Virtual Switch-based Congestion Control Enforcement for Datacenter Networks

Keqiang He[†] Eric Rozner[‡] Kanak Agarwal[*] Yu (Jason) Gu[*]
Wes Felter[‡] John Carter[*] Aditya Akella[†]

[†]University of Wisconsin–Madison [‡]IBM Research [*]IBM

Saturday 4th June, 2016, 21:54

ABSTRACT

Multi-tenant datacenters are successful because tenants can seamlessly port their workloads, applications and services to the cloud. Virtual Machine (VM) technology plays an integral role in this success by enabling a diverse set of operating systems and software to be run on a unified underlying framework. This flexibility, however, comes at the cost of dealing with out-dated, inefficient, or misconfigured TCP stacks implemented in the VMs. This paper investigates 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 vSwitch. Our scheme is light-weight, flexible, scalable and can police non-conforming flows. Our evaluation shows the computational overhead of AC⚡DC TCP is less than 4% and 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.

CCS Concepts

•Networks → Network architectures; End nodes; Programmable networks; Data center networks; *Data path algorithms; Network experimentation;*

Keywords

Load Balancing; Software-Defined Networking;

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 can yield 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 TCP/IP stack running on the VM. 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 having control over the VM TCP/IP stack, datacenter networks remain 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 its behavior is not optimized. For example, network congestion caused by non-optimzed stacks can lead to loss, increased latency and reduced throughput.

Thankfully, recent advances in optimizing TCP stacks for datacenter environments have shown that both high throughput and low latency can be achieved through novel TCP congestion control algorithms. Works such as DCTCP [3] and TIMELY [41] show great promise in providing high bandwidth and low latency by ensuring that network queues in switches do not fill up. And while these stacks are deployed in many of today's private datacenters [35, 58], ensuring that a vast majority of VMs within a public datacenter will update their TCP stacks to this new technology is a daunting, if not impossible task.

SIGCOMM'15, August 17–21, 2015, London, United Kingdom

DOI: http://dx.doi.org/10.1145/2785956.2787507

In this paper, we explore how operators can regain control of TCP's 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 [51]. 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. First, in order to ensure adoption rates are high, the approach must work without making changes to VMs. Hypervisor-based approaches that do not modify VMs 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 [53]. Therefore, limiting a VM's TCP flows in a fine-grained, dynamic nature at scale (10,000's of flows per server [44]) 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.

In this paper, we present Alternative Congestion for 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 receiver advertised window (`RWND`) on incoming ACKs. A policing mechanism ensures stacks cannot benefit from ignoring `RWND` and can also be used for non-TCP traffic.

Our scheme provides the following benefits. First, AC⚡DC allows for 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, regardless of the congestion control algorithms VMs use. 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 [35, 71]. 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 paper are as follows:

1. The design of a vSwitch-based congestion control mechanism that regains control over the TCP/IP stack implemented in the VM without requiring any changes to the VM or network hardware.

2. Prototype implementation to show our scheme is effective, scalable, simple to implement and has only 4% computational overhead over a baseline scheme.

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 paper is as follows. Background and motivation are discussed in §2. AC⚡DC's design is outlined in §3 and implementation in §4. Results are presented in §5. Related work is surveyed in §6 before concluding.

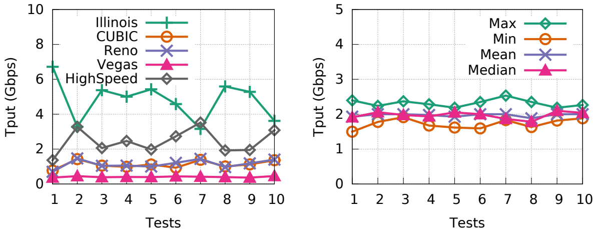
2. BACKGROUND AND MOTIVATION

In this section, we first give a brief background of congestion control in the datacenter. Then we motivate AC⚡DC by listing the benefits of moving congestion control into the vSwitch. Finally, we discuss how AC⚡DC differs from a class of related bandwidth allocation schemes.

2.1 Datacenter Transport

Today's datacenters host many applications such as search, advertising, analytics and retail that require high bandwidth and low latency. Network congestion, caused by imperfect load balancing [1], network upgrades, failures or oversubscription, can adversely impact these services. Unfortunately, congestion is not rare in datacenters. For example, Google reported congestion-based drops were observed in their fabric when the network utilization approached 25% [58]. Other studies have shown high variance and substantial increase in the 99.9th percentile latency for round-trip times in today's datacenters [43, 68]. Large tail latencies impact customer experience, result in revenue loss [3, 16], and degrade application performance [24, 32]. Therefore, significant motivation exists to reduce the impact of congestion in datacenter fabrics.

TCP, specifically its 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 [3, 34, 41, 60, 70]. Specifically, DCTCP [3] achieves high throughput and low latency by adjusting a TCP sender's rate based on the fraction of packets that experience congestion. In DCTCP, 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 flow throughput, and increase fairness and stability over traditional schemes [3, 35].



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

Figure 1: Different congestion controls lead to unfairness.

2.2 Benefits of vSwitch Congestion Control

Rather than proposing a new datacenter congestion control algorithm, this work investigates if congestion control can be moved to the vSwitch. Implementing congestion control in the vSwitch provides an element of control without cooperation from VMs. This is an important criteria in untrusted public cloud environments or simply in cases where tenants cannot update their TCP stack (such as the inability to update an OS or a dependence on a library) [35]. Allowing administrators to enforce an optimized congestion control without changing the VM is the first major benefit of our scheme.

The next major benefit is AC⚡DC allows for a *uniform* congestion control algorithm 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 exceed queue thresholds [35, 71]. Ideally, tenants shouldn't suffer based on such a simple configuration issue. Additionally, stacks with different congestion control algorithms may not share the same fabric fairly. For example, Figure 1 shows the performance of five different TCP flows on the topology in Figure 7a. Each flow selects a congestion control algorithm available in Linux: TCP CUBIC [27], TCP Illinois [39], TCP HighSpeed [20], TCP New Reno [21] and TCP Vegas [13]. Figure 1a shows aggressive stacks such as TCP Illinois and TCP HighSpeed achieve higher bandwidth and fairness is worse than all flows using the same stack (Figure 1b). A tenant should not be able to unfairly obtain higher bandwidth by simply changing its congestion control.

