Improving Datacenter Network Performance via Intelligent Network Edge

By

Keqiang He

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: June 8, 2017

The dissertation is approved by the following members of the Final Oral Committee:

Aditya Akella, Professor, Computer Sciences
Suman Banerjee, Professor, Computer Sciences
Eric Rozner, Research Staff Member, IBM Research
Michael Swift, Associate Professor, Computer Sciences
Xinyu Zhang, Assistant Professor, Electrical & Computer Engineering

Abstract

Datacenter networks are critical building blocks for modern cloud computing infrastructures. The datacenter network directly impacts the performance of applications and services running in datacenters. Today's applications are becoming more and more demanding, requiring low latency, high throughput and low packet loss rate. Thus improving datacenter network performance is both timely and important.

Datacenter networks are complicated systems with many functionalities and components spread across hardware and software. Any suboptimal functionality or component can significantly degrade network performance and affect applications. In this dissertation, we show that simple software-only solutions can go a long way to ensuring good datacenter network performance, specifically, we show how we can leverage the flexibility and high programmability of software-defined network edge (i.e., end-host networking) [101, 102] to improve the performance of three key functionalities in datacenter networks — traffic load balancing, congestion control and rate limiting.

We start from low layers and move up the stack. We first look into traffic load balancing functionality in datacenter networks. Modern datacenter networks need to deal with a variety of workloads, ranging from latency-sensitive small flows to bandwidth-hungry large flows. In-network hardware-based load balancing schemes which are based on flow hashing, e.g., ECMP, cause congestion when hash collisions occur and can perform

poorly in asymmetric topologies. Recent proposals to load balance the network require centralized traffic engineering, multipath-aware transport, or expensive specialized hardware. We propose a pure software-edge-based mechanism that avoids these limitations by (i) pushing load-balancing functionality into the soft network edge (e.g., virtual switches) such that no changes are required in the transport layer, customer VMs, or networking hardware, and (ii) load balancing on fine-grained, near-uniform units of data (flowcells) that fit within end-host segment offload optimizations used to support fast networking speeds. We design and implement such a soft-edge load balancing scheme called Presto, and evaluate it on a 10 Gbps physical testbed. We demonstrate the computational impact of packet reordering on receivers and propose a mechanism to handle reordering in the TCP receive offload functionality. Presto's performance closely tracks that of a single, non-blocking switch over many workloads and is adaptive to failures and topology asymmetry.

Optimized traffic load balancing alone is not sufficient to guarantee high-performance datacenter networks. Virtual Machine (VM) technology plays an integral role in modern multi-tenant clouds by enabling a diverse set of software to be run on a unified underlying framework. This flexibility, however, comes at the cost of dealing with outdated, inefficient, or misconfigured TCP stacks implemented in the VMs. We investigate if cloud providers can take control of a VM's TCP congestion control algorithm without making changes to the VM or network hardware. Again, we leverage the flexibility of software network edge and propose a congestion control virtualization technique called AC/DC TCP. AC/DC TCP exerts fine-grained control over arbitrary tenant TCP stacks by enforcing per-flow congestion control in the virtual switch (vSwitch) in the hypervisor. AC/DC TCP is light-weight, flexible, scalable and can police non-conforming flows. Our experiment results demonstrate that implementing an administrator-defined congestion control algorithm in the vSwitch (i.e.,

DCTCP [9]) closely tracks its native performance, regardless of the VM's TCP stack.

Presto and AC/DC TCP help reduce queueing latency in network switches (i.e., "in network" latency), but we observe that rate limiters on end-hosts can also increase network latency by an order of magnitude or even more. Rate limiters are employed to provide bandwidth allocation functionality, which is an indispensable feature of multi-tenant clouds. Rate limiters maintain a queue of outstanding packets and control the speed at which packets are dequeued into the network. This queueing introduces additional network latency. For example, in our experiments, we find that software rate limiting (HTB) increases latency by 1-3 milliseconds across a range of different environment settings. To solve this problem, we extend ECN marking into rate limiters and use a datacenter congestion control algorithm (DCTCP). Unfortunately, while this reduces latency, it also leads to throughput oscillation. Thus, this solution is not sufficient. To this end, we propose two techniques — DEM and SPRING to improve the performance of rate limiters. Our experiment results demonstrate that DEM and SPRING-enabled rate limiters can achieve high stable throughput and low latency.

Presto load balances the traffic generated by the endpoints (i.e., VMs, containers or bare-metal servers) as evenly as possible and minimize the possibility of network congestion. AC/DC TCP reduces TCP sender's speed when congestion happens in the network. Both Presto and AC/DC TCP reduce queueing latency in switches. DEM and SPRING reduce the latency caused by rate limiters on the end-hosts. They are complementary and can work together to ensure low end-to-end latency, high throughput and low packet loss rate in datacenter networks.

To my family

Acknowledgments

Pursuing a Ph.D. degree is one of my childhood dreams. Now I am finishing my Ph.D. study. I still remember the day in August 2012 when I left Beijing and landed Seattle. That was the first time I had been to a different country. Then I came to Madison to start my Ph.D. journey. Five years have passed, I owe a lot to the people who help me in this journey.

First, I would like to thank my advisor, Professor Aditya Akella for his help in my research and life. Aditya helps me identify interesting and important research problems. Finding research directions is not easy for new graduate students. When I just started my Ph.D., I was not sure what I was going to do and just read papers here and there. Aditya guides to explore the correct directions. Besides pointing out promising research directions, Aditya also teaches me the attitude of conducting solid research. Now I feel the attitude is even more important than the research itself. Aditya also gives me a lot of freedom in doing research in these 5 years. I feel lucky to have Aditya as my Ph.D. advisor.

Second, I need to thank Professor Suman Banerjee, Dr. Eric Rozner, Professor Mike Swift, and Professor Xinyu Zhang for being my committee members. I really appreciate their time in reading my Ph.D. thesis and give me constructive suggestions to help improve the quality of this thesis. Professor Suman Banerjee and Professor Mike Swift also served on my preliminary examination committee in 2016. Their questions and feedback in my prelim examination greatly helped me define the scope of this thesis.

I am fortunate to work with all of my fantastic mentors and collabo-

rators during my Ph.D. study. I would like to thank Professor Thomas Ristenpart. Tom helped me a lot on my first project, which was published in the ACM Internet Measurement Conference 2013. He is an amazing person and very humorous. I still remember the winter nights when we worked together for paper deadlines. I also want to thank Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, and John Carter at IBM Research, the Austin Research Laboratory. They helped me write papers, conduct experiments and even debug code with me. I spent two summers in Austin, Texas (in 2015 and 2016 respectively). Austin was such a great place to live and work — I remember the weather was extremely nice and we went out searching for delicious food together. I also want to thank Radhika Mysore, Pavan Konanki, Junlan Zhou, and Rui Wang at Google. I spent nearly 7 months at Mountain View, California. Radhika and Pavan discussed project ideas with me almost every week and helped review my code. Junlan and Rui helped me build a cool tool to analyze datacenter network performance. It was a really nice experience to work with them. I also need to thank Marina Thottan and Li Erran Li at Bell Labs. The summer at Bell Labs was joyful and that was my first internship. I want to thank all my collaborators of the research projects I worked on at UW-Madison: Sourav Das, Alexis Fisher, Aaron Gember-Jacobson, Junaid Khalid, Chaithan Prakash, Liang Wang, and Wenfei Wu.

I also want to thank my peers, friends and labmates at UW-Madison: Yizheng Chen, Ram Durairajan, Soudeh Ghorbani, Robert Grandl, Lei Kang, Yanfang Le, Xiujun Li, Guangyu Liu, Peng Liu, Lanyue Lu, Bozhao Qi, Shan-Hsiang Shen, Brent Stephens, Meenakshi Syamkumar, David Tran-Lam, Raajay Vishwanathan, Suli Yang, Yan Zhai, Tan Zhang, Xuan Zhang, and Shengqi Zhu. With them, life at Madison becomes more colorful.

Finally, I thank all my family members: my father, mother, brother, sister-in-law and my lovely nephew and niece. My nephew was born just

before I started my Ph.D. study. While I am doing my Ph.D., he grows up to be an adorable boy. My niece was born when I was in my fourth year in Madison, and I have not had the chance to see her in person till now. When I write this thesis, I also recall my hometown, my childhood, the cold winter mornings when I went to elementary schools, the Spring Festivals when our family members gathered together. My parents always cook delicious food for me when I get home or leave for school. They stay with me and support me no matter whether I succeed or fail. Thank you for your love!

Contents

List of Figures and Tables

1 Introduction

Cloud computing has changed the way computing is conducted. It is a rapidly growing business and many industry leaders (e.g., Amazon [12], Google [44], IBM [57, 58] and Microsoft [83]) have embraced such a business model and are deploying highly advanced cloud computing infrastructures. Market analysis [33] has predicted that the global cloud computing market will reach $270 billion by 2020. The success of cloud computing is not accidental — it is rooted in many advantages cloud computing offers over traditional computing model. The most notable feature is that tenants (customers) who rent the computing resources (e.g., CPUs, memory, storage, and network) can get equivalent computing power with *lower cost*. This is because the computing resources are shared among multiple users and server consolidation and server virtualization improve the utilization of the computing resources. Another key advantage cloud computing offers is *computing agility*. That means, tenants can rent as many computing resources as they need and can grow or shrink the computing pool based on their demands in an elastic manner. This feature is especially attractive for relatively smaller and rapidly growing businesses.

Datacenter networks are important components in modern cloud computing infrastructures. High-performance cloud computing infrastructures require high-speed, low latency, scalable and highly robust datacenter networking solutions to support a massive amount of traffic. Cisco Global Cloud Index [32] predicts that the annual global datacenter IP traffic will reach 15.3 zettabytes (ZB) by 2020, which is 3 times as large as

2015's (4.7 ZB). The tremendous growth of datacenter traffic drives the need for high-performance datacenter networking solutions.

The datacenter network is a complicated system and it covers many aspects of computer networking, ranging from TCP congestion control algorithms to switch hardware design. In the past 10 years, datacenter networking technologies have advanced significantly. For example, starting in 2009 - 2010, seminal works on datacenter network topology designs such as FatTree [6] and VL2 [46] were published. These works proposed to use multi-stage Clos networks to scale out and support hundreds of thousands of servers in a single datacenter. Recently (in 2016), Microsoft published their congestion control solution for RDMA deployments in Azure networks [136]. Despite these advances, there are still a lot of unsolved research challenges in datacenter networking.

In virtualized datacenters such as Amazon Web Service, Google Cloud Platform and Microsoft Azure, multiple Virtual Machines (VMs) or containers run on the same physical server and these VMs/containers are connected to the virtual switch (e.g., Open vSwitch [102]) in the virtualization layer. Typically all the servers (and hypervisors) in the datacenter are managed by a single entity (i.e., the cloud provider) and the software network edge can easily manage and modify traffic going out to the datacenter network core. Also, software network edge is flexible and highly programmable. Therefore, there are a lot of opportunities for innovations at the datacenter software network edge.

In this thesis, I will present our research work on improving datacenter networking performance. The major theme of this thesis is to leverage the intelligent software-defined network edge (i.e., end-host networking) to improve the performance (e.g., network throughput, latency and packet loss) for datacenter networks. In the following of this chapter, I will briefly introduce three research projects I worked on during my Ph.D. study — edge-based traffic load balancing for datacenter networks, congestion

control virtualization for multi-tenant clouds and low latency software rate limiters for cloud networks. Each of them improves the performance of one key functionality of datacenter networking and they can work together to ensure low latency, high throughput and low packet loss rate datacenter networks.

1.1 Edge-based Traffic Load Balancing

Traffic load balancing is an important functionality in datacenter networks. The goal of traffic load balancing is to minimize traffic imbalance on different network links and avoid network congestion as much as possible. The state-of-the-art approach to traffic load balancing in datacenter networks is called Equal Cost Multipathing (ECMP). When a data packet arrives at a switch, there are many paths to the destination. The switch applies a hash function to several fields in the packet header, for example, source IP address, destination IP address, transport protocol, source port number and destination port number. Based on the hash value, the switch chooses one of the potential paths. ECMP can lead to traffic imbalance and does not work well in asymmetric networks, so Weighted Cost Multipathing (WCMP) was proposed [135].

Datacenter networks need to support various kinds of network traffic generated by a diverse set of applications and services running in the datacenters. For example, flows generated by search, email, query and remote procedure calls tend to be short and small, and we call such flows mice flows. On the other hand, flows generated by big data ingestion and data backup tend to be long and large, so we call such flows elephant flows. A classic problem of ECMP and WCMP is that if two or more elephant flows are hashed onto the same network path, then *network congestion occurs because of elephant hash collision* [7, 106]. Elephant hash collision leads to two types of performance issues. First, elephant flows' throughputs are

reduced. Second, mice flows suffer from head-of-line blocking issue and their latency can be increased to tens of milliseconds [10]. Note that the baseline TCP RTT in the datacenter network environment is around 200 microseconds [56]. To solve this problem, we propose Presto. Presto has two novel components. At the sender side, we utilize the virtual switch (e.g., OVS) in the hypervisor to break elephant flows into small chunks called flowcells. A flowcell consists of several TCP segments and its maximum size is bounded to be the maximum TCP Segment Offload (TSO) [3] size, which is 64KB by default in Linux. A mice flow whose size is smaller than or equal to 64KB is one flowcell and its packets go through the same network path. So there is no packet reordering issue for flows whose sizes are smaller than or equal to 64KB. For the flows that are larger than 64KB, packets may go through different paths and packet reordering issue may arise if the congestion level on different network paths is different. Therefore at the receiver side, we modify the Generic Receive Offload (GRO) functionality in the network stack to mask packet reordering for TCP. Because Presto eliminates elephant flow collision problem and masks packet reordering for large flows below TCP layer in GRO, the performance of traffic load balancing is greatly improved. We will discuss the details of Presto in Chapter 2.

1.2 Congestion Control Virtualization

As observed by some of the largest datacenter network operators such as Google [112], network congestion is not rare in datacenter networks. Network congestion significantly inflates latency in datacenter networks, especially the tail latency. It has been observed that the 99.9th percentile latency is orders of magnitude higher than the median latency in datacenter networks [86] and queueing latency is believed to be the major contributor of the tail latency [62]. To this end, a lot of datacenter TCP congestion

control algorithms have been proposed in recent years, e.g., DCTCP [9], TIMELY [84], DCQCN [136], TCP-Bolt [117], and ICTCP [129].

Despite the recent advances in datacenter network congestion control, we still face a practical unsolved problem. Most of today's public clouds, e.g., Google Cloud Platform, Microsoft Azure and Amazon Web Services (AWS), are multi-tenant clouds. The computing resources (e.g., CPUs, memory, storage and network) are rented to tenants (i.e., customers) in the form of Virtual Machines (VMs). The practical problem is that cloud providers are not able to manage or configure the congestion control algorithms used by the VMs' TCP/IP stacks. Tenants can manage and configure their own TCP/IP stacks. So VM TCP congestion control algorithms can be outdated, inefficient or even misconfigured. Those outdated, inefficient or misconfigured TCP/IP stacks cause network congestion and fairness issues for the datacenter network. Therefore, we investigate if administrators can take control of a VM's TCP congestion control algorithm without making changes to the VM or network hardware. We propose AC/DC TCP, a scheme that exerts fine-grained control over arbitrary tenant TCP stacks by enforcing per-flow congestion control in the virtual switch (vSwitch). Our scheme is light-weight, flexible, scalable and can police non-conforming flows. In our evaluation the computational overhead of AC/DC TCP is less than one percentage point and we show implementing an administrator-defined congestion control algorithm in the vSwitch (i.e., DCTCP) closely tracks its native performance, regardless of the VM's TCP stack. We will discuss the details of AC/DC TCP in Chapter 3.

1.3 Low Latency Software Rate Limiters

The ability to create bandwidth allocations is an indispensable feature of multi-tenant clouds. Bandwidth allocations can be used to provide bandwidth reservations to a tenant or to guarantee that network bandwidth

is fairly shared between multiple competing tenants [63, 109, 111]. Bandwidth allocations are often implemented with *software rate limiters* running in the hypervisor or operating system of the end-hosts attached to the network (e.g., Linux Hierarchical Token Bucket, aka HTB). This is because software rate limiters are flexible and scalable. Unfortunately, typical software rate limiters (e.g., HTB) also increase network latency by adding an additional layer of queuing for packets. To be able to absorb bursts of incoming packets while also ensuring that network traffic does not exceed the configured rate, rate limiters maintain a queue of outstanding packets and control the speed at which packets are dequeued into the network. This queuing introduces additional network latency. For example, in our experiments, we find that software rate limiting (HTB) increases latency by 1-3 milliseconds across a range of different environment settings. This increase in latency is about an order of magnitude higher than the base latency of the network (200 microseconds). In multi-tenant clouds, this additional queuing latency can increase flow completion times, leading to possible service-level agreement (SLA) violations [126].

Inspired by recent work that reduces queuing delay for in-network devices like switches [9, 56, 84, 136], we explore how to use a congestion-control-based approach to address the latency issues associated with using software rate limiters. As a promising first step, we find that the existing datacenter congestion control protocol DCTCP [9] can be used to reduce the latency incurred by rate limiters. Unfortunately, we find that a straightforward application of DCTCP (we call it DCTCP+ECN) to software rate limiters also hurts throughput because of two problems unique to end-host networking. First, different from hardware switches in the network, end-hosts process TCP segments instead of MTU-sized packets. TCP Segmentation Offload (TSO) [3] is an optimization technique that is widely used in modern operating systems to reduce CPU overhead for fast speed networks. Because Linux has difficult driving 10Gbps (and

beyond) line-rates when TSO is not enabled, Linux uses a TSO size of 64KB by default. That means that marking the ECN bit in one 64KB segment causes 44 consecutive MTU-sized packets to have the ECN bit marked. This is because the ECN bits in the segment header are copied into each packet by the NIC. Oppositely, if a TCP segment is not marked, none of the packets in this segment is marked. This kind of coarse-grained segment-level marking leads to an inaccurate estimation of congestion level which consequently leads to throughput oscillation. The second problem with DCTCP+ECN is that the ECN mark takes one round-trip time (RTT) to get back to the source. Because of this, the congestion window computation at the source uses a stale value from one RTT ago. As a result, congestion cannot be detected at early stage, and the congestion level would be exacerbated during this one-RTT delay. To solve the problems, we present two techniques — DEM and SPRING. DEM directly sets TCP ACK's TCP ECE (Echo-Echo) bit based on real time rate limiter queue length information. In this way, congestion control loop latency is reduced to almost 0. Also coarse-grained segment-level marking is avoided. SPRING runs a queue-length-based congestion control algorithm and enforces congestion control decisions via modifying the RWND field in the TCP header. Similar to DEM, SPRING also avoids the two shortcomings of DCTCP+ECN. Compared with DEM, SPRING modifies RWND to enforce congestion control and does not rely on ECN support, so is more generic and can handle both ECN and non-ECN flows. We will discuss the details of low latency software rate limiters in Chapter 4.

1.4 Summary of Contributions and Overview

At a high level, we show that using the intelligent software network edge, we can improve the performance of key functionalities of datacenter networks. In particular, the specific contributions of this dissertation are

summarized in the following. This section also serves as an outline for the rest of this dissertation.

- Edge-based Datacenter Traffic Load Balancing. We propose two novel techniques. The first is to utilize the virtual switch (e.g., OVS) in the hypervisor to chunk flows into bounded-sized flowcells. The second is to leverage the Generic Receive Offload (GRO) functionality in the networking stack to mask packet reordering for TCP layer. To the best of our knowledge, we are the first to present these two techniques in the literature. Based on these two techniques, we build a datacenter traffic load balancing system called Presto and evaluate its performance on a real 10G testbed. Our experiment results demonstrate that Presto improves network performance significantly. We will cover the details of Presto in Chapter 2.

- Congestion Control Virtualization for Datacenter Networks. We propose AC/DC TCP, a technique that utilizes the virtual switch in the hypervisor to provide congestion control virtualization for multi-tenant clouds. To the best of our knowledge, we are one of the first two research teams[1] in the world to propose the congestion control virtualization technique for cloud networks. We validate its feasibility on a real testbed and demonstrate that it closely tracks the performance of the native congestion control algorithm (e.g., DCTCP). Chapter 3 covers the details of AC/DC TCP.

- Low Latency Software Rate Limiters. Rate limiters are important components for multi-tenant clouds. We conduct experiments to show that network latency can be increased by an order of magnitude by software rate limiters. We extend ECN into rate limiter queues and apply DCTCP to reduce latency. However, we find such

[1]The other team is from Stanford, VMware and Israel Institute of Technology [34]; we invented the same technique independently.

a straightforward solution causes TCP throughput oscillation. We identify the reasons and propose two techniques (i.e., DEM and SPRING) to address of shortcoming of the straightforward solution. Our experiments demonstrate that DEM and SPRING enable low latency software rate limiters. The research work on software rate limiters is discussed in Chapter 4.

- Related Work. We present the related work of this dissertation in Chapter 5. It covers related work in the following categories: datacenter network traffic load balancing, reducing tail latency, handling packet reordering, congestion control for datacenter networks, bandwidth allocation and rate limiters for multi-tenant clouds.

- Conclusion and Future Work. We conclude this dissertation in Chapter 6. We believe that the techniques and mechanisms presented in this dissertation will be valuable to the computer networking research community and industry. Finally, we discuss several future research topics in datacenter networking area.

2

Edge-based Load Balancing for Fast Datacenter Networks

2.1 Introduction

Datacenter networks must support an increasingly diverse set of workloads. Small latency-sensitive flows to support real-time applications such as search, RPCs, or gaming share the network with large throughput-sensitive flows for video, big data analytics, or VM migration. Load balancing the network is crucial to ensure operational efficiency and suitable application performance. Unfortunately, popular load balancing schemes based on flow hashing, *e.g.*, ECMP, cause congestion when hash collisions occur [7, 27, 29, 38, 105, 106, 133] and perform poorly in asymmetric topologies [8, 135].

A variety of load balancing schemes aim to address the problems of ECMP. Centralized schemes, such as Hedera [7] and Planck [106], collect network state and reroute elephant flows when collisions occur. These approaches are fundamentally reactive to congestion and are very coarse-grained due to the large time constraints of their control loops [7] or require extra network infrastructure [106]. Transport layer solutions such as MPTCP [128] can react faster but require widespread adoption and are difficult to enforce in multi-tenant datacenters where customers often deploy customized VMs. In-network reactive distributed load balancing schemes, *e.g.*, CONGA [8] and Juniper VCF [54], can be effective but require

specialized networking hardware.

The shortcomings of the above approaches cause us to re-examine the design space for load balancing in datacenter networks. ECMP, despite its limitations, is a highly practical solution due to its proactive nature and stateless behavior. Conceptually, ECMP's flaws are not internal to its operation but are caused by asymmetry in network topology (or capacities) and variation in flow sizes. *In a symmetric network topology where all flows are "mice", ECMP should provide near optimal load balancing*; indeed, prior work [8, 113] has shown the traffic imbalance ECMP imposes across links goes down with an increase in the number of flows and a reduction in the variance of the flow size distribution.

Can we leverage this insight to design a good proactive load balancing scheme without requiring special purpose hardware or modifications to end-point transport? The system we propose answers this in the affirmative. It relies on the datacenter network's *software edge* to transform arbitrary sized flows into a large number of near uniformly sized small sub-flows and proactively spreads those uniform data units over the network in a balanced fashion. Our scheme is fast (works at 10+ Gbps) and doesn't require network stack configurations that may not be widely supported outside the datacenter (such as increasing MTU sizes). We piggyback on recent trends where several network functions, *e.g.*, firewalls and application-level load balancers, are moving into hypervisors and software virtual switches on end-hosts [16, 73, 101]. Our work makes a strong case for moving network load balancing functionality out of the datacenter network hardware and into the software-based edge.

Several challenges arise when employing the edge to load balance the network on a sub-flow level. Software is slower than hardware, so operating at 10+ Gbps speeds means algorithms must be simple, lightweight, and take advantage of optimizations in the networking stack and offload features in the NIC. Any sub-flow level load balancing should also

be robust against reordering because packets from the same flow can be routed over different network paths which can cause out-of-order delivery. As shown in Section 2.2, reordering not only impacts TCP's congestion control mechanism, but also imposes significant computational strain on hosts, effectively limiting TCP's achievable bandwidth if not properly controlled. Last, the approach must be resilient to hardware or link failures and be adaptive to network asymmetry.

To this end, we build a proactive load balancing system called Presto. Presto utilizes edge vSwitches to break each flow into discrete units of packets, called *flowcells*, and distributes them evenly to near-optimally load balance the network. Presto uses the maximum TCP Segment Offload (TSO) size (64 KB) as flowcell granularity, allowing for fine-grained load balancing at network speeds of 10+ Gbps. To combat reordering, we modify the Generic Receive Offload (GRO) handler in the hypervisor OS to mitigate the computational burden imposed by reordering and prevent reordered packets from being pushed up the networking stack. Finally, we show Presto can load balance the network in the face of asymmetry and failures.

This chapter makes the following contributions:

1. We design and implement a system, called Presto, that near-optimally load balances links in the network. We show such a system can be built with no changes to the transport layer or network hardware and scales to 10+ Gbps networking speeds. Our approach makes judicious use of middleware already implemented in most hypervisors today: Open vSwitch and the TCP receive offload engine in the OS (Generic Receive Offload, GRO, in the Linux kernel).

2. We uncover the importance of GRO on performance when packets are reordered. At network speeds of 10+ Gbps, current GRO algorithms are unable to sustain line rate under severe reordering due

to extreme computational overhead, and hence per-packet load balancing approaches [27, 38] need to be reconsidered. We improve GRO to prevent reordering while ensuring computational overhead is limited. We argue GRO is the most natural place to handle reordering because it can mask reordering in a light-weight manner while simultaneously limiting CPU overhead by having a direct impact on the segment sizes pushed up the networking stack. In addition, our scheme distinguishes loss from reordering and adapts to prevailing network conditions to minimize the time to recover lost packets.

3. Presto achieves near-optimal load balancing in a proactive manner. For that, it leverages symmetry in the network topology to ensure that all paths between a pair of hosts are equally congested. However, asymmetries can arise due to failures. We demonstrate Presto can recover from network failures and adapt to asymmetric network topologies using a combination of fast failover and weighted multipathing at the network edge.

4. Finally, we evaluate Presto on a real 10 Gbps testbed. Our experiments show Presto outperforms existing load balancing schemes (including flowlet switching, ECMP, MPTCP) and is able to track the performance of a single, non-blocking switch (an optimal case) within a few percentage points over a variety of workloads, including trace-driven. Presto improves throughput, latency and fairness in the network and also reduces the flow completion time tail for mice flows.

2.2 Design Decisions and Challenges

In Presto, we make several design choices to build a highly robust and scalable system that provides near optimal load balancing without requir-

ing changes to the transport layer or switch hardware. We now discuss our design decisions.

