Adapting TCP for Reconfigurable Datacenter Networks

Matthew K. Mukerjee[⋆†], Christopher Canel[⋆], Weiyang Wang[°], Daehyeok Kim[⋆‡],
Srinivasan Seshan[⋆], Alex C. Snoeren[°]
[⋆]*Carnegie Mellon University,* [°]*UC San Diego,* [†]*Nefeli Networks,* [‡]*Microsoft Research*

Abstract

Reconfigurable datacenter networks (RDCNs) augment traditional packet switches with high-bandwidth reconfigurable circuits. In these networks, high-bandwidth circuits are assigned to particular source-destination rack pairs based on a schedule. To make efficient use of RDCNs, active TCP flows between such pairs must quickly ramp up their sending rates when high-bandwidth circuits are made available. Past studies have shown that TCP performs well on RDCNs with millisecond-scale reconfiguration delays, during which time the circuit network is offline. However, modern RDCNs can reconfigure in as little as 20 μs, and maintain a particular configuration for fewer than 10 RTTs. We show that existing TCP variants cannot ramp up quickly enough to work well on these modern RDCNs. We identify two methods to address this issue: First, an in-network solution that dynamically resizes top-of-rack switch virtual output queues to prebuffer packets; Second, an endpoint-based solution that increases the congestion window, `cwnd`, based on explicit circuit state feedback sent via the `ECN-echo` bit. To evaluate these techniques, we build an open-source RDCN emulator, Etalon, and show that a combination of dynamic queue resizing and explicit circuit state feedback increases circuit utilization by 1.91× with an only 1.20× increase in tail latency.

1 Introduction

Modern datacenter applications need high-bandwidth, high–port-count, low-latency, low-cost networks to connect their hosts. Unfortunately, traditional packet switches are hitting CMOS manufacturing limits and are unable to simultaneously provide both high bandwidth and large numbers of ports [43]. Thus, researchers have proposed augmenting datacenter networks with reconfigurable circuit switches (e.g., optical or wireless) that provide high bandwidth between racks *on demand* [6, 16, 20, 25, 26, 32, 38, 42, 47, 51, 57].

However, it can be challenging for endpoints to extract the full potential of *reconfigurable datacenter networks* (*RDCNs*) that combine both circuit and packet networks. Circuit switches incur non-trivial reconfiguration delays while they adjust the high-bandwidth topology, and portions of the circuit network may be unavailable during these periods. Hence, such hybrid designs often result in fluctuations between periods of high bandwidth—when a circuit is provisioned—and low bandwidth—when the packet network is in use. While periods of higher bandwidth are attractive in principle, recent proposals suggest adjusting the topology frequently. The resulting bandwidth fluctuations pose a problem for end-host applications: their active TCP connections must rapidly increase transmission rates to use the available bandwidth and then slow down again to avoid massive queuing. In this paper, we explore these adverse interactions between TCP and RDCNs, and techniques to mitigate their performance impacts.

To amortize the cost of reconfiguration, circuits must be provisioned for a long period of time relative to the reconfiguration delay of the switch. TCP interactions with RDCNs were initially explored in the context of switches with *millisecond*-scale reconfiguration delays [16, 51]. Given the sub-millisecond propagation delays found in modern datacenters, circuits in millisecond-scale reconfiguration networks are enabled for many, many round-trip times (RTTs). These early studies found that TCP was able to adapt to the link speed changes over these time periods. However, modern reconfigurable switches that can change link state on the scale of *microseconds* [20, 38, 47] have redefined these problems. At first glance, these lower reconfiguration delays result in lower overheads and allow more rapid provisioning of bandwidth to where it is needed. However, when circuit uptimes are only a few (e.g., <10) RTTs long, TCP's ramp-up is too slow to utilize the large (e.g., 10×), temporary bandwidth increase before the circuit disappears, leading to low circuit utilization (Section 3). This raises the question of whether an RDCN can employ the rapid reconfigurations needed to meet changing traffic demands while also providing good performance.

The performance issues arise from a broken assumption made by end-hosts about the network: congestion control algorithms generally assume that bandwidth does not fluctuate at short timescales. We explore the consequences of

this broken assumption for TCP, and identify two methods, at different levels in the network stack, for ramping up TCP in the environment of rapid bandwidth fluctuation.

First, we build on the insight that in RDCNs, bandwidth fluctuation is not arbitrary; it is part of a known schedule. Therefore, we can proactively modify the network to transparently influence end-host behavior. In this case, we entice TCP to ramp up earlier by eliminating packet drops due to full top-of-rack (ToR) switch virtual output queues (VOQs), thereby triggering end-hosts to increase their sending rates. We accomplish this by dynamically increasing the size of ToR VOQs in advance of provisioning a circuit (Section 5.2). Dynamic VOQ resizing does not require modifying end-hosts and, thus, works with existing TCP implementations.

Our second technique involves minor modifications to the end-host TCP stack to enable further performance improvements. At sending end-hosts, we increase the congestion window, `cwnd`, based on explicit circuit state feedback sent by the reconfigurable switch (Section 5.3). For some rack pair (S, D), we configure our emulated switch to set the `ECN-echo` (ECE) bit in the TCP headers of ACKs sent by D if there is currently a circuit enabled from S to D . The sender monitors the ECE stream, explicitly expanding and contracting `cwnd` when circuits begin and end, respectively.