Another benefit of moving congestion control to the vSwitch is it allows for different congestion control algorithms to be assigned on a per-flow basis. Today, TCP congestion control is configured at an OS-level, so all of an OS's flows are forced to use the same congestion control algorithm. A vSwitch-based approach can assign WAN flows to a congestion control algorithm that optimizes WAN performance [19, 61] 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 §3.4, 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 functionality is easy 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 easy to implement. Indeed, studies have shown that most TCP overhead comes from buffer management [40]. TCP 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 relatively easy to move its functionality to another layer. Because congestion control is light-weight, the penalty imposed for running it is low: our benchmarks show the computational overhead of AC⚡DC is less than 4%.

2.3 Tenant-Level Bandwidth Allocation

Transport layer schemes do not provide fair bandwidth allocation among tenants because a tenant with more concurrent flows ends up getting a higher share of network bandwidth. In order to provide a degree of performance isolation in the network, datacenter operators can implement a variety of bandwidth allocation schemes. Simple static rate limiters enforced on many default public cloud images dictate an upper-bound on the bandwidth available to different classes of VMs. More complex performance isolation schemes investigate how to allocate network bandwidth in datacenter networks by either guaranteeing or proportionally allocating bandwidth for tenants [10, 26, 32, 33, 36, 52, 55, 57, 72]. Some of these schemes share high-level architectural similarities to AC⚡DC. For example, EyeQ [33] provides a single dedicated switch abstraction for tenant VMs and 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 [57] provides proportional bandwidth allocation to a VM or application by forcing all traffic through a congestion controlled tunnel that is configured through weights and sender and receiver 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 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 [33] and Gatekeeper [55] 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 [1] cause congestion in the core. Communication between a pair of VMs may consist of multiple flows, each of which may traverse a distinct path due to ECMP. 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

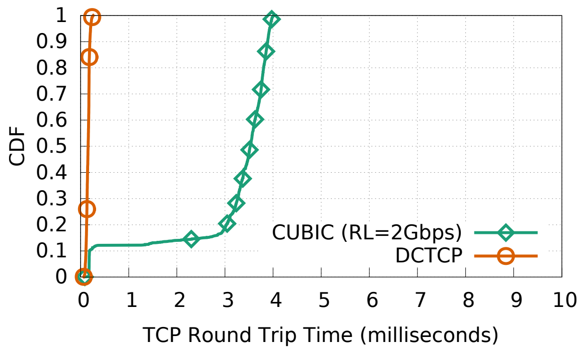


Figure 2: CDF of RTTs showing CUBIC fills buffers.

congestion on their paths. Furthermore, a scheme like Seawall [57] 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 [53] 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.

2.4 Bandwidth Allocation with Transport Control

In this work, we argue administrators should not relinquish congestion control to tenant VMs and solely rely on bandwidth arbitration to achieve low latency and high utilization in the network. It is important for a cloud provider to control *both* congestion control and bandwidth allocation. Bandwidth allocation schemes are insufficient because certain TCP stacks aggressively fill switch buffers. Consider a simple example where five flows send simultaneously on the 10 Gbps topology in Figure 7a. Even when the bandwidth is allocated "perfectly" at 2 Gbps per flow, CUBIC, the current default congestion control algorithm in Linux, saturates the output port's buffer and leads to inflated round-trip times (RTTs) for traffic sharing the same link [1]. Figure 2 shows these RTTs for CUBIC and also DCTCP, which is able to keep queueing latencies, and thus RTTs, low even though we did not enforce *any* rate limiting on it. Therefore, it is important for cloud providers to be able to exercise a desired transport-level congestion control policy.

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 policy (*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. DESIGN

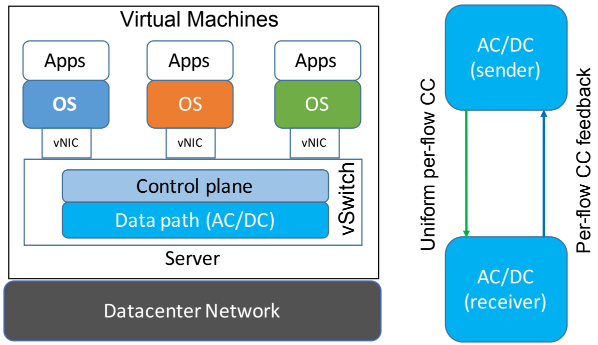


Figure 3: AC⚡DC high-level architecture.

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.

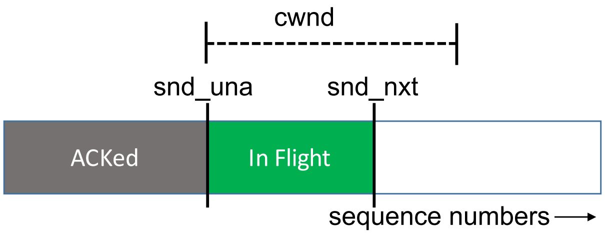


Figure 4: Variables for TCP sequence number space.

3.1 Obtaining Congestion Control State

Figure 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 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 we update `snd_una` 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` ≤ `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 [14]. 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.

3.2 Implementing DCTCP

[1] Note the servers are not over-subscribed in this scenario, so even bounding rate limiters to 10 Gbps may be deemed satisfactory by some edge-based bandwidth allocation schemes.

