hypervisor, the `vhost` process passes the userspace buffers to the kernel stack `skb` by pinning the guest VM user space and allowing direct memory access (DMA) for the physical NIC driver. The path of the `skb` in the hypervisor is shown in Figure 4.2. The Tap socket associated with the `vhost` process sends out the packet through the Tap device. Packets are then received by the virtual bridge, and the packet is passed to Qdisc. Finally, the packet is consumed by the physical NIC. Note that when `vhost` pulls a packet from the `vring`, once the packet is processed, the `kfree_skb` callback function will inform the `vring` to destroy the packet, whether it was actually transmitted or dropped by Qdisc.

The **Probe** and **Resume** steps are implemented in this setting in a very similar fashion to that of the TCP/IP stack. Implementation of **Pause** requires handling some corner cases. In particular, if the Qdisc is full, instead of calling the `kfree_skb` function to free the packet and mark the DMA as done, we mark the corresponding VM as paused, stops polling from its `vring`. This step also requires moving the `upend_idx` back to point to the position of the dropped packet. A significant difference between the hypervisor setting and the TCP/IP stack setting is the potential existence of further packets in-flight from the VM that have been pulled from the `vring` before the VM was marked as paused. The situation is further complicated as those packets can reach the Qdisc and find that it now has room. This behavior can lead to introduction of out-of-order packet delivery which can lead to TCP performance degradation. Hence, all in-flight packets between the `vring` and the Qdisc are dropped to avoid such scenarios. Note that moving the `upend_idx` makes sure that those packets are retransmitted later. We implement a callback function to resume polling from the `vring` when a packet is passed to the physical NIC driver.

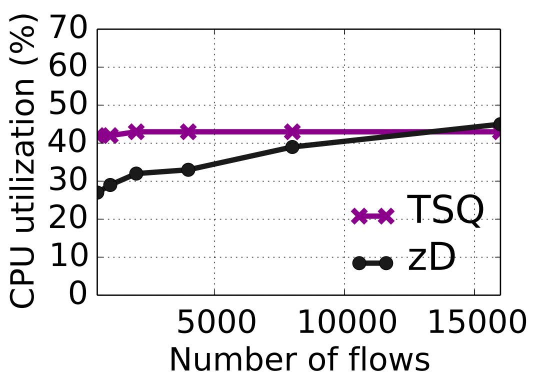
4.5 Evaluation

4.5.1 Experiments Setup

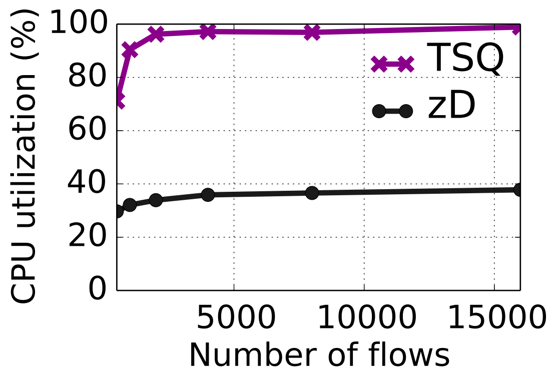
We conduct experiments between two Intel Xeon CPU E5-1620 machines, connected with a 10Gbps link. Both machines have four cores, with CPU frequency fixed to 3.6GHz. We generate traffic with neper [94], a network performance measurement tool that can generate thousands of TCP flows. The TCP flows are generated inside a virtual machine and are sent to a remote machine. We use Qemu with KVM enabled as the hypervisor. For a baseline, both VM and physical machines run Alpine Linux with kernel version 4.14.67. We run a modified version of that kernel with zD implementation. In our experiments, we ran into a known issue of `vhost` where the Rx path of a VM becomes bottlenecked on the Tx path, because both are handled with the same thread [99]. The issue is inherent in the current implementation of virtualization in the Linux kernel, affecting baselines and zD.



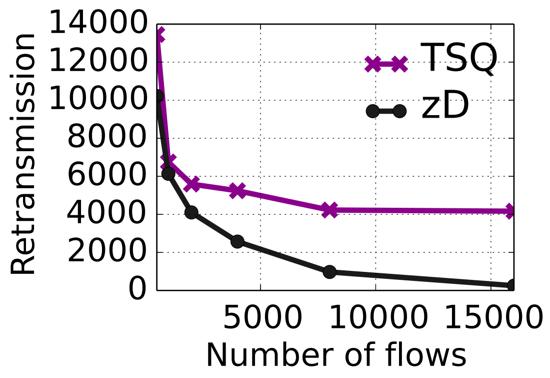
(a) Throughput



(b) VM CPU



(c) vHost CPU



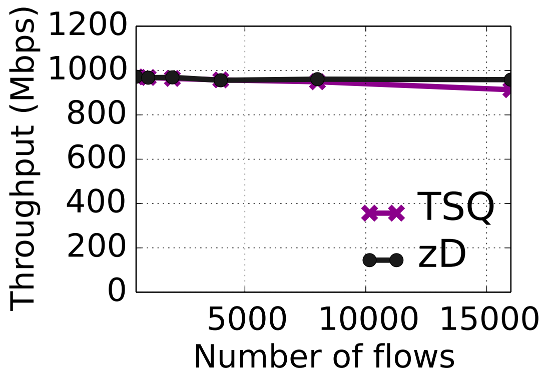
(d) Retransmission

Figure 4.10: 10Gbps network speed with a qdisc of 1000 slots in the hypervisor

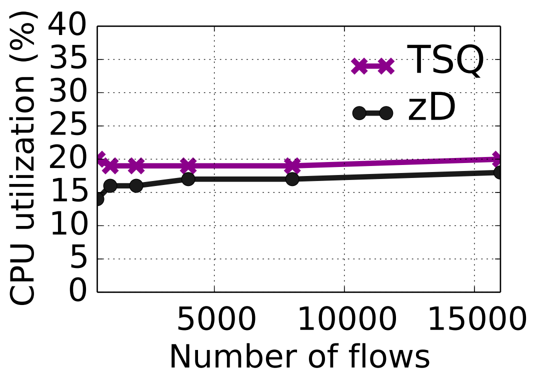
The bottleneck is resolved by allocating more CPU to the receiving path or improving the receive path architecture [45, 46]. Hence, we perform our experiments in two settings, one with 6 vCPUs assigned to the virtual machine (experimenting with a bottleneck-free end host) and another with 1 vCPU assigned to the virtual machine (exposing the Rx path bottleneck to evaluate zD under a resource constrained end host). In the first setting, we tune CPU affinity to assign 5 cores for the Rx path. None of the six cores hit 100% thus eliminating the issue. The second setting can still face that issue, however, we find that zD alleviates pressure on the Tx path, making the performance of the Rx path the main bottleneck.

The default Tx queue length is set to 1000 in both the VM and the hypervisor[3]. Ex-

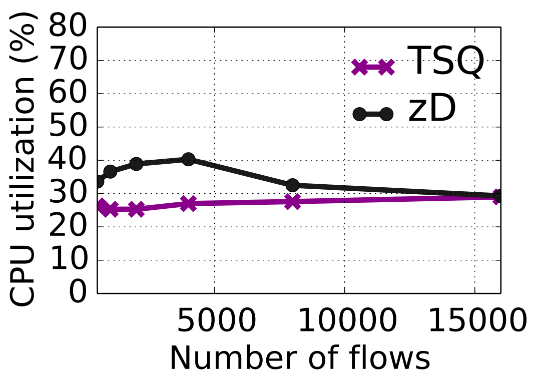
[3]Earlier work with larger scale experiments used a queue length of 4000. Note that a small queue length



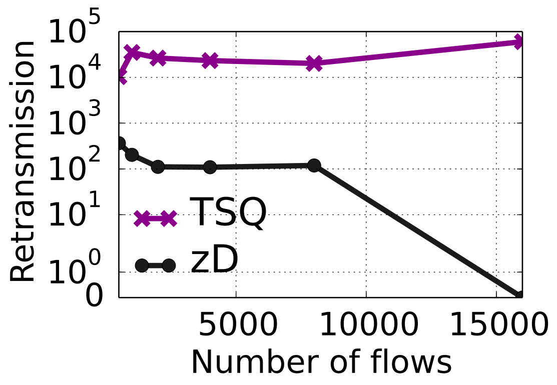
(a) Throughput



(b) VM CPU



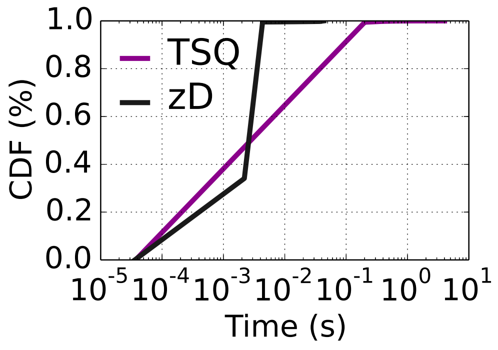
(c) vHost CPU



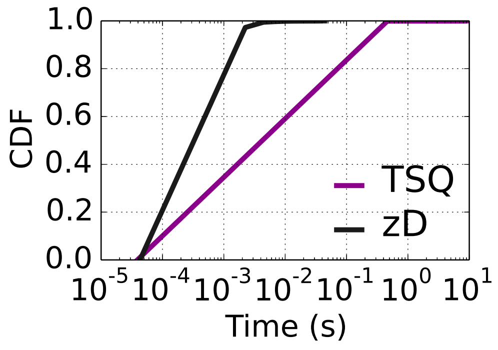
(d) Retransmission

Figure 4.11: 1Gbps network speed with a qdisc of 1000 slots in the hypervisor

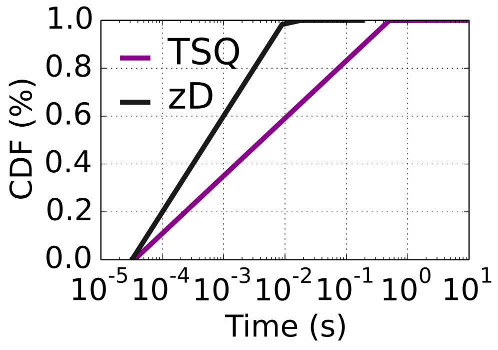
periments are run for 60 seconds each. Our primary metrics are aggregate throughput of all flows, CPU utilization inside the VM, vhost CPU utilization for its pinned core, TCP retransmissions, and RTT. We track CPU utilization in the virtual machine using `dstat` and track CPU utilization of the vhost process in the physical machine using `top`. CPU utilization is recorded every second. We track the number of TCP retransmissions using `netstat`. In all experiments, machines are running only the applications mentioned here making any CPU performance measurements correspond with network overhead.



(a) 10G network with 100 qlen



(b) 10G network with 1000 qlen



(c) 1G network with 1000 qlen

Figure 4.12: zD reduces the tail of RTT by 100x with both 10G network and 1G network

4.5.2 Overall Performance

We start by reporting the overall performance of zD in a setting where packet drops can occur in the VM and the hypervisor. These experiments represent the general case of modern cloud infrastructure. In particular, we consider three cases: 1) a high bandwidth VM with a short queue in the hypervisor, where we allocate the whole 10 Gbps to the VM but configure a short queue of 100 slots[4], 2) a high bandwidth VM with a long queue in the hypervisor, where we allocate the whole 10Gbps to the VM and configure a queue of 1000 slots, and 3) a low bandwidth VM with a long queue, where the hypervisor forces a 1 Gbps limit on the VM in a queue with 1k slots. In the high bandwidth VM setting,

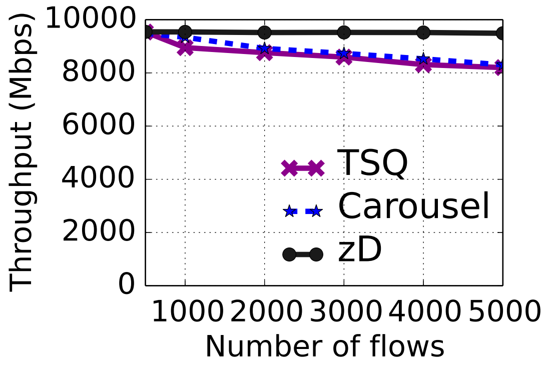
is also critical to avoid bufferbloat.