To evaluate our solutions, we design and implement an open-source RDCN emulator, *Etalon*[1], for use on public testbeds (Section 4). Experiments on 3-rack (48-host) and 8-rack (128-host) emulated testbeds show that dynamic buffer resizing and explicit circuit state feedback increase circuit utilization by 1.91× while increasing 99th percentile tail latency by 1.20× (Sections 6.1 and 6.3).

Ultimately, datacenters must adapt to reap the benefits of RDCNs. In-network changes yield impressive improvements, and if modifying end-hosts is an option, then even higher performance is feasible. We make three contributions:

1. We characterize the critical challenge of rapid bandwidth fluctuation in RDCNs. We use a combination of experimental results and simulations to identify the range of reconfiguration delays that impact TCP performance, and show that a wide range of TCP congestion control algorithms suffer from poor performance in these settings.
2. We propose two solutions, at different layers of the network stack, to ramp up TCP under rapid bandwidth fluctuation: dynamic buffer resizing and explicit circuit state feedback. Our evaluation of these techniques shows the benefits that modifying higher network layers can have for RDCNs.
3. We design and implement an emulation platform, Etalon, for evaluating hybrid networks end-to-end with real applications, and use it to demonstrate the efficacy of our proposed techniques. Etalon is open source [15].

[1]Named after an optical filter used for solar observation.

2 Background

To better understand the challenges and solutions presented in this paper, we first examine RDCNs in detail in Figure 1. While we use optical circuit switching [16, 38, 47, 51] as an illustrative example for the rest of the paper, the results generalize to other reconfigurable technologies, such as free-space optics [20, 26] and 60-GHz wireless [25, 32, 57]). We eschew older millisecond-scale reconfigurable switches [16, 51] for modern microsecond-scale switches [20, 38, 42, 47], as the nature of the challenges and solutions differ with timescale.

2.1 Hybrid Network Model

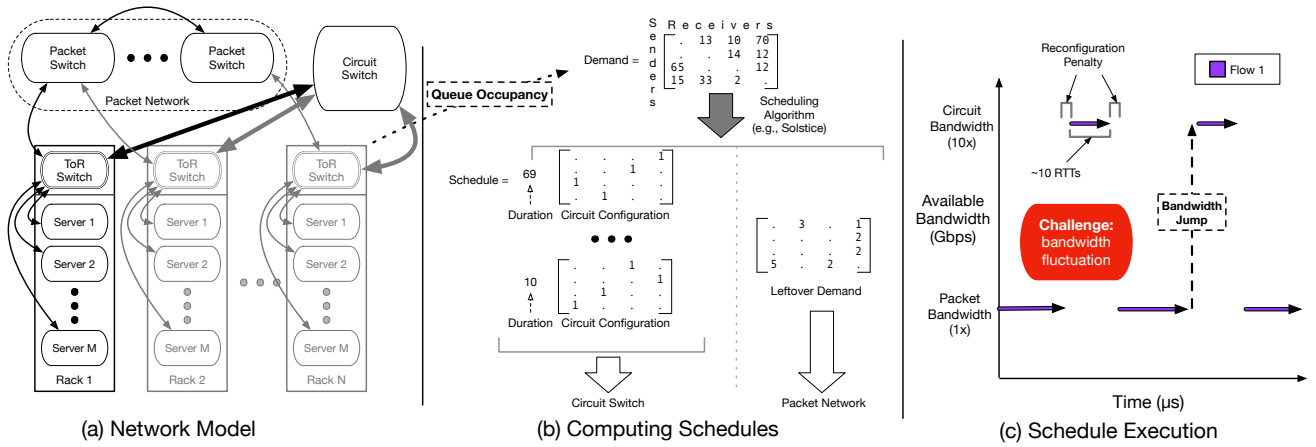
We consider an RDCN of N racks of M servers, with each rack containing a ToR switch (Figure 1(a)). ToRs connect racks to an arbitrarily-complex packet network (one or more switches) and a circuit network composed of a single, central circuit switch. The packet network is low bandwidth (e.g., 10 Gb/s), but can make forwarding decisions for individual packets. The circuit network is high bandwidth (e.g., 80-100 Gb/s), but makes forwarding decisions for many packets on much longer timescales to amortize its reconfiguration penalty.

Reconfiguration time is an inherent trade-off in circuit-switched networks. Instead of providing continuous connectivity between all endpoints, like in a packet network, a circuit network establishes exclusive, but temporary, connections between *pairs* of endpoints. Here, the endpoints are the ToRs. To expand this design to provide full connectivity, the network periodically changes which pairs of endpoints are spanned by a circuit. The physical limits of the specific underlying technology determine how long this reconfiguration takes.

Following prior work [38, 39, 47], we make the pessimistic assumption that, during circuit reconfiguration, no circuit links can be used. This allows us to apply our results to a broader set of technologies. The packet network, on the other hand, can be used at all times. Both switches source packets from $N \times N$ virtual output queues (VOQs) on the ToRs. The circuit switch itself is queue-less: It functions as a crossbar, only allowing configurations that form perfect matchings [2, 16, 38, 39, 47, 51]. I.e., a given sender is connected to exactly one receiver, and vice-versa. Thus, at any point in time, the circuit switch can drain at most one VOQ on each ToR, whereas the packet switch may drain multiple VOQs on each ToR simultaneously.