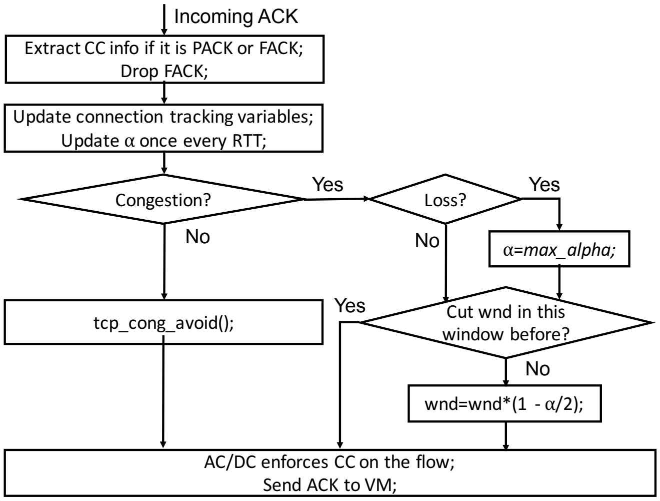

Incoming ACK
Extract CC info if it is PACK or FACK; Drop FACK;
Update connection tracking variables; Update α once every RTT;
Congestion?
Yes
No
Loss?
Yes
No
α=max_alpha;
tcp_cong_avoid();
Yes
Cut wnd in this window before?
No
wnd=wnd*(1 - α/2);
AC/DC enforces CC on the flow; Send ACK to VM;

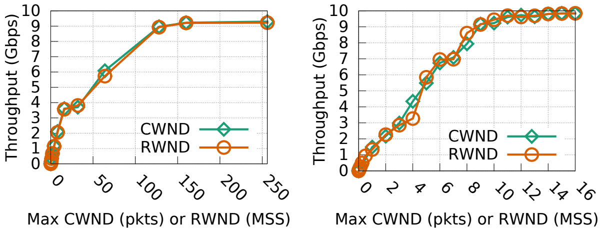
Figure 5: DCTCP congestion control in AC/DC.

This section discusses how to obtain state for and perform DCTCP's 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 receiver module strips the `congestion encountered` bits from packets in order to prevent the VM's TCP stack from decreasing rates too aggressively (recall DCTCP adjusts `CWND` proportionally to the fraction of packets that encounter congestion, 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 DCTCP, the AC/DC receiver module monitors the total and ECN-marked bytes received for a flow. Receivers piggy-back the reported totals on ACKs to the sender 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 [59]. 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 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 size, 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 size will be violated. The FACK is sent in addition to the real TCP ACK (RACK). FACKs are also discarded by the sender after logging the included data. In practice, most feedback takes the form of PACKs.



(a) MTU = 1.5kB. (b) MTU = 9kB.

Figure 6: Using `RWND` can effectively control throughput.

DCTCP congestion control Once the fraction of ECN-marked packets is obtained, implementing DCTCP's logic is straightforward. Figure 5 shows the high-level design. First, congestion control (CC) information is extracted from FACKs and PACKs. Connection tracking variables (described in §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 and no ECN), then `tcp_cong_avoid` advances the congestion window based on TCP New Reno's algorithm, using slow start or congestion avoidance as needed. If congestion was experienced, then the congestion window 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 [3, 11].

3.3 Enforcing Congestion Control

There must be a mechanism to ensure the VM's TCP flow adheres to the congestion control window determined in the vSwitch. Luckily, TCP provides built-in functionality that can easily be reprovisioned for AC/DC. Specifically, TCP's flow control allows a receiver to advertise the amount of data it is willing to process from a sender via a receive window (`RWND`). AC/DC reuses `RWND`. Specifically, the vSwitch calculates its own congestion window and overwrites `RWND`. In order to preserve TCP semantics, this value is overwritten only when it is smaller than the packet's original `RWND`. The VM's TCP flow then uses min(`CWND`, `RWND`) to limit how many packets it can send.

Ensuring a VM's TCP flows adhere to the `RWND` value 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 TCP 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. VM flows that circumvent the standard can be policed by dropping any excess packets not allowed by the calculated congestion window. Because dropped packets have to be recovered end-to-end, this incentivizes tenants to respect the standard.

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 [29, 30]. 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 [56, 64] manages buffer state and thus is an orthogonal scheme AC⚡DC can safely ignore.

While this scheme seems simple, it provides a surprising amount of flexibility. For example, TCP enables a receiver to send a TCP Window Update to update RWND [6]. AC⚡DC can create these packets to update windows without relying on a receiver's ACK. Additionally, the sender module can generate duplicate ACKs to send to a VM's TCP flow. 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 [66]). In this case, the vSwitch can send three dup-ACKs (with an appropiate RWND) to trigger the VM's TCP flow to retransmit. While our implementation currently only supports TCP traffic, we believe it can be easily extended to handle UDP traffic. Prior schemes funnel UDP traffic through VM-level congestion-controlled tunnels [33, 57], and we believe AC⚡DC can be adapted in a similar fashion to support per-flow DCTCP-friendly tunnels.

Finally, this enforcement scheme is simple and lightweight so it can scale 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 [53] and therefore do not scale gracefully. NIC or switch-based rate limiters are low-overhead, but typically only provide a handful of queues. Our enforcement algorithm reuses TCP's infrastructure so rate limiting schemes can be used at a coarser granularity (*e.g.*, VM-level).

3.4 Per-flow Differentiation

One benefit of AC⚡DC is different congestion control algorithms can be assigned 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 Compound TCP [61] 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 congestion window can be specified by snd_cwnd_clamp in Linux. AC⚡DC can provide similar functionality by bounding RWND. Figure 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 previously been studied [57, 67]. 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. We propose a unique priority-based congestion control algorithm for AC⚡DC. Specifically, DCTCP's congestion control algorithm is modified to incorporate a priority, $\beta \in [0, 1]$:

$$\text{rwnd} = \text{rwnd}(1 - (\alpha - \frac{\alpha\beta}{2})) \qquad (1)$$

Higher values of β give higher priority. When $\beta = 1$, Equation 1 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, whereas decreasing RWND can used to reduce a flow's sending rate.

4. IMPLEMENTATION

This section outlines relevant AC⚡DC implementation details. We implemented AC⚡DC in Open vSwitch (OVS) v2.3.2 [49] and added about 1200 lines of code (many are debug/comments). The implementation uses several standard techniques to reduce overhead. First, Read-Copy-Update (RCU) hash tables [25], which improve performance for read-heavy workloads, are used to keep per-flow state. Second, AC⚡DC works on TCP segments, instead of packets, which helps keep overhead low and ensures it is compatible with NIC offloading features (*i.e.*, TSO and GRO/LRO). Third, we leverage NIC checksumming so the TCP checksum does not have to be recomputed after header fields are altered. Finally, each TCP connection requires only 320 bytes of memory.