[4]We choose a small queue length to force congestion in the hypervisor. This emulates production scenarios where queue lengths are larger but the number of VMs per end host will also be much larger, making the effective queue length per VM small.

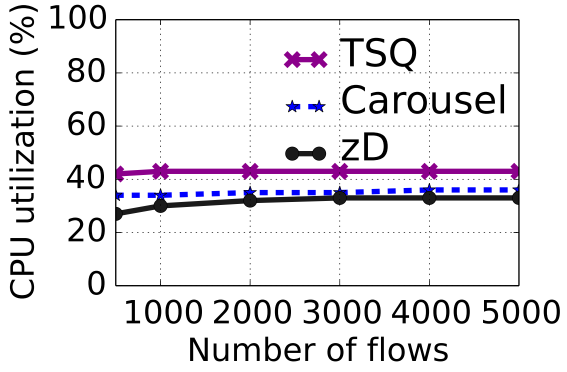
we use a `pfifo` Qdisc in the physical machine. In the low bandwidth VM setting we use `tbf` to perform rate limiting in the hypervisor. We use the default queue size 1000 for the qdisc inside the VM. The first setting represents strict performance requirements (i.e., small processing budget per packet and high probability of packet drop, as shown in recent work [39]), while the second and third represent the more general case. We vary the number of flows from 500 to 16k and measure throughput, CPU utilization inside the VM, `vhost` CPU utilization and TCP retransmissions. For both settings we measure the RTT at 4k flows. We focus on cases with a single VM to be able to better analyze the results. We further restrict the settings in the following sections, to explain the value of individual zD mechanisms. We allocate 6 vCPUs to the VM to avoid having the Rx path being the bottleneck. VM with less vCPUs will be evaluated in the micro-benchmark section.

Figure 4.9 shows the performance of the standard kernel implementation and zD for the first setting. zD performs better in terms of all metrics. In particular, zD achieves around 50% improvement on the aggregate throughput when there are more than 4k flows (Figure 4.9a). Such improvements in throughput come from the elimination of the `vhost` CPU utilization as the bottleneck (Figure 4.9c). zD saves between 40% to 50% of the thread utilization of its CPU core, which is 100% utilized in the standard implementation, making it the performance bottleneck and leading to 50% loss in network throughput. Furthermore, zD reduces tail latency by 80x from 4s to 0.05s (Figure 4.12) which is mostly due to reduction of TCP retransmissions by 1000x (Figure 4.9d). There is a slight degradation in median latency but such slight degradation is generally tolerable to significantly reduce the tail latency [32]. Note that in this scenario zD is lightweight as at low loads it consumes less CPU and achieves better network performance, compared to the standard kernel implementation.

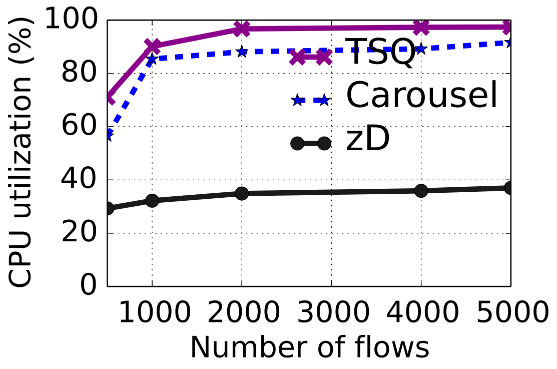
Figure 4.10 shows the results for the second case. Compared with the first setting, TSQ achieves higher throughput and less retransmission because of fewer drops on the hypervisor qdisc. But still, zD achieves higher throughput, lower VM CPU usage, lower



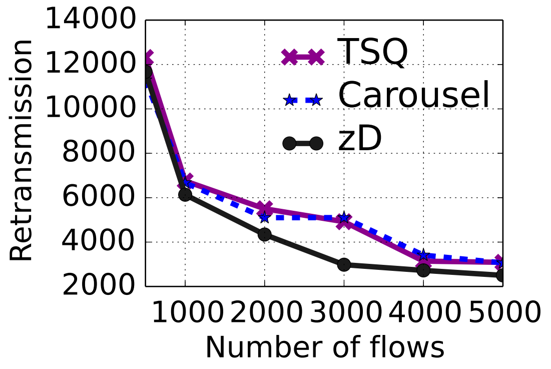
(a) Throughput



(b) VM CPU

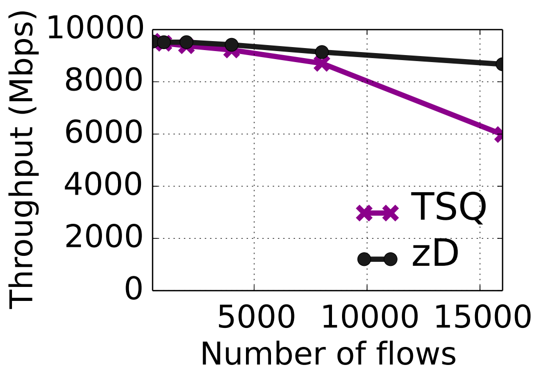


(c) vHost CPU

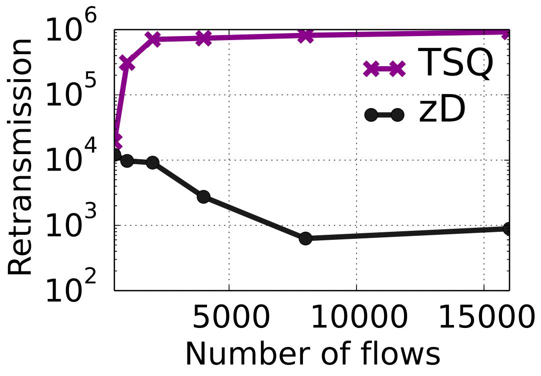


(d) Retransmission

Figure 4.13: Compared with Carousel, zD achieves higher throughput, lower VM CPU usage, lower vHost CPU usage, and fewer TCP retransmissions



(a) Throughput



(b) Retransmission

Figure 4.14: Compared with TSQ, zD achieves higher throughput and fewer TCP retransmissions when 1 vCPU is assigned to the VM.

vHost CPU usage, and fewer TCP retransmissions. We observe there are less than 100 packet drops in the hypervisor qdisc so the improvement mainly comes from the advantages of using zD in the VM. The zD vHost CPU usage is lower than that of the standard (TSQ) kernel when the number of flows is smaller than 16K. When there are 16K flows, zD has higher vhost CPU usage because it pushes much more traffic than the standard kernel. The tail latency is significantly reduced from 8s to 0.05s (Figure 4.12).

Figure 4.11 shows the results for the third setting. zD again improves all network metrics. In particular, zD improves throughput by up to 5% (Figure 4.11a) and reduces retransmissions by 1000x (Figure 4.11d). Most notably, zD reduces tail latency by 45x from 9s to 0.2s (Figure 4.12). zD also reduces VM CPU utilization by 15%. However, zD incurs higher `vhost` CPU cost by up to 40%. The higher vHost CPU usage results from the extra work of `vhost` trying to resend the packets dropped in the physical machine Qdisc instead of relying on the TCP socket in the VM to retransmit the packets. This shows a tradeoff between network performance and VM CPU on one side and hypervisor CPU on the other side. We also envision that userspace stacks can amortize the cost in the hypervisor due to their busy polling nature [87, 86].

Comparison with Carousel: We use Carousel as a baseline to examine if the combination of efficient queuing data structure and TSQ-like backpressure can improve on the performance of standard Linux Qdiscs. We implemented Carousel in Linux Qdisc using a more efficient integer priority queue data structure [42] and compared zD with carousel in the 10G network setting with a queue of 1000 slots in the hypervisor. Figure 4.13 shows that zD outperforms Carousel in all metrics. While Carousel achieves higher CPU efficiency and a higher throughput compared with TSQ, it does not fundamentally solve the problem when a queue runs out of space with a large number of flows. As discussed earlier, Carousel relies on TSQ-like backpressure to limit the number of packets per flow, which works reasonably well with a small number of flows. Unfortunately, with a large number of flows, limiting two packets per flow can still overflow the queue, leading to performance degradation.

4.5.3 Microbenchmark

zD with VM-only bottleneck: In the previous section, we looked at the general case where drops happen in both the VM and the hypervisor. In this section, we look at cases where there is a single bottleneck. We focus on the case where drops happen at the VM because it is easier to test it at large scale (i.e., large number of flows) compared to the hypervisor which requires scaling to a large number of VMs. We prevent drops in the hypervisor by a long unscheduled queue (i.e., `pfifo` with 1k slots). Note that this setting is convenient and allows for a better understanding of the performance of zD because adding more VMs causes drops in the hypervisor, which results in a similar scenario as the one we studied earlier.

We start by looking at the case where the VM is allocated 6 vCPUs, thus eliminating the Rx path bottleneck. The result is similar to what we show in Figure 4.10. When we use a queue of 1000 slots in the hypervisor, regardless whether zD is implemented in the hypervisor or not, the performance is similar because the hypervisor queue is not easily congested due to the high-speed NIC and the low latency of the networking stack.

To highlight the value of zD, we rerun the experiment in the setting where 1 vCPU is assigned to the virtual machine. zD's value is clear in its impact on throughput as shown in Figure 4.14a. In particular, zD can maintain 43% higher throughput at 16k flows. This significant improvement is mostly due to reduction in retransmission rate (Figure 4.14b). We find that zD and the standard kernel exhibit similar CPU performance for both the VM and the `vhost` thread when the number of flows is larger than 2k. Both systems have 100% VM CPU utilization and their `vhost` CPU utilization was about 49% for the Linux kernel and 43% for zD. The reason for degradation of kernel implementation moving from allocating 6 vCPUs to a single vCPU is the higher retransmission rate in the latter case. This is mostly due to the Rx path congestion which leads ACK packets to be delayed.

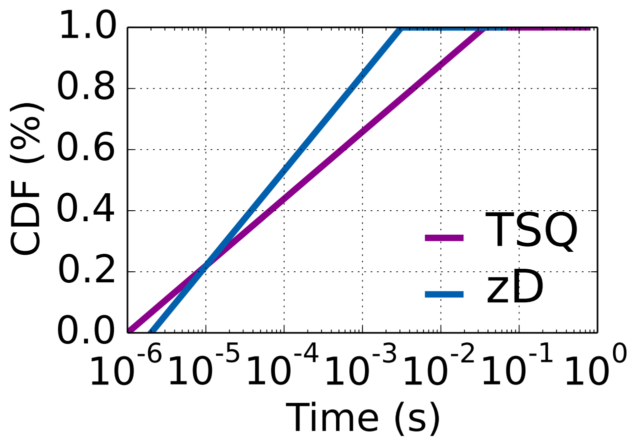


Figure 4.15: CDF of flow RTT with hypervisor-only zD.

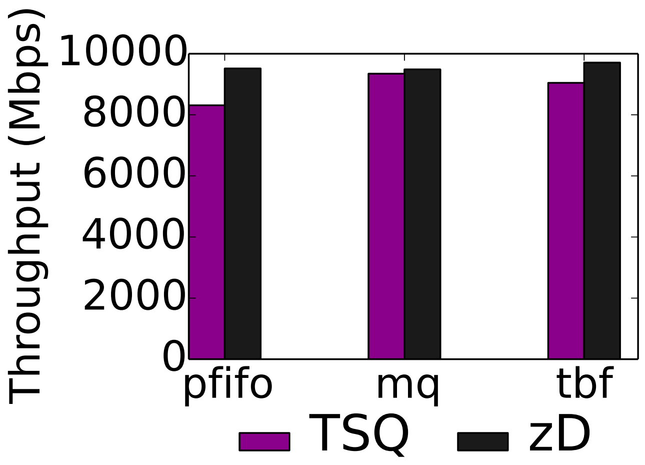


Figure 4.16: Throughput under different Qdiscs

This causes TCP spurious retransmissions, where senders timeout and retransmit packets whose ACK is delayed. zD achieves lower TCP retransmission by reducing the number of interrupts in VM Tx path reducing the Rx path congestion.

zD with Hypervisor-only bottleneck: Next we quantify the benefit of zD when it is only implemented in the physical machine. We use `tbf` Qdisc to set a bandwidth limit of 1Gbps in the physical machine with only 100 flows. This setting forces packet drops to happen only in the hypervisor. In both settings, the flows achieve the targetted aggregate

Table 4.1: Drops and retransmission in case of standard Linux kernel and hypervisor-only zD implementation.