2.2 Computing Circuit Schedules

The circuit scheduler is tasked with estimating demand on the network and then computing a set of configurations that best satisfies this demand (Figure 1(b)). During reconfiguration, the circuit network is unavailable. Therefore, the circuit scheduler must balance the competing objectives of (1) servicing demand from many different rack pairs by reconfiguring



(a) Network Model (b) Computing Schedules (c) Schedule Execution

Figure 1: Overview of RDCNs.

frequently and (2) achieving high circuit uptime by allowing a configuration to persist for a (relatively) long time. To achieve a high relative uptime of 90%, schedules typically hold a particular circuit state for 9× the duration of a reconfiguration.

Network scheduling in most RDCNs entails mapping rack-level demand to a set of circuit configurations (port-matchings) with corresponding time duration[2]. Any "leftover" demand is handled by the lower-bandwidth packet switch. Borrowing terminology from prior work [47], we refer to a single circuit uptime as a *day* and the reconfiguration period, during which the circuit switch is offline, as a *night*. Night length is determined by switch technology and is generally 10-30 μs [20,38,39,47]. To allow for at least 90% link uptime, the average day length must be ≥9× greater than the night length, or ~90-270 μs. A series of one or more day/night pairs that implement a set schedule is a *week*. Weeks should be sufficiently long (e.g., 2 ms) to amortize schedule computation.

Scheduling is a three-step loop: 1) Demand for the next week is estimated (e.g., through ToR VOQ occupancy); 2) An algorithm computes the schedule for the next week; 3) The schedule is disseminated to the switch. Scheduling algorithms for RDCNs (e.g., Solstice [39] and Eclipse [2]) use skew and sparsity in demand to minimize the number of configurations. Prior work on circuit scheduling informs, but is orthogonal to, our investigation into the resulting bandwidth fluctuations.

2.3 Schedule Execution

Once a schedule is disseminated to the circuit switch, it runs the circuit configurations for their respective duration (Figure 1(c)). After reconfiguration, a flow may transition from using the packet network to using the circuit network and vice versa. Because of the short day length, flows likely spend only a few (e.g., <10) RTTs on the circuit network each day. Therefore, transport protocols must cope with large (e.g., 10×) bandwidth variations based on which network a flow traverses.

[2]A major exception being RotorNet [42], which uses a predetermined schedule. It still suffers from the same bandwidth fluctuations.

Prior work has avoided the bandwidth fluctuation problem by segregating traffic into mice and elephant flows and routing them exclusively over the packet and circuit networks, respectively [16,38]. Each flow encounters one bandwidth regime, albeit the elephant flows must pause during circuit downtime. However, recent work has proven that such segregation is sub-optimal [19]. We adopt a non-segregated approach, treating the hybrid network as indivisible and routing all traffic over available circuit links, thus trading off reduced network complexity for bandwidth fluctuation.

3 TCP in RDCNs: Trends and Challenges

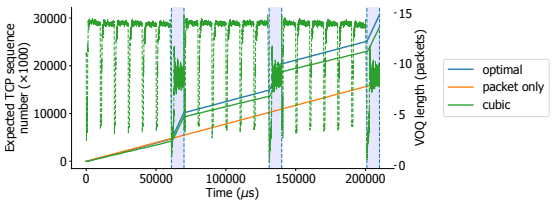
This section investigates how TCP's interactions with RDCNs are evolving with the underlying hardware trends.

3.1 Evolving Reconfiguration Delays

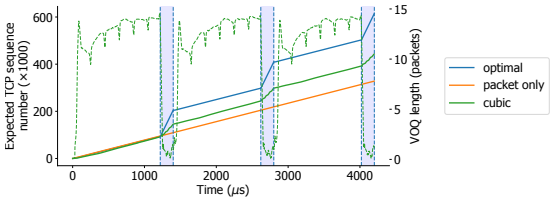
Circuit networks are characterized by their inherently high bandwidth. However, that comes at the cost of flexibility. With the exception of pathological examples, switching flexibility enables the circuit network to better serve diverse workloads. Improving flexibility by reducing the reconfiguration time has been an ongoing challenge for hardware designers, with the hope that circuit technologies will eventually be capable of approximating packet switching.

A decade ago, the best MEMS (Micro-Electro-Mechanical Systems) optical switches, which reconfigure by physically rotating laser-directing mirrors, changed paths on the order of a few milliseconds. If we generously assume that such a switch can reconfigure in 1 ms, our 90% uptime target implies that each configuration will persist for 9 ms. On network timescales, this is an eternity: Assuming a conservative RTT of 60 μs, 9 ms is 150 RTTs.

To understand how TCP behaves on RDCNs, we use an emulator, described in Section 4, and analyze the expected TCP sequence number of flows over time. We run an 8-rack schedule with 1 ms nights and 9 ms days, using the CUBIC [24]



(a) A decade ago: 1-ms reconfiguration delays and 9-ms circuit uptimes give TCP time to saturate the link.



(b) Today: TCP struggles to fill the link for circuits with short 20 μs reconfiguration delays and 180 μs uptimes.