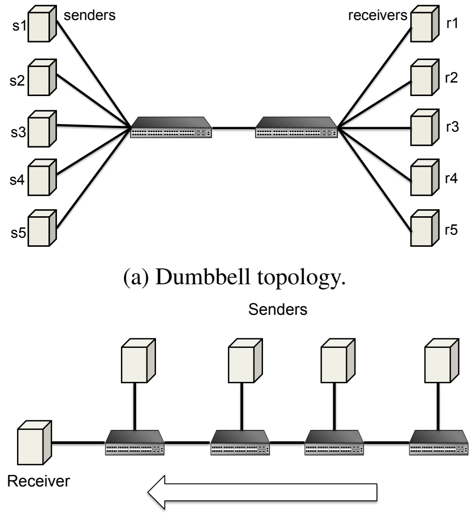
While our scheme is implemented in software, we are currently investigating the possibility of implementing AC⚡DC in the NIC. Today, "smart-NICs" implement OVS-offload functionality [38, 45], and naturally provide a mechanism to support hypervisor bypass (*e.g.*, SR-IOV).

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 (Broadcom Trident+), each with a buffer of 9MB shared by forty-eight 10G ports and four 40G ports. The switches have dynamic memory management enabled by default.

System settings We run Linux kernel 3.18.0 which implements DCTCP as a pluggable module [46]. We set RTO_{min} to 10 ms [35, 66] and also set tcp_low_latency, tcp_no_metrics_s tcp_sack to 1. Results are obtained with MTU sizes of 1.5kB and 9kB, as networks are typically configured with one of these settings. Due to space constraints, a subset



(a) Dumbbell topology.

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

Figure 7: Experiment topologies.

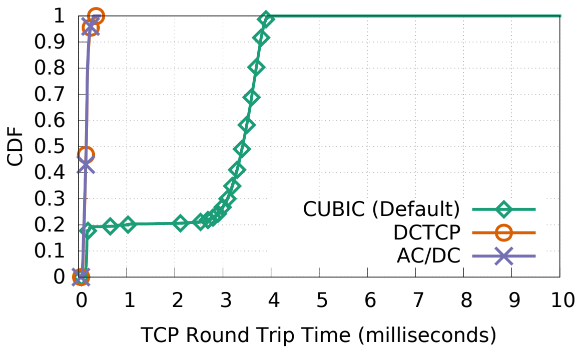


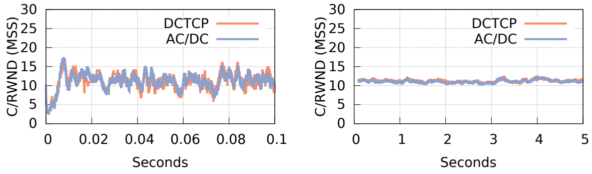
Figure 8: RTT of schemes on dumbbell topology.

of the results are presented and unless otherwise noted, the MTU is set to 9kB.

Experiment details To understand the performance of AC/DC, 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 performance metrics used are: TCP RTT (measured by sockperf [47]), TCP throughput (measured by iperf), packet loss rate (by collecting switch counters) and Jain's fairness index [31]. In §5.2, flow completion time (FCT) [18] is used to quantify application-level performance.

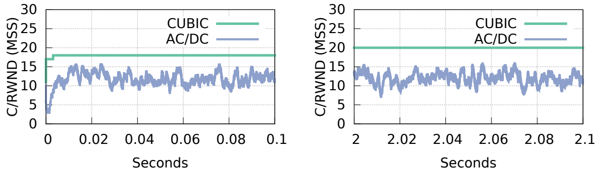
5.1 Microbenchmarks



(a) First 100 ms of a flow.

(b) Moving average.

Figure 9: AC/DC's `RWND` tracks DCTCP's `CWND`. MTU size = 1.5kB.



(a) Starting from 0 sec.

(b) Starting from 2 sec.

Figure 10: Who limits TCP throughput when AC/DC is run with CUBIC? MTU = 1.5kB.

We first evaluate AC/DC's performance using a set of microbenchmarks. The microbenchmarks are conducted on topologies shown in Figure 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 7a. The average per-flow throughput of AC/DC, DCTCP and CUBIC are all 1.98Gbps. Figure 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.

Second, each sender in Figure 7b starts a long-lived flow to the single 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 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 50th and 99.9th 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 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 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 DCTCP's `CWND` by using `tcpprobe` [63]. We align the `RWND` and `CWND` values by timestamps and sequence numbers and show a timeseries in Figure 9. Figure 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 9b shows the windows over a 100ms moving average are also similar. This suggests that it is easy to accurately recreate congestion control in the vSwitch. Note we obtained results for 9kB MTU that were similar.

CC Variants	50th percentile RTT (µs)		99th percentile RTT (µs)		Avg Tput (Gbps)		Fairness Index	
	mtu=1.5kB	mtu=9kB	mtu=1.5kB	mtu=9kB	mtu=1.5kB	mtu=9kB	mtu=1.5kB	mtu=9kB
CUBIC*	3232	3448	3641	3865	1.89	1.98	0.85	0.98
DCTCP*	128	142	232	259	1.89	1.98	0.99	0.99
CUBIC	128	142	231	252	1.89	1.98	0.99	0.99
Reno	120	149	235	248	1.89	1.97	0.99	0.99
DCTCP	129	149	232	266	1.88	1.98	0.99	0.99
Illinois	134	152	215	262	1.89	1.97	0.99	0.99
HighSpeed	119	147	224	252	1.88	1.97	0.99	0.99
Vegas	126	143	216	251	1.89	1.97	0.99	0.99

Table 1: AC⚡DC works with various kinds of congestion control variants. CUBIC*: CUBIC + standard OVS, switch WRED/ECN marking off. DCTCP*: DCTCP + standard OVS, switch WRED/ECN marking on. Others: different CCs + AC⚡DC, switch WRED/ECN marking on.