2.2.1 Design Decisions

Load Balancing in the Soft Edge A key design decision in Presto is to implement the functionality in the soft edge (*i.e.*, the vSwitch and hypervisor) of the network. The vSwitch occupies a unique position in the networking stack in that it can easily modify packets without requiring any changes to customer VMs or transport layers. Functionality built into the vSwitch can be made aware of the underlying hardware offload features presented by the NIC and OS, meaning it can be fast. Furthermore, an open, software-based approach prevents extra hardware cost and vendor lock-in, and allows for simplified network management. These criteria are important for providers today [88]. Thanks to projects like Open vSwitch, soft-switching platforms are now fast, mature, open source, adopted widely, remotely configurable, SDN-enabled, and feature-rich [73, 100, 102]. Presto is built on these platforms.

Reactive vs Proactive Load Balancing The second major design decision in Presto is to use a proactive approach to congestion management. Bursty behavior can create transient congestion that must be reacted to before switch buffers overflow to prevent loss (timescales range from 100s of µs to around 4 ms [106]). This requirement renders most of the centralized reactive schemes ineffective as they are often too slow to react to any but the largest network events, *e.g.*, link failures. Furthermore, centralized schemes can hurt performance when rerouting flows using stale information. Distributed reactive schemes like MPTCP [128] and CONGA [8] can respond to congestion at faster timescales, but have a high barrier to deployment. Furthermore, distributed reactive schemes must take great care to avoid oscillations. Presto takes a proactive, correct-by-design approach

to congestion management. That is, if small, near-uniform portions of traffic are equally balanced over a symmetric network topology, then the load-balancing can remain agnostic to congestion and leave congestion control to the higher layers of the networking stack. Presto is only reactive to network events such as link failures. Fortunately, the larger timescales of reactive feedback loops are sufficient in these scenarios.

Load Balancing Granularity ECMP has been shown to be ineffective at load balancing the network, and thus many schemes advocate load balancing at a finer granularity than a flow [8, 27, 38, 54]. A key factor impacting the choice of granularity is operating at high speed. Operating at 10+ Gbps incurs great computational overhead, and therefore host-based load balancing schemes must be fast, light-weight and take advantage of optimizations provided in the networking stack. For example, per-packet load balancing techniques [27] cannot be employed at the network edge because TSO does not work on a per-packet basis. TSO, commonly supported in OSes and NICs, allows for large TCP segments (typically 64 KB in size) to be passed down the networking stack to the NIC. The NIC breaks the segments into MTU-sized packets and copies and computes header data, such as sequence numbers and checksums. When TSO is disabled, a host incurs 100% utilization of one CPU core and can only achieve around 5.5 Gbps [70]. Therefore, per-packet schemes are unlikely to scale to fast networks without hardware support. Limiting overhead by increasing the MTU is difficult because VMs, switches, and routers must all be configured appropriately, and traffic leaving the datacenter must use normal 1500 byte packets. Furthermore, per-packet schemes [27, 38] are likely to introduce significant reordering into the network.

Another possibility is to load balance on flowlets [8, 54]. A flow is comprised of a series of bursts, and a flowlet is created when the inter-arrival time between two packets in a flow exceeds a threshold inactivity timer. In practice, inactivity timer values are between 100-500 μs [8]. These

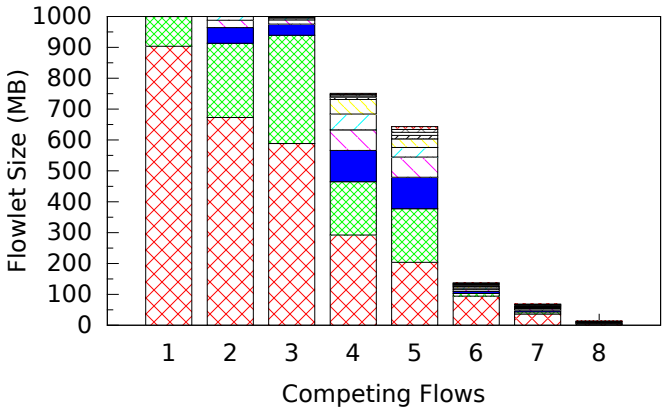


Figure 2.1: Stacked histogram of flowlet sizes (in MB) for a 1 GB scp file transfer. We vary the number of nuttcp [94] background flows and denote them as *Competing Flows*. The size of each flowlet is shown within each bar, and flowlets are created whenever there is a 500 µs delay between segments. The top 10 flowlet sizes are shown here. We also analyzed the results of a 1 GB nuttcp, ftp, and a simple custom client/server transfer and found them to be similar.

values intend to strike a good balance between load balancing on a sub-flow level and acting as a buffer to limit reordering between flowlets. Flowlets are derived from traffic patterns at the sender, and in practice this means the distribution of flowlet sizes is not uniform. To analyze flowlet sizes, a simple experiment is shown in Figure 2.1. We connect a sender and a receiver to a single switch and start an scp transfer designed to emulate an elephant flow. Meanwhile, other senders are hooked up to the same switch and send to the same receiver. We vary the number of these competing flows and show a stacked histogram of the top 10 flowlet sizes for a 1 GB scp transfer with a 500 µs inactivity timer. The graph shows flowlet sizes can be quite large, with more than half the transfer being attributed to a single flowlet for up to 3 competing flows. Using a smaller inactivity timer, such 100µs, helps (90% of flowlet sizes are 114KB or less), but does not prevent a long tail: 0.1% of flowlets are larger than 1 MB, with the

largest ranging from 2.1-20.5 MB. Collisions on large flowlet sizes can lead to congestion. The second problem with flowlets is that small inactivity thresholds, such as 100 μs, can lead to significant reordering. Not only does this impact TCP performance (profiled in Section 2.5), but it also needlessly breaks small flows into several flowlets. With only one flow in the network, we found a 50 KB mice flow was broken into 4-5 flowlets on average. Small flows typically do not need to be load balanced on a sub-flow level and need not be exposed to reordering.

The shortcomings of the previous approaches lead us to reconsider on what granularity load balancing should occur. Ideally, sub-flow load balancing should be done on near uniform sizes so that the network paths have a near equal amount of loads. Also, the unit of load balancing should be small to allow for fine-grained load balancing, but not so small as to break small flows into many pieces or as to be a significant computational burden. As a result, we propose load balancing on 64 KB units of data we call *flowcells*. Flowcells have a number of advantages. First, the maximum segment size supported by TSO is 64 KB, so flowcells provide a natural interface to high speed optimizations provided by the NIC and OS and can scale to fast networking speeds. Second, an overwhelming fraction of mice flows are less than 64 KB in size and thus do not have to worry about reordering [19, 46, 69]. Last, since most bytes in datacenter networks originate from elephant flows [9, 19, 69], this ensures that a significant portion of datacenter traffic is routed on uniform sizes. While promising, this approach must combat reordering to be effective. Essentially we make a trade-off: the sender avoids congestion by providing fine-grained, near-uniform load balancing, and the receiver handles reordering to maintain line-rate.

Per-Hop vs End-to-End Multipathing The last design consideration is whether multipathing should be done on a local, per-hop level (*e.g.*, ECMP), or on a global, end-to-end level. In Presto, we choose the latter: pre-

configured end-to-end paths are allocated in the network and path selection (and thus multipathing) is realized by having the network edge place flowcells onto these paths. Presto can be used to load-balance in an ECMP style per-hop manner, but the choice of end-to-end multipathing provides additional benefits due to greater control of how flowcells are mapped to paths. Per-hop multipathing can be inefficient under asymmetric topologies [135], and load-balancing on a global end-to-end level can allow for weighted scheduling at the vSwitch to rebalance traffic. This is especially important when failure occurs. The second benefit is flowcells can be assigned over multiple paths very evenly by iterating over paths in a round-robin, rather than randomized, fashion.

2.2.2 Reordering Challenges

Due to the impact of fine-grained, flowcell-based load balancing, Presto must account for reordering. Here, we highlight reordering challenges. The next section shows how Presto deals with these concerns.

Reordering's Impact on TCP The impact of reordering on TCP is well-studied [77, 97]. Duplicate acknowledgments caused by reordering can cause TCP to move to a more conservative sender state and reduce the sender's congestion window. Relying on parameter tuning, such as adjusting the DUP-ACK threshold, is not ideal because increasing the DUP-ACK threshold increases the time to recover from real loss. Other TCP settings such as Forward Acknowledgement (FACK) assume un-acked bytes in the SACK are lost and degrade performance under reordering. A scheme that introduces reordering should not rely on careful configuration of TCP parameters because (i) it is hard to find a single set of parameters that work effectively over multiple scenarios and (ii) datacenter tenants should not be forced to constantly tune their networking stacks. Finally, many reordering-robust variants of TCP have been proposed [22, 23, 134], but

as we will show, GRO becomes ineffective under reordering. Therefore, reordering should be handled below the transport layer.

Computational Bottleneck of Reordering Akin to TSO, Generic Receive Offload (GRO) mitigates the computational burden of receiving 1500 byte packets at 10 Gbps. GRO is implemented in the kernel of the hypervisor, and its handler is called directly by the NIC driver. It is responsible for aggregating packets into larger segments that are pushed up to OVS and the TCP/IP stack. GRO is implemented in the Linux kernel and is used even without virtualization. Similar functionality can be found in Windows (RSC [107]) and hardware (LRO [47]).

Because modern CPUs use aggressive prefetching, the cost of receiving TCP data is now dominated by per-packet, rather than per-byte, operations. As shown by Menon [82], the majority of this overhead comes from buffer management and other routines not related to protocol processing, and therefore significant computational overhead can be avoided by aggregating "raw" packets from the NIC into a single sk_buff. Essentially, spending a few cycles to aggregate packets within GRO creates less segments for TCP and prevents having to use substantially more cycles at higher layers in the networking stack. Refer to [64, 82] for detailed study and explanation.

To better understand the problems reordering causes, a brief description of the TCP receive chain in Linux follows. First, interrupt coalescing allows the NIC to create an interrupt for a batch of packets [21, 87], which prompts the driver to poll the packets into an aggregation queue. Next, the driver invokes the GRO handler, located in the kernel, which *merges* the packets into larger segments. The merging continues, possibly across many polling events, until a segment reaches a threshold size, a certain age, or cannot be combined with the incoming packet. Then, the combined, larger segment is *pushed up* to the rest of the TCP/IP networking stack. The GRO process is done on a per-flow level. With GRO disabled,

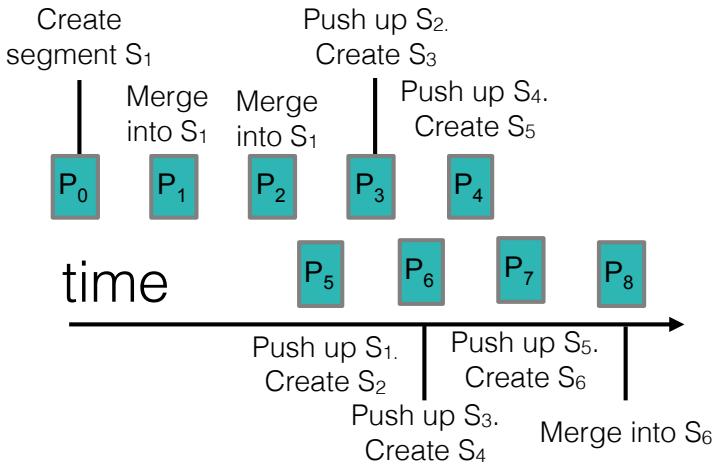


Figure 2.2: GRO pushes up small segments (S_i) during reordering.

throughput drops to around 5.7-7.1 Gbps and CPU utilization spikes to 100% (Section 2.5 and [70]). Receive offload algorithms, whether in hardware (LRO) [13, 47] or in software (GRO), are usually *stateless* to make them fast: no state is kept beyond the segment being merged.

We now uncover how GRO breaks down in the face of reordering. Figure 2.2 shows the impact of reordering on GRO. Reordering does not allow the segment to grow: each reordered packet cannot be merged with the existing segment, and thus the previously created segment must be pushed up. With extreme reordering, GRO is effectively disabled because small MTU-sized segments are constantly pushed up. This causes (i) severe computational overhead and (ii) TCP to be exposed to significant amounts of reordering. We term this the *small segment flooding* problem.

Determining where to combat the reordering problem has not previously taken the small segment flooding problem into account. Using a reordering buffer to deal with reordered packets is a common solution (*e.g.*, works like [27] re-sort out-of-order packets in a shim layer below TCP), but a buffer implemented above GRO cannot prevent small segment flooding. Implementing a buffer below GRO means that the NIC must

be changed, which is (i) expensive and cumbersome to update and (ii) unlikely to help combat reordering over multiple interrupts.

In our system, the buffer is implemented in the GRO layer itself. We argue this is a natural location because GRO can directly control segment sizes while simultaneously limiting the impact of reordering. Furthermore, GRO can still be applied on packets pushed up from LRO, which means hardware doesn't have to be modified or made complex. Implementing a better GRO algorithm has multiple challenges. The algorithm should be light-weight to scale to fast networking speeds. Furthermore, an ideal scheme should be able to distinguish loss from reordering. When a gap in sequence numbers is detected (*e.g.*, when P_5 is received after P_2 in Figure 2.2), it is not obvious if this gap is caused from loss or reordering. If the gap is due to reordering, GRO should not push segments up in order to try to wait to receive the missing gap and merge the missing packets into a preestablished segment. If the gap is due to loss, however, then GRO should immediately push up the segments to allow TCP to react to the loss as fast as possible. Ideally, an updated GRO algorithm should ensure TCP does not perform any worse than a scheme with no reordering. Finally, the scheme should adapt to prevailing network conditions, traffic patterns and application demands.

2.3 Design

This section presents the design of Presto by detailing the sender, the receiver, and how the network adapts in the case of failures and asymmetry.

2.3.1 Sender

Global Load Balancing at the Network Edge In Presto, a centralized controller is employed to collect the network topology and disseminate

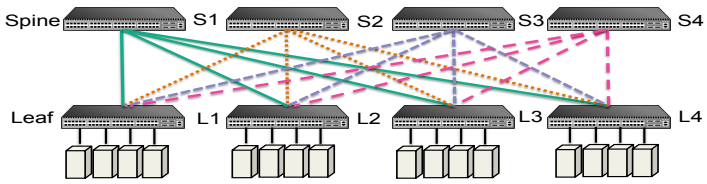


Figure 2.3: Our testbed: 2-tier Clos network with 16 hosts.

corresponding load balancing information to the edge vSwitches. The goal of this design is to ensure the vSwitches, as a whole, can load balance the network in an even fashion, but without requiring an individual vSwitch to have detailed information about the network topology, updated traffic matrices or strict coordination amongst senders. At a high level, the controller partitions the network into a set of multiple spanning trees. Then, the controller assigns each vSwitch a unique forwarding label in each spanning tree. By having the vSwitches partition traffic over these spanning trees in a fine-grained manner, the network can load balance traffic in a near-optimal fashion.

The process of creating spanning trees is made simple by employing multi-stage Clos networks commonly found in datacenters. For example, in a 2-tier Clos network with ν spine switches, the controller can easily allocate ν disjoint spanning trees by having each spanning tree route through a unique spine switch. Figure 2.3 shows an example with four spine switches and four corresponding disjoint spanning trees. When there are γ links between each spine and leaf switch in a 2-tier Clos network, the controller can allocate γ spanning trees per spine switch. Note that 2-tier Clos networks cover the overwhelming majority of enterprise datacenter deployments and can support tens of thousands of physical servers [8]. The controller ensures links in the network are equally covered by the allocated spanning trees.

Once the spanning trees are created, the controller assigns a unique forwarding label for each vSwitch in every spanning tree and installs the

relevant forwarding rules into the network. Forwarding labels can be implemented in a variety of ways using technologies commonly deployed to forward on labels, such as MPLS [28], VXLAN [8, 73], or IP encapsulation [27]. In Presto, label switching is implemented with shadow MACs [5]. Shadow MACs implement label-switching for commodity Ethernet by using the destination MAC address as an opaque forwarding label that can easily be installed in L2 tables. Each vSwitch is assigned one shadow MAC per spanning tree. Note Shadow MACs are extremely scalable on existing chipsets because they utilize the large L2 forwarding table. For example, Trident II-based switches [14, 26, 35] have 288k L2 table entries and thus 8-way multipathing (*i.e.*, each vSwitch has 8 disjoint spanning trees) can scale up to 36,000 physical servers. To increase scalability, shadow MAC tunnels can be implemented from edge switch to edge switch instead of from vSwitch to vSwitch. Switch-to-switch tunneling has been proposed in previous works such as MOOSE [81] and NetLord [91]. Tunneling requires $\mathcal{O}(|switches| \times |paths|)$ rules instead of $\mathcal{O}(|vSwitches| \times |paths|)$ rules. All shadow MAC labels can route to a destination edge switch that forwards the packet to the correct destination by forwarding on L3 information.

Finally, we note that shadow MACs are also compatible with network virtualization (both L2 and L3 address space virtualization). Tunneling techniques such as VXLAN encapsulate packets in Ethernet frames, which means shadow MACs should still allow path selection in virtualized environments by modifying outer Ethernet headers. VXLAN hardware offload is supported in modern NICs and has little performance overhead [124].

Load Balancing at the Sender After the controller installs the shadow MAC forwarding rules into the network, it creates a mapping from each physical destination MAC address to a list of corresponding shadow MAC addresses. These mappings provide a way to send traffic to a specific destination over different spanning trees. The mappings are pushed from

Algorithm 1 Pseudo-code of flowcell creation

```
1: if bytecount + len(skb) > threshold then
2:   bytecount ← len(skb)
3:   current_mac ← (current_mac + 1) % total_macs
4:   flowcellID ← flowcellID + 1
5: else
6:   bytecount ← bytecount + len(skb)
7: end if
8: skb ← update(skb, current_mac, flowcellID)
9: sendToNIC(skb)
```

the controller to each vSwitch in the network, either on-demand or pre-emptively. In Presto, the vSwitch on the sender monitors outgoing traffic (*i.e.*, maintains a per-flow counter in the datapath) and rewrites the destination MAC address with one of the corresponding shadow MAC addresses. The vSwitch assigns the same shadow MAC address to all consecutive segments until the 64 KB limit is reached. In order to load balance the network effectively, the vSwitch iterates through destination shadow MAC addresses in a round-robin fashion. This allows the edge vSwitch to load balance over the network in a very fine-grained fashion.

Sending each 64 KB worth of flowcells over a different path in the network can cause reordering and must be carefully addressed. To assist with reordering at the receiver (Presto's mechanisms for combatting reordering are detailed in the next section), the sender also includes a sequentially increasing *flowcell ID* into each segment. In our setup the controller installs forwarding rules solely on the destination MAC address and ARP is handled in a centralized manner. Therefore, the source MAC address can be used to hold the flowcell ID. Other options are possible, *e.g.*, some schemes include load balancing metadata in the reserved bits of the VXLAN header [51] and implementations could also stash flowcell IDs into large IPv6 header fields.[1] Note that since the flowcell ID and the

[1] In our implementation, TCP options hold the flowcell ID for simplicity and ease of

Algorithm 2 Pseudo-code of Presto GRO flush function

```
1: for each flow f do
2:   for S ∈ f.segment_list do
3:     if f.lastFlowcell == getFlowcell(S) then
4:       f.expSeq ← max(f.expSeq, S.endSeq)
5:       pushUp(S)
6:     else if getFlowcell(S) > f.lastFlowcell then
7:       if f.expSeq == S.startSeq then
8:         f.lastFlowcell ← getFlowcell(S)
9:         f.expSeq ← S.endSeq
10:        pushUp(S)
11:      else if f.expSeq > S.startSeq then
12:        f.lastFlowcell ← getFlowcell(S)
13:        pushUp(S)
14:      else if timeout(S) then
15:        f.lastFlowcell ← getFlowcell(S)
16:        f.expSeq ← S.endSeq
17:        pushUp(S)
18:      end if
19:    else
20:      pushUp(S)
21:    end if
22:  end for
23: end for
```

shadow MAC address are modified before a segment is handed to the NIC, the TSO algorithm in the NIC replicates these values to all derived MTU-sized packets. The pseudo-code of flowcell creation is presented in Algorithm 1. Since this code executes in the vSwitch, retransmitted TCP packets run through this code for each retransmission.

debugging.

2.3.2 Receiver

The main challenge at the receiver is dealing with reordering that can occur when different flowcells are sent over different paths. The high-level goal of our receiver implementation is to mitigate the effects of the small segment flooding problem by (i) not so aggressively pushing up segments if they cannot be merged with an incoming packet and (ii) ensuring that segments pushed up are delivered in-order.

Mitigating Small Segment Flooding Let's use Figure 2.2 as a motivating example on how to combat the small segment flooding problem. Say a polling event has occurred, and the driver retrieves 9 packets from the NIC (P_0-P_8). The driver calls the GRO handler, which merges consecutive packets into larger segments. The first three packets (P_0-P_2) are merged into a segment, call it S_1 (note: in practice S_1 already contains in-order packets received before P_0). When P_5 arrives, a new segment S_2, containing P_5, should be created. Instead of pushing up S_1 (as is done currently), both segments should be kept. Then, when P_3 is received, it can be merged into S_1. Similarly, P_6 can be merged into S_2. This process can continue until P_4 is merged into S_1. At this point, the gap between the original out-of-order reception (P_2-P_5) has been filled, and S_1 can be pushed up and S_2 can continue to grow. This means the size of the segments being pushed up is increased, and TCP is not exposed to reordering.

The current default GRO algorithm works as follows. An interrupt by the NIC causes the driver to poll (multiple) packets from the NIC's ring buffer. The driver calls the GRO handler on the received batch of packets. GRO keeps a simple doubly linked list, called gro_list, that contains segments, with a flow having at most one segment in the list. When packets for a flow are received in-order, each packet can be merged into the flow's preexisting segment. When a packet cannot be merged, such as with reordering, the corresponding segment is pushed up (ejected from the linked list and pushed up the networking stack) and a new segment

is created from the packet. This process is continued until all packets in the batch are serviced. At the end of the polling event, a flush function is called that pushes up all segments in the gro_list.

Our GRO algorithm makes the following changes. First, multiple segments can be kept per flow in a doubly linked list (called segment_list). To ensure the merging process is fast, each linked list is kept in a hash table (keyed on flow). When an incoming packet cannot be merged with any existing segment, the existing segments are kept and a new segment is created from the packet. New segments are added to the head of the linked list so that merging subsequent packets is typically $\mathcal{O}(1)$. When the merging is completed over all packets in the batch, the flush function is called. The flush function decides whether to push segments up or to keep them. Segments may be kept so reordered packets still in flight have enough time to arrive and can then be placed in-order before being pushed up. Reordering can cause the linked lists to become slightly out-of-order, so at the beginning of flush an insertion sort is run to help easily decide if segments are in-order.

The pseudo-code of our flush function is presented in Algorithm 2. For each flow, our algorithm keeps track of the next expected in-order sequence number (f.expSeq) and the corresponding flowcell ID of the most recently received in-order sequence number (f.lastFlowcell). When the merging is completed, the flush function iterates over the sorted segments (S), from lowest sequence number to highest sequence number in the segment_list (line 2). The rest of the code is presented in the subsections that follow.

How to Differentiate Loss from Reordering? In the case of no loss or reordering, our algorithm keeps pushing up segments and updating state. Lines 3-5 deal with segments from the same flowcell ID, so we just need to update f.expSeq each time. Lines 6-10 represent the case when the current flowcell ID is fully received and we start to receive the next flowcell ID. The problem, however, is when there is a gap that appears between the

sequence numbers of the segments. When a gap is encountered, it isn't clear if it is caused from reordering or from loss. If the gap is due to reordering, our algorithm should be conservative and try to wait to receive the packets that "fill in the gap" before pushing segments up to TCP. If the gap is due to loss, however, then we should push up the segments immediately so that TCP can react to the loss as quickly as possible.

To solve this problem, we leverage the fact that all packets carrying the same flowcell ID traverse the same path and should be in-order. This means incoming sequence numbers can be monitored to check for gaps. A sequence number gap within the same flowcell ID is assumed to be a loss (because all the packets in the same flowcell should go through the same network path), and not reordering, so those packets are pushed up immediately (lines 3-5). Note that because a flowcell consists of many packets (a 64 KB flowcell consists of roughly 42 1500 byte packets), when there is a loss it is likely that it occurs within flowcell boundaries. The corner case, when a gap occurs on the flowcell boundary, leads us to the next design question.

How to Handle Gaps at Flowcell Boundaries? When a gap is detected in sequence numbers at flowcell boundaries, it is not clear if the gap is due to loss or reordering. Therefore, the segment should be held long enough to handle reasonable amounts of reordering, but not so long that TCP cannot respond to loss promptly. Previous approaches that deal with reordering typically employ a large static timeout (10ms) [27]. Setting the timeout artificially high can handle reordering, but hinders TCP when the gap is due to loss. Setting a low timeout is difficult because many dynamic factors, such as delays between segments at the sender, network congestion, and traffic patterns (multiple flows received at the same NIC affect inter-arrival time), can cause large variations. As a result, we devise an adaptive timeout scheme, which monitors recent reordering events and sets a dynamic timeout value accordingly. Presto tracks cases when there is

reordering, but no loss, on flowcell boundaries and keeps an exponentially-weighted moving average ($EWMA$) over these times. Presto then applies a timeout of $\alpha * EWMA$ to a segment when a gap is detected on flowcell boundaries. Here α is an empirical parameter that allows for timeouts to grow. As a further optimization, if a segment has timed out, but a packet has been merged into that segment in the last $\frac{1}{\beta} * EWMA$ time interval, then the segment is still held in hopes of preventing reordering. We find α and β work over a wide range of parameters and set both of them to 2 in our experiments. A timeout firing is dealt with in lines 14-18.

How to Handle Retransmissions? Retransmitted packets are pushed up immediately in order to allow the TCP stack to react without delay. If the flowcell ID of the retransmission is the same as the expected flowcell ID, then the retransmission will be pushed up immediately because its sequence number will be $\leqslant f.expSeq$. If the flowcell ID is larger than the expected flowcell ID (when the first packet of a flowcell is a retransmission), then the packet is pushed up (line 13). If a retransmitted packet has a smaller flowcell ID than the next expected flowcell ID (a stale packet), then it will be pushed up immediately (line 20). Note we ensure overflow is handled properly in all cases.

2.3.3 Failure Handling and Asymmetry

When failures occur, Presto relies on the controller to update the forwarding behavior of the affected vSwitches. The controller can simply prune the spanning trees that are affected by the failure, or more generally enforce a weighted scheduling algorithm over the spanning trees. Weighting allows for Presto to evenly distribute traffic over an asymmetric topology. Path weights can be implemented in a simple fashion by duplicating shadow MACs used in the vSwitch's round robin scheduling algorithm. For example, assume we have three paths in total (p_1, p_2 and p_3) and their updated

weights are 0.25, 0.5 and 0.25 respectively. Then the controller can send the sequence of p_1, p_2, p_3, p_2 to the vSwitch, which can then schedule traffic over this sequence in a round robin fashion to realize the new path weights. This way of approximating path weights in the face of network asymmetry is similar to WCMP [135], but instead of having to change switch firmware and use scarce on-chip SRAM/TCAM entries, we can push the weighted load balancing entirely to the network edge.