	Drop in Qdisc	TCP retransmission
Linux kernel with TSQ	6.67×10^5	1×10^5
zD	562	110

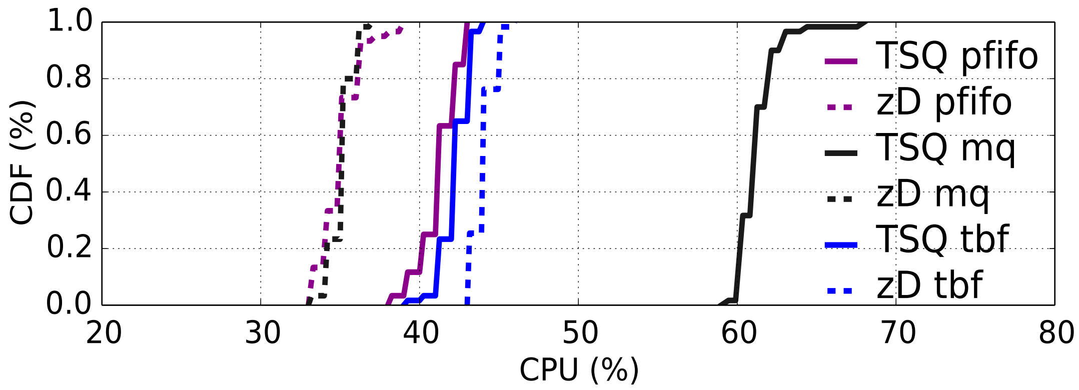


Figure 4.17: CPU usage in VM under different Qdiscs

rate. However, zD improves RTT by avoiding packet drops. Figure 4.15 shows the CDF of flow RTT. With vanilla kernel, the 99.99^{th} percentile is large at around 0.8s. The 99.99^{th} percentile of zD is less than 0.1s. We present the number of drops in Qdisc and the number of TCP retransmission for both the standard kernel and zD in Table 4.1. zD reduces the TCP retransmission caused by packet drop in Qdisc by 1000x. Note that CPU utilization is low and comparable for both implementations due to the limited scale of the experiment.

Interaction with different Qdisc: We explore zD performance under different Qdiscs to show that zD can operate with different underlying policies. We implement a FIFO policy for the Paused-flows Queue, which is compatible with all queuing policies used here. However, each queuing policy requires a different implementation of the `has_room` function. We conduct our experiments for `pfifo`, `mq`, and `tbf` Qdiscs and measure the throughput and CPU usage inside the VM with 4k flows. The chosen setting is similar to the case where drops happen only at the VM.

Figure 4.16 shows that zD achieves around 12% throughput improvement with `pfifo`

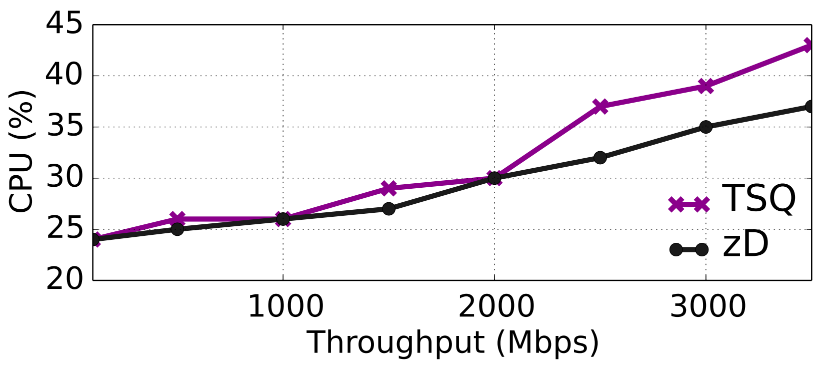


Figure 4.18: CPU usage in VM under light traffic load

and around 8% throughput improvement with `tbf`. With `mq`, both zD and the standard kernel have similar throughput because the instances of `vhost` process scale as the number of queues increases. Figure 4.17 shows the CDF of CPU usage in the VM under different Qdiscs. zD reduces the CPU usage by 7% under `pfifo` and by 25% under `mq` but has slightly higher CPU usage under `tbf`. The higher CPU usage with `tbf` results from the extra work performed by zD to stop and resume the vring. Although dropping packets directly can save hypervisor CPU, the dropped packets need to be recovered by the TCP retransmission mechanism thus wasting CPU in the VM.

zD at light loads: zD achieves its goals by adding more coordination between traffic sources and packet queue, which might cause significant overhead at light loads. However, we find that this is not the case. To conduct experiments with low loads, we use 10 instances of iperf as the traffic generator, each generating 100 TCP flows. We control the load by setting a rate limit in the application layer, reducing demand of individual flows. Figure 4.18 shows the effect of varying the TCP loads on CPU usage in VM. zD has similar CPU usage as the standard kernel because there is little packet drop in Qdisc when the traffic load is light. Hence, no coordination between traffic sources and the packet queue is needed. As the traffic load increases, getting closer to an aggregate rate of 3 Gbps, the number of drops in Qdisc also increases and zD starts to outperform the standard kernel.

4.6 Limitations of zD

We show that zD can improve network and CPU performance by applying backpressure from the scheduled buffer to the source buffers. Our work on zD has some modest limitations. In particular, the overhead of backpressure in the hypervisor can, in some cases, cause the `vhost` thread to consume more CPU than a standard implementation. We believe that with further engineering this overhead can be eliminated completely. A minor limitation of our evaluation approach is our focus on simple scheduling policies in the Paused-flows queue. However, we find that recent work on efficient packet schedulers in software has thoroughly handled the issue [42], allowing us to focus more on handling cases of congestion.

4.7 Summary

Packet queuing and scheduling is a standard operation at end hosts. Congestion of scheduled queues at end hosts typically incurs packet drops which lead to high CPU cost as well as degradation in network performance. In this chapter, we show that by augmenting existing architectures with three simple mechanisms, CPU and network performance can be significantly improved under high loads, improving tail latency by 100x. Our work on zD should extend the scalability of current end-host stacks and motivate revisiting the queuing architecture in other network elements.

CHAPTER 5
ANALYZING THE CPU COST OF NETWORKING STACK

Modern servers at large scale operators handle tens of thousands of clients simultaneously [60, 100, 38]. This scale will only grow as NIC speeds increase [101, 102, 103] and servers get more CPU cores [84, 104]. For example, a server with a 400 Gbps NIC [102] can serve around 80k HD video clients and 133k SD video clients.[1] This scale is critical not only for video on demand but also for teleconferencing and AR/VR applications. The focus of the community has been on scaling servers in terms of packets transmitted per second [52, 106, 53, 107, 45, 54], with little attention paid to developing complete stacks that can handle large numbers of flows well [57, 59].

We envisage servers delivering large volumes of data to millions of clients simultaneously. Our goal is to identify bottlenecks that arise when servers reach that scale. In particular, we take a close look at network stack components that become the bottleneck as the number of flows increases. We find that as the number of flows increases competition between flows can lead to overall performance degradation, requiring fine-grain scheduling. Further, the overhead of per-flow bookkeeping and flow coordination increases. Thus, we categorize problems that arise due to increase of number of concurrent flows into two categories:

1) Admission Control to the Stack: The admission policy determines the frequency at which a flow can access the stack and how many packets it can send per access. The backpressure mechanism determines how a flow is paused (e.g., denied admission) and resumed (i.e., granted admission). The frequency at which a flow gets access to network resources and the duration of each access determine the throughput it can achieve. As the number of flows increases, admission control becomes critical for the efficiency of the

[1]HD and SD videos consume up to 5 Mbps and 3 Mbps, respectively [105].

stack. For example, admitting and alternating between flows at a high frequency can reduce Head-of-Line (HoL) blocking and improve fairness but at the expense of CPU overhead, which can become a bottleneck, leading to throughput loss.

2) Per-packet Overhead within the Stack: The overhead of most per-packet operations is almost constant or a function of packet length (e.g., checksum, routing, and copying). However, the overhead of some operations depends entirely on the number of flows or clients serviced by the system. For example, the overhead of demultiplexing is tied to the number of flows in the system. The overhead of scheduling, for some scheduling policies, depends on the number of flows in the system (e.g., fair queueing).

We focus our attention on the Linux stack. Despite its well documented inefficiencies (e.g., the overhead of system calls, interrupts, and per-packet memory allocation [57, 108]), the Linux networking stack remains the only fully implemented, publicly available networking stack. Further, emerging network processing engines sometimes interface with different parts of the stack to make use of its functionality [109]. Hence, our focus on Linux is critical for two reasons: 1) our results are immediately useful to a wide range of server operators, and 2) we are able to identify all possible bottlenecks that might not appear in other stacks because they lack the functionality. However, a major goal of our study is to focus on performance issues that are common with other stacks, avoiding the well documented issues that are Linux-specific. Thus, we avoid documenting problems like the overhead of system calls, slow memory allocation, and reliance on interrupts [57, 108].

We are not the first to identify some of the bottlenecks and performance impairments we discuss in this paper. For instance, there have been several proposals to improve the performance of data structures used in scheduling [55, 38, 42]. There has also been work improving the performance of transport layer [57] and backpressure [110]. In our study, each of these problems represents an instance of a broader category. *The contribution of this work is defining these broad categories and identifying new instances of each of them.*

5.1 Measurement Setup

Testbed: We conduct our experiments on two dual-socket servers. Each server is equipped with two Intel E5-2680 v4 @ 2.40GHz processors. The socket directly connected to the NIC is considered the local socket. Each server has an Intel XL710 Dual Port 40G NIC Card with multi-queue enabled. The machines belong to the same rack. Both machines use Ubuntu Server 18.04 with Linux kernel 5.3.0. We bind multiple IP addresses to each server so the number of TCP flows that can be generated is not limited by the number of ports available for a single IP address.

Testbed Tuning: To reduce cache synchronization between different cores and improve interrupts affinity, we configure each transmit/receive queue pair to always use the same core. We enable Receiver Packet Steering (RPS), which computes a hash source and destination IPs and ports and determines which CPU to send the packet to based on the hash. Then, the packet is assigned to the appropriate per-CPU queue for further processing by `softirq`. We limit all network processing to exclusively use the local socket. We find that using both cores leads to performance degradation at 200k flows. We compare the aggregate throughput of 200k flows when they all use the local socket and when they are divided between the two CPU sockets. We observe that aggregate throughput drops by 15% with equal or more than 200k flows when the two sockets are used. When CPU usage reaches 50%, the CPU resource is saturated because we bind the application and interrupts to only one socket.

We enabled different hardware offload functions including GSO, GRO, and LRO to lower CPU utilization. We also enabled interrupt moderation which generates interrupts per batch of packets, rather than per packet. We use TCP CUBIC as the default transport protocol, providing it with maximum buffer size, to avoid memory bottlenecks. The entire set of parameters is shown in Table 5.1.