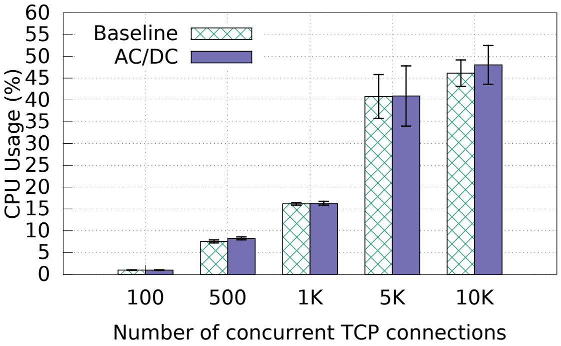


Figure 11: CPU overhead: sender side (1.5kB MTU).

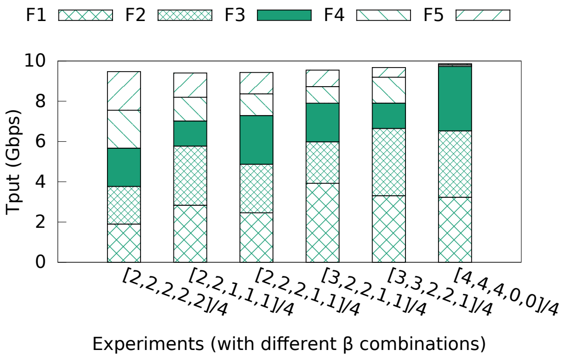


Figure 13: QoS-based CC.

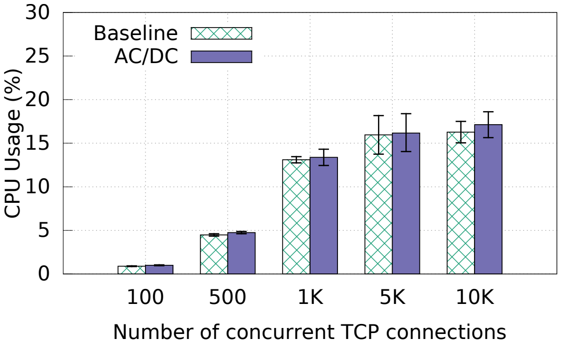
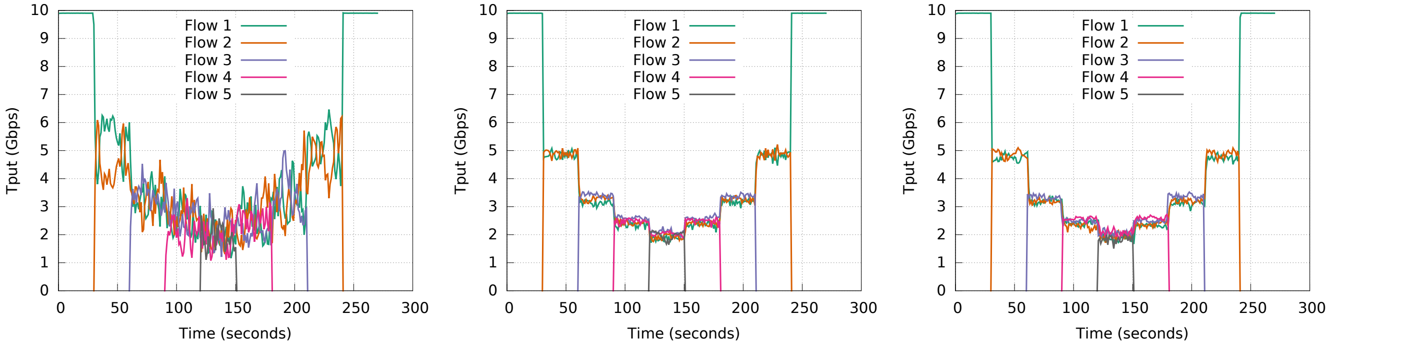


Figure 12: CPU overhead: receiver side (1.5kB MTU).

We were also interested to see how often AC⚡DC's congestion window takes effect. We rerun the experiment, 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 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`. Each flow is given time to perform the TCP handshake and when all are connected, each TCP client sends with a demand of 10 Mbps by sending 128 kB bursts every 100 milliseconds (so 1,000 connections saturate the 10 Gbps link). The CPU overhead of AC⚡DC compared to the CPU overhead of baseline (*i.e.*, just OVS) is shown for the sender in Figure 11 and the receiver in Figure 12. Even when scaling to 10,000 connections, the CPU overhead of our scheme is less than 4%. Note experiments with 9kB MTU have similar trends, but we show the 1.5kB MTU case because smaller MTU sizes incur higher overhead.

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 1 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 7a). The first two rows of the table, CUBIC* and DCTCP*, show the performance of each stack with an un-

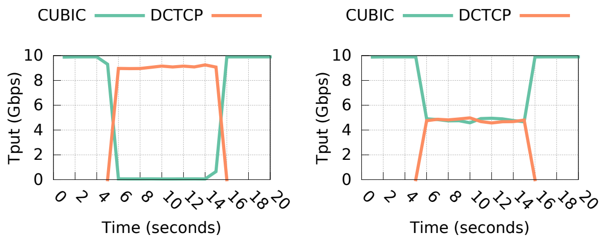


(a) CUBIC convergence test.

(b) DCTCP convergence test.

(c) AC⚡DC convergence test.

Figure 14: Convergence tests.



(a) Default.

(b) AC⚡DC.

Figure 15: (a) CUBIC gets little throughput when competing with DCTCP. (b) With AC⚡DC, CUBIC and DCTCP flows get fair share.

modified 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. 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 1) for each flow in the dumbbell topology. Figure 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 aim to show AC⚡DC can mimic DCTCP's behavior in converging to fair throughputs. We repeat the experiment originally performed by Alizadeh [3] and Judd [35] by adding a new flow every 30 seconds on a bottleneck link, and then reverse the process. The result is shown in Figure 14. Figure 14a shows CUBIC's problems converging to fair allocations. Figures 14b and 14c show DCTCP's and AC⚡DC's performance, respectively. AC⚡DC tracks DCTCP's behavior. CUBIC's drop rate is 0.17% while DCTCP's and AC⚡DC's is 0%.