As an added optimization, Presto can leverage any fast failover features that the network supports, such as BGP fast external failover, MPLS fast reroute, or OpenFlow failover groups. Fast failover detects port failure and can move corresponding traffic to a predetermined backup port. Hardware failover latency ranges from several to tens of milliseconds [30, 98]. This ensures traffic is moved away from the failure rapidly and the network remains connected when redundant links are available. Moving to backup links causes imbalance in the network, so Presto relies on the controller learning of the network change, computing weighted multipath schedules, and disseminating the schedules to the edge vSwitches.

2.4 Methodology

Implementation We implemented Presto in Open vSwitch v2.1.2 [95] and Linux kernel v3.11.0 [78]. In OVS, we modified 5 files and ~600 lines of code. For GRO, we modified 11 files and ~900 lines of code.

Testbed We conducted our experiments on a physical testbed consisting of 16 IBM System x3620 M3 servers with 6-core Intel Xeon 2.53GHz CPUs, 60GB memory, and Mellanox ConnectX-2 EN 10GbE NICs. The servers were connected in a 2-tier Clos network topology with 10 Gbps IBM RackSwitch G8264 switches, as shown in Figure 2.3.

Experiment Settings We ran the default TCP implementation in the

Linux kernel (TCP CUBIC [52]) and set parameters tcp_sack, tcp_fack, tcp_low_latency to 1. Further, we tuned the host Receive Side Scaling (RSS) [108] and IRQ affinity settings and kept them the same in all experiments. We send and receive packets from the hypervisor OS instead of VMs. LRO is not enabled on our NICs.

Workloads We evaluate Presto with a set of synthetic and realistic workloads. Similar to previous works [6, 7, 106], our synthetic workloads include: *Shuffle*: Each server in the testbed sends 1GB data to every other server in the testbed in random order. Each host sends two flows at a time. This workload emulates the shuffle behavior of Hadoop workloads. *Stride(8)*: We index the servers in the testbed from left to right. In stride(8) workload, server[i] sends to server[(i+8) mod 16]. *Random*: Each server sends to a random destination not in the same pod as itself. Multiple senders can send to the same receiver. *Random Bijection*: Each server sends to a random destination not in the same pod as itself. Different from random, each server only receives data from one sender. Finally, we also evaluate Presto with trace-driven workloads from real datacenter traffic [69].

Performance Evaluation We compare Presto to ECMP, MPTCP, and a single non-blocking switch used to represent an optimal scenario. ECMP is implemented by enumerating all possible end-to-end paths and randomly selecting a path for each flow. MPTCP uses ECMP to determine the paths of each of its sub-flows. The MPTCP implementation is still under active development, and we spent significant effort in finding the most stable configuration of MPTCP on our testbed. Ultimately, we found that Mellanox mlx_en4 driver version 2.2, MPTCP version 0.88 [90], subflow count set to 8, OLIA congestion control algorithm [72], and configured buffer sizes as recommended by [72, 96, 105] gave us the best trade-offs in terms of throughput, latency, loss and stability. Unfortunately, despite our efforts, we still occasionally witness some stability issues with MPTCP

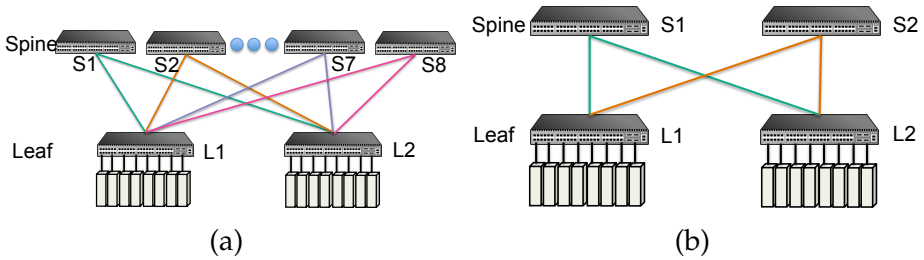


Figure 2.4: (a) Scalability benchmark and (b) Oversub. benchmark topology.

that we believe are due to implementation bugs.

We evaluate Presto on various performance metrics, including: round trip time (a single TCP packet, measured by sockperf [115]), throughput (measured by nuttcp), mice flow completion time (time to send a 50 KB flow and receive an application-layer acknowledgement), packet loss (measured from switch counters), and fairness (Jain's fairness index [60] over flow throughputs). Mice flows (50KB) are sent every 100 ms and elephant flows last 10 seconds. Each experiment is run for 10 seconds over 20 runs. Error bars on graphs denote the highest and lowest value over all runs.

2.5 Microbenchmarks

We first evaluate the effectiveness of Presto over a series of microbenchmarks: (i) Presto's effectiveness in preventing the small segment flooding problem and reordering, (ii) Presto's CPU overhead, (iii) Presto's ability to scale to multiple paths, (iv) Presto's ability to handle congestion, (v) comparison to flowlet switching, and (vi) comparison to local, per-hop load balancing.

Presto's GRO Combats Reordering To examine Presto's ability to handle packet reordering, we perform a simple experiment on the topology shown in Figure 2.4b. Here two servers attached to leaf switch L1 send traffic to

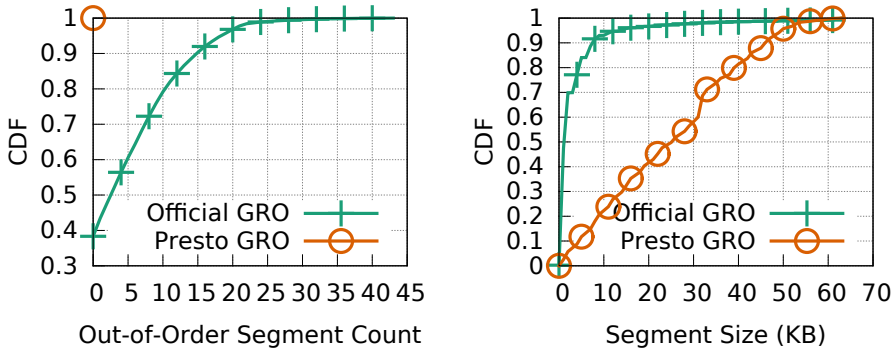


Figure 2.5: (a) Illustration of the modified GRO's effectiveness on masking reordering. (b) In case of massive packet reordering, official GRO cannot merge packets effectively such that lots of small packets are processed by TCP which poses great processing overhead for CPU.

their own receivers attached to leaf switch L2 by spreading flowcells over two network paths. This setup can cause reordering for each flow, so we compare Presto's GRO to an unmodified GRO, denoted "Official GRO". The amount of reordering exposed to TCP is presented in Figure 2.5a. To quantify packet reordering, we show a CDF of the *out-of-order segment count*: *i.e.*, the number of segments from other flowcells between the first packet and last packet of each flowcell. A value of zero means there is no reordering and larger values mean more reordering. The figure shows Presto's GRO can completely mask reordering while official GRO incurs significant reordering. As shown in Section 2.2, reordering can also cause smaller segments to be pushed up the networking stack, causing significant processing overhead. Figure 2.5b shows the received TCP segment size distribution. Presto's GRO pushes up large segments, while the official GRO pushes up many small segments. The average TCP throughputs in official GRO and Presto GRO are 4.6 Gbps (with 86% CPU utilization) and 9.3 Gbps (with 69% CPU utilization), respectively. Despite the fact that

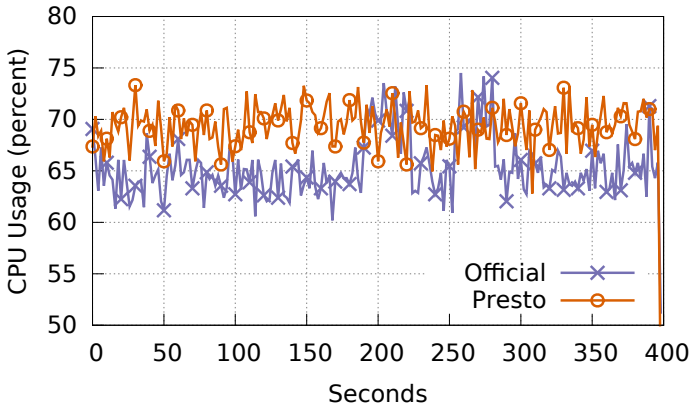


Figure 2.6: Presto incurs 6% CPU overhead on average.

official GRO only obtains about half the throughput of Presto's GRO, it still incurs more than 24% higher CPU overhead. Therefore, an effective scheme must deal with both reordering and small segment overhead.

Presto Imposes Limited CPU Overhead We investigate Presto's CPU usage by running the stride workload on a 2-tier Clos network as shown in Figure 2.3. For comparison, official GRO is run with the stride workload using a non-blocking switch (so there is no reordering). Note both official GRO and Presto GRO can achieve 9.3 Gbps. The receiver CPU usage is sampled every 2 seconds over a 400 second interval, and the time-series is shown in Figure 2.6. On average, Presto GRO only increases CPU usage by 6% compared with the official GRO. The minimal CPU overhead comes from Presto's careful design and implementation. At the sender, Presto needs just two memcpy operations (1 for shadow MAC rewriting, 1 for flowcell ID encoding). At the receiver, Presto needs one memcpy to rewrite the shadow MAC back to the real MAC and also incurs slight overhead because multiple segments are now kept per flow. The overhead of the latter is reduced because these segments are largely kept in reverse sorted order, which means merge on an incoming packet is usually $\mathcal{O}(1)$. The

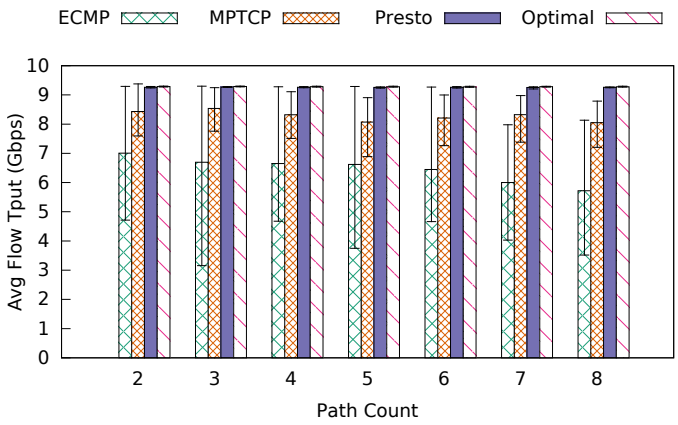


Figure 2.7: Throughput comparison in scalability benchmark. We denote the non-blocking case as Optimal.

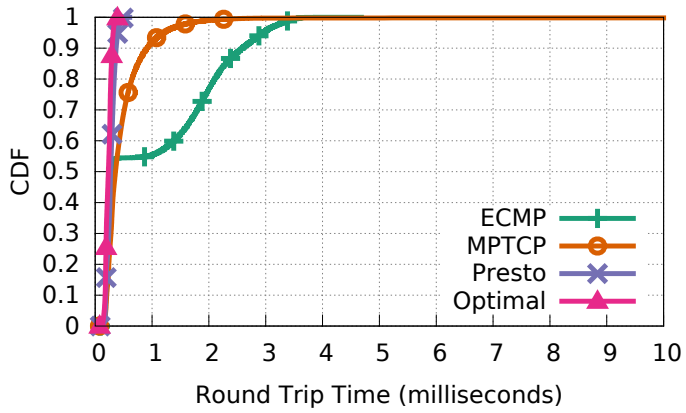


Figure 2.8: Round trip time comparison in scalability benchmark.

insertion sort is done at the beginning of each flush event over a small number of mostly in-order segments, which amortizes overhead because it is called infrequently compared to merge.

Presto Scales to Multiple Paths We analyze Presto's ability to scale in the number of paths by setting the number of flows (host pairs) equal to the number of available paths in the topology shown in Figure 2.4a. The

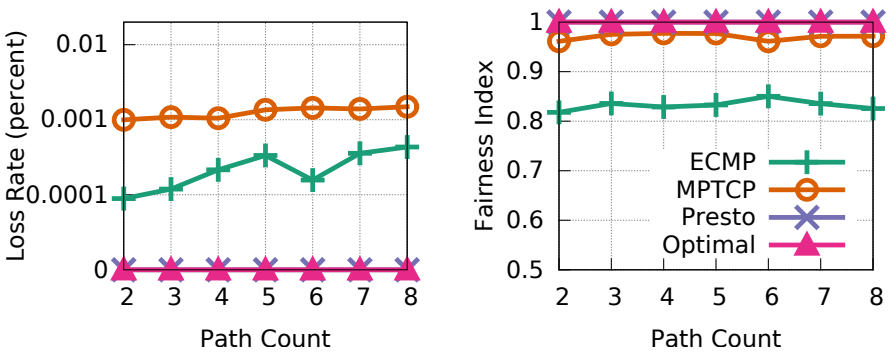


Figure 2.9: (a) Loss rate and (b) Fairness index comparison in scalability benchmark.

number of paths is varied from 2 to 8, and Presto always load-balances over all available paths. Figure 2.7 shows Presto's throughput closely tracks Optimal. ECMP (and MPTCP) suffer from lower throughput when flows (or subflows) are hashed to the same path. Hashing on the same path leads to congestion and thus increased latency, as shown in Figure 2.8. Because this topology is non-blocking and Presto load-balances in a near optimal fashion, Presto's latency is near Optimal. Packet drop rates are presented in Figure 2.9a and show Presto and Optimal have no loss. MPTCP has higher loss because of its bursty nature [8] and its aggression in the face of loss: when a single loss occurs, only one subflow reduces its rate. The other schemes are more conservative because a single loss reduces the rate of the whole flow. Finally, Figure 2.9b shows Presto, Optimal and MPTCP achieve almost perfect fairness.

Presto Handles Congestion Gracefully Presto's ability to handle congestion is analyzed by fixing the number of spine and leaf switches to 2 and varying the number of flows (host pairs) from 2 to 8, as shown in Figure 2.4b. Each flow sends as much as possible, which leads to the

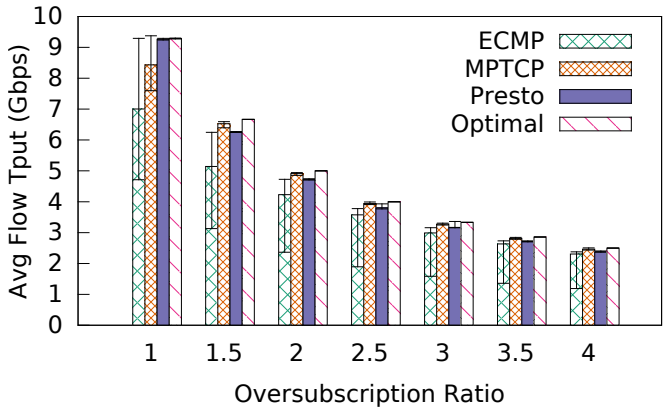


Figure 2.10: Throughput comparison in oversubscription benchmark.

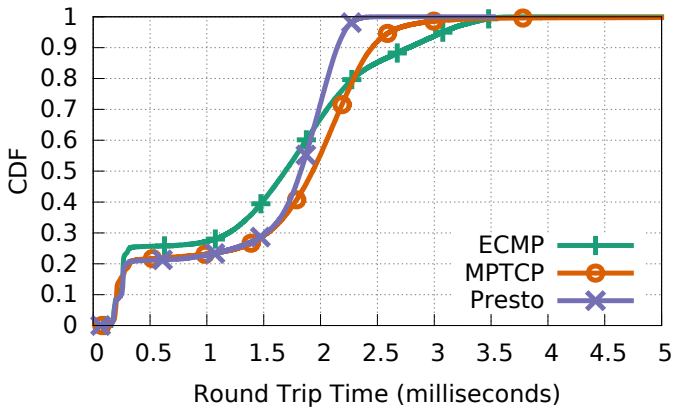


Figure 2.11: Round trip time comparison in oversubscription benchmark.

network being oversubscribed by a ratio of 1 (two flows) to 4 (eight flows). Figure 2.10 shows all schemes track Optimal in highly oversubscribed environments. ECMP does poorly under moderate congestion because the limited number of flows can be hashed to the same path. Presto does no worse in terms of latency (Figure 2.11) and loss (Figure 2.12a). The long tail latency for MPTCP is caused by its higher loss rates. Both Presto and MPTCP have greatly improved fairness compared with ECMP (Fig-

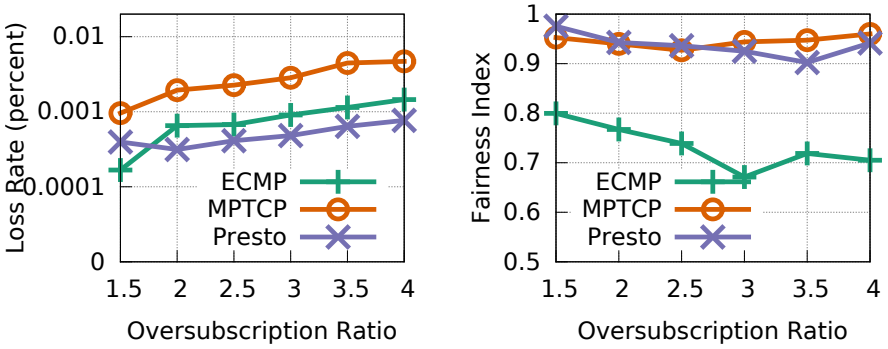


Figure 2.12: (a) Loss rate and (b) Fairness index comparison in oversubscription benchmark.

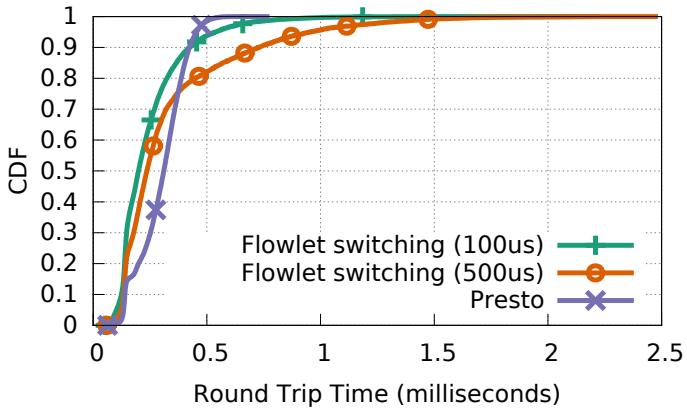


Figure 2.13: Round trip time comparison of flowlet switching and Presto in Stride workload. The throughputs of Flowlet switching with 100 µs gap, 500 µs gap and Presto are 4.3 Gbps, 7.6 Gbps and 9.3 Gbps respectively.

ure 2.12b).

Comparison to Flowlet Switching We first implemented a flowlet load-balancing scheme in OVS that detects inactivity gaps and then schedules flowlets over disjoint paths in a round robin fashion. The receiver for flowlets uses official GRO. Our flowlet scheme is not a direct reflection

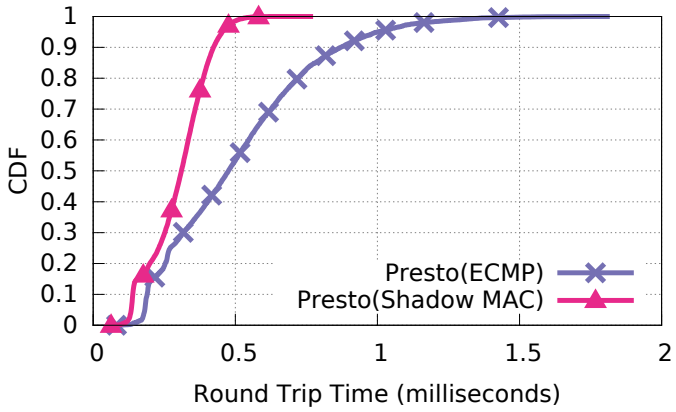


Figure 2.14: Round trip time comparison between Presto + shadow MAC and Presto + ECMP.

of CONGA because (i) it is not congestion-aware and (ii) the flowlets are determined in the software edge instead of the networking hardware. Presto is compared to 500 µs and 100 µs inactivity timers in the stride workload on the 2-tier Clos network (Figure 2.3). The throughput of the schemes are 9.3 Gbps (Presto), 7.6 Gbps (500 µs), and 4.3 Gbps (100 µs). Analysis of the 100 µs network traces show 13%-29% packets in the connection are reordered, which means 100 µs is not enough time to allow packets to arrive in-order at the destination and thus throughput is severely impacted. Switching flowlets with 500 µs prevents most reordering (only 0.03%-0.5% packets are reordered), but creates very large flowlets (see Figure 2.1). This means flowlets can still suffer from collisions, which can hurt throughput (note: while not shown here, 500 µs outperforms ECMP by over 40%). Figure 2.13 shows the latencies. Flowlet 100 µs has low throughput and hence lower latencies. However, since its load balancing isn't perfect, it can still cause increased congestion in the tail. Flowlet 500 µs also has larger tail latencies because of more pronounced flowlet collisions. As compared to the flowlet schemes, Presto decreases 99.9^{th} percentile latency by 2x-3.6x.

Comparison to Local, Per-Hop Load Balancing Presto sends flowcells in a round robin fashion over pre-configured end-to-end paths. An alternative is to have ECMP hash on flowcell ID and thus provide per-hop load balancing. We compare Presto + shadow MAC with Presto + ECMP using a stride workload on our testbed. Presto + shadow MAC's average throughput is 9.3 Gbps while Presto + ECMP's is 8.9 Gbps. The round trip time CDF is shown in Figure 2.14. Presto + shadow MAC gives better latency performance compared with Presto + ECMP. The performance difference comes from the fact that Presto + shadow MAC provides better fine-grained flowcell load balancing because randomization in per-hop multipathing can lead to corner cases where a large fraction of flowcells get sent to the same link over a small timescale by multiple flows. This transient congestion can lead to increased buffer occupancy and higher delays.

2.6 Evaluation

In this section, we analyze the performance of Presto for (i) synthetic workloads, (ii) trace-driven workloads, (iii) workloads containing north-south cross traffic, and (iv) failures. All tests are run on the topology in Figure 2.3.

Synthetic Workloads Figure 2.15 shows the average throughputs of elephant flows in the shuffle, random, stride and random bijection workloads. Presto's throughput is within 1-4% of Optimal over all workloads. For the shuffle workload, ECMP, MPTCP, Presto and Optimal show similar results because the throughput is mainly bottlenecked at the receiver. In the non-shuffle workloads, Presto improves upon ECMP by 38-72% and improves upon MPTCP by 17-28%.

Figure 2.16 shows a CDF of the mice flow completion time (FCT) for each workload. The stride and random bijection workloads are non-

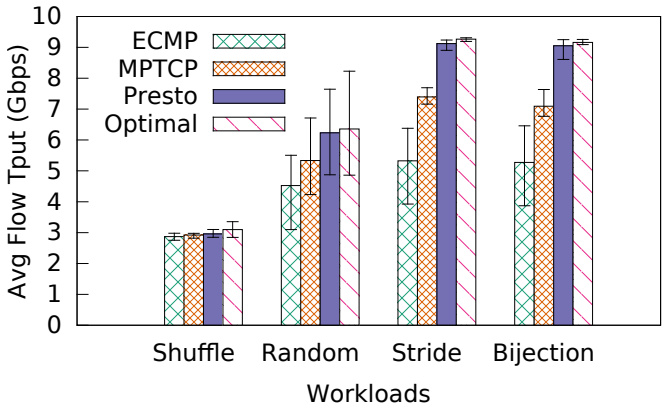
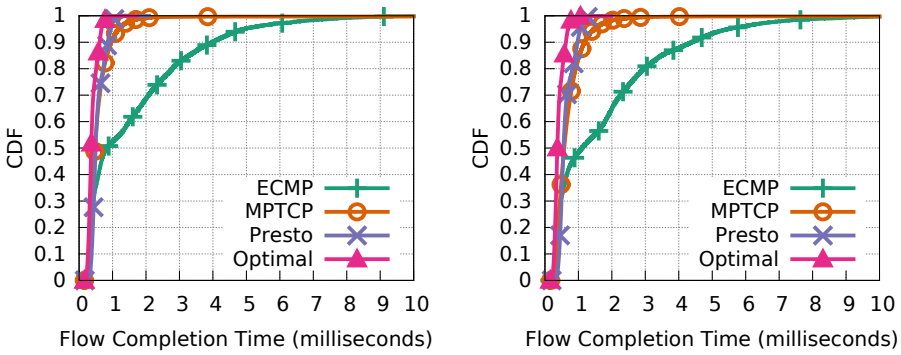


Figure 2.15: Elephant flow throughput for ECMP, MPTCP, Presto and Optimal in shuffle, random, stride and random bijection workloads.

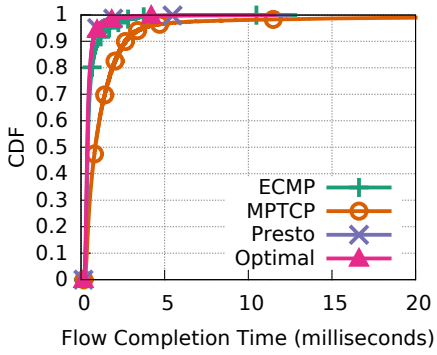
blocking, and hence the latency of Presto closely tracks Optimal: the 99.9^{th} percentile FCT for Presto is within 350 µs for these workloads. MPTCP and ECMP suffer from congestion, and therefore the tail FCT is much worse than Presto: ECMP's 99.9^{th} percentile FCT is over 7.5x worse (~11ms) and MPTCP experiences timeout (because of higher loss rates and the fact that small sub-flow window sizes from small flows can increase the chances of timeout [105]). We used the Linux default timeout (200 ms) and trimmed graphs for clarity. The difference in the random and shuffle workloads is less pronounced (we omit random due to space constraints). In these workloads elephant flows can collide on the last-hop output port, and therefore mice FCT is mainly determined by queuing latency. In shuffle, the 99.9^{th} percentile FCT for ECMP, Presto and Optimal are all within 10% (MPTCP again experiences TCP timeout) and in random, the 99.9^{th} percentile FCT of Presto is within 25% of Optimal while ECMP's is 32% worse than Presto.

Trace-driven Workload We evaluate Presto using a trace-driven workload based on traffic patterns measured in [69]. Each server establishes a long-lived TCP connection with every other server in the testbed. Then each



(a) Stride

(b) Random Bijection



(c) Shuffle

Figure 2.16: Mice FCT of ECMP, MPTCP, Presto and Optimal in stride, random bijection, and shuffle workloads.

Percentile	ECMP	Optimal	Presto
50%	1.0	−12%	−9%
90%	1.0	−34%	−32%
99%	1.0	−63%	−56%
99.9%	1.0	−61%	−60%

Table 2.17: Mice (<100KB) FCT in trace-driven workload [69]. Negative numbers imply shorter FCT.

server continuously samples flow sizes and inter-arrival times and each time sends to a random receiver that is not in the same rack. We scale the flow size distribution by a factor of 10 to emulate a heavier workload. Mice flows are defined as flows that are less than 100 KB in size, and elephant flows are defined as flows that are greater than 1 MB. The mice FCT, normalized to ECMP, is shown in Table 2.17. Compared with ECMP, Presto has similar performance at the 50^{th} percentile but reduces the 99^{th} and 99.9^{th} percentile FCT by 56% and 60%, respectively. Note MPTCP is omitted because its performance was quite unstable in workloads featuring a large number of small flows. The average elephant throughput (not shown) for Presto tracks Optimal (within 2%), and improves upon ECMP by over 10%.