Traffic Generation: We generate traffic with `neper` [111], a network performance

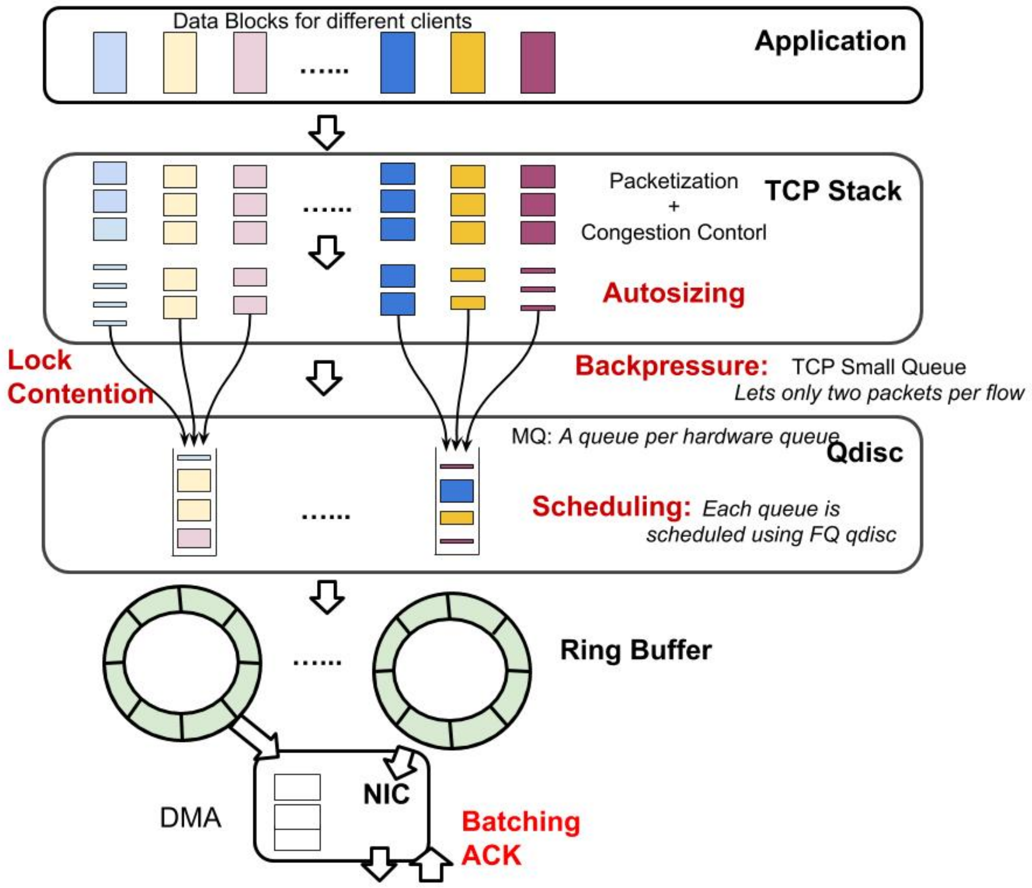

Data Blocks for different clients
Application
TCP Stack
Packetization
+
Congestion Contorl
Autosizing
Lock
Contention
Backpressure: TCP Small Queue
Lets only two packets per flow
MQ: A queue per hardware queue
Qdisc
Scheduling: Each queue is
scheduled using FQ qdisc
Ring Buffer
DMA
NIC
Batching
ACK

Figure 5.1: Schematic of the packet transmission path with identified pain points marked in red.

measurement tool to generate up to 300k concurrent flows. With 40 Gbps aggregate throughput, the per-flow rate can range from 133 Kbps, which is a typical flow rate for web service [112], to 400 Mbps, which might be large data transfer [113]. We ran experiments with different numbers of threads ranging from 200 to 2000 and observed that using more threads causes higher overhead in book-keeping and context switching, leading to degraded throughput when the server needs to support hundreds of thousands of flows. The results shown in this paper are with 200 threads if not specified otherwise. For the rest of the paper, we use flows and clients interchangeably.

Figure 5.1 visualizes our assumed stack architecture. Our focus is on the overhead of the transport and scheduling components of the stack. We experiment with differ-

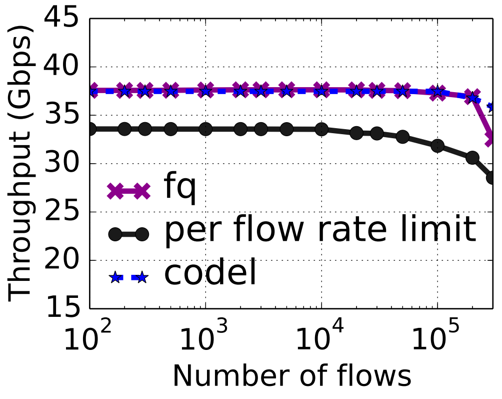
Table 5.1: Tuning parameters

Parameter	Tuned
RX-Ring	MAX [4096]
net.core.netdev_max_backlog	65536
net.core.tcp_max_syn_backlog	65536
net.ipv4.tcp_rmem	8192 65536 16777216
net.ipv4.tcp_wmem	8192 87380 16777216
net.ipv4.tcp_mem	768849 1025133 1537698
net.core.somaxconn	65535
net.netfilter.nf_conntrack_max	600000
TSO,GSO	enabled
interrupt moderation	enabled
irqbalance	disabled

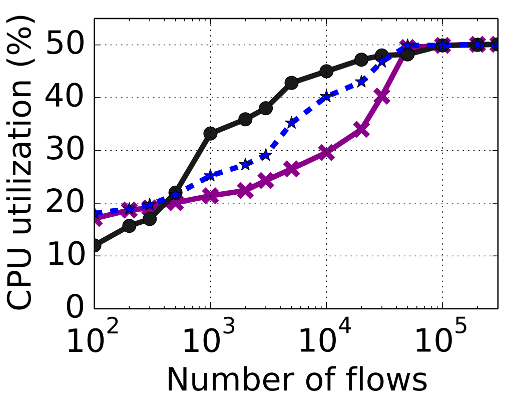
ent scheduling algorithms using different Queuing Disciplines (qdiscs). In our multi-core multi-queue scheme, the multiqueue qdisc (`mq`) is used to avoid having a single lock for all hardware queues. All scheduling algorithms are implemented by per-queue within `mq`. Thus, enforcement of scheduling policies is somewhat distributed. By default, `mq` handles packets FIFO in its queues. However, we use Fair Queue (`fq`) [96] as the default qdisc combined with `mq`. `fq` is designed to avoid HoL blocking by ensuring fairness among flows sharing the same queue. In some experiments, we use `fq_codel` qdisc [114] to reduce latency within the qdisc.

Measurement Collection: Experiments are run for 100 seconds each. In all experiments, machines are running only the applications mentioned here making any CPU performance measurements correspond with packet processing. We track overall CPU utilization using `dstat` [115] and track average flow RTT using `ss` [116]. We track the TCP statistics using `netstat` [117] and `tcpdump` [118]. Performance statistics of specific functions in kernel is obtained using `perf` [119].

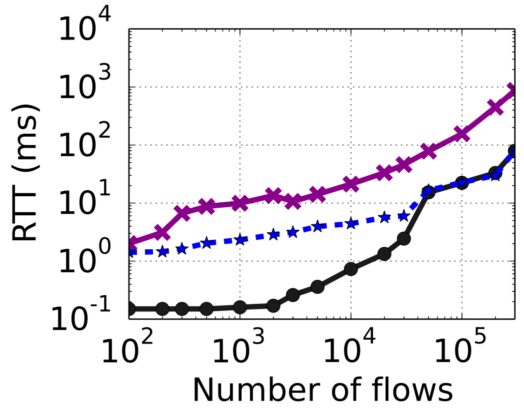
The entire set of tuning parameters is shown in table 5.1.



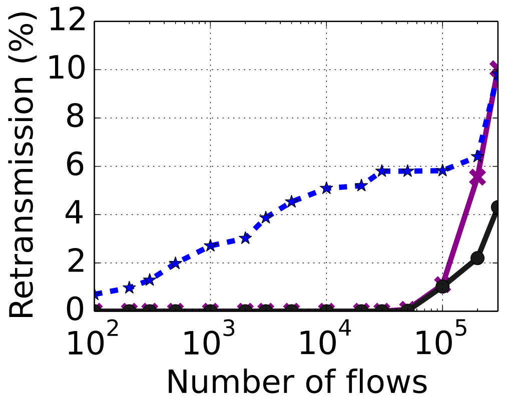
(a) Aggregate Throughput



(b) CPU Usage



(c) RTT



(d) Retransmission

Figure 5.2: Overall performance of the network stack as a function of the number of flows

5.2 Overall Stack Performance

We start by observing the overall performance of the stack as we increase the number of flows. Our goal is to observe how bottlenecks arise as we increase the number of flows, affecting overall server performance. In particular, we look at aggregate throughput, CPU utilization, average RTT, and retransmissions. Figure 5.2 shows a summary of our results. The Linux stack can maintain line rate up to 200k flows (Figure 5.2a). As the number of flows increases, the CPU utilization steadily increases until it becomes the bottleneck. Recall that we are only using a single socket, which means that 50% utilization means full utilization in our case (Figure 5.2b). The increase in CPU utilization can be explained by the inefficiency of packet admission control (increasing the packet rate and flow scheduling overhead) and per-packet processing components, which we explore thoroughly in the next two sections.

To our surprise, the average latency introduced by the stack increases to around *one second* at 200k flows. A part of this latency can be explained by well-documented Linux stack inefficiencies due to relying on interrupts [120, 51, 57, 108]. This can be seen as we apply a per-flow rate limit, targeting an aggregate rate that is 90% of NIC capacity. The rate is applied by setting `SO_MAX_PACING_RATE`. Targeting a lower rate than capacity and preventing any single flow from bursting eliminates all admission control issues and any resource contention between flows. It also avoids overwhelming the NIC. Thus, we get minimal latency till 2k flows. Enforcing this per-flow rate leads to higher CPU utilization, leading to loss in throughput at 50k flows.

We try to understand how much of the latency is caused by transport and scheduling components of the stack. Thus, we use `fq_codel` with a target latency of 100μs. CoDel drops packets if their queueing delay exceeds the target delay. This allows for a reduction of latency by 2.5 to 10$\times$ compared to using `fq` without CoDel. It also matches the latency of per-flow rate limiting between 50k and 300k flows. This leads us to the conclusion that a significant part of the latency difference between the `fq` case and the `fq_codel` case is caused by inefficiencies in transport and scheduling on the egress path of the Linux stack, which is our focus in this paper. Note that CoDel comes at a price of higher CPU utilization due to packet drop and retransmission (Figure 5.2d).

5.3 Admission Control to the Stack

Network stacks are typically optimized to maximize the number of packets per second they can handle. However, we find that as the number of flows increases, a lot of room arises for optimizing the number of packets per second that enter the stack. Such optimization can improve the CPU efficiency of the stack, without compromising network utilization. There are three parts of the stack that determine the number of packets entering:

Automatic packet sizing: determines the number of packets a block of data is broken into. For instance, a 64KB block of data can be handled by the stack as a single TCP segment of 64KB or many smaller segments. Sending a large segment can hurt network performance by bursting [96]. Breaking it up into small pieces increases CPU utilization, potentially leading to low network utilization.

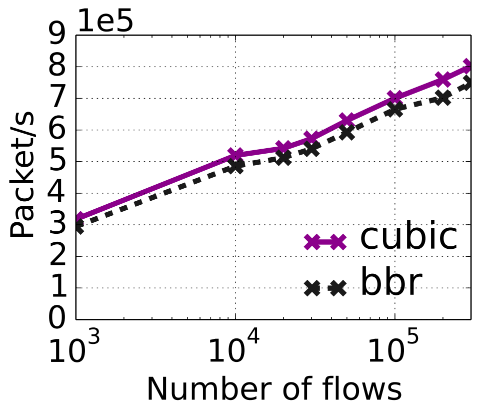
Backpressure: rate limits or pauses flows based on the load on the network stack. Backpressure typically limits the number of packets a flow can have outstanding in the stack. However, as the number of packets increases, even having a single packet per flow can overwhelm the server's capacity.