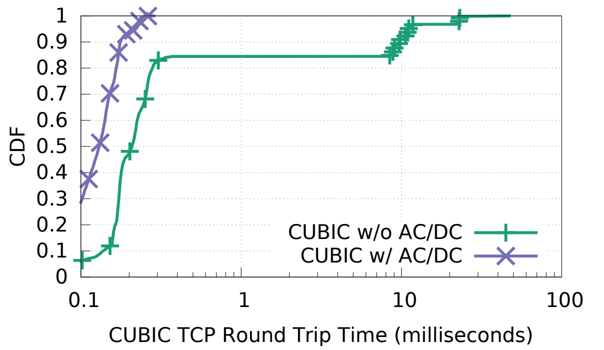
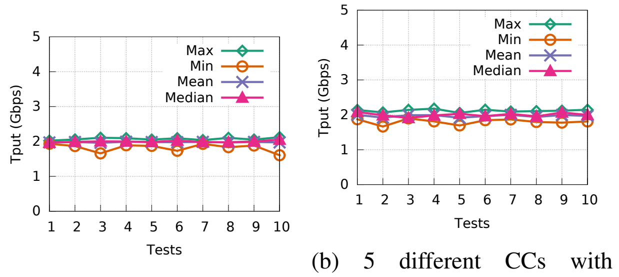


Figure 16: CUBIC experiences high RTT when competing with DCTCP. AC⚡DC fixes this issue.

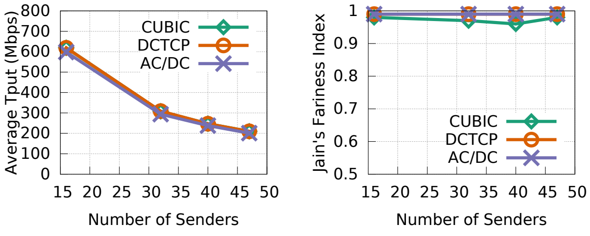


(a) All DCTCP.

(b) 5 different CCs with AC⚡DC.

Figure 17: AC⚡DC improves TCP fairness.

The second experiment is also repeated from Judd's paper [35]. ECN-capable and non-ECN-capable flows do not coexist well because switches drop non-ECN packets when the queue length is larger than the predefined marking threshold. Figure 15a shows the throughput of CUBIC suffers when CUBIC (with no ECN) and DCTCP (with ECN) traverse the same bottleneck link. Figure 15b shows this problem is alleviated with AC⚡DC because it forces all flows to become ECN-capable when they enter fabric. Figure 16 shows CUBIC's 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.



(a) average throughput. (b) fairness.

Figure 18: Many to one incast: throughput and fairness.

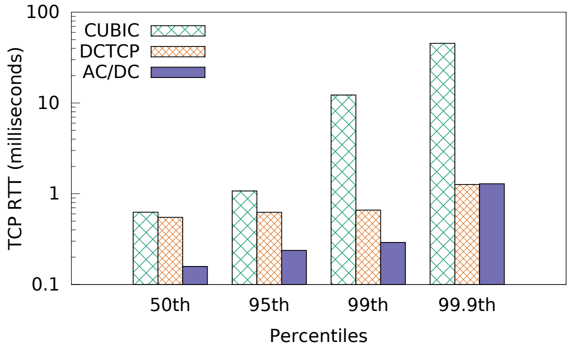


Figure 20: TCP RTT measurement results when almost all ports are congested.

However, implementing 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 1. Figure 17a shows what happens if all flows are configured to use DCTCP and Figure 17b 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 1b).

5.2 Macrobenchmarks

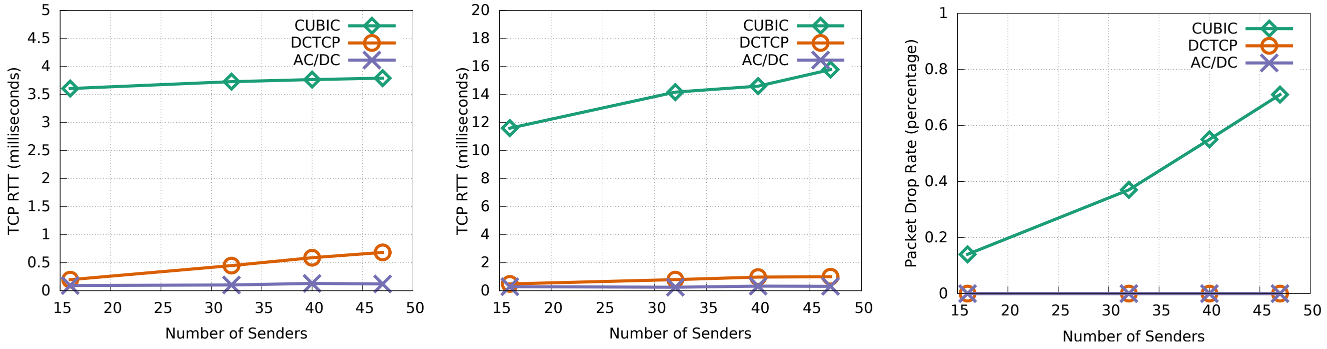
In this section we attach all servers to a single switch and run a variety of well-known workloads to better understand how well AC⚡DC can track DCTCP's performance. Experiments are run for 10 minutes. A simple TCP application sends messages of specified sizes to measure flow completion times.

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 18 shows throughput and fairness results. Both DCTCP and AC⚡DC get comparable throughput as CUBIC and both offer a fairness index greater than 0.99. Figure 19 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 99.9th 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⚡DC's performance is better than DCTCP when the number of senders increases (Figure 19a). 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 [35]. 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 checks whether AC⚡DC can interact well with the switch's dynamic buffer allocation scheme. 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 20. 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 99.9th percentile RTT for CUBIC is very high. The packet drop rate for both DCTCP and AC⚡DC is 0%.