Percentile	ECMP	Optimal	Presto	MPTCP
50%	1.0	−34%	−20%	−12%
90%	1.0	−83%	−79%	−73%
99%	1.0	−89%	−86%	−73%
99.9%	1.0	−91%	−87%	TIMEOUT

Table 2.18: FCT comparison (normalized to ECMP) with ECMP load balanced north-south traffic. Optimal means all the hosts are attached to a single switch.

Impact of North-South Cross Traffic Presto load balances on "east-west" traffic in the datacenter, *i.e.*, traffic originating and ending at servers in the datacenter. In a real datacenter environment "north-south" traffic (*i.e.*, traffic with an endpoint outside the datacenter) must also be considered. To study the impact of north-south traffic on Presto, we attach an additional server to each spine switch in our testbed to emulate remote users. The 16 servers establish a long-lived TCP connection with each remote user. Next, each server starts a flow to a random remote user every 1 millisecond. This emulates the behavior of using ECMP to load balance north-south traffic.

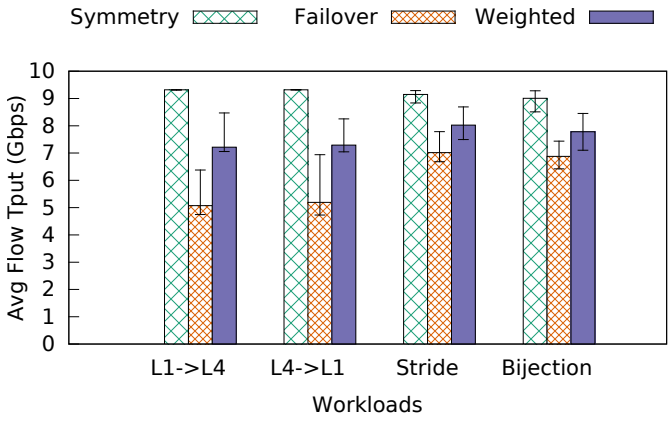


Figure 2.19: Presto's throughput in symmetry, failover and weighted multipathing stages for different workloads.

The flow sizes for north-south traffic are based on the distribution measurement in [55]. The throughput to remote users is limited to 100Mbps to emulate the limitation of an Internet WAN. Along with the north-south flows, a stride workload is started to emulate the east-west traffic. The east-west mice FCT is shown in Table 2.18 (normalized to ECMP). ECMP, MPTCP, Presto, and Optimal's average throughput is 5.7, 7.4, 8.2, and 8.9Gbps respectively. The experiment shows Presto can gracefully co-exist with north-south cross traffic in the datacenter.

Impact of Link Failure Finally, we study the impact of link failure. Figure 2.19 compares the throughputs of Presto when the link between spine switch S1 and leaf switch L1 goes down. Three stages are defined: symmetry (the link is up), failover (hardware fast-failover moves traffic from S1 to S2), and weighted (the controller learns of the failure and prunes the tree with the bad link). Workload L1→L4 is when each node connected to L1 sends to one node in L4 (L4→L1 is the opposite). Despite the asymmetry in the topology, Presto still achieves reasonable average throughput at each stage. Figure 2.20 shows the round trip time of each stage in a random bijection workload. Due to the fact that the network is no longer

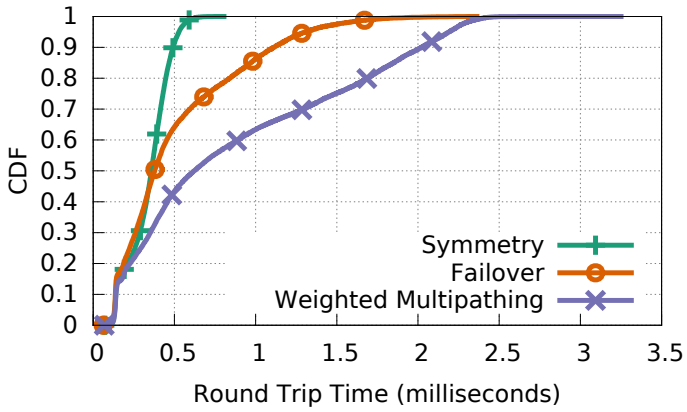


Figure 2.20: Presto's RTT in symmetry, fast failover and weighted multipathing stages in random bijection workload.

non-blocking after the link failure, failover and weighted multipathing stages have larger round trip time.

2.7 Summary

In this chapter, we present Presto: a near uniform sub-flow distributed load balancing scheme that can near optimally load balance the network at fast networking speeds. Our scheme makes a few changes to the hypervisor soft-edge (vSwitch and GRO) and does not require any modifications to the transport layer or network hardware, making the bar for deployment lower. Presto is explicitly designed to load balance the network at fine granularities and deal with reordering without imposing much overhead on hosts. Presto is flexible and can also deal with failures and asymmetry. Finally, we show the performance of Presto can closely track that of an optimal non-blocking switch, meaning elephant throughputs remain high while the tail latencies of mice flow completion times do not grow due to congestion.

3

Virtual Congestion Control Enforcement for Datacenter Networks

3.1 Introduction

Multi-tenant datacenters are a crucial component of today's computing ecosystem. Large providers, such as Amazon, Microsoft, IBM, Google and Rackspace, support a diverse set of customers, applications and systems through their public cloud offerings. These offerings are successful in part because they provide efficient performance to a wide-class of applications running on a diverse set of platforms. Virtual Machines (VMs) play a key role in supporting this diversity by allowing customers to run applications in a wide variety of operating systems and configurations.

And while the flexibility of VMs allows customers to easily move a vast array of applications into the cloud, that same flexibility inhibits the amount of control a cloud provider yields over VM behavior. For example, a cloud provider may be able to provide virtual networks or enforce rate limiting on a tenant VM, but it cannot control the VM's TCP/IP stack. As the TCP/IP stack considerably impacts overall network performance, it is unfortunate that cloud providers cannot exert a fine-grained level of control over one of the most important components in the networking stack.

Without control over the VM TCP/IP stack, datacenter networks re-

main at the mercy of inefficient, out-dated or misconfigured TCP/IP stacks. TCP behavior, specifically congestion control, has been widely studied and many issues have come to light when it is not optimized. For example, network congestion caused by non-optimzed stacks can lead to loss, increased latency and reduced throughput.

Thankfully, recent advances optimizing TCP stacks for datacenters have shown high throughput and low latency can be achieved through novel TCP congestion control algorithms. Works such as DCTCP [9] and TIMELY [84] provide high bandwidth and low latency by ensuring network queues in switches do not fill up. And while these stacks are deployed in many of today's private datacenters [66, 112], ensuring a vast majority of VMs within a public datacenter will update their TCP stacks to a new technology is a daunting, if not impossible, task.

In this chapter, we explore how operators can regain authority over TCP congestion control, regardless of the TCP stack running in a VM. Our aim is to allow a cloud provider to utilize advanced TCP stacks, such as DCTCP, without having control over the VM or requiring changes in network hardware. We propose implementing congestion control in the virtual switch (vSwitch) running on each server. Implementing congestion control within a vSwitch has several advantages. First, vSwitches naturally fit into datacenter network virtualization architectures and are widely deployed [102]. Second, vSwitches can easily monitor and modify traffic passing through them. Today vSwitch technology is mature and robust, allowing for a fast, scalable, and highly-available framework for regaining control over the network.

Implementing congestion control within the vSwitch has numerous challenges, however. First, in order to ensure adoption rates are high, the approach must work without making changes to VMs. Hypervisor-based approaches typically rely on rate limiters to limit VM traffic. Rate limiters implemented in commodity hardware do not scale in the number

of flows and software implementations incur high CPU overhead [104]. Therefore, limiting a VM's TCP flows in a fine-grained, dynamic nature at scale (10,000's of flows per server [89]) with limited computational overhead remains challenging. Finally, VM TCP stacks may differ in the features they support (*e.g.*, ECN) or the congestion control algorithm they implement, so a vSwitch congestion control implementation should work under a variety of conditions.

This chapter presents Administrator Control over Datacenter TCP (AC/DC TCP, or simply AC/DC), a new technology that implements TCP congestion control within a vSwitch to help ensure VM TCP performance cannot impact the network in an adverse way. At a high-level, the vSwitch monitors all packets for a flow, modifies packets to support features not implemented in the VM's TCP stack (*e.g.*, ECN) and reconstructs important TCP parameters for congestion control. AC/DC runs the congestion control logic specified by an administrator and then enforces an intended congestion window by modifying the receive window (RWND) on incoming ACKs. A policing mechanism ensures stacks cannot benefit from ignoring RWND.

Our scheme provides the following benefits. First, AC/DC allows administrators to enforce a uniform, network-wide congestion control algorithm without changing VMs. When using congestion control algorithms tuned for datacenters, this allows for high throughput and low latency. Second, our system mitigates the impact of varying TCP stacks running on the same fabric. This improves fairness and additionally solves the ECN co-existence problem identified in production networks [66, 130]. Third, our scheme is easy to implement, computationally lightweight, scalable, and modular so that it is highly complimentary to performance isolation schemes also designed for virtualized datacenter environments. The contributions of this chapter are as follows:

1. The design of a vSwitch-based congestion control mechanism that

regains control over the VM's TCP/IP stack without requiring any changes to the VM or network hardware.

2. A prototype implementation to show our scheme is effective, scalable, simple to implement, and has less than one percentage point computational overhead in our tests.

3. A set of results showing DCTCP configured as the host TCP stack provides nearly identical performance to when the host TCP stack varies but DCTCP's congestion control is implemented in the vSwitch. We demonstrate how AC/DC can improve throughput, fairness and latency on a shared datacenter fabric.

The outline of this chapter is as follows. Background and motivation are discussed in §3.2. AC/DC's design is outlined in §3.3 and implementation in §3.4. Results are presented in §3.5.

3.2 Background and Motivation

This section first gives a brief background of congestion control in the datacenter. Then the motivation for moving congestion control into the vSwitch is presented. Finally, AC/DC is contrasted from a class of related bandwidth allocation schemes.

3.2.1 Datacenter Transport

Today's datacenters host applications such as search, advertising, analytics and retail that require high bandwidth and low latency. Network congestion, caused by imperfect load balancing [7], network upgrades or failures, can adversely impact these services. Unfortunately, congestion is not rare in datacenters. For example, recently Google reported congestion-based drops were observed when network utilization

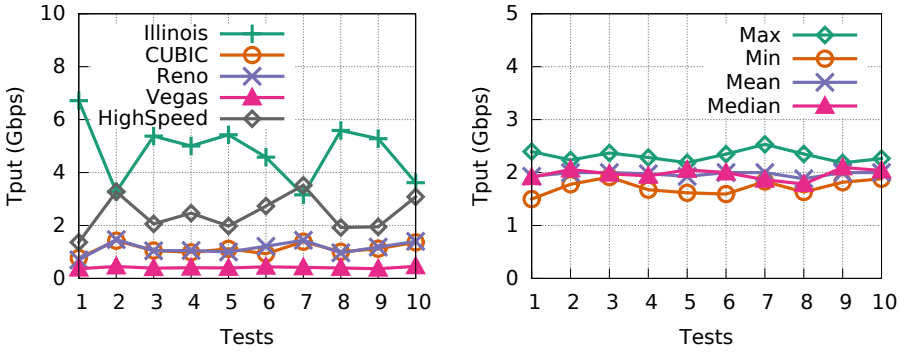
approached 25% [112]. Other studies have shown high variance and substantial increase in the 99.9^{th} percentile latency for round-trip times in today's datacenters [86, 125]. Large tail latencies impact customer experience, result in revenue loss [9, 37], and degrade application performance [48, 62]. Therefore, significant motivation exists to reduce congestion in datacenter fabrics.

TCP's congestion control algorithm is known to significantly impact network performance. As a result, datacenter TCP performance has been widely studied and many new protocols have been proposed [9, 65, 84, 117, 129]. Specifically, DCTCP [9] adjusts a TCP sender's rate based on the fraction of packets experiencing congestion. In DCTCP, the switches are configured to mark packets with an ECN bit when their queue lengths exceed a threshold. By proportionally adjusting the rate of the sender based on the fraction of ECN bits received, DCTCP can keep queue lengths low, maintain high throughput, and increase fairness and stability over traditional schemes [9, 66]. For these reasons, we implement DCTCP as the vSwitch congestion control algorithm in AC/DC.

3.2.2 Benefits of AC/DC

Allowing administrators to enforce an optimized congestion control without changing the VM is the first major benefit of our scheme. This is an important criteria in untrusted public cloud environments or simply in cases where servers cannot be updated due to a dependence on a specific OS or library. [66]

The next benefit is AC/DC allows for *uniform* congestion control to be implemented throughout the datacenter. Unfairness arises when stacks are handled differently in the fabric or when conservative and aggressive stacks coexist. Studies have shown ECN-capable and ECN-incapable flows do not exist gracefully on the same fabric because packets belonging to ECN-incapable flows encounter severe packet drops when their packets



(a) 5 different CCs. (b) All CUBIC.

Figure 3.1: Different congestion controls lead to unfairness.

exceed queue thresholds [66, 130]. Additionally, stacks with different congestion control algorithms may not share the same fabric fairly. For example, Figure 3.1 shows the performance of five different TCP flows on the topology in Figure 3.7a. Each flow selects a congestion control algorithm available in Linux: CUBIC [53], Illinois [80], HighSpeed [41], New Reno [42] and Vegas [25]. Figure 3.1a shows aggressive stacks such as Illinois and HighSpeed achieve higher bandwidth and thus fairness is worse than all flows using the same stack (Figure 3.1b).

Another benefit of AC/DC is it allows for different congestion control algorithms to be assigned on a per-flow basis. A vSwitch-based approach can assign WAN flows to a congestion control algorithm that optimizes WAN performance [40, 118] and datacenter flows to one that optimizes datacenter performance, even if these flows originate from the same VM (*e.g.*, a webserver). Additionally, as shown in §3.3.4, a flexible congestion control algorithm can provide relative bandwidth allocations to flows. This is useful when tenants or administrators want to prioritize flows assigned to the same quality-of-service class. In short, adjusting congestion control algorithms on a per-flow basis allows for enhanced flexibility and

performance.

Finally, congestion control is not difficult to port. While the entire TCP stack may seem complicated and prone to high overhead, the congestion control aspect of TCP is relatively light-weight and simple to implement. Indeed, studies show most TCP overhead comes from buffer management [82], and in our evaluation the computational overhead of AC/DC is less than one percentage point. Porting is also made easy because congestion control implementations in Linux are modular: DCTCP's congestion control resides in tcp_dctcp.c and is only about 350 lines of code. Given the simplicity of congestion control, it is not hard to move its functionality to another layer.

3.2.3 Tenant-Level Bandwidth Allocation

While AC/DC enforces congestion control, transport layer schemes do not provide fair bandwidth allocation among tenants because a tenant with more concurrent flows can obtain a higher share of bandwidth. In order to provide performance isolation in the network, datacenter operators can implement a variety of bandwidth allocation schemes by either guaranteeing or proportionally allocating bandwidth for tenants [17, 50, 62, 63, 75, 103, 109, 111, 132]. Some of these schemes share high-level architectural similarities to AC/DC. For example, EyeQ [63] handles bandwidth allocation at the edge with a work-conserving distributed bandwidth arbitration scheme. It enforces rate limits at senders based on feedback generated by receivers. Similarly, Seawall [111] provides proportional bandwidth allocation to a VM or application by forcing all traffic through a congestion-controlled tunnel configured through weights and endpoint feedback.

The fundamental difference between these schemes and our approach is the design goals determine the granularity on which they operate. Performance isolation schemes generally focus on *bandwidth allocation on a*

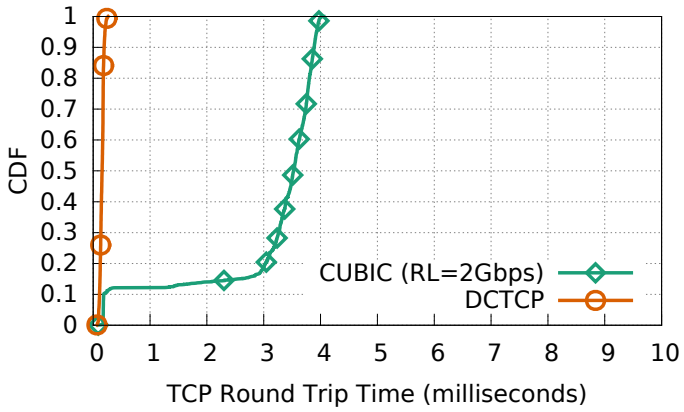


Figure 3.2: CDF of RTTs showing CUBIC fills buffers.

VM-level and are not sufficient to relieve the network of congestion because they do not operate on flow-level granularity. For example, the single switch abstraction in EyeQ [63] and Gatekeeper [109] explicitly assumes a congestion-free fabric for optimal bandwidth allocation between pairs of VMs. This abstraction doesn't hold in multi-pathed topologies when failure, traffic patterns or ECMP hash collisions [7] cause congestion in the core. Communication between a pair of VMs may consist of multiple flows, each of which may traverse a distinct path. Therefore, enforcing rate limits on a VM-to-VM level is too coarse-grained to determine how specific flows should adapt in order to mitigate the impact of congestion on their paths. Furthermore, a scheme like Seawall [111] cannot be easily applied to flow-level granularity because its rate limiters are unlikely to scale in the number of flows at high networking speeds [104] and its allocation scheme does not run at fine-grained round-trip timescales required for effective congestion control. Additionally, Seawall violates our design principle by requiring VM modifications to implement congestion-controlled tunnels.

The above points are not intended to criticize any given work, but rather support the argument that it is important for a cloud provider to enforce *both* congestion control and bandwidth allocation. Congestion control can

ensure low latency and high utilization, and bandwidth allocation can provide tenant-level fairness. Bandwidth allocation schemes alone are insufficient to mitigate congestion because certain TCP stacks aggressively fill switch buffers. Consider a simple example where five flows send simultaneously on the 10 Gbps topology in Figure 3.7a. Even when the bandwidth is allocated "perfectly" at 2 Gbps per flow, CUBIC saturates the output port's buffer and leads to inflated round-trip times (RTTs) for traffic sharing the same link. Figure 3.2 shows these RTTs for CUBIC and also DCTCP, which is able to keep queueing latencies, and thus RTTs, low even though no rate limiting was applied. Therefore, it is important for cloud providers to exercise a desired congestion control.

In summary, our vision regards enforcing tenant congestion control and bandwidth allocation as *complimentary* and we claim an administrator should be able to combine any congestion control (*e.g.*, DCTCP) with any bandwidth allocation scheme (*e.g.*, EyeQ). Flow-level congestion control and tenant performance isolation need not be solved by the same scheme, so AC/DC's design goal is to be modular in nature so it can co-exist with any bandwidth allocation scheme and its associated rate limiter (and also in the absence of both).

3.3 Design

This section provides an overview of AC/DC's design. First, we show how basic congestion control state can be inferred in the vSwitch. Then we study how to implement DCTCP. Finally, we discuss how to enforce congestion control in the vSwitch and provide a brief overview of how per-flow differentiation can be implemented.

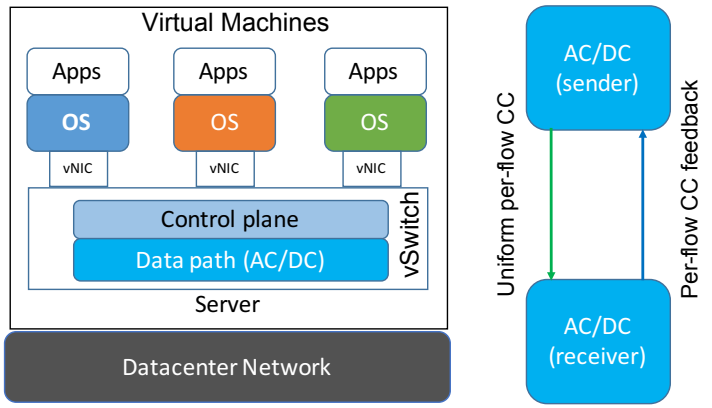


Figure 3.3: AC/DC high-level architecture.

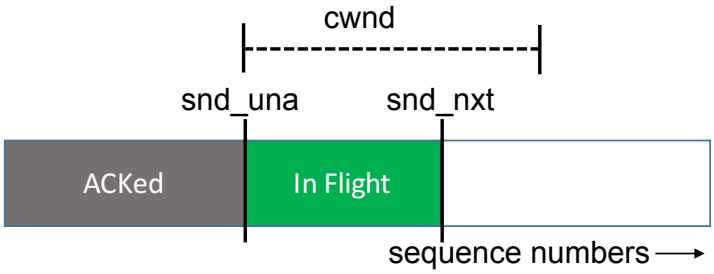


Figure 3.4: Variables for TCP sequence number space.

3.3.1 Obtaining Congestion Control State

Figure 3.3 shows the high-level structure of AC/DC. Since it is implemented in the datapath of the vSwitch, all traffic can be monitored. The sender and receiver modules work together to implement per-flow congestion control (CC).

We first demonstrate how congestion control state can be inferred. Figure 3.4 provides a visual of the TCP sequence number space. The snd_una variable is the first byte that has been sent, but not yet ACKed. The snd_nxt variable is the next byte to be sent. Bytes between snd_una and snd_nxt are in flight. The largest number of packets that can be sent

and unacknowledged is bounded by CWND. snd_una is simple to update: each ACK contains an acknowledgement number (ack_seq), and snd_una is updated when ack_seq $>$ snd_una. When packets traverse the vSwitch from the VM, snd_nxt is updated if the sequence number is larger than the current snd_nxt value. Detecting loss is also relatively simple. If ack_seq $\leqslant$ snd_una, then a local dupack counter is updated. Timeouts can be inferred when snd_una $<$ snd_nxt and an inactivity timer fires. The initial CWND is set to a default value of 10 [31]. With this state, the vSwitch can determine appropriate CWND values for canonical TCP congestion control schemes. We omit additional details in the interest of space.

3.3.2 Implementing DCTCP

This section discusses how to obtain DCTCP state and perform its congestion control.

ECN marking DCTCP requires flows to be ECN-capable, but the VM's TCP stack may not support ECN. Thus, all egress packets are marked to be ECN-capable on the sender module. When the VM's TCP stack does not support ECN, all ECN-related bits on ingress packets are stripped at the sender and receiver modules in order to preserve the original TCP settings. When the VM's TCP stack does support ECN, the AC/DC modules strip the congestion encountered bits in order to prevent the VM's TCP stack from decreasing rates too aggressively (recall DCTCP adjusts CWND proportional to the fraction of congested packets, while traditional schemes conservatively reduce CWND by half). A reserved bit in the header is used to determine if the VM's TCP stack originally supported ECN.

Obtaining ECN feedback In DCTCP, the fraction of packets experiencing congestion needs to be reported to the sender. Since the VM's TCP stack may not support ECN, the AC/DC receiver module monitors the total and ECN-marked bytes received for a flow. Receivers piggy-back the reported

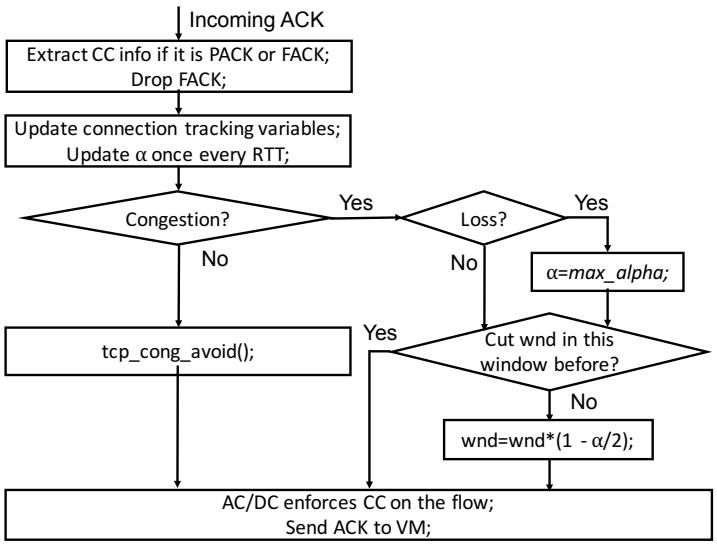


Figure 3.5: DCTCP congestion control in AC/DC.

totals on ACKs by adding an additional 8 bytes as a TCP Option. This is called a Piggy-backed ACK (PACK). The PACK is created by moving the IP and TCP headers into the ACK packet's skb headroom in Linux [114]. The totals are inserted into the vacated space and the memory consumed by the rest of the packet (*i.e.*, the payload) is left as is. The IP header checksum, IP packet length and TCP Data Offset fields are recomputed by the virtual switch and the TCP checksum is calculated by the NIC. The PACK option is stripped at the sender so it is not exposed to the VM's TCP stack.

If adding a PACK creates a packet larger than the MTU, the NIC offload feature (*i.e.*, TSO) will replicate the feedback information over multiple packets, which skews the feedback. Therefore, a dedicated feedback packet called a Fake ACK (FACK) is sent when the MTU will be violated (i.e., the orginal packet size plus the number of embeded bytes are larger than the MTU size). The FACK is sent in addition to the real TCP ACK. FACKs are also discarded by the sender after logging the included data. In practice, most feedback takes the form of PACKs.

DCTCP congestion control Once the fraction of ECN-marked packets is obtained, implementing DCTCP's logic is straightforward. Figure 3.5 shows the high-level design. First, congestion control (CC) information is extracted from FACKs and PACKs. Connection tracking variables (described in §3.3.1) are updated based on the ACK. The variable α is an EWMA of the fraction of packets that experienced congestion and is updated roughly once per RTT. If congestion was not encountered (no loss or ECN), then tcp_cong_avoid advances CWND based on TCP New Reno's algorithm, using slow start or congestion avoidance as needed. If congestion was experienced, then CWND must be reduced. DCTCP's instructions indicate the window should be cut at most once per RTT. Our implementation closely tracks the Linux source code, and additional details can be referenced externally [9, 18].

3.3.3 Enforcing Congestion Control

There must be a mechanism to ensure a VM's TCP flow adheres to the congestion control window size determined in the vSwitch. Luckily, TCP provides built-in functionality that can be reprovisioned for AC/DC. Specifically, TCP's flow control allows a receiver to advertise the amount of data it is willing to process via a receive window (RWND). Similar to other works [68, 116], the vSwitch overwrites RWND with its calculated CWND. In order to preserve TCP semantics, this value is overwritten only when it is smaller than the packet's original RWND. The VM's flow then uses $\min(\texttt{CWND}, \texttt{RWND})$ to limit how many packets it can send.