Batching ingress packets: reduces CPU overhead by amortizing the cost of traversing the stack over a batch of packets. Batching balances CPU utilization and packet latency. A large batch size can mean delaying packets for a long time till a full batch is ready while a small batch size means that the stack is invoked more frequently. Another drawback of a small batch size at the NIC is not giving LRO enough time to coalesce packets belonging to the same flow. The chance of such coalescing decreases as the number of flows increases, further increasing the CPU overhead by delivering small packets, at a high rate, to the stack.

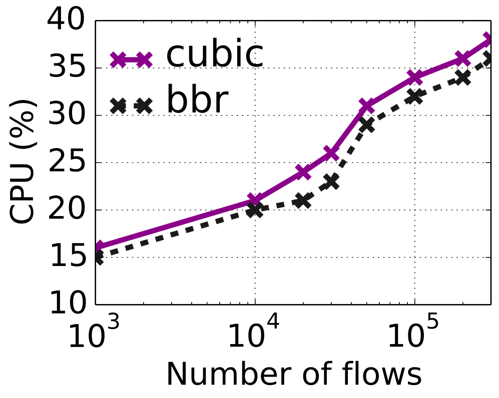
For the rest of this section, we take a closer look at the implementation of each of these operations in the kernel, showing that at a large number of flows, better algorithms are needed to perform these tasks.

5.3.1 Automatic Packet Sizing

Packet autosizing decides the size of data in a burst, by ensuring that flows send a packet at least every 1ms [96]. The algorithm is triggered if a flow is sending at a rate lower than 512 Mbps (i.e., a thousand Maximum Segment Sized (MSS) segments every second, assuming an MSS of 64KB). When triggered, it reduces the size of the segments transmitted every 1ms, where inter-packet gap is enforced through a pacer. Autosizing infers the rate



(a) Packet Rate



(b) CPU Usage

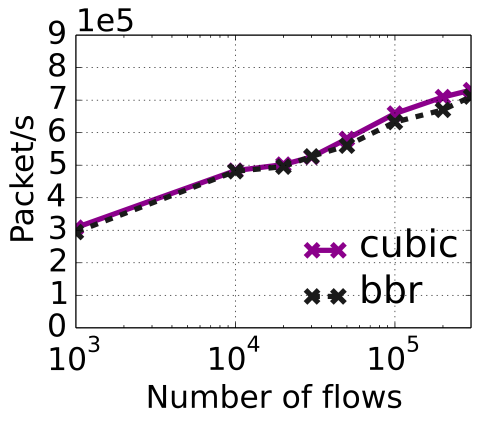
Figure 5.3: CUBIC v.s. BBR with 5% drop rate.

of a flow by dividing the number of bytes sent during an RTT (i.e., the `cwnd`) over the measured RTT. This allows for maintaining the same average sending rate while spreading packet transmission over time. The technique provides a tradeoff between CPU utilization and network performance by increasing the number of packets per second handled by the server while lowering the size of bursts the network deals with.

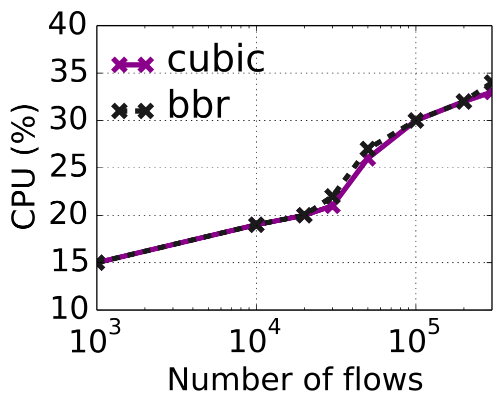
The CPU tradeoff becomes even more costly when the number of flows increases. In particular, the same aggregate rate of 512 Mbps can result in a packet rate of 1k packets per second for one flow or 1M packets per second for 1k flows in the worst case.[2] This increase in packet rate causes the overloading of the stack , leading to delay in packet transmission (Figure 5.2c). The increased delay makes the autosizing algorithm misbehave. In particular, RTT increases when the stack is overloaded, leading to underestimation of the rates of all flows handled by the stack. This causes autosizing to reduce the size of bursts unnecessarily, creating more packets, increasing the congestion at the server [110].

Reducing delay introduced in the stack can help autosizing infer the rates of flows more accurately. However, as we will show later, scheduling flows, including delaying packets,

[2]The number of packets is typically much smaller than the worst case scenario due to imperfect pacing. Delays in dispatching packets, resulting from imperfect pacing, require sending larger packets to maintain the correct average rate, leading to a lower packet rate. However, the CPU cost of autosizing increases with the number of flows even with imperfect pacing.



(a) Packet Rate



(b) CPU Usage

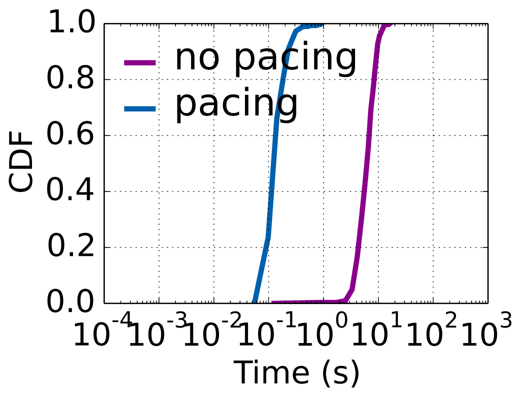
Figure 5.4: CUBIC v.s. BBR with 0% drop rate.

is essential to scaling the end host. This means that autosizing-like algorithms need to differentiate between network congestion and end-host congestion. This will be useful in avoiding generating extra packets which might congest the end host but not the network.

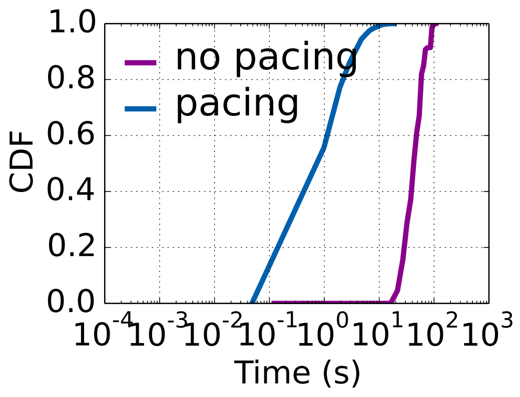
Autosizing has the side effect of causing different congestion control algorithms to have different CPU costs. In particular, algorithms that react more severely to congestion (e.g., CUBIC which halves its window on a packet drop) send at lower rates, forcing autosizing to create more packets. On the other hand, algorithms that react mildly to congestion (e.g., BBR) can generate a smaller number of packets and maintain the high rate. Figure 5.3 show the difference between CUBIC and BBR at 5% drop rate induced by a `netem` Qdisc at the receiver. The relationship between number of flows and packet rate is similar at 0% drop but there is no difference between BBR and CUBIC at 0% drop rate (Figure 5.4). We set MTU size to 7000 to eliminate the CPU bottleneck.

5.3.2 Backpressure and Scheduling

When a flow has a packet to send, its thread attempts to enqueue the packet to the packet scheduler (i.e., Qdisc in the kernel stack). In order to avoid HoL blocking, TCP Small Queue (TSQ) limits the number of packets enqueued to the qdisc to only two packets per flow [14]. TSQ combined with fair queueing at the qdisc can ensure fairness between a



(a) 300 flows



(b) 30k flows

Figure 5.5: CDF of flow rate showing that FQ fails to ensure fairness among flows

small number of flows. However, as the number of flows increases two configurations are possible. First, we can allow only a small queue buildup in the qdisc, dropping any excess packets. This is appealing because it limits bufferbloat at the server [13, 110]. But this means that not all flows get their packets enqueued in the qdisc, hindering the fair queueing functionality as the qdisc is not aware of all active flows. Further, limiting the capacity of the qdisc leads to drops. The current approach in Linux is to immediately retry to enqueue the dropped packet. This leads to poor CPU utilization as threads keep retrying to enqueue packets. Figure 5.6 shows the CPU utilization for different values of maximum queue length at the qdisc. As we decrease the maximum queue length the CPU utilization increases. Further, thread scheduling has no notion of per-flow fairness, leading to severe unfairness between flows.

The second configuration is to allow long queues, which has the drawback of causing bufferbloat, but should allow for better fairness as all flows get a chance to enqueue their packets and it is up to the scheduler to ensure fairness. However, we find that unfairness still exists due to the distributed nature of scheduling in a multi-queue system [56]. Figure 5.5 compares the CDF of rates achieved when `fq` is used with a small number of 300 and 30k flows. The two scenarios are contrasted with the per-flow pacing scenario which achieves best possible fairness by rate limiting all flows to the same rate, with aggregate rate below

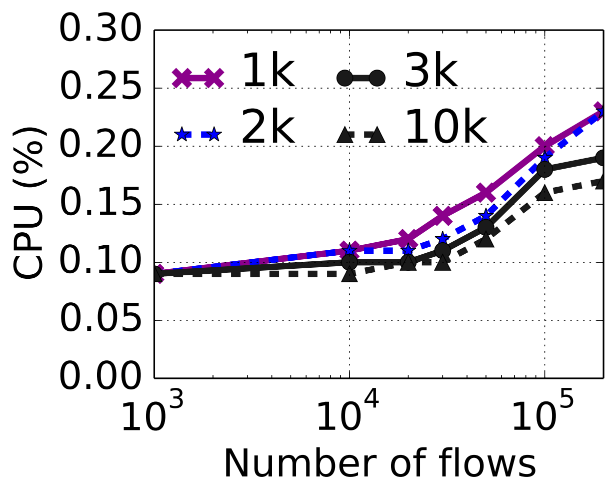


Figure 5.6: CPU usage as function of Qdisc queue length

NIC capacity, thus avoiding creating any bottlenecks. In the 30k flows scenario, the largest rate is two orders of magnitude greater than the smallest rate. This is caused by the batching on the NIC queue.

Packet transmission in an end-host refers to the process of a packet traversing from the user space, to the kernel space, and finally to the NIC in packet transmission process. The application generates a packet and copies it into the kernel space TCP buffer. Packets from the TCP buffer are then queued into the Qdisc. Then there are two ways of dequeuing a packet from the Qdisc to the driver buffer: 1)dequeue a packet immediately, and 2) schedule a packet to be dequeued later through the softriq, which calls `net_tx_action` to retrieve packet from the Qdisc (Figure 5.8). The `net_tx_action` function calls into the Qdisc layer and starts to dequeue skb through `dequeue_skb` function. Multiple packets can be returned by some queues, and a list of skb may be sent to the NIC, blocking packets from other queues. We observe that there are many more requeue operations in Qdisc when pacing is not used than when pacing is used, indicating that pacing prevents the NIC from being overwhelmed by a subset of queues.

Addressing this problem requires combining two techniques. The first relies on per-flow scheduling instead of per-packet scheduling [110, 109]. These approaches allow a flow to

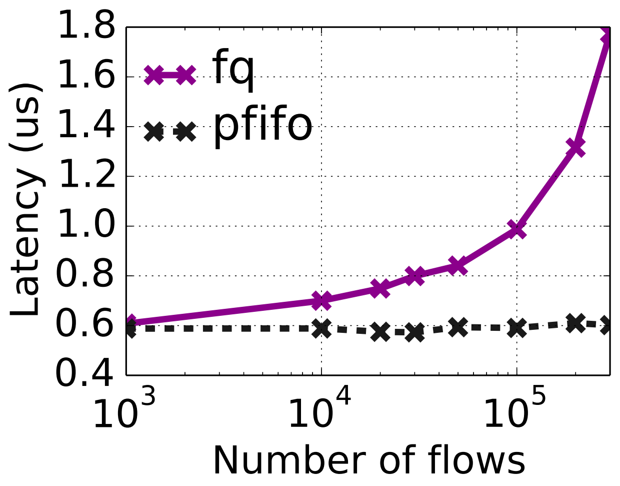


Figure 5.7: Average execution time of FQ enqueue operation

enqueue a packet only when there is room for it in the scheduler, avoiding unnecessary drops and retries. Second, distributed multi-queue scheduling allows for enforcing accurate scheduling policies in a multi-queue setting [56].