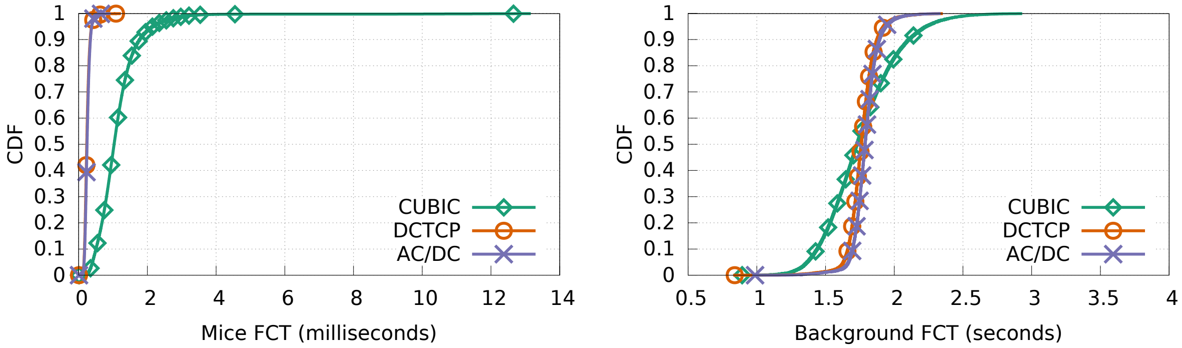
Concurrent stride workload In concurrent stride, 17 servers are attached to a single switch. Each server i sends a 512MB flow to servers $[i+1, i+4]$ mod 17 in sequential fashion to emulate background traffic. Simultaneously, each server i sends 16KB messages every 100 ms to server $(i+8)$ mod 17. The FCT for small flows (16KB) and background flows (512MB) are shown in Figure 21. For small flows, DCTCP and AC⚡DC reduce the median FCT by 77% and 76% respectively. At the 99.9th 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 22. For small flows, DCTCP and AC⚡DC reduce median FCT by 72% and 71% when compared to CUBIC. At the 99.9th percentile, DCTCP and AC⚡DC reduce FCTs by 55% and 73% respectively. For large flows, CUBIC, DCTCP and AC⚡DC have almost identical performance.



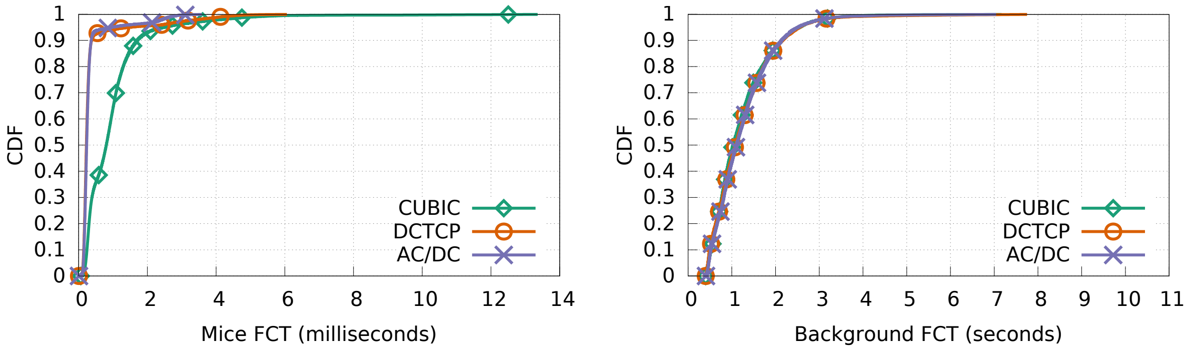
(a) 50th percentile RTT (µs). (b) 99.9th percentile RTT (µs). (c) packet drop rate.

Figure 19: Many to one incast: RTT and packet drop rate.



(a) Mice flow completion times. (b) Background flow completion times.

Figure 21: CDF of mice and background FCTs in concurrent stride workload.

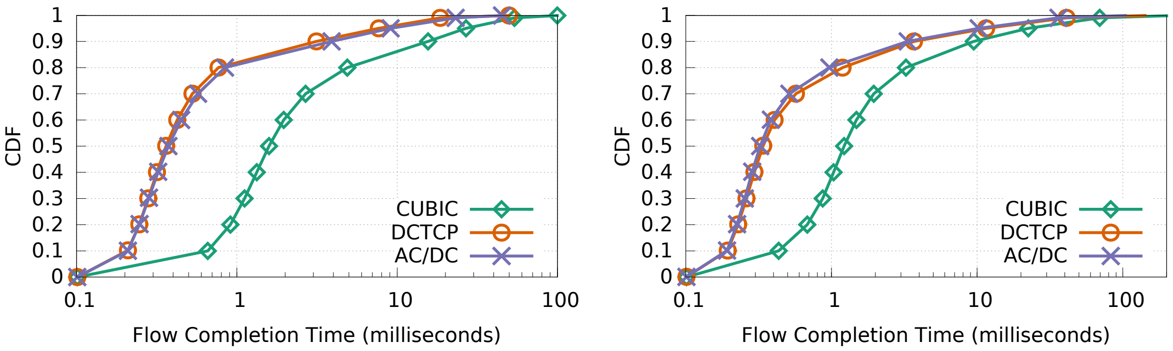


(a) Mice flow completion times. (b) Background flow completion times.

Figure 22: CDF of mice and background FCTs in shuffle 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 [3] and a data-mining workload [2, 23], whose flow size distribution has a heavier tail. Figure 23 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 99.9th 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 99.9th percentile, DCTCP and AC⚡DC reduce FCTs by 36% and 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.



(a) Web-search workload.

(b) Data-mining workload.

Figure 23: CDF of mice flow's (<10KB) FCT in web-search and data-mining workloads.

6. RELATED WORK

This section discusses five different classes of related work.

Congestion control for DCNs DCTCP [3] is a seminal TCP variant for datacenter networks. Judd [35] proposed simple yet practical fixes to enable DCTCP in production networks. TCP-Bolt [60] is a variant of DCTCP for PFC-enabled lossless Ethernet (DCB). DCQCN [73] is a rate-based congestion control scheme (built on DCTCP and QCN) to support RDMA deployments in PFC-enabled lossless networks. TIMELY [41] and DX [37] use accurate network latency as the signal to perform congestion control. TCP ex Machina [69] uses computer-generated congestion control rules. PERC [34] proposes proactive congestion control to improve convergence. ICTCP's [70] 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 [12, 62] to implement TCP Offload Engine (TOE) in specialized NICs are not widely deployed for reasons noted in [42, 65]. AC/DC is designed to work with commodity NICs.