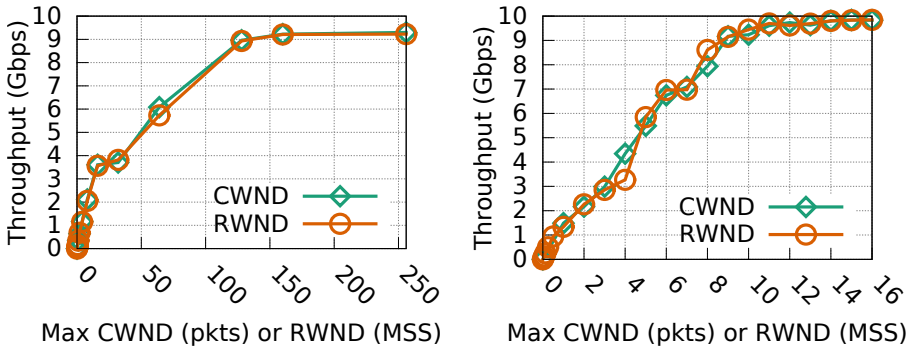
This enforcement scheme must be compatible with TCP receive window scaling to work in practice. Scaling ensures RWND does not become an unnecessary upper-bound in high bandwidth-delay networks and provides a mechanism to left-shift RWND by a window scaling factor [59]. The window scaling factor is negotiated during TCP's handshake, so AC/DC monitors handshakes to obtain this value. Calculated congestion windows

are adjusted accordingly. TCP receive window auto-tuning [110] manages buffer state and thus is an orthogonal scheme AC/DC can safely ignore.

Ensuring a VM's flow adheres to RWND is relatively simple. The vSwitch calculates a new congestion window every time an ACK is received. This value provides a bound on the number of bytes the VM's flow is now able to send. VMs with unaltered TCP stacks will naturally follow our enforcement scheme because the stacks will simply follow the standard. Flows that circumvent the standard can be policed by dropping excess packets not allowed by the calculated congestion window, which incentivizes tenants to respect the standard.

While simple, this scheme provides a surprising amount of flexibility. For example, TCP enables a receiver to send a TCP Window Update to update RWND [11]. AC/DC can create these packets to update windows without relying on ACKs. Additionally, the sender module can generate duplicate ACKs to to trigger retransmissions. This is useful when the VM's TCP stack has a larger timeout value than AC/DC (*e.g.*, small timeout values have been recommended for incast [122]). Another useful feature is when AC/DC allows a TCP stack to send *more* data. This can occur when a VM TCP flow aggressively reduces its window when ECN feedback is received. By removing ECN feedback in AC/DC, the VM TCP stack won't reduce CWND. In a similar manner, DCTCP limits loss more effectively than aggressive TCP stacks. Without loss or ECN feedback, VM TCP stacks grow CWND. This causes AC/DC's RWND to become the limiting window, and thus AC/DC can increase a flow's rate instantly when RWND $<$ CWND. Note, however, AC/DC cannot force a connection to send more data than the VM's CWND allows.

Another benefit of AC/DC is that it scales in the number of flows. Traditional software-based rate limiting schemes, like Linux's Hierarchical Token Bucket, incur high overhead due to frequent interrupts and contention [104] and therefore do not scale gracefully. NIC or switch-based



(a) MTU = 1.5KB.

(b) MTU = 9KB.

Figure 3.6: Using RWND can effectively control throughput.

rate limiters are low-overhead, but typically only provide a handful of queues. Our enforcement algorithm does not rate limit or buffer packets because it exploits TCP flow control. Therefore, rate limiting schemes can be used at a coarser granularity (*e.g.*, VM-level).

Finally, we outline AC/DC's limitations. Since AC/DC relies on sniffing traffic, schemes that encrypt TCP headers (*e.g.*, IPSec) are not supported. Our implementation only supports TCP, but we believe it can be extended to handle UDP similar to prior schemes [63, 111]. Implementing per-flow DCTCP-friendly UDP tunnels and studying its impact remains future work, however. And finally, while MPTCP supports per-subflow RWND [43], it is not included in our case study and a more detailed analysis is future work.

3.3.4 Per-flow Differentiation

AC/DC can assign different congestion control algorithms on a per-flow basis. This gives administrators additional flexibility and control by assigning flows to specific congestion control algorithms based on policy. For example, flows destined to the WAN may be assigned CUBIC and

flows destined within the datacenter may be set to DCTCP.

Administrators can also enable per-flow bandwidth allocation schemes. A simple scheme enforces an upper-bound on a flow's bandwidth. Traditionally, an upper-bound on a flow's CWND can be specified by the parameter snd_cwnd_clamp in Linux. AC/DC can provide similar functionality by bounding RWND. Figure 3.6 shows the behavior is equivalent. This graph can also be used to convert a desired upper-bound on bandwidth into an appropriate maximum RWND (the graph is created on an uncongested link to provide a lower bound on RTT).

In a similar fashion, administrators can assign different bandwidth priorities to flows by altering the congestion control algorithm. Providing differentiated services via congestion control has been studied [111, 123]. Such schemes are useful because networks typically contain only a limited number of service classes and bandwidth may need to be allocated on a finer-granularity. The the following equation is the original DCCP congestion control law, where α is a parameter in DCTCP congestion control algorithm and it is basically a measure of the extent of congestion in the network.

$$cwnd = cwnd(1 - \frac{\alpha}{2}) \tag{3.1}$$

We propose a unique priority-based congestion control scheme for AC/DC. Specifically, DCTCP's congestion control algorithm is modified to incorporate a priority, $\beta \in [0, 1]$:

$$rwnd = rwnd(1 - (\alpha - \frac{\alpha\beta}{2})) \tag{3.2}$$

Higher values of β give higher priority. When $\beta = 1$, Equation 3.2 simply converts to DCTCP congestion control. When $\beta = 0$, flows aggressively back-off (RWND is bounded by 1 MSS to avoid starvation). This equation alters multiplicative decrease instead of additive increase because increasing RWND cannot guarantee the VM flow's CWND will allow the

flow to increase its sending rate.

3.4 Implementation

This section outlines relevant implementation details. AC/DC was implemented in Open vSwitch (OVS) v2.3.2 [95] and about 1200 lines of code (many are debug/comments) were added. A high-level overview follows. A hash table is added to OVS, and flows are hashed on a 5-tuple (IP addresses, ports and VLAN) to obtain a flow's state. The flow entry state is 320 bytes and is used to maintain the congestion control state mentioned in §3.3. SYN packets are used to create flow entries, and FIN packets, coupled with a course-grained garbage collector, are used to remove flow entries. Other TCP packets, such as data and ACKs, trigger updates to flow entries. There are many more table lookup operations (to update flow state) than table insertions or deletions (to add/remove flows). Thus, Read-Copy-Update (RCU) hash tables [49] are used to enable efficient lookups. Additionally, individual spinlocks are used on each flow entry in order to allow for multiple flow entries to be updated simultaneously.

Putting it together, the high-level operation on a data packet is as follows. An application on the sender generates a packet that is pushed down the network stack to OVS. The packet is intercepted in the function ovs_dp_process_packet, where the packet's flow entry is obtained from the hash table. Sequence number state is updated in the flow entry and ECN bits are set on the packet if needed (see §3.3). If the packet's header changes, the IP checksum is recalculated. Note TCP checksumming is offloaded to the NIC. The packet is sent over the wire and received at the receiver's OVS. The receiver updates congestion-related state, strips off ECN bits, recomputes the IP checksum, and pushes the packet up the stack. ACKs eventually triggered by the packet are intercepted, where the congestion information is added. Once the ACK reaches the sender, the AC/DC

module uses the congestion information to compute a new congestion window. Then it modifies RWND with a memcpy, strips off ECN feedback and recomputes the IP checksum before pushing the packet up the stack. Since TCP connections are bi-directional, two flow entries are maintained for each connection.

Our experiments in §3.5.1 show the CPU overhead of AC/DC is small and several implementation details help reduce computational overhead. First, OVS sits above NIC offloading features (*i.e.*, TSO and GRO/LRO) in the networking stack. Briefly, NIC offloads allow the host to pass large data segments along the TCP/IP stack and only deal with MTU-sized packets in the NIC. Thus, AC/DC operates on a segment, rather than a per-packet, basis. Second, congestion control is a relatively simple algorithm, and thus the computational burden is not high. Finally, while AC/DC is implemented in software, it may be possible to further reduce the overhead with a NIC implementation. Today, "smart-NICs" implement OVS-offload functionality [79, 92], naturally providing a mechanism to reduce overhead and support hypervisor bypass (*e.g.*, SR-IOV).

3.5 Results

This section quantifies the effects of AC/DC and determines if the performance of DCTCP implemented in the vSwitch (*i.e.*, AC/DC) is equivalent to the performance of DCTCP implemented in the host TCP stack.

Testbed The experiments are conducted on a physical testbed with 17 IBM System x3620 M3 servers (6-core Intel Xeon 2.53GHz CPUs, 60GB memory) and Mellanox ConnectX-2 EN 10GbE NICs. Our switches are IBM G8264, each with a buffer of 9MB shared by forty-eight 10G ports.

System settings We run Linux kernel 3.18.0 which implements DCTCP as a pluggable module. We set $\mathtt{RTO}_{\mathtt{min}}$ to 10 ms [66, 122] and set parameters tcp_no_metrics_save, tcp_sack and tcp_low_latency to 1. Results are obtained

with MTU sizes of 1.5KB and 9KB, as networks typically use one of these settings. Due to space constraints, a subset of the results are presented and unless otherwise noted, the MTU is set to 9KB.

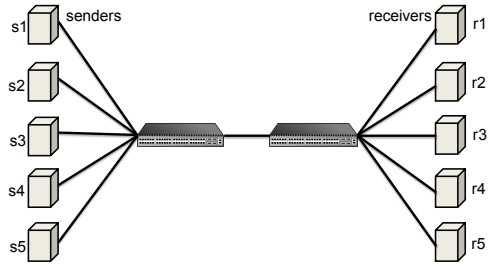
Experiment details To understand AC/DC performance, three different congestion control configurations are considered. The baseline scheme, referred to as *CUBIC*, configures the host TCP stack as CUBIC (Linux's default congestion control), which runs on top of an unmodified version of OVS. Our goal is to be similar to *DCTCP*, which configures the host TCP stack as DCTCP and runs on top of an unmodified version of OVS. Our scheme, *AC/DC*, configures the host TCP stack as CUBIC (unless otherwise stated) and implements DCTCP congestion control in OVS. In DCTCP and AC/DC, WRED/ECN is configured on the switches. In CUBIC, WRED/ECN is not configured.

The metrics used are: TCP RTT (measured by sockperf [115]), TCP throughput (measured by iperf), loss rate (by collecting switch counters) and Jain's fairness index [61]. In §3.5.2, flow completion time (FCT) [39] is used to quantify application performance. All benchmark tools are run in a container on each server, rather than in a VM.

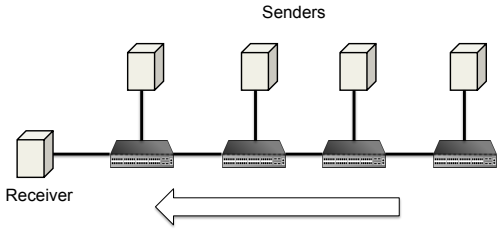
3.5.1 Microbenchmarks

We first evaluate AC/DC's performance using a set of microbenchmarks. The microbenchmarks are conducted on topologies shown in Figure 3.7.

Canonical topologies We aim to understand the performance of our scheme on two simple topologies. First, one long-lived flow is started per server pair (s_i to r_i) in Figure 3.7a. The average per-flow throughput of AC/DC, DCTCP and CUBIC are all 1.98Gbps. Figure 3.8 is a CDF of the RTT from the same test. Here, increases in RTT are caused by queueing delay in the switch. AC/DC achieves comparable RTT with DCTCP and significantly outperforms CUBIC.



(a) Dumbbell topology.



(b) Multi-hop, multi-bottleneck (parking lot) topology.

Figure 3.7: Experiment topologies.

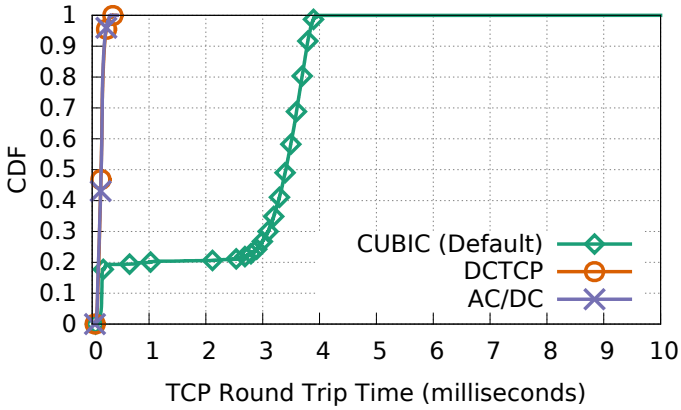
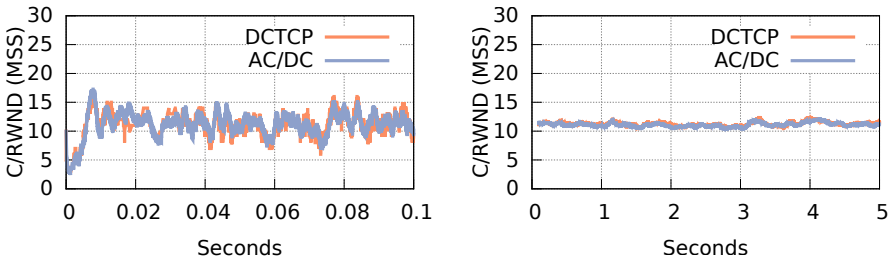


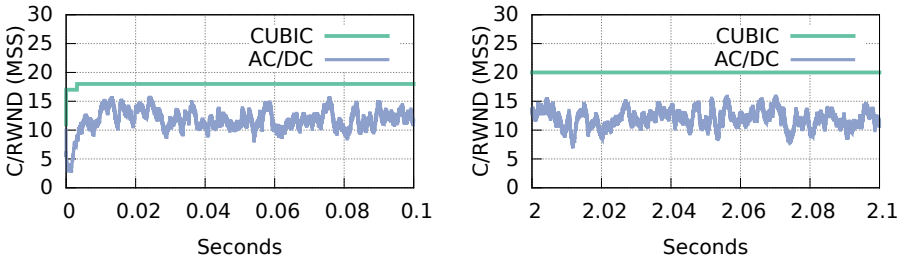
Figure 3.8: RTT of schemes on dumbbell topology.

Second, each sender in Figure 3.7b starts a long-lived flow to the receiver. Each flow traverses a different number of bottleneck links. CUBIC has an average per-flow throughput of 2.48Gbps with a Jain's fairness



(a) First 100 ms of a flow. (b) Moving average.

Figure 3.9: AC/DC's RWND tracks DCTCP's CWND (1.5KB MTU).



(a) Starting from 0 sec. (b) Starting from 2 sec.

Figure 3.10: Who limits TCP throughput when AC/DC is run with CUBIC? (1.5 KB MTU)

index of 0.94, and both DCTCP and AC/DC obtain an average throughput of 2.45Gbps with a fairness index of 0.99. The 50^{th} and 99.9^{th} percentile RTT for AC/DC (DCTCP, CUBIC) are 124µs (136µs, 3.3ms) and 279µs (301µs, 3.9ms), respectively.

Tracking window sizes Next, we aim to understand how accurately our AC/DC tracks DCTCP's performance at a finer level. The host's TCP stack is set to DCTCP and our scheme runs in the vSwitch. We repeat the experiment in Figure 3.7a and measure the RWND calculated by AC/DC. Instead of over-writing the RWND value in the ACKs, we simply log the value to a file. Thus, congestion is enforced by DCTCP and we can capture

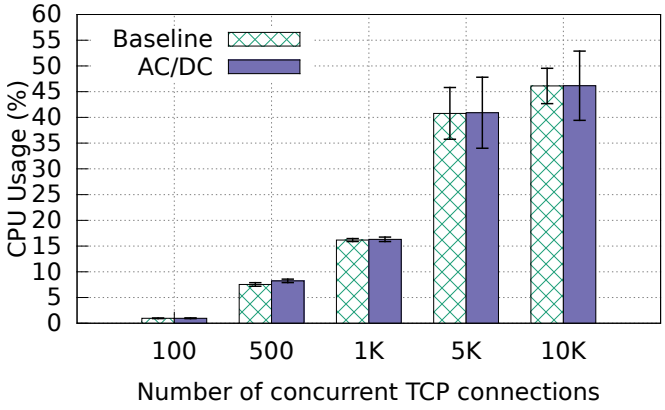


Figure 3.11: CPU overhead: sender side (1.5KB MTU).

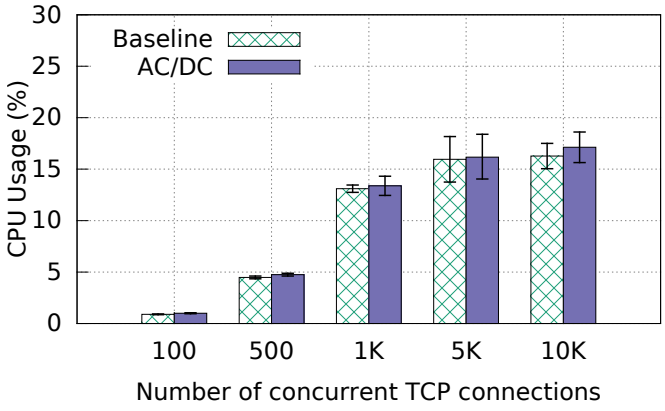


Figure 3.12: CPU overhead: receiver side (1.5KB MTU).

DCTCP's CWND by using tcpprobe [120]. We align the RWND and CWND values by timestamps and sequence numbers and show a timeseries in Figure 3.9. Figure 3.9a shows both windows for the first 100 ms of a flow and shows that AC/DC's calculated window closely tracks DCTCP's. Figure 3.9b shows the windows over a 100ms moving average are also similar. This suggests it is possible to accurately recreate congestion control in the vSwitch. These results are obtained with 1.5KB MTU. Trends for 9KB MTU

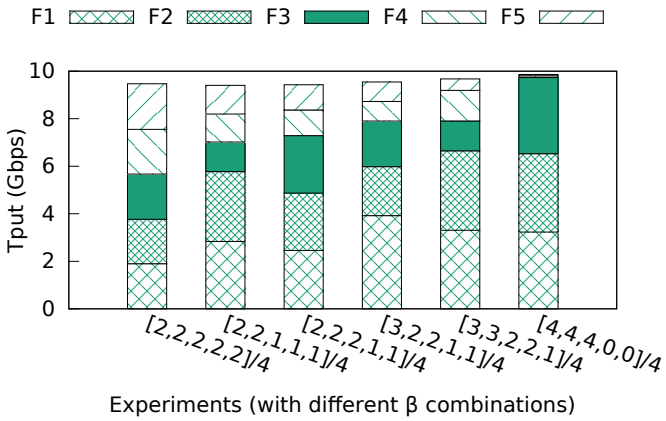


Figure 3.13: AC/DC provides differentiated throughput via QoS-based CC. β values are defined on a 4-point scale.

are similar but the window sizes are smaller.

We were also interested to see how often AC/DC's congestion window takes effect. We rerun the experiment (MTU is still 1.5KB), but set the host TCP stack to CUBIC. The RWND computed by AC/DC is both written into the ACK and logged to a file. We again use tcpprobe to measure CUBIC's CWND. Figure 3.10 is a timeseries (one graph from the start of the experiment and one graph 2 seconds in) that shows AC/DC's congestion control algorithm is indeed the limiting factor. In the absence of loss or ECN markings, traditional TCP stacks do not severely reduce CWND and thus AC/DC's RWND becomes the main enforcer of a flow's congestion control. Because DCTCP is effective at reducing loss and AC/DC hides ECN feedback from the host TCP stack, AC/DC's enforcement is applied often.

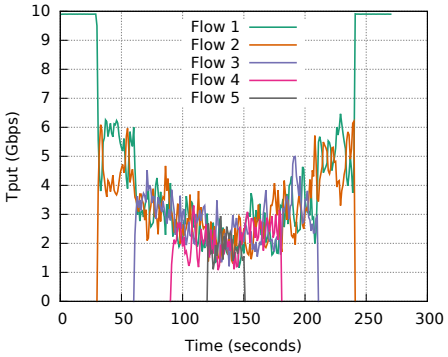
CPU overhead We measure the CPU overhead of AC/DC by connecting two servers to a single switch. Multiple simultaneous TCP flows are started from one server to the other and the total CPU utilization is measured on the sender and receiver using sar utility in Linux. Each flow is given time to perform the TCP handshake and when all are connected,

CC Variants	50^{th} percentile RTT (µs)		99^{th} percentile RTT (µs)		Avg Tput (Gbps)	
	mtu=1.5KB	mtu=9KB	mtu=1.5KB	mtu=9KB	mtu=1.5KB	mtu=9KB
CUBIC*	3232	3448	3641	3865	1.89	1.98
DCTCP*	128	142	232	259	1.89	1.98
CUBIC	128	142	231	252	1.89	1.98
Reno	120	149	235	248	1.89	1.97
DCTCP	129	149	232	266	1.88	1.98
Illinois	134	152	215	262	1.89	1.97
HighSpeed	119	147	224	252	1.88	1.97
Vegas	126	143	216	251	1.89	1.97

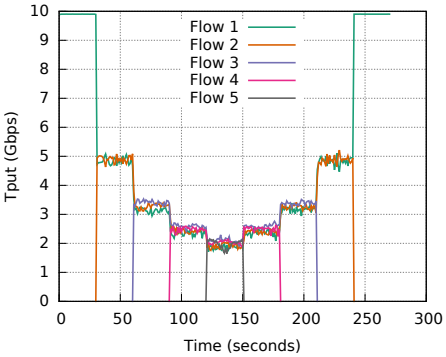
Table 3.14: AC/DC works with many congestion control variants. CUBIC*: CUBIC + standard OVS, switch WRED/ECN marking off. DCTCP*: DCTCP + standard OVS, switch WRED/ECN marking on. Others: different congestion control algorithms + AC/DC, switch WRED/ECN marking on.

each TCP client sends with a demand of 10 Mbps by sending 128KB bursts every 100 milliseconds (so 1,000 connections saturate the 10 Gbps link). The system-wide CPU overhead of AC/DC compared to the system-wide CPU overhead of baseline (*i.e.*, just OVS) is shown for the sender in Figure 3.11 and the receiver in Figure 3.12. Error bars show standard deviation over 50 runs. While AC/DC increases CPU usage in all cases, the increase is negligible. The largest difference is less than one percentage point: the baseline and AC/DC have 16.27% and 17.12% utilization, respectively for 10K flows at the receiver. The results are shown with 1.5KB MTU because smaller packets incur higher overhead. Note experiments with 9KB MTU have similar trends.

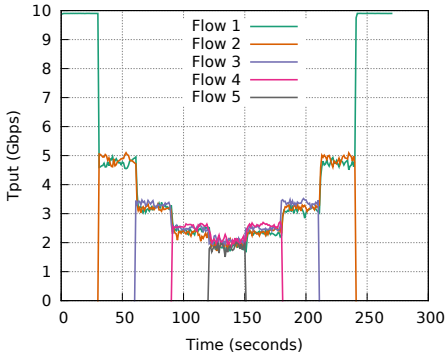
AC/DC flexibility AC/DC aims to provide a degree of control and flexibility over tenant TCP stacks. We consider two cases. First, AC/DC should work effectively, regardless of the tenant TCP stack. Table 3.14 shows the performance of our scheme when various TCP congestion control algorithms are configured on the host. Data is collected over 10 runs lasting 20 seconds each on the dumbbell topology (Figure 3.7a). The first two rows of the table, CUBIC* and DCTCP*, show the performance of each



(a) CUBIC convergence test.



(b) DCTCP convergence test.

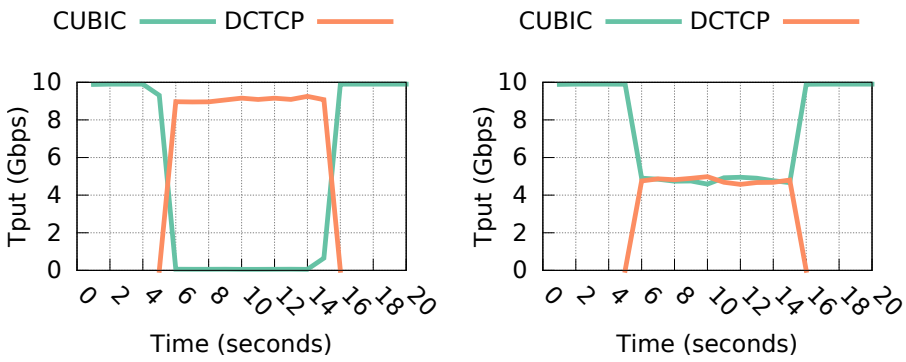


(c) AC/DC convergence test.

Figure 3.15: Convergence tests: flows are added, then removed, every 30 secs. AC/DC performance matches DCTCP.

stack with an unmodified OVS. The next six rows show the performance of a given host stack with AC/DC running DCTCP in OVS. The table shows AC/DC can effectively track the performance of DCTCP*, meaning it is compatible with popular delay-based (Vegas) and loss-based (Reno, CUBIC, etc) stacks.

Second, AC/DC enables an administrator to assign different congestion control algorithms on a per-flow basis. For example, AC/DC can provide the flexibility to implement QoS through differentiated congestion control.



(a) Default. (b) AC/DC.

Figure 3.16: (a) CUBIC gets little throughput when competing with DCTCP. (b) With AC/DC, CUBIC and DCTCP flows get fair share.

We fix the host TCP stack to CUBIC and alter AC/DC's congestion control for each flow by setting the β value (in Equation 3.2) for each flow in the dumbbell topology. Figure 3.13 shows the throughput achieved by each flow, along with its β setting. AC/DC is able to provide relative bandwidth allocation to each flow based on β. Flows with the same β value get similar throughputs and flows with higher β values obtain higher throughput. The latencies (not shown) remain consistent with previous results.

Fairness Three different experiments are used to demonstrate fairness. First, we show AC/DC can mimic DCTCP's behavior in converging to fair throughputs. We repeat the experiment originally performed by Alizadeh [9] and Judd [66] by adding a new flow every 30 seconds on a bottleneck link and then reversing the process. The result is shown in Figure 3.15. Figure 3.15a shows CUBIC's problems converging to fair allocations. Figures 3.15b and 3.15c show DCTCP and AC/DC performance, respectively. AC/DC tracks DCTCP's behavior. CUBIC's drop rate is 0.17% while DCTCP's and AC/DC's is 0%.