Figure 2: TCP CUBIC performance, then and now. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.

variant of TCP. 16 emulated hosts on rack 1 each send a flow to a counterpart host on rack 2. In this experiment, we consider a circuit switch that delivers 8× higher bandwidth than the packet network. The ToR VOQ capacity is 16 packets. The expected sequence number is measured as packets leave the hybrid switch, as described in Section 4.2, and then averaged across experimental runs.

Figure 2a shows the results of this experiment. Circuit uptimes are delineated by the blue vertical shaded regions. Since the sequence number represents the number of bytes transferred, the slope of a line corresponds directly to a flow's achieved bandwidth. We always compare to two baselines: (1) *optimal*, which is calculated based on the line rate of the packet and circuit links (taking into account that the network is offline during reconfigurations), and (2) *packet only*, which is calculated assuming that flows always use the packet network only (and that the packet network is always available). Intuitively, circuit utilization is how well the slope of a line matches that of *optimal* during the blue shaded regions. Experimental results can never exceed *optimal*, and any improvement over *packet only* illustrates the benefit of the hybrid network over a purely packet network. Throughout our analysis, we use the length of the ToR VOQs, described further in Section 4.2, to understand TCP's behavior. For a particular line on a sequence number graph, the dotted line in the same color reports the corresponding average VOQ length.

We can see that TCP roughly keeps up with the optimal throughput of the hybrid network, saturating the link. During the circuit days, which are shaded in blue, the VOQs contain approximately 8 packets. Since the VOQs never empty except for a brief moment when the days begin, there are always

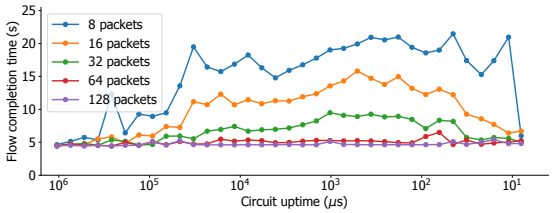


Figure 3: Simulated flow completion time for transferring 25 GB of data using 10 flows, for various buffer sizes.

packets to send over the network, resulting in high utilization.

Let us consider how hybrid networks have evolved over the past decade. Figure 2b shows the same results for a modern schedule with 20 μs nights and 180 μs days. The ratio of circuit uptime is the same, 90%, but TCP performs differently. Because the number of RTTs in a day has decreased from ~150 to ~3, TCP does not ramp up before the circuit ends. For TCP CUBIC, 3 RTTs is simply not enough time to increase the congestion window (`cwnd`) [24]. Moreover, packet drops during the subsequent reconfiguration period and the transition to the packet network cut TCP's sending rate. While TCP does recover while using the packet network, the process repeats at the next cycle, with TCP unable to ramp up to use the circuit network's full bandwidth regardless of how many periods elapse. We can also see this manifested in the VOQ length: When a day begins, the VOQs drain immediately to fill the larger bandwidth-delay product (BDP) of the circuit network, but there are insufficient outstanding packets to do so completely and insufficient time for TCP to ramp up, so the VOQs stay empty throughout the day.

To extend these results to a wider range of circuit uptimes, we ran a simulation that transfers 25 GB of data from one rack to another using 10 TCP CUBIC flows[3]. Figure 3 shows the resulting flow completion times (FCTs) for five orders of magnitude of circuit uptimes, where each line corresponds to a different amount of queuing at the ToR switch. In all cases, the RTT is 60 μs. The takeaways here are twofold: First, considering the smaller queue sizes (8-32 packets), the FCTs are low for short (e.g., 10 μs) and long (e.g., 10 ms) days, yet degrade by $2-4\times$ for moderate values (e.g., 1 ms). For short circuits, the network is effectively approximating packet switching, whereas for long circuits, the uptime is sufficient that fluctuating bandwidth is not an issue. Unfortunately, today's RDCNs fall in the middle region. Techniques to adapt TCP for RDCNs, like those presented in this paper, are necessary as long as circuit reconfiguration and propagation delays place us in a regime similar to this. We predict that the order-of-magnitude improvement in underlying circuit technology that is required to reach the "short circuits" region, where the network approximates packet switching and TCP ramp-up is no longer a problem, is still many years away.

[3]The simulator uses 1500B packets while the Etalon emulator uses 9000B packets. For the experiment in Figure 3, we scale the buffer size by 6× for easier comparison. I.e., for "8 packets", the buffer was $8\times6=48$ packets.

We find inspiration, however, from a second takeaway: With larger queue sizes (e.g., 64 and 128 packets), the flows complete quickly regardless of the circuit uptime. This indicates that large buffers build up enough excess in-flight data to burst packets quickly when more bandwidth becomes available. We discuss this further in Section 5.1, and this realization become the basis for the dynamic buffer resizing technique that we propose in Section 5.2.

3.2 Categorizing TCP Variants

Before diving into our technical solutions, it is important to remember that TCP comes in many shapes and sizes. The experiments above use TCP CUBIC, a common loss-based variant, but there are dozens of other variants designed for a plethora of network contexts, from low latency to high bandwidth to frequent loss, and more. This section gives an overview of the broad classes of TCP variants and demonstrates that no existing variant works well for hybrid networks.