5.3.3 Batching Ingress Packets

The two previous sections discuss controlling the packet rate on the egress path. In this section, we consider controlling the packet rate on the ingress path. A receiver has little control on the number of incoming packets, aside from flow control. By coalescing packets belonging to the same flow on the ingress path using techniques like LRO, the receiver can improve the CPU efficiency of the receive path by reducing the number of packets it has to process. This is important even for servers as ACK coalescing can help reduce the overhead of the ingress path. Batching algorithms deliver packets to the software stack once the number of outstanding packets in the NIC reaches a certain maximum batch size or some timer expires. As the number of flows increases, the chances of such coalescing decrease as the likelihood of two incoming packets belong to the same flow decreases (Figure 5.9). In the Linux setting, this is especially bad as increasing the number of incoming packets results in an increase in interrupts, leading to severe degradation in CPU efficiency.

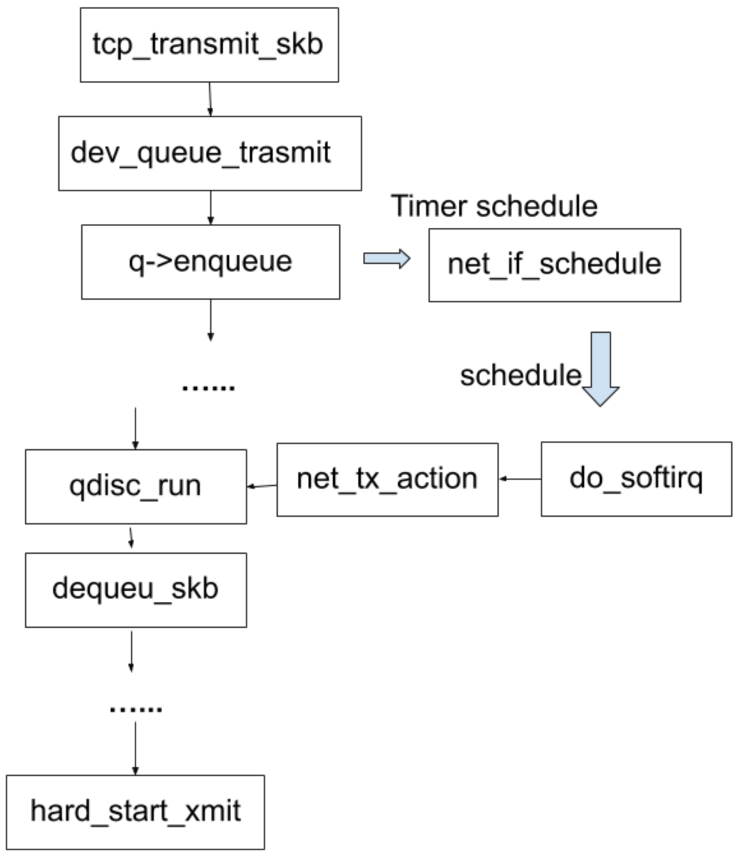


Figure 5.8: Packet transmission function call graph

Better batching techniques that prioritize short flows, and give LRO more time with long flows, can significantly help improve the performance of the ingress path. Some coarse grain adaptive batching techniques have been proposed [121, 122], however, we believe that better performance can be achieved with fine-grain per-flow adaptive batching.

5.4 Per-packet Overhead

To identify the operations whose overhead increases as the number of flows increases, we use `perf` [119] and observe the CPU utilization and latency of different kernel functions as we change the number of flows. In particular, operations whose computational complexity is a function of the number of flows will have higher CPU utilization as we increase the

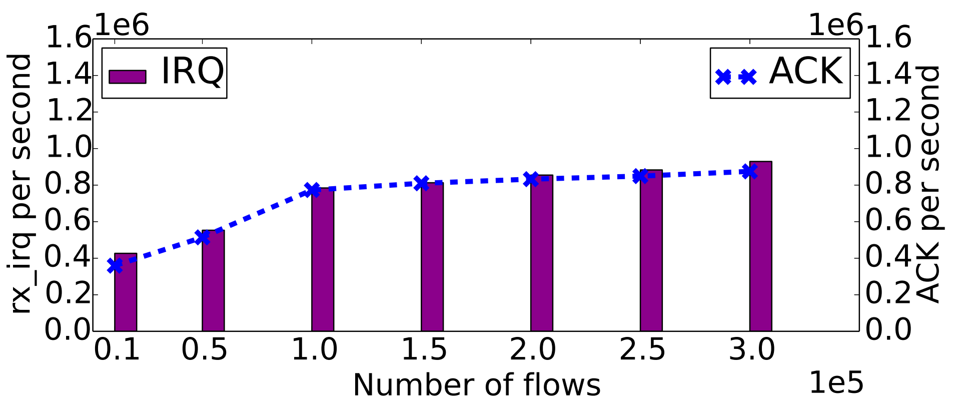


Figure 5.9: Rates of RX interrupts and ACK per second

number of flows. Operations that are bottlenecked on a different type of resource will have higher latency as we increase the number of flows. Figures 5.12a and 5.12b show the top four functions in each category. There is an overlap between functions with high latency and functions with high CPU utilization; this is typical because high CPU utilization can lead to high latency (e.g., `fq_dequeue` and `inet_lookup`). However, there are function with high latency but low CPU utilization (e.g., `tcp_ack` and `dev_queue_xmit`). Through further profiling of the code of these functions, we find that there are two types of bottlenecks that arise: cache pressure and lock contention. Note that the overhead of the `tg3_poll_work` function is part of inefficiency of the Linux reception path [123] and is not the focus of our work.

Data structures: There are two operations whose complexity is a function of the number of flows: packet scheduling and packet demultiplexing. The overhead of packet scheduling is captured by the CPU utilization of `fq_enqueue` and `fq_dequeue`. The two functions handle adding and removing packets to the `fq` Qdisc, which sorts flows in a red-black tree based on the soonest transmission time of their packets. The overhead of enqueue and dequeue operations in $O(\log(n))$, where n is the number of flows. The overhead of packet demultiplexing is captured by the CPU utilization of `inet_lookup` which matches incoming packets to their flows using a hashmap. As the number of flows increases, the chances of collision increases, increasing the overhead of finding the match. Further, finding a match, in the case of collision, requires processing information of flows whose hash collide. This increases the cache miss ratio of the function (Figure 5.12c), further increasing the latency of the function.

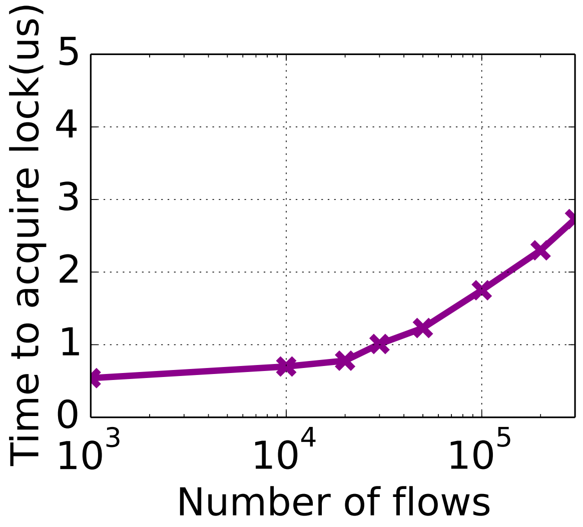


Figure 5.10: Aggregate cache misses

Data structure overhead requires reexamining all complex data structures used in the stack, taking into account that the stack can process millions of packets per second coming from millions of flows. For example, the overhead of scheduling can be reduced by using simpler policies, at the expense of network performance (Figure 5.7). We compare the `fq` with `pfifo_fast` qdiscs in terms of enqueueing latency. The time to enqueue a packet into `pfifo_fast` queue is almost constant while the enqueue time for `fq` increases with the number of flows. This is because the FQ uses a tree structure to keep track of every flow and the complexity of insertion operation is $O(\log(n))$. The cache miss when fetching flow information from the tree also contributes to the latency with large number of flows.

Cache pressure: One of the functions with the highest cache miss ratio is `tcp_ack`, which clears the TCP window based on received acknowledgements. The function does use any complex data structures or wait on locks. Its overhead stems from the overhead

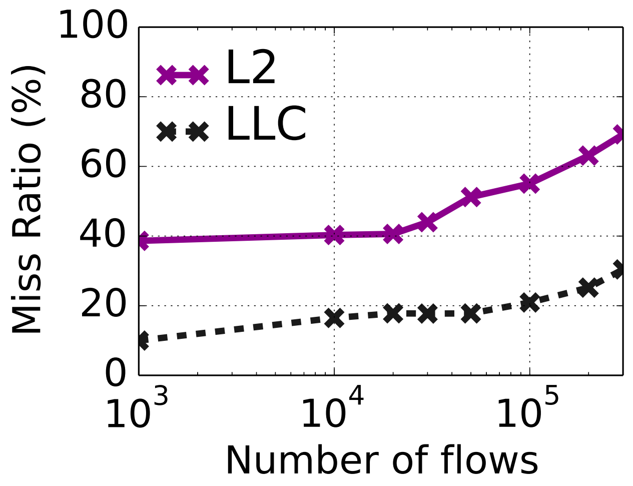


Figure 5.11: Time to acquire Qdisc lock

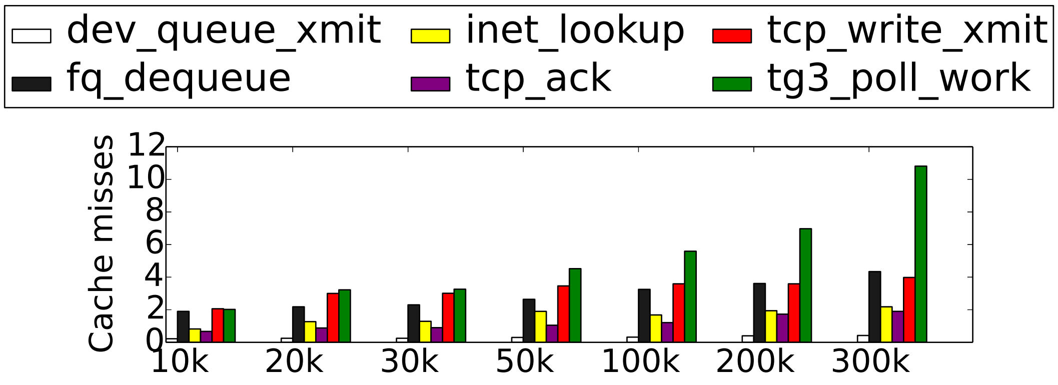
of fetching flow information and modifying it. To understand the impact of cache misses on throughput performance, we measure the cache miss ratio and aggregated throughput with a varying number of flows. As shown in Figure 5.10, the cache miss ratio in both L2 cache and Last Level Cache (LLC) increases as the number of flows increases. While cache misses are not a huge bottleneck in our setting, we believe that as the number of flows increases, with tighter requirements on latency, cache miss ratio will have to be minimized.

Lock contention: Another source of high latency is lock contention when accessing shared resources. The biggest critical section in the Linux stack is the one used to protect access to the qdisc, done in `dev_queue_xmit`. The overhead of acquiring the qdisc lock is well documented [124, 38]. Increasing the number of flows exacerbates the problem. Figure 5.11 shows that as the time to acquire lock increases by 4 times as the number of flows increases from 1k to 300k. Another factor contributing to the increase in lock acquisition time is the increase in packet rate which we have shown to increase as the number of flows increases (Figure 5.4a). Distributed and lazy coordination between independent multiple queues can help alleviate the problem by reducing the need for locking [38, 56].