The second experiment is also repeated from Judd's paper [66]. ECN-

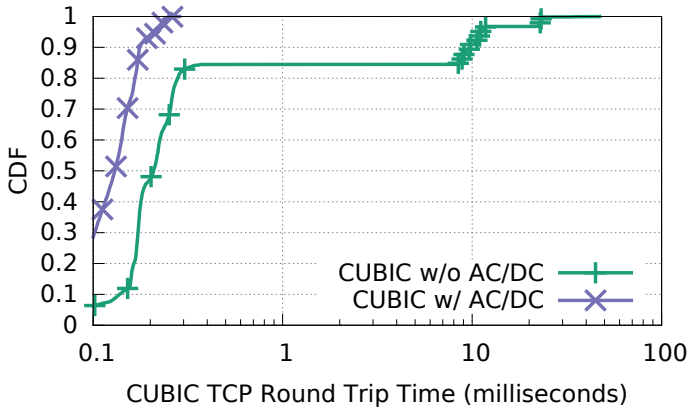
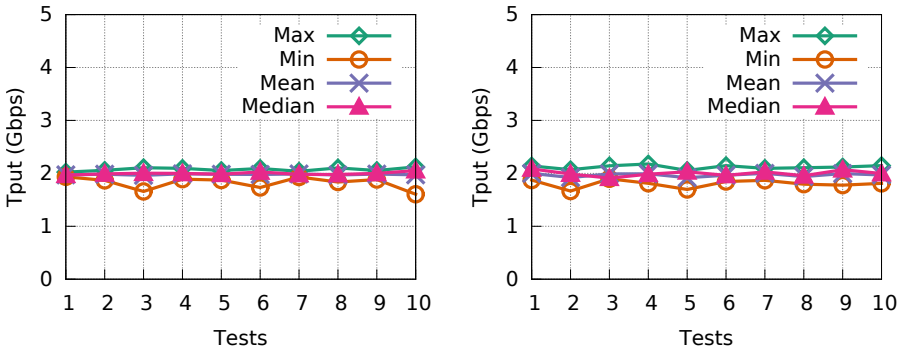


Figure 3.17: CUBIC experiences high RTT when competing with DCTCP. AC/DC fixes this issue.

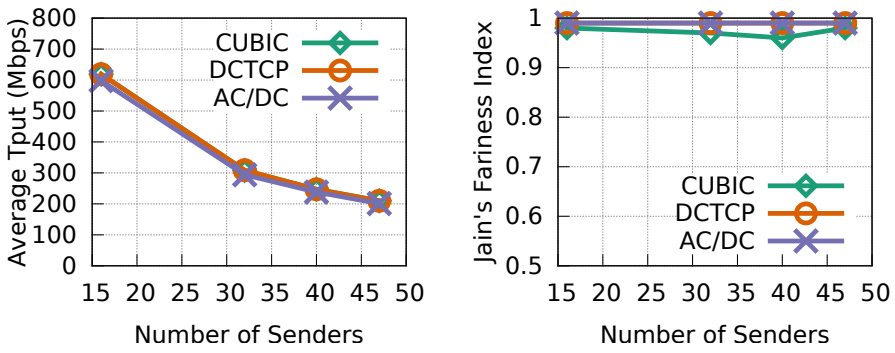


(a) All DCTCP.

(b) 5 different CCs (AC/DC).

Figure 3.18: AC/DC improves fairness when VMs implement different CCs. DCTCP performance shown for reference.

capable and non-ECN-capable flows do not coexist well because switches drop non-ECN packets when the queue length is larger than the marking threshold. Figure 3.16a shows the throughput of CUBIC suffers when CUBIC (with no ECN) and DCTCP (with ECN) traverse the same bottleneck link. Figure 3.16b shows AC/DC alleviates this problem because it forces all flows to become ECN-capable. Figure 3.17 shows CUBIC's



(a) Average throughput.

(b) Fairness.

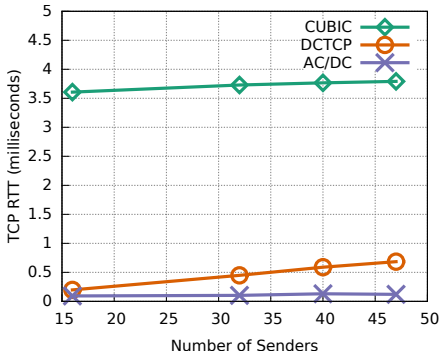
Figure 3.19: Many to one incast: throughput and fairness.

RTT is extremely high in the first case because switches drop non-ECN packets (the loss rate is 0.18%) and thus there is a significant number of retransmissions. However, AC/DC eliminates this issue and reduces latency.

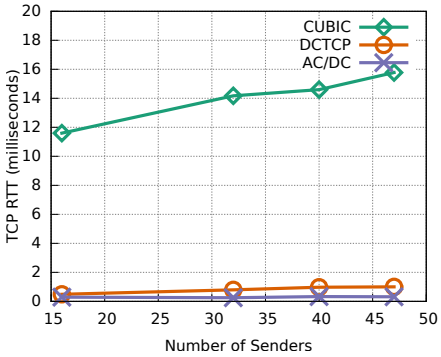
The last experiment examines the impact of having multiple TCP stacks on the same fabric. Five flows with different congestion control algorithms (CUBIC, Illinois, HighSpeed, New Reno and Vegas) are started on the dumbbell topology. This is the same experiment as in Figure 3.1. Figure 3.18a shows what happens if all flows are configured to use DCTCP and Figure 3.18b shows when the five different stacks traverse AC/DC. We can see AC/DC closely tracks the ideal case of all flows using DCTCP, and AC/DC and DCTCP provide better fairness than all CUBIC (Figure 3.1b).

3.5.2 Macrobenchmarks

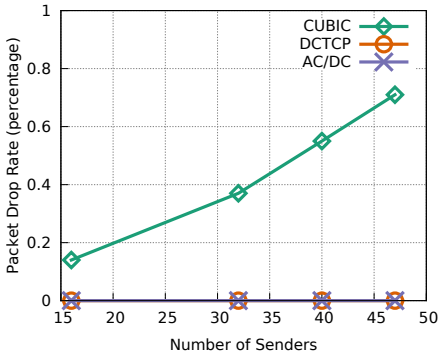
In this section we attach all servers to a single switch and run a variety of workloads to better understand how well AC/DC tracks DCTCP's performance. Experiments are run for 10 minutes. A simple TCP application



(a) 50^{th} percentile RTT.



(b) 99.9^{th} percentile RTT.



(c) Packet drop rate.

Figure 3.20: Many to one incast: RTT and packet drop rate. AC/DC can reduce DCTCP's RTT by limiting window sizes.

sends messages of specified sizes to measure FCTs.

Incast In this section, we evaluate incast scenarios. To scale the experiment, 17 physical servers are equipped with four NICs each and one flow is allocated per NIC. In this way, incast can support up to 47-to-1 fan-in (our switch only has 48 ports). We measure the extent of incast by increasing the number of concurrent senders to 16, 32, 40 and 47. Figure 3.19 shows throughput and fairness results. Both DCTCP and AC/DC obtain a fairness index greater than 0.99 and get comparable throughput as CUBIC.

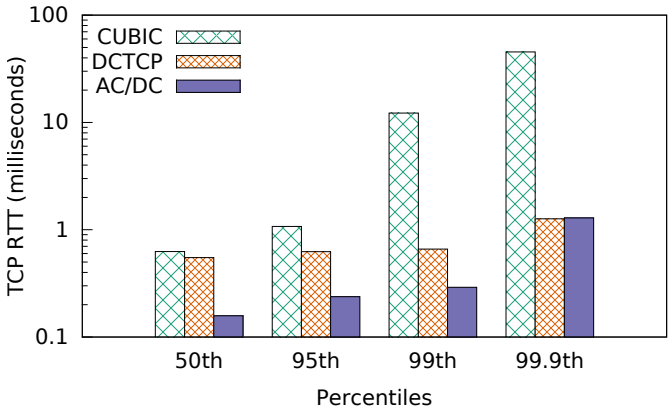


Figure 3.21: TCP RTT when almost all ports are congested.

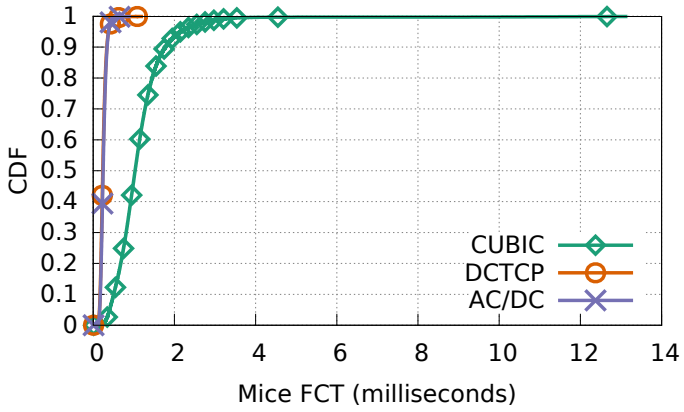
Figure 3.20 shows the RTT and packet drop rate results. When there are 47 concurrent senders, DCTCP can reduce median RTT by 82% and AC/DC can reduce by 97%; DCTCP can reduce 99.9^{th} percentile RTT by 94% and AC/DC can reduce by 98%. Both DCTCP and AC/DC have 0% packet drop rate. It is curious that AC/DCâŁ™s performance is better than DCTCP when the number of senders increases (Figure 3.20a). The Linux DCTCP code puts a lower bound of 2 packets on CWND. In incast, we have up to 47 concurrent competing flows and the network's MTU size is 9KB. In this case, the lower bound is too high, so DCTCP's RTT increases gradually with the number of senders. This issue was also found in [66]. AC/DC controls RWND (which is in bytes) instead of CWND (which is in packets) and RWND's lowest value can be much smaller than 2*MSS. We verified modifying AC/DC's lower bound caused identical behavior.

The second test aims to put pressure on the switch's dynamic buffer allocation scheme, similar to an experiment in the DCTCP paper [9]. To this end, we aim to congest every switch port. The 48 NICs are split into 2 groups: group A and B. Group A has 46 NICs and B has 2 (denoted B_1 and B_2). Each of the 46 NICs in A sends and receives 4 concurrent flows within A (*i.e.*, NIC i sends to $[i+1, i+4]$ mod 46). Meanwhile, all of the

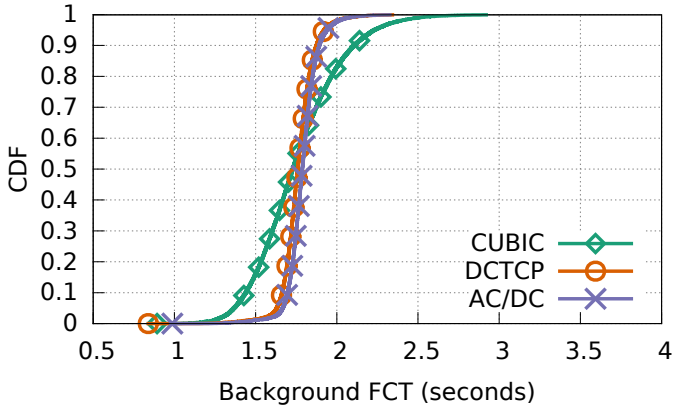
NICs in A send to B_1, creating a 46-to-1 incast. This workload congests 47 out of 48 switch ports. We measure the RTT between B_2 and B_1 (i.e., RTT of the traffic traversing the most congested port) and the results are shown in Figure 3.21. The average throughputs for CUBIC, DCTCP, and AC/DC are 214, 214 and 201 Mbps respectively, all with a fairness index greater than 0.98. CUBIC has an average drop rate of 0.34% but the most congested port has a drop rate as high as 4%. This is why the 99.9^{th} percentile RTT for CUBIC is very high. The packet drop rate for both DCTCP and AC/DC is 0%.

Concurrent stride workload In concurrent stride, 17 servers are attached to a single switch. Each server i sends a 512MB flow to servers $[i+1, i+4]$ mod 17 in sequential fashion to emulate background traffic. Simultaneously, each server i sends 16KB messages every 100 ms to server $(i+8)$ mod 17. The FCT for small flows (16KB) and background flows (512MB) are shown in Figure 3.22. For small flows, DCTCP and AC/DC reduce the median FCT by 77% and 76% respectively. At the 99.9^{th} percentile, DCTCP and AC/DC reduce FCT by 91% and 93%, respectively. For background flows, DCTCP and AC/DC offer similar completion times. CUBIC has longer background FCT because its fairness is not as good as DCTCP and AC/DC.

Shuffle workload In shuffle, each server sends 512MB to every other server in random order. A sender sends at most 2 flows simultaneously and when a transfer is finished, the next one is started until all transfers complete. Every server i also sends a 16 KB message to server $(i+8)$ mod 17 every 100 ms. This workload is repeated for 30 runs. The FCT for each type of flow is shown in Figure 3.23. For small flows, DCTCP and AC/DC reduce median FCT by 72% and 71% when compared to CUBIC. At the 99.9^{th} percentile, DCTCP and AC/DC reduce FCTs by 55% and 73% respectively. For large flows, CUBIC, DCTCP and AC/DC have almost identical performance.



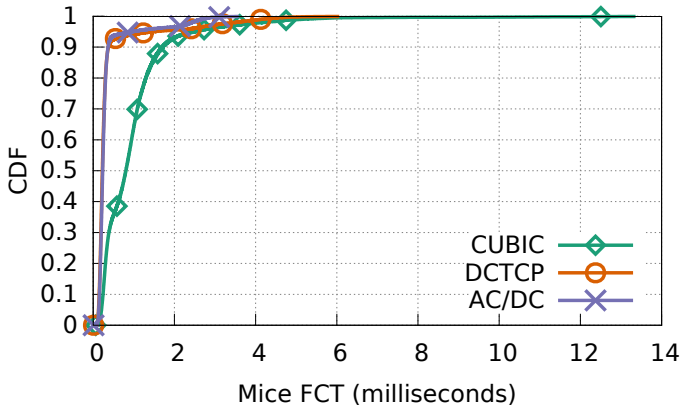
(a) Mice flow completion times.



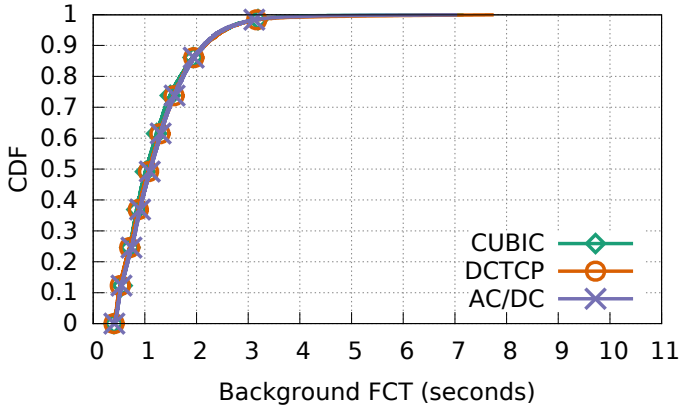
(b) Background flow completion times.

Figure 3.22: CDF of mice and background FCTs in concurrent stride workload.

Trace-driven workloads Finally, we run trace-driven workloads. An application on each server builds a long-lived TCP connection with every other server. Message sizes are sampled from a trace and sent to a random destination in sequential fashion. Five concurrent applications on each server are run to increase network load. Message sizes are sampled from a web-search [9] and a data-mining workload [8, 46], whose flow size



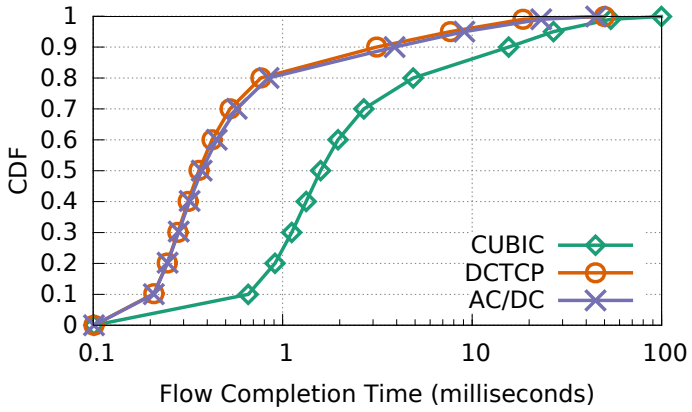
(a) Mice flow completion times.



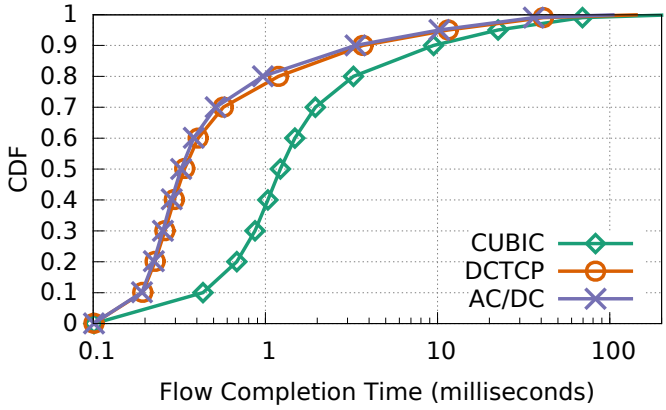
(b) Background flow completion times.

Figure 3.23: CDF of mice and background FCTs in shuffle workload.

distribution has a heavier tail. Figure 3.24 shows a CDF of FCTs for mice flows (smaller than 10KB) in the web-search and data-mining workloads. In the web-search workload, DCTCP and AC/DC reduce median FCTs by 77% and 76%, respectively. At the 99.9^{th} percentile, DCTCP and AC/DC reduce FCTs by 50% and 55%, respectively. In the data-mining workload, DCTCP and AC/DC reduce median FCTs by 72% and 73%, respectively. At the 99.9^{th} percentile, DCTCP and AC/DC reduce FCTs by 36% and



(a) Web-search workload.



(b) Data-mining workload.

Figure 3.24: CDF of mice (flows<10KB) FCT in web-search and data-mining workloads.

53% respectively.

Evaluation summary The results validate that congestion control can be accurately implemented in the vSwitch. AC/DC tracks the performance of an unmodified host DCTCP stack over a variety of workloads with little computational overhead. Furthermore, AC/DC provides this functionality over various host TCP congestion control configurations.

3.6 Summary

Today's datacenters host a variety of VMs (virtual machines) in order to support a diverse set of tenant services. Datacenter operators typically invest significant resources in optimizing their network fabric, but they cannot control one of the most important components of avoiding congestion: TCP's congestion control algorithm in the VM. In this chapter, we present a technology that allows administrators to regain control over arbitrary tenant TCP stacks by enforcing congestion control in the vSwitch. Our scheme, called AC/DC TCP, requires no changes to VMs or network hardware. Our approach is scalable, light-weight, flexible and provides a policing mechanism to deal with non-conforming flows. In our evaluation the CPU overhead is less than one percentage point and our scheme is shown to effectively enforce an administrator-defined congestion control algorithm over a variety of tenant TCP stacks.

4

Low Latency Software Rate Limiters for Cloud Networks

4.1 Introduction

Bandwidth allocation is an indispensable feature in multi-tenant clouds. It guarantees the performance of various applications from multiple tenants running on the same physical server. Bandwidth allocation is often implemented by *software rate limiters*[1] in the operating system (e.g., Linux Hierarchical Token Bucket, aka HTB) due to their flexibility and scalability. However, rate limiters either uses traffic policing (i.e., dropping packets when packet arrival rate is above the desired rate) or traffic shaping (i.e., queueing packets in a large queue to absorb burst and send packets to the network based on token and bucket algorithms). Thus, bandwidth allocation, low latency and low loss rate can be not achieved at the same time. We conduct performance measurements in a public cloud platform (CloudLab [1]) and find that one of the most widely used software rate limiters, HTB, dramatically increases network latency.

Previous work mainly focused on solving "in-network" queueing latency (i.e., latency in switches) [9, 56, 84], but little research effort has been done to solve the latency, packet loss and burstiness issues for the software rate limiters on the end-host. To this end, we explore how we can address the performance issues associated with rate limiters on the

[1]In this chapter, we use software rate limiter and rate limiter interchangeably.

end-host in this chapter. Inspired by the efforts to reduce queueing latency in hardware switches, we first extend ECN into software rate limiters and configure DCTCP on the end-points. However, in our tests, we find simply applying DCTCP+ECN on the end-host causes a problem — TCP throughput tends to oscillate (between 50% to 95% in some cases, see Section 4.3). And throughput oscillation would significantly degrade application performance.

There are two issues with simply applying DCTCP+ECN on end-host rate limiters. The first is that, different from hardware switches in the network, end-hosts process TCP segments instead of MTU-sized packets. TCP Segmentation Offload (TSO) [3] is an optimization technique that is widely used in modern operating systems to reduce CPU overhead for fast speed networks. TSO is enabled in Linux by default. Because of TSO, TCP segment size can reach 64KB in the default setting. That means, if we mark the ECN bits in one TCP segment, a bunch of consecutive MTU-sized packets are marked because the ECN bits in the header are copied into each packet in the NIC. On the other hand, if a TCP segment is not marked, then none of the packets in this segment is marked. This kind of coarse-grained segment-level marking leads to an inaccurate estimation of congestion level, and consequently leads to throughput oscillation. The second issue with DCTCP+ECN is that its congestion control loop latency is large. There are three types of traffic in datacenter networks — 1) intra-datacenter traffic (i.e., east-west traffic) 2) inter-datacenter traffic and 3) traffic that goes across WANs to remote clients. DCTCP+ECN's control loop latency is one RTT. But RTT is affected by many factors. For intra-datacenter traffic, RTT is affected by "in-network" congestion. For traffic that goes across WANs to remote clients, its RTT can be tens of milliseconds. Congestion control loop latency is not bounded and congestion control actions can rely on stale congestion feedback. The long congestion control loop latency is further exacerbated by the fact that traffic on the end-host

is bursty because of TCP windowing scheme and TSO optimizations [71].

To address the shortcomings of DCTCP+ECN, we design our first mechanism — *direct ECE marking (DEM)*. In DEM, when an ACK is received, DEM checks the real-time queue length in its corresponding rate limiter in the virtual switch, and if the queue length exceeds a pre-configured threshold, DEM *marks the ACK packet instead of data packets*. By marking TCP ACK, we mean set TCP ACK's ECN-Echo (ECE) bit. Directly marking TCP ECE avoids the two shortcomings in DCTCP+ECN. First, the congestion control loop latency is almost reduced to 0 because it is based on real-time queue length, not the queue length one RTT ago. Second, it can avoid coarse-grained segment-level ECN marking on the transmitting path. Our design (Section 4.4) and experiment (Section 4.5) show that DEM eliminates throughput oscillation.

There is a prerequisite to deploying DEM — end-hosts are able to react to ECN marking correctly (e.g., DCTCP). However, in public clouds, this prerequisite does not always hold because the cloud operator does not have access to the network stack configuration in tenants' VMs. Even the tenant VM is configured with DCTCP, not all the flows are ECN-capable. For example, TCP flows between cloud and clients are usually not ECN-capable because ECN is not widely used in WANs [74]. To make our solution more generic, we adopt the mechanism from AC/DC [56], i.e., performing TCP CWND computation outside of tenants' VMs and enforcing congestion control decisions via rewriting RWND. We also design and implement a window-based TIMELY [84]-like congestion control algorithm. We name this mechanism SPRING. SPRING completely gets rid of ECN marking and can work for both ECN-capable and non ECN-capable flows. SPRING also avoids the two issues that DCTCP+ECN has because for each return ACK, we compute CWND according to instantaneous rate limiter queue length and queue length gradient and rewrite RWND in TCP ACK header.

The contributions of this chapter are as follows:

1. We point out and measure the latency caused by software rate limiters on the end-host in multi-tenant clouds, and also show that simply applying DCTCP+ECN is not sufficient to achieve constant bandwidth saturation.

2. We identify the limitations of DCTCP+ECN in end-host networking and propose DEM to solve the throughput oscillation problem. We also propose SPRING to make the solution more generic and deployable.

3. We perform a preliminary evaluation to show our solutions can achieve high bandwidth saturation, low latency, low oscillation and throughput fairness with negligible CPU overhead.

4.2 Background

4.2.1 Bandwidth Allocation and Rate Limiters

Bandwidth allocation is a must for cloud networks [63, 109, 111]. Software Rate limiters, e.g., Linux Traffic Control (TC), are the commonly used methods to provide per-VM/container bandwidth allocation because of their flexibility and scalability. In Linux traffic control, there are two kinds of rate limiting methods. The first method is traffic shaping. In traffic shaping, packets are first pushed into a queue and delayed, then packets are scheduled and sent to the network based on token and bucket-based mechanisms. Shaping traffic ensures the traffic speed meets the desired rate. A nice outcome of traffic shaping that it effectively makes bursty traffic (TCP traffic is bursty because of offloading optimizations such TSO) more smooth. Therefore, shaping traffic is good for switch buffers and avoid packet drops in the switches, this is especially true because modern datacenter networks are built with shallow-buffered switches. Another good property of traffic shaping is that it can effectively absorb bursty

traffic and avoid packet loss on the end-host. The second rate limiting method is traffic policing. Traffic policing monitors the packet arrival rate, it only pushes a packet into the queue when traffic rate is not above the desired rate, otherwise the packet is dropped. Traffic policing is a less accurate and less effective rate limiting method compared with traffic shaping [4].

4.2.2 High throughput and Low Latency Datacenter Networks

Datacenter networks need to support high throughput, low latency, and low loss rate to meet various kinds of application's SLAs. There have been many research work about reducing "in-network" latency and packet loss rate caused by network congestion [9, 15, 48, 56, 84, 136]. The work on reducing datacenter network latency can be roughly classified into two categories: 1) congestion control based such as DCTCP [9] and DCQCN [136] and 2) priority-based such as PIAS [15] and QJUMP [48]. However, for bandwidth allocation functionality, prioritizing some flows is not realistic in a virtualized setting because the hypervisor has no way of knowing which VM traffic is more important. Also, if high priority flows always come in, then low priority flows suffer from starvation. Further, flows having the same priority still lead to queueing latency. To date, little research effort has been done to explore latency, packet loss and burstiness issues of rate limiters on the end-host. We will show that queueing latency in the rate limiters on the end-host is not negligible, and it can increase end-to-end latency significantly. We will also demonstrate that end-host rate limiters have their own unique characteristics and we should rethink high throughput, low latency, and low loss rate solutions for end-host networking.