At the highest level, the goal of TCP congestion control is to maximize the sending rate while fairly sharing the available bandwidth between flows and avoiding overloading the network. Determining the sending rate involves looking at signals from the network to infer its current state. The efficacy of a congestion control algorithm depends on how well it gleans information from such signals. Below, we discuss the three main categories of signals that TCP variants use to detect congestion (packet loss, network delay, and explicit network feedback), and discuss how they are effected by RDCNs.

3.2.1 Loss-based Congestion Control

Packet loss is the most commonly used congestion signal. The intuition here is that when the sender determines that packets are being lost, it assumes that the losses are a result of network congestion. Examples include TCP CUBIC [24], Reno [31], BIC [54], Illinois [40], and Highspeed [18]. In the absence of loss, these protocols increase their transmission rates to probe for available bandwidth, until a loss occurs. Different protocols choose different approaches for this probing, but all of them limit the aggressiveness of their probing to coexist reasonably with other TCP variants. This results in poor performance in RDCNs since these protocols cannot ramp up quickly enough to make use of the high-bandwidth circuits.

3.2.2 Delay-based Congestion Control

Another technique is to use measurements of the packet RTT in the congestion control protocol. Such protocols use RTT increases as an indicator of queue buildup in the network, and therefore congestion. Examples include BBR [5], TCP Vegas [4], and TIMELY [45]. Vegas and BBR are both rate-based protocols that use the difference between the offered load and the achieved throughput to detect queue buildup. TIMELY uses high-fidelity NIC timers to measure the RTT and then paces transmission based on delay gradients.

Like other TCP variants, delay-based variants are typically conservative in their probing for available bandwidth to ensure fair coexistence. An additional interaction with RDCNs is that, due to topology differences, the circuit and packet networks typically have different propagation delays (as discussed in Section 4.4: 30 μs vs. 10 μs, respectively). This poses a challenge for TCP variants that use changes in RTT as an indication of queuing.

3.2.3 Explicit Feedback–based Congestion Control

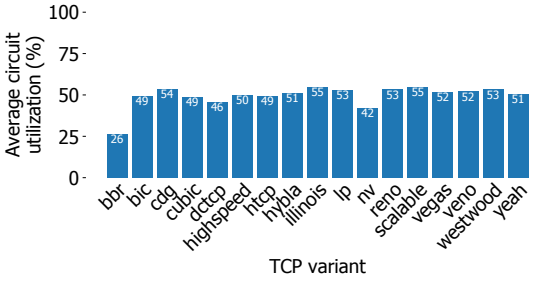
Finally, some TCP variants rely on explicit feedback from the network to detect congestion. Two manifestations of this are XCP [33] and DCTCP [1]. DCTCP responds to switch signals sent when a switch is likely to drop packets soon. If the act of a switch accepting a packet would increase its internal buffer length beyond a threshold (which is set to be lower than the total capacity of the queue), then the switch accepts the packet but sets the ECN flag in its TCP header. This flag is then communicated back to the sender via the packet's ACK. The sender monitors this stream of ECN marks to estimate when the network is close to being congested.

For domains where a single entity controls the senders and the network, coordination in this manner is a direct technique for improving performance. In Section 5.3, we propose a similar technique for adapting to bandwidth fluctuations in RDCNs that uses the ECE bit in ACKs to notify a sender when one of its flows transitions between the packet network and a high-bandwidth circuit.

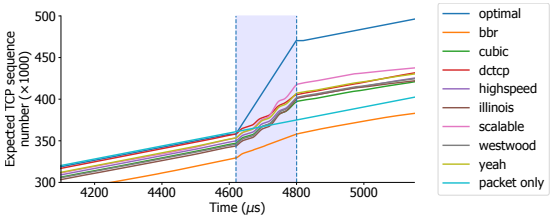
3.2.4 A Common Underlying Issue

We repeat the experiment in Section 3.1 with the 17 TCP variants pre-installed on Ubuntu 18.04. Figure 4a shows the average circuit utilization, which does not exceed 55%. Figure 4b visualizes the expected TCP sequence number over time for a selection of 8 of the 17 variants. Most perform slightly better than CUBIC, with BBR and NV falling behind, but no variant is aggressive enough to overcome the limitation that 3 RTTs is insufficient time to ramp up.

These results demonstrate that the challenge of TCP not ramping up quickly enough is not isolated to CUBIC, and cannot be solved by simply choosing a more appropriate TCP variant. Instead, we need a technique that more-directly interacts with the basic properties of TCP. Furthermore, the fact that all of the (sometimes quite) different TCP variants perform similarly (i.e., with surprisingly little variation, given their technical differences) suggests that they are all hampered in a similar way. If we address this common issue, then we have the potential to improve circuit utilization across the board, for all of these TCP variants.



(a) Circuit utilization vs. TCP variant.



(b) Expected TCP seq. num. vs. time, for many TCP variants. Circuit days are shared in blue.

Figure 4: With 20 μs reconfiguration delays and 180 μs circuit days, no TCP variant performs well.