Bandwidth allocation Many bandwidth allocation schemes have been proposed. Gatekeeper [55] and EyeQ [33] abstract the network as a single switch and provide bandwidth guarantees by managing each server's access link. Oktopus [10] provides fixed performance guarantees within virtual clusters. SecondNet [26] enables virtual datacenters with static bandwidth guarantees. Proteus [72] allocates bandwidth for applications with dynamic demands. Seawall [57] provides bandwidth proportional to a defined weight by forcing traffic through congestion-based edge-to-edge tunnels. NetShare [36] utilizes hierarchical weighted max-min fair sharing to tune relative bandwidth allocation for services. FairCloud [52] identifies trade-offs in minimum guarantees, proportionality and high utilization, and designs schemes over this space. Silo [32] 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 [48] 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 NIC hardware to perform packet scheduling. Silo's pacer injects void packets into original packet sequence to achieve pacing. FasTrack [48] offloads functionality from the server into the switch for certain flows. AC/DC prevents TCP flows from sending in the first place and can be used in conjunction with these schemes.

Low latency DCNs Many schemes have been proposed to reduce latency in datacenter networks. HULL [4] uses phantom queues to leave bandwidth headroom to support low latency. pFabric [5] is a clean-slate design which utilizes priority and minimal switch buffering to achieve low latency. Fastpass [50] uses a centralized arbiter to perform per-packet level scheduling. QJUMP [24] uses priority queueing and rate limiting to bound latency. Traffic engineering [1, 54] and load balancing [2, 22, 28] can also reduce latency. Because AC/DC works on the transport level, it is largely complimentary to these works.

Performance-enhancing proxies Several schemes improve end-to-end protocol performance via a middlebox or proxy [7, 8, 9, 15, 17]. AC/DC fits into this class of works, but is unique in providing a mechanism to alter a VM's TCP congestion control algorithm by modifying the vSwitch.

7. CONCLUSION

Today's datacenters host a variety of virtual machines (VMs) 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 paper, 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. Our results show the CPU overhead is less than 4% and our scheme effectively enforces an administrator-defined congestion control algorithm over a variety of tenant TCP stacks.

Acknowledgement

We would like to thank Jon Crowcroft (our shepherd) and the anonymous reviewers for their valuable feedback. This work is supported in part by IBM Corporation, National Science Foundation (grants CNS-1302041, CNS-1330308 and CNS-1345249) and the Wisconsin Institute on Software-Defined Datacenters of Madison.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *NSDI*, 2012.

[5] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. 2013.

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

[7] H. Balakrishnan, V. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. TCP Performance Implications of Network Path Asymmetry. RFC 3449, 2002.

[8] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *MobiCom*, 1995.

[9] M. Balakrishnan, T. Marian, K. Birman, H. Weatherspoon, and E. Vollset. Maelstrom: Transparent Error Correction for Lambda Networks. In *NSDI*, 2008.

[10] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[11] S. Bensley, L. Eggert, D. Thaler, P. Balasubramanian, and G. 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.

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

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

[14] 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.

[15] P. Davern, N. Nashid, C. J. Sreenan, and A. Zahran. HTTPEP: A HTTP Performance Enhancing Proxy for Satellite Systems. *International Journal of Next-Generation Computing*, 2011.

[16] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[17] G. Dommety and K. Leung. Mobile IP Vendor/Organization-Specific Extensions. RFC 3115, 2001.

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

[19] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

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

[21] 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.

[22] S. Ghorbani, B. Godfrey, Y. Ganjali, and A. Firoozshahian. Micro Load Balancing in Data Centers with DRILL. In *HotNets*, 2015.

[23] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[24] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[25] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The Read-Copy-Update Mechanism for Supporting Real-time Applications on Shared-memory Multiprocessor Systems with Linux. *IBM Systems Journal*, 2008.

[26] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[27] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 2008.

[28] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.

[29] V. Jacobson, B. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, 1992. http://www.rfc-editor.org/rfc/rfc1323.txt.

[30] V. Jacobson and R. Braden. TCP Extensions for Long-delay Paths. RFC 1072, 1988. http://www.rfc-editor.org/rfc/rfc1072.txt.

[31] R. K. Jain, D.-M. W. Chiu, and W. 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.

[32] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*, 2015.

[33] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[34] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

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

[36] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *ACM SIGCOMM Computer Communication Review*, 2012.

[37] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *USENIX Annual Technical Conference*, 2015.

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

[39] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A Loss-and Delay-based Congestion Control Algorithm for High-speed Networks. *Performance Evaluation*, 2008.

[40] A. Menon and W. Zwaenepoel. Optimizing TCP Receive Performance. In *USENIX Annual Technical Conference*, 2008.

[41] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

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

[43] J. C. Mogul and R. R. Kompella. Inferring the Network Latency Requirements of Cloud Tenants. In *HotOS*, 2015.

[44] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. In *NSDI*, 2013.

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

[46] net:tcp: DCTCP congestion control algorithm. https://lwn.net/Articles/598432/.

[47] Network Benchmarking Utility. https://github.com/mellanox/sockperf.

[48] R. Niranjan Mysore, G. Porter, and A. Vahdat. FasTrak: Enabling Express Lanes in Multi-tenant Data Centers. In *CoNEXT*, 2013.

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

[50] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized Zero-queue Datacenter Network. In *SIGCOMM*, 2014.

[51] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[52] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[53] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for End-host Rate Limiting. In *NSDI*, 2014.

[54] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *SIGCOMM*, 2014.

[55] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV*, 2011.

[56] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *ACM SIGCOMM Computer Communication Review*, 1998.

[57] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[58] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

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

[60] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for Improved Data Center Networks. In *INFOCOM*, 2014.

[61] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

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

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

[64] TCP Receive Window Auto-Tuning. https://technet.microsoft.com/en-us/magazine/2007.01.cableguy.aspx.

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

[66] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM*, 2009.

[67] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. *SIGOPS Oper. Syst. Rev.*, 2002.

[68] G. Wang and T. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.

[69] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-generated Congestion Control. In *SIGCOMM*, 2013.

[70] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNEXT*, 2010.

[71] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang. Tuning ECN for Data Center Networks. In *CoNEXT*, 2012.

[72] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.

[73] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.