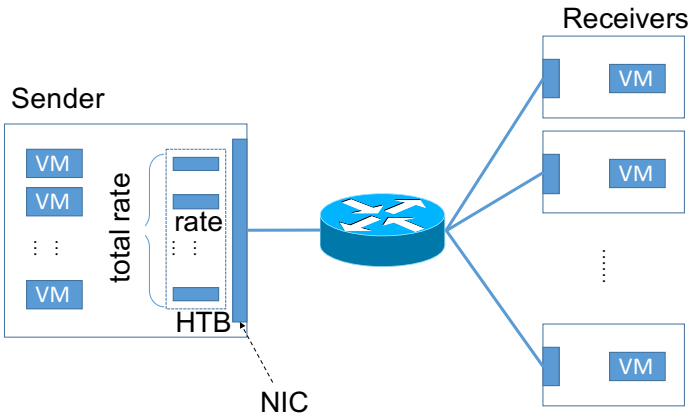


Figure 4.1: Experiment setup

4.3 Measurement and Analysis

4.3.1 Performance of Linux HTB

We first measure the performance of Linux HTB rate limiter. Compared with other software rate limiters, HTB ensures that a minimum amount of bandwidth is guaranteed to each traffic class and if the required minimum amount of bandwidth is not fully used, the remaining bandwidth is distributed to other classes. The distribution of spare bandwidth is in proportion to the minimum bandwidth specified to a class [2]. We set up servers in CloudLab, and each server is equipped with 10 Gbps NICs. The experiment setup is shown in Figure 4.1. In the experiments, we configure a rate limiter for each sender VM; for each sender-receiver VM pair, we use iperf to send background traffic. We configure HTB to control the bandwidth for each pair, and vary the number of flows between each pair. We also control the total sending rate of all pairs (i.e., using the hierarchical feature of HTB). With these settings, we run sockperf between sender and receiver pairs to measure TCP RTT. We conducted two sets of experiments. In the first set of experiments, we have one sender VM and one receiver

numReceiver	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
numFlows/receiver	-	1	1	1	1	2	2	2	2	8	8	8	8	16	16	16	16
rate/receiver (Gbps)	-	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
total rate (Gbps)	-	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
total tput (mbps)	-	951	1910	3820	7380	945	1910	3820	7500	958	1915	3827	7627	960	1920	3829	7655
b/w saturation (%)	-	95.1	95.5	95.5	95.3	94.5	95.5	95.5	93.8	95.8	95.8	95.7	95.3	96	96	95.7	95.7
50% RTT (us)	116	957	883	643	1583	1513	1078	1047	853	2316	1766	1529	1110	3192	2373	1880	1262
99.9% RTT (us)	237	1115	1000	706	1673	1701	1203	1132	933	2527	1939	1637	1208	3320	2511	2016	1486

Table 4.2: HTB experiments for one receiver VM

numReceiver	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
numFlows/receiver	-	1	1	1	1	2	2	2	2	8	8	8	8	16	16	16	16
rate/receiver (Gbps)	-	2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5
total rate (Gbps)	-	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
total tput (mbps)	-	9420	9420	9410	9410	9410	9420	9420	9420	9224	9392	9321	9401	9182	9161	9225	9296
b/w saturation (%)	-	94.2	94.2	94.1	94.1	94.1	94.2	94.2	94.2	92.2	93.9	93.2	94.0	91.8	91.6	92.3	93.0
50% RTT (us)	118	475	797	1515	1658	699	916	1006	1036	1512	1407	1410	1587	2023	2040	2064	1986
99.9% RTT (us)	135	551	849	1626	1751	983	989	1102	1115	1697	1673	1532	1768	2147	2182	2185	2100

Table 4.3: HTB experiments for two receiver VMs

VM. We specify the speed of the sender side rate limiter to 1Gbps, 2Gbps, 4Gbps and 8Gbps. We vary the number of iperf flows (1, 2, 8 and 16) from the sender to receiver. In the second set of experiments, we configure two rate limiters on the sender server and set up two VMs (one rate limiter each VM). We configure the minimum rate of each rate limiter to 2Gbps, 3Gbps, 4Gbps and 5Gbps and the total rate of the two rate limiters is always 10Gbps.

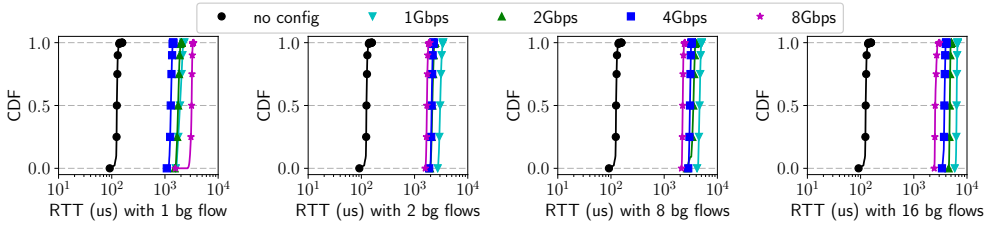


Figure 4.4: HTB experiment: one receiver VM, varying rate limiting and number of background flows

The experiments results are shown in Table 4.2 and 4.3. In all experi-

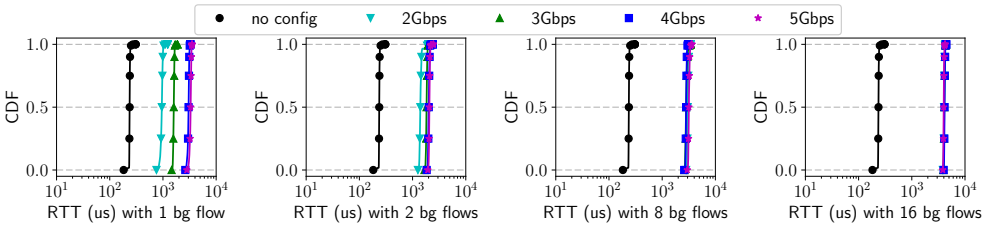


Figure 4.5: HTB experiment: two receiver VMs, varying rate limiting and number of background flows

ment scenarios, network saturation ratio is from 91%-96%. Note that in Table 4.3, if the sum of individual rate limiter's rate is smaller than the configured total rate, HTB would allow all flows to utilize and compete for the spare bandwidth. Another observation is that more flows lead to lower bandwidth saturation, because there is more competition between flows, which leads to throughput oscillation.

We further look into the scenario of one receiver VM. We visualize the TCP RTT results in Figure 4.4; each subfigure shows the CDF of sockperf trace RTT with different rate limiter speed. Different subfigures show scenarios with a varying number of background iperf flows. We can draw three conclusions based on the measurement data. First, TCP RTT increases dramatically when packets go through a congested rate limiter. In the baseline case where no HTB is configured and no iperf background flow running, the median TCP RTT is 62us. While with one background iperf flow and rate limiter speed from 1Gbps to 8Gbps, the median RTT increases to 957us-1583us, which is 15-25X larger compared with the baseline case. Second, TCP RTT increases with more background flows running. For example, with 1Gbps rate limiting, the median RTT is 957us for one background flow and 3192us for 16 background flows. Third, TCP RTT decreases with larger rate limiter speed configured [2]. Because rate limiter

[2]The only exception is the case with one background flow and 8Gbps rate limiting, which is suspected to be experiment noise.

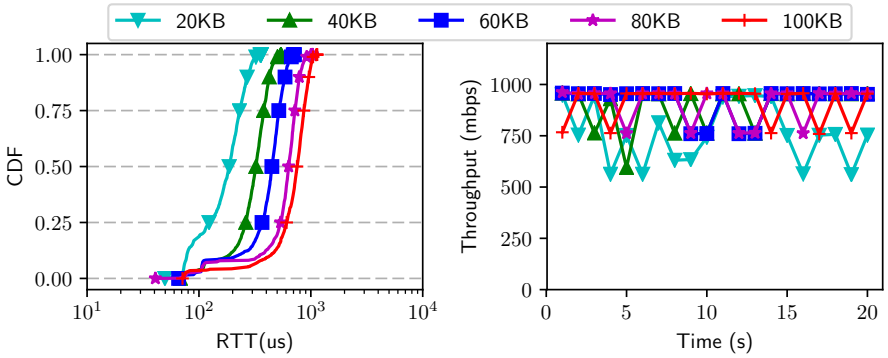
speed determines the dequeue speed of HTB queue, thus, with larger dequeue speed, the queue tends to be drained faster.

In the experiments with two receiver VMs (Figure 4.5), we can draw the same conclusions regarding RTT increase and the impact of the number of background flows. The difference is that TCP RTT increases with larger rate limiting speed configured. In these experiments, we did not constrain the total rate, allowing HTB to utilize spare bandwidth. Thus, the dequeue speed is constant in each figure (10Gbps/numFlow). The possible reason for the trend is that enqueue speed is higher when rate limiter speed is higher. For a fixed dequeue speed, larger enqueue speed implies higher latency.

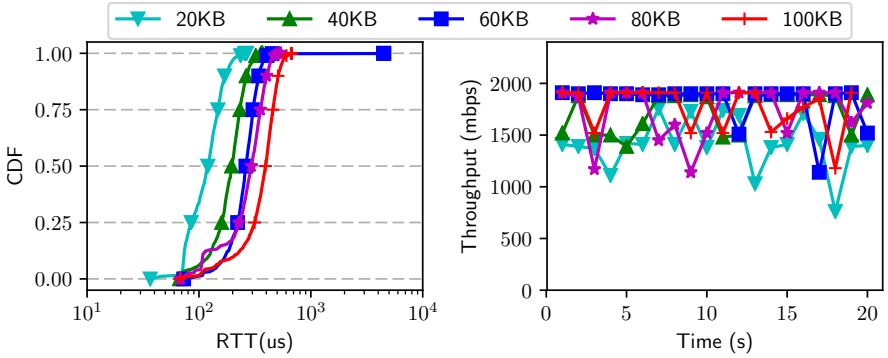
4.3.2 Strawman Solution: DCTCP + ECN

Inspired by solutions to reduce queueing latency in switches [9], we test a strawman solution. In the strawman solution, we implement ECN marking in Linux HTB queues and enable DCTCP at the end-points. For ECN marking, there is a tunable parameter — *marking threshold*. When the queue length exceeds the marking threshold, all enqueued packets would have their ECN bits set; otherwise, packets are not modified. DCTCP reacts to ECN marking and adjusts sender's congestion window based on the ratio of marked packets [9].

We set up experiments with one sender and one receiver. We configure the HTB rate limiter to be 1Gbps and 2Gbps, and vary ECN marking threshold. The experiment results are shown in Figure 4.6. We observe that TCP RTT can be reduced significantly (<1ms) by extending ECN into rate limiter queues. For example, with the marking threshold set to 60KB, median TCP RTT is 224us (Figure 4.6a), which is less than 1/4 of native HTB's (957us). A smaller ECN marking threshold can achieve even lower latency — with the threshold from 100KB to 20KB, median TCP RTT is reduced from 375 us to 93us.



(a) Rate Limiting: 1Gbps



(b) Rate Limiting: 2Gbps

Figure 4.6: DCTCP experiments, 1 flow, varying threshold

While latency can be improved, we observe a negative effect of DCTCP + ECN on the throughput, as shown in Figure 4.6. TCP throughput appears to have large oscillation, which implies applications cannot get constantly high throughput and the bandwidth is not fully utilized. For example, with 2Gbps rate limiting (Figure 4.6b), even we set the threshold to be 100KB (much larger than the best theoretical value according to [9]), there is still occasional low throughput (e.g., 1000Mbps) within a 20-second experiment duration.

4.3.3 Throughput Oscillation Analysis

Directly applying the existing ECN marking technique to rate limiter queue causes TCP throughput oscillation. There are two reasons. First, end-host networking stacks enable optimization techniques such TSO (TCP Segmentation Offload [3]) to improve throughput and reduce CPU overhead. Therefore, end-host networking stack (including the software rate limiters) processes TCP segments instead of MTU-sized packets. The maximum TCP segment size is 64KB by default. A TCP segment's IP header is copied into each MTU-sized packet in the NIC using TSO. So marking one TCP segment in the rate limiter queue results in a bunch of consecutive MTU-sized packets to be marked. For example, marking a 64KB segment means 44 consecutive Ethernet frames are marked. Such coarse-grained segment-level marking finally causes the accuracy of congestion estimation in DCTCP to be greatly decreased.

Second, ECN marking happens on the transmitting path, and it takes one RTT for congestion feedback to travel back to the sender before congestion control actions are taken. Moreover, TCP RTT can be affected by the "in network" latency. "In network" latency can be milliseconds or tens of milliseconds. Thus, this one RTT control loop latency would cause the ECN marks to be "outdated", not precisely reflecting the instantaneous queue length when the marks are used for congestion window update in DCTCP. Without congestion control based on instantaneous queue length, the one-RTT control loop latency exacerbates the incorrect segment-level ECN marking. Thus, congestion window computation in DCTCP tends to change more dramatically, leading to the throughput oscillation.

4.3.4 Call for Better Software Rate Limiters

We list the design requirements for better software rate limiters, as shown in Table 4.7. First the rate limiter should be able to provide high network

R1	high throughput
R2	low throughput oscillation
R3	low latency
R4	generic

Table 4.7: Software rate limiter design requirements

	Stable Tput	Low Latency	Generic
Raw HTB	✓	✗	✓
DCTCP+ECN	✗	✓	✗

Table 4.8: Raw HTB and DCTCP+ECN can not meet the design requirements

throughput. Second, network throughput should be stable with low oscillation. Third, flows traversing the rate limiter should experience low latency. Finally, the rate limiter can be able to handle various kinds of traffic — ECN-capable and non ECN-capable. Neither raw HTB nor HTB with DCTCP+ECN can meet the four design requirements (see Table 4.8). For Raw HTB, it can achieve high and stable throughput but with very high latency as our measurement results show earlier. Raw HTB is generic and can handle both ECN-capable and non ECN-capable flows. For HTB with DCTCP+ECN, low latency requirement can be satisfied but it can not achieve stable high throughput and can only handle ECN-capable flows. Therefore, this is a need for better software rate limiters.

Fortunately, software rate limiters are implemented on the end-host, which gives us opportunities to design and implement better software rate limiters for cloud networks. First, end-host has enough memory to store per-flow information. Second, we have sufficient programmability (e.g., loadable OVS module). For example, we can correlate an incoming ACK with its outgoing queue length; we can compute per-flow window

Algorithm 3 Pseudo-code of Direct ECE Marking Algorithm

```
1: for each incoming TCP ACK p do
2:   q ← rate_limiter_queue(p)
3:   if len(q) > K then
4:     tcp(p).ece ← 1
5:   end if
6: end for
```

size and encode it in packets before they arrive at VMs.

4.4 Design

4.4.1 Direct ECE Marking

In this subsection, we introduce a technique called Direct ECE Marking (DEM). DEM requires that VMs and containers are configured with DCTCP congestion control algorithm. In DEM, we monitor rate limiter queue occupancy and process each incoming TCP ACK. If the current rate limiter queue occupancy is above a threshold K, we directly set the ACK's TCP ECE (ECN Echo) bit to 1. To get the correct rate limiter queue occupancy for the TCP ACK, we need to inspect the TCP ACK and determine which queue the incoming TCP ACK's data packet belongs to. In other words, we need to determine the queue that this TCP ACK's reverse flow goes to. The pseudo-code of DEM is presented in Algorithm 3. DEM can be implemented in the virtual switch (e.g., OVS) in the hypervisor. OVS rate limiters directly call the Linux HTB implementation so it can get the rate limiter queue information. Also, OVS processes all the packets so it can inspect and modify all the incoming TCP ACKs.

The difference between DEM and existing ECN marking schemes is that it directly marks TCP ECE bit based on current queue occupancy instead of the queue occupancy one TCP RTT ago if using the existing ECN

marking schemes. Therefore, congestion control actions depend on real-time queueing information and control loop latency is reduced to almost 0. Control loop latency is the time it takes to forward the TCP ACK from the virtual switch to the VM or container. In this way, "in network" latency does not cause side-effects for end-host congestion control. Note that if we perform ECN marking on the outgoing path, then congestion control loop latency can be very large (e.g., RTT of the flows to remote clients is tens of ms). Besides reducing control loop latency, DEM also avoids coarse-grained segment-level ECN marking, which leads to inaccurate congestion level estimation, as we discussed before. Therefore, DEM makes rate limiter congestion control more timely and effective.

DEM only turns TCP ECE bit from 0 to 1, it never does the opposite. If congestion happens both in the rate limiter on the end-host and in the switch(es) on the network path, Then TCP ECE bit is always 1. If congestion only happens in the rate limiter, then DEM turns TCP ECE bit from 0 to 1. If congestion only happens in the network (i.e., in the switches), then TCP ECE is kept as 1. If neither network switches nor the rate limiter is congested, then TCP ECE is always 0. So DEM does not affect the correctness of end-to-end congestion control and is complementary with "in network" congestion control schemes.

4.4.2 SPRING

DEM has two limitations. First is that it relies on DCTCP transport in VMs and containers. For containers, cloud administrators are able to configure server's congestion control algorithm to DCTCP. So such an assumption is reasonable. However, for VMs, tenants have the flexibility to tune their congestion control settings. Therefore, assuming that every VM uses DCTCP as the congestion control algorithm is not realistic in practice. Second, DEM needs ECN support in the network. As mentioned before, ECN is not widely supported in WAN traffic [74]. To address

Algorithm 4 Pseudo-code of SPRING Algorithm

```
1:  for each packet p do
2:    q ← rate_limiter_queue(p)
3:    current_qlen ← len(q)
4:    new_gradient ← current_qlen – q.prev_qlen
5:    q.prev_qlen ← current_qlen
6:    q.gradient ← (1 – α)*q.gradient + α*new_gradient
7:    q.normalized_gradient ← q.gradient / K1
8:    if p is an incoming TCP ACK then
9:      f ← getReverseFlow(p)
10:     if current_qlen < K1 then
11:       f.rwnd ← f.rwnd + MSS
12:     else if current_qlen > K2 then
13:       f.ssthresh ← f.rwnd
14:       f.rwnd ← f.rwnd*(1 – β*(1 – K2 / current_qlen))
15:       f.rwnd ← max(f.rwnd, MSS)
16:     else if q.gradient ⩽ 0 then
17:       f.rwnd ← f.rwnd + MSS
18:     else
19:       f.rwnd ← f.rwnd*(1 – β*q.normalized_gradient)
20:       f.rwnd ← max(f.rwnd, MSS)
21:     end if
22:   end if
23: end for
```

the limitations and make our solution more generic, we present SPRING (shown in Algorithm 4).

SPRING modifies TCP ACK's receiver's advertised window size (also known as RWND) to enforce congestion control [34, 56]. It uses real-time rate limiter queue length as congestion control signal and a TIMELY-like [84] congestion control law. For each packet, we get its corresponding rate limiter queue length. If the packet is outgoing, we get the length of the queue that the packet is to be enqueued. If the packet is an incoming TCP ACK, we get the length of the queue that its reverse flow goes to (TCP is bidirectional). We maintain a gradient for the rate limiter queue

length using Exponentially Weighted Moving Average (EWMA) (line 2–6). We set two thresholds, K1 and K2 (K1 $<$ K2). The queue length gradient is normalized by dividing it using K1 (line 7). Note that gradient is a per-queue defined parameter. If the processed packet is an incoming TCP ACK, we first need to get its reverse flow (i.e., the TCP ACK's corresponding data packet flow). Then, we manage a running RWND for each flow based on a TIMELY-like congestion control law (line 10– line 20). There are 4 cases: if the current rate limiter queue length is smaller than K1, that means this is no congestion, so we increase the flow's RWND by one MSS (Maximum Segment Size). If the current rate limiter queue length is larger than K2, that means congestion happens in the rate limiter queue, so we multiplicatively decrease the RWND. If the current rate limiter queue length is between K1 and K2, we check the gradient of rate limiter queue occupancy. If the gradient is smaller than or equal to 0, that means the queue is being drained or its size is not increasing, we increase RWND by one MSS. Otherwise, we multiplicatively decrease the RWND based on the normalized gradient.

Note that TIMELY [84] is a rate-based congestion control algorithm while SPRING is a window-based. TIMELY uses accurate latency measurement provided by the NIC while SPRING performs congestion control based on real-time rate limiter queue length because SPRING checks the rate limiter queue length when receiving incoming TCP ACKs. Because congestion control decisions are enforced via modifying RWND field in TCP ACK headers, SPRING has the following good properties: 1) the solution does not relies on DCTCP transport in VMs and ECN support in the network, so it is generic and can support not only east-west traffic (i.e., intra-datacenter traffic) but also north-south traffic (i.e., inter-datacenter traffic and traffic between cloud and clients). 2) the solution avoids coarse-grained segment-level ECN marking and its control loop latency is almost 0, so congestion control is more effective compared with the strawman

solution—DCTCP in VMs/containers and ECN marking in rate limiter queues.

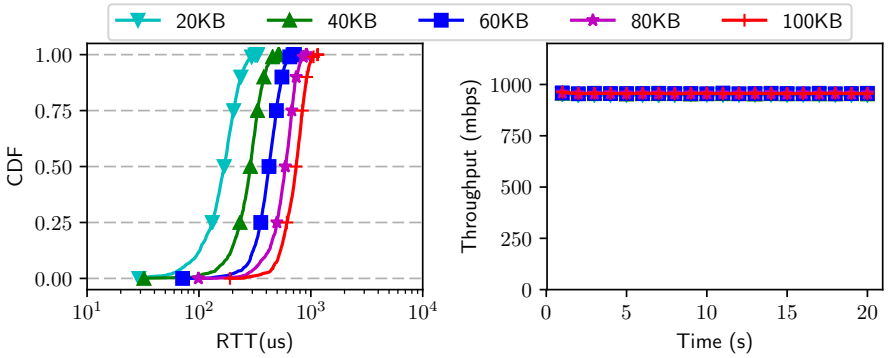
4.4.3 Remarks

Both DEM and SPRING avoid long and unpredictable congestion control loop latency and avoid throughput oscillation due to coarse-grained segment-level ECN marking. DEM relies on ECN support in the network and DCTCP transports configured in the end-points. Compared with DEM, SPRING is a more generic solution. DEM and SPRING share the same limitation, that is they do not support IPSec (because they need to modify TCP header). However, SSL/TLS is supported. Furthermore, SPRING needs to maintain per-flow information in the hypervisor. Maintaining per-flow information in switches is conventionally considered to be challenging. In SPRING we only need to maintain the information of the connections from the VMs/Containers running on the end-host. Also, recent advances like OVS ConnTrack [45] has made connection tracking on the end-host more effective.

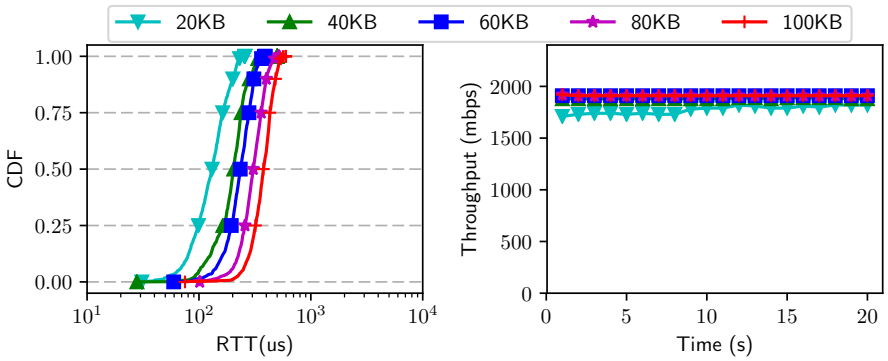
4.5 Evaluation

We evaluate the performance of DEM and SPRING in this section. We use CloudLab as our testbed and configure rate limiters using Linux HTB (Hierarchical Token Bucket). We modify OVS and HTB to implement DEM and SPRING. In the following, we will show the throughput, latency, fairness and CPU overhead of DEM and SPRING enabled rate limiters.

DEM performance We setup two servers. One acts as the sender and the other as the receiver. On the sender side, we configure rate limiter (HTB) to different speeds (500Mbps, 1Gbps, 2Gbps, 4Gbps, 6Gbps and 8Gbps). We also enable DCTCP on the two servers. DEM directly sets TCP ECE bit



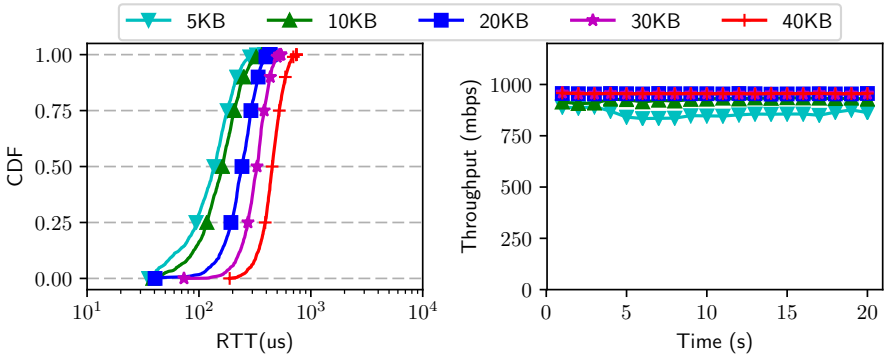
(a) Rate Limiting: 1Gbps



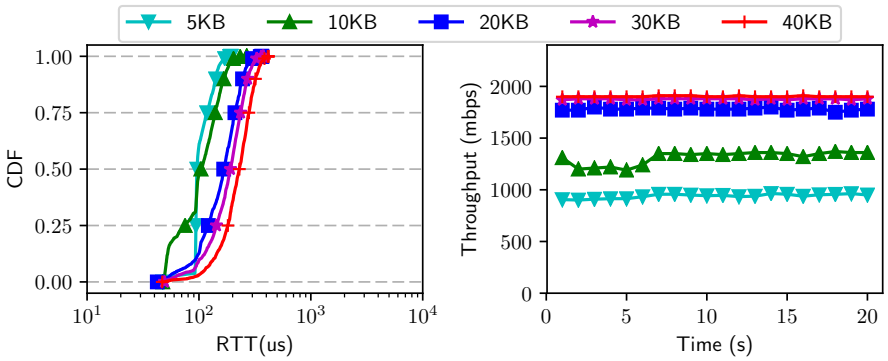
(b) Rate Limiting: 2Gbps

Figure 4.9: DEM experiments, 1flow, varying threshold

if real time rate limiter queue length is above a pre-configured threshold K. For each rate limiter speed, we vary threshold K. Then we measure the throughput using iperf and TCP RTT using sockperf. The results are shown in Figure 4.9. We can see that DEM can greatly reduce the latency caused by rate limiters (latency is decreased by around 10 times). Also it gives high and stable TCP throughput (throughput is the same as raw HTB). We also benchmark the performance of DEM with multiple iperf flows and the results show similar trends.



(a) Rate Limiting: 1Gbps, K2=K1+10KB



(b) Rate Limiting: 2Gbps, K2=K1+20KB

Figure 4.10: SPRING experiments, $\alpha = 0.5$, $\beta = 0.5$, 1 flow, varying threshold

SPRING performance We run SPRING-enabled rate limiters on the sender side. The rate limiter (HTB) is configured with different speeds (500Mbps, 1Gbps, 2Gbps, 4Gbps, 6Gbps and 8Gbps). Then we send an iperf flow (to measure throughput) and a sockperf flow (to measure TCP RTT). The experiment results for rate limiter speed of 1Gbps and 2Gbps are shown in Figure 4.10. Similar to DEM, when the algorithm parameters are appropriate, SPRING gives very stable and high throughput saturation while latency is close to the case where there is no congestion. We also

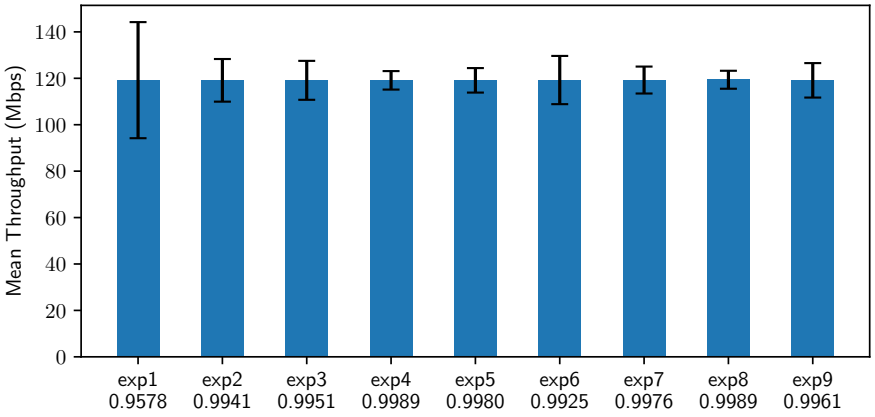


Figure 4.11: SPRING throughput fairness: 9 runs, 8 flows per run, rate-limiting=1Gbps, fairness index at the bottom

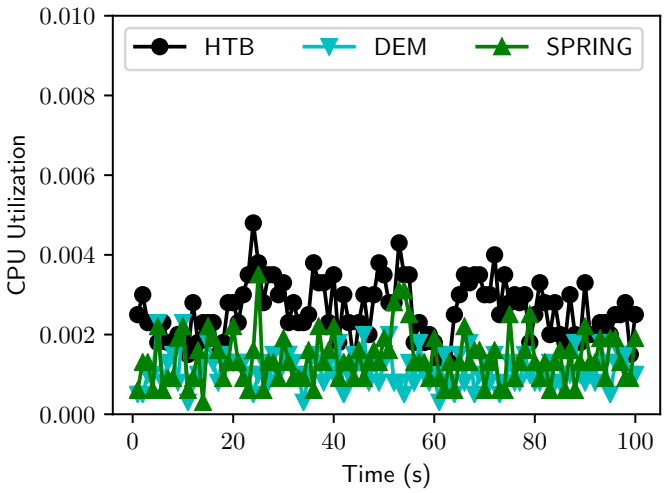


Figure 4.12: CPU overhead

increase the number of concurrent iperf flows and the results show similar trends.