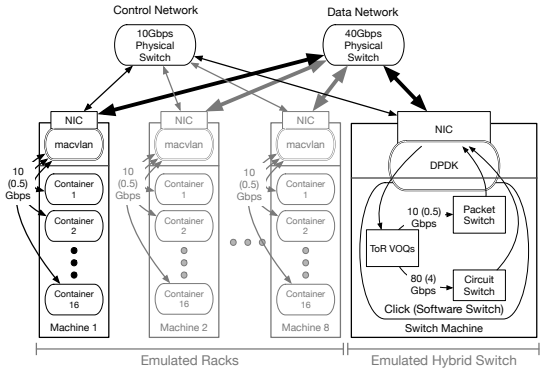


Figure 5: Etalon emulating 8 racks of 16 servers.

4 The Etalon Emulator

One of the key challenges in understanding TCP performance on RDCNs is performing repeatable experiments at scales appropriate for modern distributed cloud applications (i.e., across dozens or hundreds of hosts). In this section, we present our open-source emulator, Etalon [15], which measures the end-to-end performance of *real applications and end-host stacks* on emulated RDCNs in public testbeds.

4.1 Overview

Figure 5 presents an overview of Etalon. Each of the N physical machines emulates a rack of M servers using Docker containers [11, 44]. Containers are connected to the physical NIC using macvlan [12], which virtualizes a physical NIC into multiple virtual NICs, connecting them with a lightweight layer-2 software switch. `tc` [49] controls link bandwidths between the containers and the virtual switch, emulating a host-to-ToR link.

A separate physical machine emulates the reconfigurable datacenter network, as described in Section 4.2. Therefore, a cluster of $N+1$ physical machines can emulate $N \times M$ virtual hosts. For convenience, each physical host is connected to separate control and data networks, but this is not necessary. The experiment harness communicates with the testbed using RPyC [48]. Section 4.3 explains how time dilation enables Etalon to emulate many hosts with high-bandwidth links on a small testbed.

4.2 Click Software Switch

The $(N+1)^{\text{st}}$ machine emulates the RDCN itself, namely the ToR VOQs and the hybrid switch. This host runs a Click [34] software switch that uses DPDK [13] to process packets at line rate. We choose to emulate ToR VOQs in the software switch to make circuit and packet link emulation straightforward.

Figure 6 shows the software switch's internals. Packets enter the switch via DPDK [13] and are sent to an emulated ToR VOQ based on their *(source rack, destination rack)* pair. To achieve line rate, Etalon uses the Click elements `FromDPDKDevice` [7] and `ToDPDKDevice` [8] to exchange packets with the NIC. Packets are pulled from each VOQ by either the packet switch or the circuit switch. In Figure 6, packet uplink i is connected to the N VOQs in ToR i, pulling packets from these VOQs in a *round-robin fashion*. A packet pulled by a packet uplink enters the packet switch, where it is multiplexed over a packet downlink and transmitted using DPDK. If a packet would be dropped in the packet switch, it is held at the ToR VOQ (similar to PFC [28]). Circuit link i is connected to the ith VOQ of each of the N ToRs via a *pull switch*. A settable "input" value on pull switch i connects circuit link i to exactly one VOQ at a time. After packets traverse the circuit link, they are transmitted using DPDK. Before releasing a packet, the Click hybrid switch logs its IP and TCP headers, the current circuit state, and timing information. These logs are used offline to analyze the expected sequence number over time. The experiment harness communicates with the software switch using Click's control socket.

Our software switch contains three control elements (shown in gray in Figure 6): a demand estimator, a scheduler, and a schedule executor. The demand estimator estimates rack-to-rack demand using ToR VOQ occupancy. The scheduler computes a schedule from this demand, which is then run by the schedule executor by modifying the circuit link pull switches' "input" value (as described above). Our scheduler element is pluggable: We implement Solstice [39] as an example, but modify its objective to schedule maximal demand within a set window W (like Eclipse [2]), rather than scheduling all demand in unbounded time. We integrate Solstice, however, purely for implementation completeness. For our evaluation, a simple fixed strobe schedule, as described in

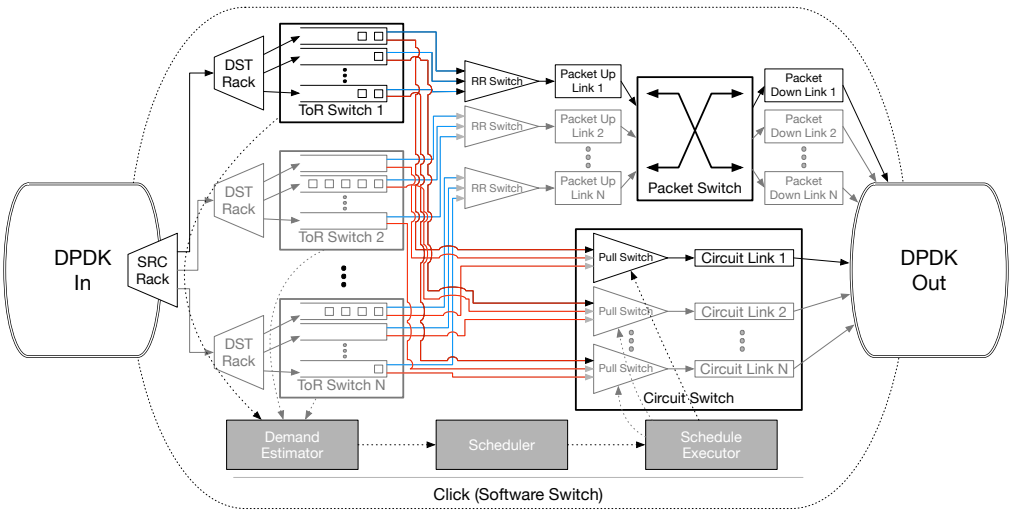


Figure 6: The Click software switch emulates the ToRs, the packet and circuit networks, and scheduling elements.