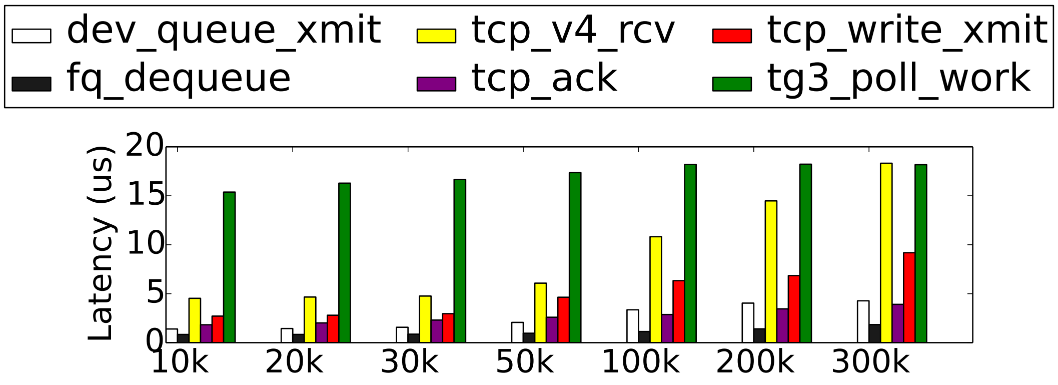
5.5 Summary

In this section, we identify the different bottlenecks that arise when we scale the number of flows to hundreds of thousands in a fully implemented stack. Despite the progress made in improving the performance of network stacks, we have not been able to identify a single, publicly available stack that addresses all the problems we discuss in this section. As we present throughout the section, there have been efforts to address some of the individual problems in isolation. However, integrating and testing such solutions at the scale of hundreds of thousands to millions of long-lived simultaneously-active flows remains an open problem. This is critical for new stacks that support Quic, which are being designed from scratch to support scalable servers [125, 126]. We hope that our work sheds some light on the pain points that stack designers should pay attention to when building next generation stacks that scale to terabits per second and millions of flows.

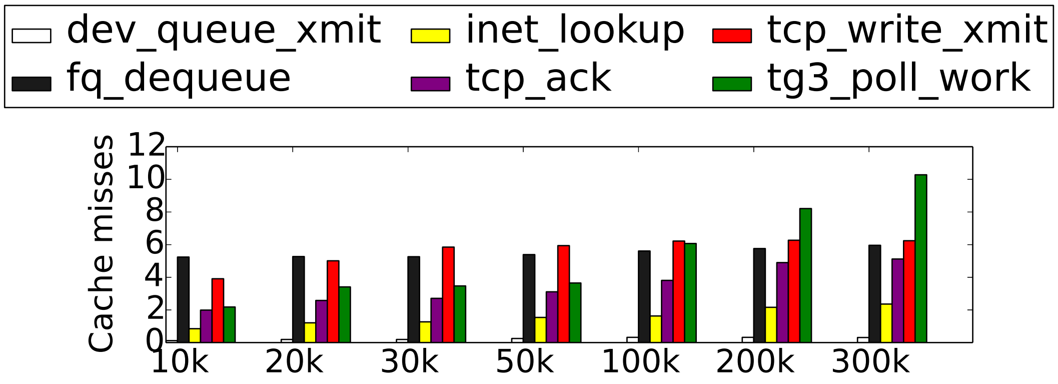
Another observation is that hardware offload solutions are not a panacea. Admission control issues require careful coordination between the software part of the stack, including the application, and the hardware part of the stack. Careful hardware and data structure design can help reduce the latency of complex operations [44]. However, data structure issues do not disappear when implemented in hardware. We hope that this work can highlight some of the needed improvements in the interface between the software and the hardware components of the network stack.



(a) CPU usage



(b) Average function execution time



(c) Cache misses

Figure 5.12: Function profiling results for top functions

CHAPTER 6
CONCLUDING REMARKS

The thesis has identified and addressed a variety of network congestion problems caused by limited bandwidth, buffer space, and CPU resources, both in the Internet and in the data center networks. In summary, the contributions of this thesis are:

A Framework to Mitigate Inter-AS Congestion through CP/ISP Collaboration: We design a framework to be deployed in an access ISP network for joint inter-intra-domain routing. We consider practical deployment issues and evaluate different design choices. We develop a resource allocation strategy that can be deployed by ISPs that maximizes the allocation to the CPs within the ISP capacity constraints while insuring fairness among CP allocations. Our evaluation shows that such framework is beneficial to both CPs and ISPs, improving total throughput of CPs within an ISP and improving ISP throughput. We also show that the benefits of Unison can be achieved even if only a subset of CPs connecting to an ISP agrees to participate.

A Backpressure Mechanism for Congested Queuing System at End Hosts: Congestion of scheduled queues at end hosts typically incurs packet drops which lead to high CPU cost as well as degradation in net- work performance. This work presents design, implementation, and evaluation of a backpressure mechanism for handling congestion of scheduled buffers. We show that by augmenting existing architectures with three simple mechanisms, CPU and network performance can be significantly improved under high loads, improving tail latency by 100x. This work should extend the scalability of current end-host stacks and motivate revisiting the queuing architecture in other network elements.

A Measurement Study on CPU Efficiency with a Focus on Large Number of Flows : CPU efficiency of the networking stack is critical for next generation servers where a single machine serves hundreds of thousands and even a million requests. We conduct a measurement study on end host CPU efficiency to identify the bottlenecks caused by the increasing number of concurrent flows. We categorize problems that arise due to increase of number of concurrent flows into two categories; namely, admitting more packets into the network stack than can be handled efficiently, and increasing per-packet overhead within the stack. We show that these problems contribute to high CPU usage and network performance degradation in terms of aggregate throughput and RTT.

6.1 Future Directions

The work done in this thesis can be extended in several directions:

Mitigating Inter-AS Congestion at Scale: Our work Unison is a first step towards designing an Internet-wide framework to optimize inter-and-intra domain routing with the collaboration between access ISPs and CPs. Our work focuses on a one-ISP-multiple-CPs setting and we designed a framework to be deployed on an access ISP network. A natural extension of the work is to consider a more challenging situation where multiple ISPs and multiple CPs are involved. In this case the content provider needs to synchronize with multiple ISPs, and the scalability of the control architecture will be considered a significant challenge. Unison suggests aggregating flows to limit the scale of the problem in the one-ISP-multiple-CPs setting, but more questions need to be answered as the scale of the problem increases: How can we apply congestion control to the aggregate flows? Should we consider breaking the flow to enforce dynamic congestion control policies?

Applying zD for UDP, Ingress Traffic, and Userspace Stacks: Our zD work focuses on the TCP stack in the kernel, mostly due to the ubiquity of such a setting. As QUIC gains a larger share of Internet traffic, handling backpressure on UDP flows becomes more important. Such backpressure is particularly important because UDP packet drops are physical drops, as UDP does not provide reliability. This puts more stress on the QUIC stack to recover these losses. We do not envision any significant engineering or research challenges extending zD to the UDP stack. The situation is similar for ingress traffic where drops are caused by NIC buffers overwhelming a kernel buffer. We envision that CPU overhead can be saved if zD is applied in such scenarios. The situation is different for userspace Network Stacks (e.g., DPDK). While we believe the building blocks of zD can be mapped to such stacks, we envision that porting it will require engineering effort.

Improving Granularity of Backpressure: In our zD work, the granularity of the scheduled buffer decides the granularity of Backpressure Interface. For example, the backpressure performed from the hypervisor to a VM considers all traffic from the VM as an aggregate flow because packets lose flow-level information when it passes through from the VM to the hypervisor. A future direction is to enforce flow-level backpressure from hypervisor to VMs to increase the granularity of the backpressure mechanism. The challenge is to allow VMs and the hypervisor to share more flow information.

Building a Network Stack for a Million Flows: Our work identifies the bottlenecks of the network stack when the server scales up in terms of number of flows. A promising direction is to build a network stack that is capable of serving a million flows simultaneously. The stack should define APIs that: 1) allow for the implementation of different transport and scheduling algorithms, 2) allow for coordination between different components of the stack that ultimately perform the same task (e.g., determine the number of packets to be processed or transmitted), and 3) provide a scalable way of accessing shared data structures (e.g., scheduled queues and routing tables).

REFERENCES

[1] *Cisco global cloud index 2015–2020*, 2020.

[2] A. Dhamdhere and C. Dovrolis, "The Internet is Flat: Modeling the Transition from a Transit Hierarchy to a Peering Mesh," in *Proc. ACM CoNEXT*, 2010.

[3] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig, "Interdomain Traffic Engineering with BGP," *IEEE Communications magazine*, 2003.

[4] 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 *Proc. ACM SIGCOMM*, 2017.

[5] 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 *Proc. ACM SIGCOMM*, 2017.

[6] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang, "Unreeling Netflix: Understanding and improving multi-CDN movie delivery," in *Proc. of IEEE INFOCOM*, 2012.

[7] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. M. Munafo, and S. Rao, "Dissecting video server selection strategies in the Youtube CDN," in *Proc. of IEEE ICDCS*, 2011.

[8] R. Andrews and S. Higginbotham, *YouTube sucks on French ISP Free, and French regulators want to know why*, (Date last accessed 1-April-2019), 2013.

[9] J. Brodkin, *Time Warner, net neutrality foes cry foul over Netflix Super HD demands*, (Date last accessed 1-April-2019), 2013.

[10] ——, *Why YouTube buffers: The secret deals that make—and break—online video*, (Date last accessed 1-April-2019), 2013.

[11] S. Buckley, *France Telecom and Google entangled in peering fight*, (Date last accessed 1-April-2019), 2013.

[12] C. Dovrolis, "The evolution and economics of internet interconnections," *Submitted to Federal Communications Commission*, 2015.

[13] V. Cerf, V. Jacobson, N. Weaver, and J. Gettys, “BufferBloat: What’s Wrong with the Internet?” *ACM Queue*, vol. 9, 2011.

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

[15] J. LaVoie, E. Nahum, and R. Flynn, “Profiling tcp: An in-depth analysis of processing costs,” *IBM Research Report*, 2007.

[16] H.-W. Jin and C. Yoo, “Impact of protocol overheads on network throughput over high-speed interconnects: Measurement, analysis, and improvement,” *The Journal of Supercomputing*, vol. 41, no. 1, pp. 17–40, 2007.

[17] D. O. Awduche and B. Jabbari, “Internet traffic engineering using multi-protocol label switching (mpls),” *Computer Networks*, vol. 40, no. 1, pp. 111–129, 2002.

[18] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, “Merlin: A language for provisioning network resources,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, ser. CoNEXT ’14, ACM, 2014, pp. 213–226.

[19] V. Heorhiadi, M. K. Reiter, and V. Sekar, “Simplifying Software-Defined Network Optimization Using SOL,” in *Proc. USENIX NSDI*, 2016.

[20] R. Mahajan, D. Wetherall, and T. Anderson, “Negotiation-based routing between neighboring ISPs,” in *Proc. USENIX NSDI*, 2005.

[21] G. Shrimali, A. Akella, and A. Mutapcic, “Cooperative Interdomain Traffic Engineering Using Nash Bargaining and Decomposition,” *IEEE/ACM Transactions on Networking (TON)*, 2010.

[22] P Godfrey, I. Ganichev, S. Shenker, and I. Stoica, “Pathlet Routing,” *Proc. of ACM SIGCOMM*, 2009.

[23] H. Esquivel, C. Muthukrishnan, F. Niu, S. Chawla, and A. Akella, “RouteBazaar: An Economic Framework for Flexible Routing,” *Technical Report TR1654, Department of Computer Sciences*, 2009.

[24] V. Kotronis, R. Klöti, M. Rost, P. Georgopoulos, B. Ager, S. Schmid, and X. Dimitropoulos, “Stitching Inter-domain Paths over IXPs,” in *Proc. ACM SOSR*, 2016.

[25] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, “SDX: A Software Defined Internet Exchange,” in *Proc. ACM SIGCOMM*, 2014.

[26] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever, “An Industrial-Scale Software Defined Internet Exchange Point,” in *Proc. USENIX NSDI*, 2016.

[27] W. Jiang, R. Zhang-Shen, J. Rexford, and M. Chiang, “Cooperative content distribution and traffic engineering in an isp network,” in *ACM SIGMETRICS Performance Evaluation Review*, ACM, vol. 37, 2009, pp. 239–250.