Throughput fairness We run an experiment to check throughput fairness

of SPRING. We fix the rate limiter (HTB) speed to 1Gbps and send 8 concurrent iperf flows through the rate limiter. Figure 4.11 shows the results. We perform the experiment for 9 runs and each run lasts for 20 seconds. We can see that in all runs, TCP throughput fairness index is above 0.95.

CPU overhead The operations of DEM and SPRING are pretty light-weight. So their CPU overhead should be very small. We conduct an experiment to validate this—we run raw HTB, DEM-enabled HTB and SPRING-enabled HTB and fix the rate limiter speed to 2Gbps. We send TCP traffic to saturate the rate limiterand and measure the CPU usage of sender server (the sender server has 40 cores). Figure 4.12 shows the experiment results. Indeed, both DEM and SPRING incurs very little CPU overhead and surprisingly, it is slightly smaller than raw HTB's. We think it might be caused by the fact that DEM and SPRING reduce throughput slightly.

4.6 Summary

A lot of recent work has been focusing on solving in network latency in datacenter networks. In this chapter, we focus on a less explored topic — latency increase caused by rate limiters on the end-host. We show that latency can be increased by an order of magnitude by the rate limiters in cloud networks, and simply extending ECN into rate limiters is not sufficient. To this end, we propose two techniques — DEM and SPRING to improve the performance of rate limiters. Our experiment results demonstrate that DEM and SPRING enabled rate limiters can achieve high (and stable) throughput and low latency.

5 Related Work

Load Balancing in Datacenters MPTCP [105, 128] is a transport protocol that uses subflows to transmit over multiple paths. CONGA [8] and Juniper VCF [54] both employ congestion-aware flowlet switching [113] on specialized switch chipsets to load balance the network. RPS [38] and DRB [27] evaluate per-packet load balancing on symmetric 1 Gbps networks at the switch and end-host, respectively. The CPU load and feasibility of end-host-based per-packet load balancing for 10+ Gbps networks remains open. Hedera [7], MicroTE [20] and Planck [106] use centralized traffic engineering to reroute traffic based on network conditions. FlowBender [67] reroutes flows when congestion is detected by end-hosts and Fastpass [99] employs a centralized arbiter to schedule path selection for each packet. As compared to these schemes, Presto is the only one that proactively load-balances at line rate for fast networks in a near uniform fashion without requiring additional infrastructure or changes to network hardware or transport layers. Furthermore, to the best of our knowledge, Presto is the first work to explore the interactions of fine-grained load balancing with built-in segment offload capabilities used in fast networks.

Reducing Tail Latency DeTail [133] is a cross-layer network stack designed to reduce the tail of flow completion times. DCTCP [9] is a transport protocol that uses the portion of marked packets by ECN to adaptively adjust sender's TCP's congestion window to reduce switch buffer occupancy. HULL [10] uses Phantom Queues and congestion notifications to cap link utilization and prevent congestion. In contrast, Presto is a load

balancing system that naturally improves the tail latencies of mice flows by uniformly spreading traffic in fine-grained units. QJUMP [48] utilizes priority levels to allow latency-sensitive flows to "jump-the-queue" over low priority flows. PIAS [15] uses priority queues to mimic the Shortest Job First principle to reduce FCTs. Last, a blog post by Casado and Pettit [29] summarized four potential ways to deal with elephants and mice, with one advocating to turn elephants into mice at the edge. We share the same motivation and high-level idea and design a complete system that addresses many practical challenges of using such an approach.

Handling Packet Reordering TCP performs poorly in the face of reordering, and thus several studies design a more robust alternative [22, 23, 134]. Presto takes the position that reordering should be handled below TCP in the existing receive offload logic. In the lower portion of the networking stack, SRPIC [131] sorts reordered packets in the driver after each interrupt coalescing event. While this approach can help mitigate the impact of reordering, it does not sort packets across interrupts, have a direct impact on segment sizes, or distinguish between loss and reordering.

Congestion control for DCNs DCTCP [9] is a seminal TCP variant for datacenter networks. Judd [66] proposed simple yet practical fixes to enable DCTCP in production networks. TCP-Bolt [117] is a variant of DCTCP for PFC-enabled lossless Ethernet. DCQCN [136] is a rate-based congestion control scheme (built on DCTCP and QCN) to support RDMA deployments in PFC-enabled lossless networks. TIMELY [84] and DX [76] use accurate network latency as the signal to perform congestion control. TCP ex Machina [127] uses computer-generated congestion control rules. PERC [65] proposes proactive congestion control to improve convergence. ICTCP's [129] receiver monitors incoming TCP flows and modifies RWND to mitigate the impact of incast, but this cannot provide generalized congestion control like AC/DC. Finally, efforts [24, 119] to implement TCP Offload Engine (TOE) in specialized NICs are not widely deployed for

reasons noted in [85, 121]. vCC [34] is a concurrently designed system that shares AC/DC's goals and some of its design details. The paper is complementary in that some items not addressed in AC/DC are presented, such as a more detailed analysis of the ECN-coexistence problem, an exploration of the design space, and a theoretical proof of virtualized congestion control's correctness. AC/DC provides an in-depth design and thorough evaluation of a DCTCP-based virtualized congestion control algorithm on a 10 Gbps testbed.

Bandwidth allocation Many bandwidth allocation schemes have been proposed. Gatekeeper [109] and EyeQ [63] abstract the network as a single switch and provide bandwidth guarantees by managing each server's access link. Oktopus [17] provides fixed performance guarantees within virtual clusters. SecondNet [50] enables virtual datacenters with static bandwidth guarantees. Proteus [132] allocates bandwidth for applications with dynamic demands. Seawall [111] provides bandwidth proportional to a defined weight by forcing traffic through congestion-based edge-to-edge tunnels. NetShare [75] utilizes hierarchical weighted max-min fair sharing to tune relative bandwidth allocation for services. FairCloud [103] identifies trade-offs in minimum guarantees, proportionality and high utilization, and designs schemes over this space. Silo [62] provides guaranteed bandwidth, delay and burst allowances through a novel VM placement and admission algorithm, coupled with a fine-grained packet pacer. AC/DC is largely complimentary to these schemes because it is a transport-level solution.

Rate limiters SENIC [93] identifies the limitations of NIC hardware rate limiters (*i.e.*, not scalable) and software rate limiters (*i.e.*, high CPU overhead) and uses the CPU to enqueue packets in host memory and the NIC. Silo's pacer injects void packets into an original packet sequence to achieve pacing. FasTrack [93] offloads functionality from the server into the switch for certain flows.

6 Conclusion and Future Work

6.1 Conclusion

In this dissertation, we present three research projects — 1) an edge-based traffic load balancing system (i.e., Presto) for datacenter networks, 2) virtualized congestion control technique for multi-tenant clouds (AC/DC TCP) and 3) low latency software rate limiters for cloud networks. All of them leverage the flexibility and high programmability of software-defined network edge (i.e., end-host networking) to improve the performance of datacenter networks. Each of these three projects focuses on one crucial functionality in datacenter networks. In the following, we provide a short conclusion for each project in turn.

Presto: Edge-based Load Balancing for Fast Datacenter Networks. Modern datacenter networks are built with multi-stage Clos networks. There are usually tens to hundreds of network paths between two servers in the same datacenter. The state-of-the-art traffic load balancing uses flow-level schemes (e.g., ECMP and WCMP). However, flow-level load balancing schemes suffer from the elephant flow collision problem. Elephant flow collisions lead to reduced throughput and increased latency. We propose Presto to address this classic problem. Presto is an end-host-based traffic load balancing system. At the sender side, Presto uses virtual switch (OVS) to chunk elephant flows into flowcells (a flowcell consists of multiple consecutive TCP segments and its maximum size is 64KB) and spread the flowcells evenly over multiple network paths. At the receiver

side, we design and implement improved Generic Receive Offload (GRO) functionality in Linux networking subsystem to mask packet reordering for TCP. Presto makes sure mice flows (smaller than or equal to 64KB in size) are not exposed to packet ordering issue. Note that in realistic datacenter networks, the overwhelming majority of the flows are mice flows. For elephant flows, Presto's improved GRO logic puts flowcells in order before they are pushed up to TCP. Presto eliminates the elephant flow collision problem and demonstrates that subflow-level traffic load balancing is possible and effective.

AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In multi-tenant clouds, tenants manage their own Virtual Machines (VMs). VM TCP stacks' congestion control algorithms can be outdated, inefficient or even misconfigured. Those outdated, inefficient or even misconfigured VM TCP stacks can cause severe network congestion and throughput fairness issue. Network congestion and throughput unfairness affect the performance of the applications running in clouds (e.g., increased task completion time). To address this problem, we present AC/DC TCP, a virtual congestion control enforcement technique for datacenter networks. The key idea of AC/DC TCP is to implement an intended congestion control algorithm (i.e., DCTCP) in the virtual switch in the hypervisor and the congestion control decisions are enforced via modifying TCP header's RWND field. Our experiment results show that enforcing an intended congestion control algorithm in the network virtualization layer can greatly reduce network latency and improve throughput fairness. Also, AC/DC TCP's CPU and memory overhead is negligible.

Low Latency Software Rate Limiters for Cloud Networks. Rate limiters are employed to provide bandwidth allocation feature in multi-tenant clouds. For example, in Google Cloud Platform, different kinds of VMs are allocated with different maximum bandwidths. However, we find that

rate limiters can increase network latency by an order of magnitude or even higher. That is because traffic shaping (the underlying mechanism of rate limiters) maintains a queue to absorb bursty traffic and dequeues packets into the network based on preconfigured rate. Queueing latency in rate limiter queue inflates end-to-end network latency. To solve this problem, we first extend ECN into rate limiter queues and apply DCTCP on the end-host. Though this straightforward scheme reduces network latency significantly, it can also lead to TCP throughput oscillation because of coarse-grained segment-level ECN marking and long congestion control loop latency. To address the shortcomings of the straightforward scheme, we present DEM, which directly sets TCP ECE bit in reverse ACKs and SPRING which runs a queue-length-based congestion control algorithm and enforces congestion control decisions via modifying RWND field in reverse ACKs. Both DEM and SPRING enable low latency, high network saturation rate limiters. DEM relies on ECN support while SPRING is generic and can handle both ECN flows and non-ECN flows.

The techniques and mechanisms proposed in this dissertation are original. We believe the research work presented in this dissertation will be valuable to the computer networking research community.

6.2 Lessons Learned

Darkness Is Before the Dawn. The first important thing I learned during my Ph.D. study is that sometimes we have to go through the darkness before the dawn. The Presto project is such an example. Initially, we only implemented the Presto sender side logic, but we found that even in a simple topology, network throughput could not reach 9.3Gbps. We later found that it was caused by packet reordering. Before starting the project, we thought modern TCP stacks should be smart enough to tolerate a certain amount of packet reordering. However, we found it was not the

case for high speed networks. I did extensive experiments to figure out how to improve the performance. After exploring in the darkness for around 2 months, we finally designed the first version of the improved GRO logic to mark packet reordering for TCP. The story tells me sometimes doing research is just like walking in the dark, but we should not lose hope. We will be fine if we keep exploring in the right direction.

Simple Things Can Work Well. I also learned that simple things can work pretty well in practice, maybe this is especially true in system and networking research. For example, the DCTCP congestion control law is extremely simple but this work has a huge impact in datacenter networking research. I think all of techniques or mechanisms introduced in this thesis are simple, especially, AC/DC TCP, DEM and Spring. We find simple things can boost performance significantly. I like simple and elegant solutions a lot.

Start Simple and Build Software Step by Step. We need to write code to validate our ideas. A key thing I learned is that we should start small and build our prototypes step by step. Once we finish a small step, we can check whether the code we wrote works or not. In this way, we can have a deep understanding of the code we wrote. Also we can find bugs or performance issues at the early stage. All the projects presented in this thesis involve some kernel level programming. The practice works pretty well for me.

Old Ideas or Techniques Can Reborn in New Use Cases. TCP was born in the late 1980s. But we observe TCP can not meet our performance goals in datacenters networks. So new research opportunities come. ECN-based or latency-based congestion control are not really new neither. But because of datacenter networks, they come to play important roles now. I have observed this phenomenon a lot of times in the last 5 years in networking research. I think applying old ideas or techniques is not a problem. The key thing is how we apply them, how we can improve upon

them and how we can leverage the innovations in hardware or software to make the old ideas become more effective in the new settings.

6.3 Future Work

In the following, we list potential research topics in datacenter networking we may explore in the future.

Automatic Datacenter Network Topology Analysis. Datacenter network topology determines how scalable the network is, how resilient it is to link or switch failures and how easily the network can be incrementally deployed. Today's practice is that network architects need to manually infer (usually based on experiences) many key characteristics related to the candidate network topologies. So we lack a scientific and formal method to evaluate different network topologies. Therefore, there is a need to build a network topology analysis framework to help network architect analyze and compare candidate network topologies. To compare different network topologies, we need to set up metrics to quantify different network topologies. Our first goal is to identify a set of metrics (e.g., cost, wiring complexity, bandwidth, reliability, routing convergence) that can accurately quantify datacenter network topologies. Also, we need to define a set of workloads and traffic patterns to run against the network. Next, we want to investigate whether we can design and implement an automatic topology analysis framework to gain more insights and help design better network topologies.

Applying Machine Learning Techniques to Traffic Load Balancing. Datacenter networks are shared by a large amount of applications hosted in clouds. Based on the fact that traffic is multiplexed and the conjecture that the majority of traffic should be used by some top applications, predictable traffic patterns may exist in the datacenter networks. First, we want to measure traffic load on each link in a real datacenter network for a long

time (e.g., a few months). Based on the measurement data, we analyze whether traffic loads can be predictable or not. If so, we can utilize big-data systems and machine learning techniques to help us apply better traffic load balancing schemes to reduce the possibility of network congestion and improve the performance of applications. We believe such a machine-learning-aided load balancing system can be applied to different scenarios. For example, it can be applied in both intra-datacenter networks and wide-area inter-datacenter networks.

Near Real Time Network Congestion Monitoring. Network congestion is a crucial performance hurdle for high performance cloud computing services like search, query and remote procedure calls. Studies have shown that end-to-end network latency (TCP RTT) can be increased by tens of milliseconds due to network congestion. Such a huge network latency can affect customer's experience and can have significant negative impacts on revenue. Therefore, one research question is whether we can monitor network congestion in (near) real time manner. If the answer is yes, how we should provide such information to network administrators or developers in the cloud? How we can quickly reroute network traffic to bypass congested network paths? This research problem is challenging because network congestion information should be obtained in (near) real time manner such that application traffic can be rerouted to avoid buffer building up. In today's data center networks, the base line end-to-end latency is around 40 to 200 microseconds, so we need to reduce the "monitoring and action" control loop latency as small as possible. To achieve this goal, we may need to explore recent advances in fast software packet processing (e.g., DPDK [36]) and modern hardware features in the switches.

Bibliography

[1] CloudLab. https://www.cloudlab.us/.

[2] HTB Linux queuing discipline manual - user guide. http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm.

[3] Offloading the Segmentation of Large TCP Packets. https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/offloading-the-segmentation-of-large-tcp-packets.

[4] OpenvSwitch QoS. https://github.com/openvswitch/ovs/blob/master/FAQ.md#qos.

[5] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter. Shadow MACs: Scalable Label-switching for Commodity Ethernet. In *HotSDN*, 2014.

[6] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[7] Mohammad Al-fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[8] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

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

[10] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *NSDI*, 2012.

[11] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, 2009. http://www.rfc-editor.org/rfc/rfc5681.txt.

[12] Amazon Web Services. https://aws.amazon.com/.

[13] Gianni Antichi, Christian Callegari, and Stefano Giordano. Implementation of TCP Large Receive Offload on Open Hardware Platform. In *Proceedings of the First Edition Workshop on High Performance and Programmable Networking*, HPPN '13, 2013.

[14] Arista 7250QX Series 10/40G Data Center Switches. http://www.arista.com/assets/data/pdf/Datasheets/7250QX-64_Datasheet.pdf.

[15] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, 2015.

[16] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. Enabling End Host Network Functions. In *SIGCOMM*, 2015.

[17] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[18] Stephen Bensley, Lars Eggert, Dave Thaler, Praveen Balasubramanian, and Glenn Judd. Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters. Internet-Draft draft-ietf-tcpm-dctcp-01, 2015. http://www.ietf.org/internet-drafts/draft-ietf-tcpm-dctcp-01.txt.

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

[20] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[21] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2006.

[22] Ethan Blanton and Mark Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 2002.

[23] Stephan Bohacek, Joao P. Hespanha, Junsoo Lee, Chansook Lim, and Katia Obraczka. TCP-PR: TCP for Persistent Packet Reordering. In *ICDCS*, pages 222–231, 2003.

[24] Boosting Data Transfer with TCP Offload Engine Technology. http://www.dell.com/downloads/global/power/ps3q06-20060132-Broadcom.pdf.

[25] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

[26] Broadcom Extends Leadership with New StrataXGS Trident II Switch Series Optimized for Cloud-Scale Data Center Networks. http://www.broadcom.com/press/release.php?id=s702418, 2012.

[27] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *CoNEXT*, 2013.

[28] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, 2012.

[29] Martin Casado and Justin Pettit. Of Mice and Elephants. http://networkheresy.com/2013/11/01/of-mice-and-elephants/, November 2013.

[30] Wu Chou and Min Luo. Agile Network, Agile World: the SDN Approach. http://huaweiempresas.com/img_eventos/140314/SDN_Workshop_2.pdf.

[31] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. RFC 6928, 2013. http://www.rfc-editor.org/rfc/rfc6928.txt.

[32] Cisco Global Cloud Index: Forecast and Methodology, 2015-2020. http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf.

[33] Cloud Computing Market. http://www.cloudcomputingmarket.com/.

[34] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *SIGCOMM*, 2016.

[35] Sujal Das. Smart-Table Technology—Enabling Very Large Server, Storage Nodes and Virtual Machines to Scale Using Flexible Network Infrastructure Topologies. http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP101-R.pdf, July 2012.

[36] Data Plane Development Kit (DPDK). http://dpdk.org/.

[37] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[38] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.

[39] Nandita Dukkipati and Nick McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[40] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

[41] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, 2003. http://www.rfc-editor.org/rfc/rfc3649.txt.

[42] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, 2004. https://tools.ietf.org/html/rfc3782.

[43] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. Tcp extensions for multipath operation with multiple addresses. RFC 6824, RFC Editor, January 2013. http://www.rfc-editor.org/rfc/rfc6824.txt.

[44] Google Compute Engine. https://cloud.google.com/compute/.

[45] Thomas Graf and Justin Pettit. Connection Tracking & Stateful NAT. In *Open vSwitch 2014 Fall Conference*, 2014.

[46] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[47] Leonid Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Linux Symposium*, 2005.

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

[49] Dinakar Guniguntala, Paul E McKenney, Josh Triplett, and Jonathan Walpole. The Read-Copy-Update Mechanism for Supporting Real-time Applications on Shared-memory Multiprocessor Systems with Linux. *IBM Systems Journal*, 2008.

[50] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[51] H. Chen and W. Song. Load Balancing without Packet Reordering in NVO3. Internet-Draft. https://tools.ietf.org/html/draft-chen-nvo3-load-banlancing-00, October 2014.

[52] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 2008.

[53] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 2008.

[54] Douglas Richard Hanks. *Juniper QFX5100 Series: A Comprehensive Guide to Building Next-Generation Networks*. O'Reilly Media, Inc., 2014.

[55] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure. In *IMC*, 2013.

[56] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Jason Gu, Wes Felter, John Carter, and Aditya Akella. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 244–257. ACM, 2016.

[57] IBM Bluemix. http://www.ibm.com/cloud-computing/bluemix/.

[58] IBM SoftLayer. http://www.softlayer.com/.

[59] Van Jacobson, Bob Braden, and Dave Borman. TCP Extensions for High Performance. RFC 1323, 1992. http://www.rfc-editor.org/rfc/rfc1323.txt.

[60] Raj Jain, Dah-Ming Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR*, 1998.

[61] Rajendra K Jain, Dah-Ming W Chiù, and William R Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. Technical Report DEC-TR-301, Digital Equipment Corporation, 1984.

[62] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*, 2015.

[63] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[64] JLS2009: Generic Receive Offload. https://lwn.net/Articles/358910/.

[65] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

[66] Glenn Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *NSDI*, 2015.

[67] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *CoNEXT*, 2014.

[68] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Explicit Window Adaptation: A Method to Enhance TCP Performance. *IEEE/ACM Transactions on Networking*, 2002.

[69] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *IMC*, 2009.

[70] Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT*, 2013.

[71] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. Bullet trains: a study of nic burst behavior at microsecond timescales. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 133–138. ACM, 2013.

[72] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. MPTCP is Not Pareto-optimal: Performance Issues and a Possible Solution. In *CoNEXT*, 2012.

[73] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, et al. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[74] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. On the state of ecn and tcp options on the internet. In *International Conference on Passive and Active Network Measurement*, pages 135–144. Springer, 2013.

[75] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *ACM SIGCOMM Computer Communication Review*, 2012.

[76] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate Latency-based Congestion Feedback for Datacenters. In *USENIX Annual Technical Conference*, 2015.

[77] Ka-Cheong Leung, Victor OK Li, and Daiqin Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *Parallel and Distributed Systems, IEEE Transactions on*, 2007.

[78] Linux Kernel. https://www.kernel.org.

[79] LiquidIO Server Adapters . http://www.cavium.com/LiquidIO_Server_Adapters.html.

[80] Shao Liu, Tamer Başar, and Ravi Srikant. TCP-Illinois: A Loss- and Delay-based Congestion Control Algorithm for High-speed Networks. *Performance Evaluation*, 2008.

[81] Andrew Moore Malcolm Scott and Jon Crowcroft. Addressing the scalability of Ethernet with MOOSE. In *DC-CAVES*, 2009.

[82] Aravind Menon and Willy Zwaenepoel. Optimizing TCP Receive Performance. In *USENIX Annual Technical Conference*, 2008.

[83] Microsoft Azure. https://azure.microsoft.com/en-us/?b=16.33.

[84] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[85] Jeffrey C Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS*, 2003.

[86] Jeffrey C Mogul and Ramana Rao Kompella. Inferring the Network Latency Requirements of Cloud Tenants. In *HotOS*, 2015.

[87] Jeffrey C Mogul and KK Ramakrishnan. Eliminating Receive Livelock in An Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, 1997.

[88] Timothy Prickett Morgan. A Rare Peek Into The Massive Scale of AWS. http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/, November 2014.

[89] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable Rule Management for Data Centers. In *NSDI*, 2013.

[90] MPTCP Linux Kernel Implementation. http://www.multipath-tcp.org.

[91] Jayaram Mudigonda, Praveen Yalagandula, Jeffrey C. Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. In *SIGCOMM*, 2011.

[92] Netronome FlowNIC. https://netronome.com/product/flownics/.

[93] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. FasTrak: Enabling Express Lanes in Multi-tenant Data Centers. In *CoNEXT*, 2013.

[94] Nuttcp. http://www.nuttcp.net/Welcome%20Page.html.

[95] Open vSwitch. http://openvswitch.org.

[96] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. On the Benefits of Applying Experimental Design to Improve Multipath TCP. In *CoNEXT*, 2013.

[97] Vern Paxson. End-to-end Internet Packet Dynamics. In *ACM SIGCOMM Computer Communication Review*, 1997.

[98] Ivan Pepelnjak. NEC+IBM: Enterprise OpenFlow You can Actually Touch. http://blog.ipspace.net/2012/02/necibm-enterprise-openflow-you-can.html.

[99] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[100] Justin Pettit. Open vSwitch and the Intelligent Edge. In *OpenStack Summit*, 2014.

[101] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending Networking into the Virtualization Layer. In *HotNets*, 2009.

[102] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[103] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[104] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-host Rate Limiting. In *NSDI*, 2014.

[105] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[106] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *SIGCOMM*, 2014.

[107] Receive Segment Coalescing (RSC). http://technet.microsoft.com/en-us/library/hh997024.aspx.

[108] Receive Side Scaling (RSS). https://technet.microsoft.com/en-us/library/hh997036.aspx.

[109] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV*, 2011.

[110] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP Buffer Tuning. *ACM SIGCOMM Computer Communication Review*, 1998.

[111] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[112] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[113] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[114] SKB in Linux Networking. http://vger.kernel.org/~davem/skb.html.

[115] Sockperf: A Network Benchmarking Utility over Socket API. https://code.google.com/p/sockperf/.

[116] N. T. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. Bershad. Receiver Based Management of Low Bandwidth Access Links. In *INFOCOM*, 2000.

[117] Brent Stephens, Alan L Cox, Anubhav Singla, Jenny Carter, Colin Dixon, and Wes Felter. Practical DCB for Improved Data Center Networks. In *INFOCOM*, 2014.

[118] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[119] TCP Offload Engine (TOE). http://www.chelsio.com/nic/tcp-offload-engine/.

[120] TCP Probe. http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe.

[121] TOE. http://www.linuxfoundation.org/collaborate/workgroups/networking/toe.

[122] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM*, 2009.

[123] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. *SIGOPS Oper. Syst. Rev.*, 2002.

[124] VXLAN Performance Evaluation on VMware vSphere 5.1. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-VXLAN-Perf.pdf.

[125] Guohui Wang and TS Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.

[126] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[127] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated Congestion Control. In *SIGCOMM*, 2013.

[128] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.

[129] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNEXT*, 2010.

[130] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ECN for Data Center Networks. In *CoNEXT*, 2012.

[131] Wenji Wu, Phil Demar, and Matt Crawford. Sorting Reordered Packets with Interrupt Coalescing. *Computer Networks*, 2009.

[132] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.

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

[134] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: A Reordering-robust TCP with DSACK. In *ICNP*, 2003.

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

[136] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.