Section 4.5, is sufficiently illustrative.

4.3 Time Dilation

As the goal of Etalon is to emulate RDCNs on public testbeds, the machine emulating the hybrid network likely has only a single high-speed NIC. However, we wish to emulate a switch with N high-speed ports. We solve this problem with *time dilation (TD)*. Originally proposed for VMs [21, 22, 50] and recently containers [35, 55, 56], TD provides accurate emulation of higher-bandwidth links by "slowing down" the rest of the machine. We refer to the constant factor by which time is dilated as the *time dilation factor (TDF)*. We implement an open-source interposition library called LibVirtualTime (LibVT) [37], which applies TD to many common syscalls without requiring applications changes. We catch: `clock_gettime()`, `gettimeofday()`, `sleep()`, `usleep()`, `alarm()`, `select()`, `poll()`, and `setitimer()`. Extending LibVT to other syscalls is trivial. We verify that common network benchmarks (iperf [29], iperf3 [30], netperf [27], sockperf [41], flowgrind [58, 59], ping [46]) perform correctly with TD. We also limit CPU time for containers with respect to TD. Using time dilation to emulate high-speed links is one of Etalon's main advantages.

4.4 Etalon Testbeds

We use two testbeds for our experiments: a CloudLab cluster emulating 8 racks (128 hosts) and a local cluster emulating 3 racks (48 hosts). We use the large CloudLab cluster to validate the Etalon emulator, and run our experiments on the small local cluster. For the contributions in this paper, we do not require a large cluster or complex workload.

The local testbed uses four servers to emulate three racks of 16 machines plus the hybrid switch, as described in Section 4.2 and Figure 5. Each physical machine has 2 × 20-core 2.8 GHz Intel Xeon E5-2680v2 and is connected to a 40 Gb/s Ethernet data network (with jumbo frames). We also use Etalon on the public CloudLab APT cluster [9, 14], where nine R320 machines emulate eight racks of 16 hosts each and the hybrid switch. Each APT machine has an 8-core 2.1 GHz Intel Xeon E5-2450 and is connected to a 40 Gb/s Ethernet data network (with jumbo frames). At the time of our experiments, the CloudLab APT cluster was configured for 56 Gb/s InfiniBand, so we manually reconfigured the switches into 40 Gb/s Ethernet mode.

We use a TDF of 20× across our experiments. For example, in our 3-rack cluster, we emulate a 3-port 10 Gb/s (0.5 Gb/s)[4] packet switch and a 3-port 80 Gb/s (4 Gb/s) circuit switch. Outside of TD, the network can produce 3×0.5 Gb/s $+ 3 \times 4$ Gb/s = 13.5 Gb/s of total traffic, far below our data network's 40 Gb/s physical link speed. Each per-container link (i.e., each intra-rack link between the ToR and a host) is limited to 10 Gb/s (0.5 Gb/s).

Packet switch up/down links have 5 μs (100 μs with TDF) of delay each. Prior work assumes that the circuit network consists of ToRs connected to a pod-level central circuit switch, thus requiring long fibres (in the case of an optical network) [16]. To model this, we conservatively configure the circuit delay as 30 μs (600 μs), or 3× higher than the total packet link delay ($5\ \mu s \times 2 = 10\ \mu s$ total). To avoid out-of-order packet delivery, if there is a circuit scheduled between racks S and D, then we disable the packet switch for rack pair (S, D) both during the reconfiguration leading up to a circuit and during the circuit period itself. Therefore, between S and D there exists exactly one connection: either packet or circuit. This is an example of non-segregated routing: Mice and elephant flows traverse the same links.

[4] Values in parenthesis represent bandwidth/delay outside of TD, i.e., actual bandwidth and delay values.

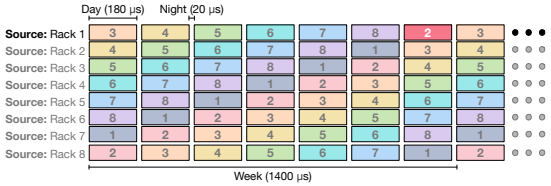


Figure 7: An 8-rack strobe schedule. All *(src rack, dst rack)* pairs can communicate 1/7th of the time.

4.5 Schedule and Workload

This paper focuses specifically on congestion control mechanisms, and therefore we believe that a simple traffic pattern is sufficient. With a couple of exceptions, we use the following schedule and workload for all experiments. Extending this investigation to complex workloads is future work.

Schedule We use a strobe schedule that, for a cluster with R racks, creates a circuit from a rack to each other rack $\frac{1}{R-1}$ of the time[5]. Figure 7 shows how, for an 8-rack schedule, a rack connects to each other rack in turn, repeating after a week. Solstice [39] would produce this schedule for an all-to-all workload such as a MapReduce shuffle [10]. For our experiments, a 25-rack cluster (described next) with 20 μs nights and 180 μs days (90% uptime) yields a week of duration $(20\ \mu s + 180\ \mu s) \times (25-1) = 4800\ \mu s$. The time between circuit days is $4800 - 180 = 4620\ \mu s$.