[28] I. Poese, B. Frank, G. Smaragdakis, S. Uhlig, A. Feldmann, and B. Maggs, “Enabling content-aware traffic engineering,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 21–28, 2012.

[29] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann, “Improving content delivery using provider-aided distance information,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, ACM, 2010, pp. 22–34.

[30] M. Wichtlhuber, J. Kessler, S. Bücker, I. Poese, J. Blendin, C. Koch, and D. Hausheer, “Soda: Enabling cdn-isp collaboration with software defined anycast,” in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, IEEE, 2017, pp. 1–9.

[31] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *Prof. of ACM SIGCOMM ’11*, 2011.

[32] 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 *Prof. of USENIX NSDI ’12*, 2012.

[33] 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 *Proc. of ACM SIGCOMM ’15*, 2015.

[34] D. Katabi, M. Handley, and C. Rohrs, “Congestion control for high bandwidth-delay product networks,” in *ACM SIGCOMM computer communication review*, ACM, vol. 32, 2002, pp. 89–102.

[35] N. Dukkipati, “Rate control protocol (rcp): Congestion control to make flows complete quickly,” 2008.

[36] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, “Pfabric: Minimal near-optimal datacenter transport,” in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 43, 2013, pp. 435–446.

[37] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 44, 2014, pp. 307–318.

[38] A. Saeed, N. Dukkipati, V. Valancius, T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End-Hosts," in *Proc. of ACM SIGCOMM '17*, 2017.

[39] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat, "Picnic: Predictable virtualized nic," in *Proc. of ACM SIGCOMM '19*, 2019.

[40] K. He, W. Qin, Q. Zhang, W. Wu, J. Yang, T. Pan, C. Hu, J. Zhang, B. Stephens, A. Akella, and Y. Zhang, "Low latency software rate limiters for cloud networks," in *Proc. of ACM APNet'17*, 2017.

[41] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: scalable NIC for end-host rate limiting," in *Proc. of USENIX NSDI '14*, 2014.

[42] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *Proc. of USENIX NSDI '19*, 2019.

[43] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *Prof. of USENIX NSDI '19*, 2019.

[44] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. of ACM SIGCOMM '19*, 2019.

[45] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *Proc. of USENIX NSDI '19*, 2019.

[46] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proc. of ACM SOSP '17*, 2017.

[47] M. D. et al., "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proc. of USENIX NSDI '18*, 2018.

[48] 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 *Proc. of ACM SIGCOMM '15*, 2015.

[49] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "Hpcc: High precision congestion control," in *Proc. of ACM SIGCOMM '19*, 2019.

[50] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 41, 2011, pp. 50–61.

[51] *Intel DPDK: Data plane development kit*, `https://www.dpdk.org/`, 2014.

[52] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[53] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with flexnic," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 67–81.

[54] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "{ix}: A protected dataplane operating system for high throughput and low latency," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 49–65.

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

[56] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-queue fair queuing," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[57] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "Mtcp: A highly scalable user-level {tcp} stack for multicore systems," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 489–502.

[58] M. Rotaru, F. Olariu, E. Onica, and E. Rivière, "Reliable messaging to millions of users with migratorydata," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, 2017, pp. 1–7.

[59] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, "Tas: Tcp acceleration as an os service," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[60] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "Acceltcp: Accelerating network applications with stateful TCP offloading," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 77–92.

[61] S. Tse and G. Choudhury, "Real-time traffic management in at&t's sdn-enabled core ip/optical network," *Optical Fiber Communication Conference*, 2018.

[62] M. Birk, G. Choudhury, B. Cortez, A. Goddard, N. Padi, A. Raghuram, K. Tse, S. Tse, A. Wallace, and K. Xi, "Evolving to an sdn-enabled isp backbone: Key technologies and applications," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 129–135, 2016.

[63] B. Donnet and O. Bonaventure, "On bgp communities," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 55–59, 2008.

[64] *MANIC: Measurement and ANalysis of Internet Congestion*, (Date last accessed 18-August-2019).

[65] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. P. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy, "Inferring persistent interdomain congestion," in *Proc. ACM SIGCOMM*, 2018.

[66] M. Moradi, Y. Zhang, Z. M. Mao, and R. Manghirmalani, "Dragon: Scalable, flexible, and efficient traffic engineering in software defined isp networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2744–2756, 2018.

[67] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make your Virtual SDNs Programmable," in *Proc. ACM SIGCOMM HotSDN Workshop*, 2014.

[68] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations," in *Proc. ACM SIGCOMM*, 2016.

[69] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz, "Route Flap Damping Exacerbates Internet Routing Convergence," in *ACM SIGCOMM Computer Communication Review*, 2002.

[70] B. Zhang, D. Pei, D. Massey, and L. Zhang, "Timer Interaction in Route Flap Damping," in *Proc. IEEE ICDCS*, 2005.

[71] A. A. Gilroy, "The net neutrality debate: Access to broadband networks," *Congressional Research Service, Washington, DC*, 2019.

[72] "Using the CPLEX Callable Library and CPLEX Mixed Integer Library," *CPLEX Optimization, Incline Village*, 1993.

[73] *The gurobi optimizer*, (Date last accessed 15-May-2018).

[74] O. Tilmans, T. Bühler, S. Vissicchio, and L. Vanbever, “Mille-Feuille: Putting ISP traffic under the Scalpel,” in *Proc. ACM HotNets*, 2016.

[75] J. Jaffe, “Bottleneck Flow Control,” *IEEE Transactions on Communications*, 1981.

[76] M. Demirci and M. Ammar, “Fair Allocation of Substrate Resources among Multiple Overlay Networks,” in *Proc. IEEE MASCOTS*, 2010.

[77] J. Kleinberg, Y. Rabani, and É. Tardos, “Fairness in routing and load balancing,” in *Proc. of IEEE FOCS*, 1999.

[78] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving High Utilization with Software-Driven WAN,” in *Proc. ACM SIGCOMM*, 2013.

[79] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, “The Internet Topology Zoo,” *IEEE Journal on Selected Areas in Communications*, 2011.

[80] M. Roughan, M. Thorup, and Y. Zhang, “Traffic Engineering with Estimated Traffic Matrices,” in *Proc. ACM IMC*, 2003.

[81] M. Zink, K. Suh, Y. Gu, and J. Kurose, “Characteristics of Youtube Network Traffic at a Campus Network–Measurements, Models, and Implications,” *Computer networks*, 2009.

[82] M. M. Waldrop, “The chips are down for moore’s law,” *Nature News*, vol. 530, 2016.

[83] 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*, vol. 28, 2000.

[84] D. Geer, “Chip makers turn to multicore processors,” *IEEE Computer*, vol. 38, 2005.

[85] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, “Extending networking into the virtualization layer,” in *Proc. of ACM HotNets-VIII*, 2009.

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

[87] *Intel DPDK: Data plane development kit*, `https://www.dpdk.org/`, 2014.

[88] W. Almesberger, J. H. Salim, and A. Kuznetsov, “Differentiated services on linux,” in *Proc. of IEEE GLOBECOM ’99*, 1999.

[89] R. Russell, “virtio: towards a de-facto standard for virtual I/O devices,” *ACM SIGOPS Operating Systems Review*, vol. 42, 2008.

[90] M Tsirkin, “vhost-net and virtio-net: Need for Speed,” in *Proc. KVM Forum*, 2010.

[91] 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 *Proc. of USENIX NSDI ’15*, 2015.

[92] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, “Understanding TCP incast throughput collapse in datacenter networks,” in *Proc. of the ACM workshop on Research on enterprise networking (WREN ’09)*, 2009.

[93] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, “EyeQ: Practical Network Performance Isolation at the Edge,” in *Proc. of USENIX NSDI ’13*, 2013.

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

[95] A. N. Kuznetsov, *pfifo-tc: PFIFO Qdisc*, `https://linux.die.net/man/8/tc-pfifo/`, 2002.

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

[97] Y. Cheng and N. Cardwell, “Making linux tcp fast,” in *Netdev 1.2 Conference*, 2016.

[98] H.-k. J. Chu, “Zero-copy TCP in Solaris,” in *Proc. of USENIX ATC ’96*, 1996.

[99] J. Tan, C. Liang, H. Xie, Q. Xu, J. Hu, H. Zhu, and Y. Liu, “VIRTIO-USER: A New Versatile Channel for Kernel-Bypass Networks,” in *Proc. of the ACM Workshop on Kernel-Bypass Networks (KBNets ’17)*, 2017.

[100] T. Zhang, J. Wang, J. Huang, J. Chen, Y. Pan, and G. Min, “Tuning the aggressive tcp behavior for highly concurrent http connections in intra-datacenter,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3808–3822, 2017.

[101] *High-performance, feature-rich netxtreme® e-series dual-port 100g pcie ethernet nic*, `https://www.broadcom.com/products/ethernet-connectivity/network-adapters/100gb-nic-ocp/p2100g`.

[102] "Ieee standard for ethernet - amendment 10: Media access control parameters, physical layers, and management parameters for 200 gb/s and 400 gb/s operation," *IEEE Std 802.3bs-2017*, pp. 1–372, 2017.

[103] *IEEE 802.3 Industry Connections Ethernet Bandwidth Assessment Part II*, 2020.

[104] *Microprocessor trend data*, `https://github.com/karlrupp/microprocessor-trend-data`, 2018.

[105] *Netflix Help Center: Internet Connection Speed Recommendations*, `https://help.netflix.com/en/node/306`, 2020.

[106] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 1–16.

[107] M. Hock, M. Veit, F. Neumeister, R. Bless, and M. Zitterbart, "Tcp at 100 gbit/s–tuning, limitations, congestion control," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, IEEE, 2019, pp. 1–9.

[108] J. D. Brouer, "Network stack challenges at increasing speeds," in *Proc. Linux Conf*, 2015, pp. 12–16.

[109] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, *et al.*, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ACM, 2019, pp. 399–413.

[110] Y. Zhao, A. Saeed, E. Zegura, and M. Ammar, "Zd: A scalable zero-drop network stack at end hosts," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 220–232.

[111] *neper: a Linux networking performance tool*, `https://github.com/google/neper`, 2020.

[112] F. R. P. Cavalcanti and S. Andersson, *Optimizing wireless communication systems*. Springer, 2009, vol. 386.

[113] Q.-C. Chen, X.-H. Yang, and X.-L. Wang, "A peer-to-peer based passive web crawling system," in *2011 International Conference on Machine Learning and Cybernetics*, IEEE, vol. 4, 2011, pp. 1878–1883.

[114] *FlowQueue-Codel*, `https://tools.ietf.org/id/draft-ietf-aqm-fq-codel-02.html`, 2020.

[115] *dstat-Linux man page*, `https://linux.die.net/man/1/dstat`, 2020.

[116] *ss-Linux man page*, `https://linux.die.net/man/8/ss`, 2020.

[117] *netstat-Linux man page*, `https://linux.die.net/man/8/netstat`, 2020.

[118] *Tcpdump Manual*, `https://linux.die.net/man/8/tcpdump`, 2020.

[119] *Perf Manual*, `https://www.man7.org/linux/man-pages/man1/perf.1.html`, 2020.

[120] J. C. Mogul and K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[121] L. Sun and P. Kostic, *Adaptive hardware interrupt moderation*, US Patent App. 13/534,607, 2014.

[122] Y. Li, L. Cornett, M. Deval, A. Vasudevan, and P. Sarangam, *Adaptive interrupt moderation*, US Patent 9,009,367, 2015.

[123] C. Benvenuti, *Understanding Linux network internals.* " O'Reilly Media, Inc.", 2006.

[124] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "{senic}: Scalable {nic} for end-host rate limiting," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 475–488.

[125] 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, 2017.

[126] J Iyengar and M Thomson, "Quic: A udp-based multiplexed and secure transport; draft-ietf-quic-transport-24," *Internet Engineering Task Force: Newark, DE, USA*, 2019.