Workload The 16 emulated hosts on rack 1 each use flowgrind [58, 59] to send a TCP flow to their counterpart host on rack 2. The other racks are idle. We choose a flow duration of 3000 weeks, or 14.4 seconds for a 25-rack strobe schedule with 4800 μs per week. Each set of three weeks is treated as one experimental run, and the 1000 runs are averaged when reporting expected sequence number and VOQ length.

To better illustrate our contributions, we take advantage of our simple workload, described above, to emulate a larger testbed. Since our workload involves only racks 1 and 2, from the perspective of either of those racks, the cluster size effects only the duration between circuits. Therefore, we mimic a larger cluster by artificially lengthening this duration. We use this technique to run a 25-rack strobe schedule on our 3-rack local testbed.

4.6 Validating Etalon

We validate Etalon on the 8-rack CloudLab cluster using a strobe schedule while sending TCP traffic between pairs of racks for 2 seconds. ACKs are diverted around the switch for this one experiment to avoid ACK loss. By bypassing the hybrid switch, ACKs are transported instantly across the core of the emulated network, enticing TCP to ramp up faster, thus nullifying the bandwidth fluctuation described in Section 3. This is, of course, an unrealistic technique for actual networks, but we employ it to validate the Etalon emulator. We present timing and bandwidth results in Table 1. These results demonstrate that the emulator is sufficiently accurate to achieve the desired circuit schedule times (night and day lengths) and packet and circuit bandwidths.

[5] A rack need not connect to itself, so there are $R-1$ days in a week.

	Expected	Experimental Mean	Std. Dev.
Circuit day	180μs	180.25μs	0.04μs
Week length	1400μs	1400.02μs	0.05μs
Packet utilization	10 Gbps	9.93 Gbps	0.75 Gbps
Circuit utilization	80 Gbps	79.99 Gbps	1.60 Gbps

Table 1: Validating Etalon's timing and throughput.

5 Overcoming Rapid Bandwidth Fluctuation

As discussed in Section 3, the short uptimes (e.g., < 3 RTTs) of modern reconfigurable datacenter networks create bandwidth fluctuations such that TCP is unable to fully utilize the available bandwidth. This section describes two distinct techniques, implemented at different network layers, that adapt TCP to this challenge: 1) dynamic VOQ resizing that transparently prebuffers packets at the ToR before a circuit activates, and 2) explicit circuit state feedback to end-hosts that directly triggers a `cwnd` increase. Before introducing our techniques, however, we first consider a simpler, static-buffer approach that illustrates the bandwidth and latency trade-off that our solutions must navigate.

Section 3.2 presented the general signals that TCP variants use to infer network congestion. Increasing switch buffer sizes masks packet loses caused by queue overflows. For loss-based TCP variants like CUBIC and New Reno, removing this congestion signal triggers ramp-up. However, this technique is not appropriate for delay-based variants such as TIMELY or Vegas. We demonstrate in Section 6.2 that for variants which rely at least partially on loss as a congestion signal, hiding congestion-based drops is effective at increasing circuit utilization. Our second technique, explicit circuit state feedback via the `ECE-echo` bit, is more general since it provides a direct signal to end-hosts, but of course has the trade-off that it requires modifying end-hosts.

5.1 Leveraging VOQs to Increase Bandwidth

It is well understood that switch buffer sizing presents a trade-off between high bandwidth and low latency [5, 17]. The more traffic that can be queued up in the network, the better it will be able to saturate links in the presence of transient demand—or, in our case, capacity variations—because packets will be available to burst immediately in the event of a capacity increase. However, when packets incur queuing latency, the effective round-trip time increases. Because latency

(and tail latency in particular) impacts short flow—and potentially job—completion times, much work on datacenter transport protocols has focused on achieving high bandwidth while keeping queues short [1, 5]. As a starting point toward mitigating the effects of bandwidth fluctuation in RDCNs, we experiment with various sizes of static ToR VOQs and demonstrate that loading the network with excess traffic does help saturate the high-bandwidth circuits, but, as expected, at the cost of high latency.

Using the schedule and workload described in Section 4.5, we configure the hosts to run TCP CUBIC, vary the size of the ToR VOQs from 4 to 128 packets, and examine the impact on circuit utilization. Figure 8a. shows utilization measured as the aggregate achieved bandwidth of all of a rack's flows versus the maximum bandwidth the flows should have been able to achieve, averaged over all of the circuit periods in an experiment. For small buffers, circuit utilization is low. Large buffers fare better, with 64 packets building up a sufficient "backlog" of packets to absorb the bandwidth fluctuations.

Figure 8b shows the expected TCP sequence number during the lead-up to a circuit day, for various queue sizes, as measured by the software switch (Section 4.2). The slope of each line is a flow's achieved bandwidth. The *optimal* and *packet only* baselines are computed as in Figure 2. Larger buffers yield a steady convergence to *optimal*. While TCP grows at a rate of one packet per RTT, regardless of buffer size, larger switch buffers allow flows to queue up a packet backlog which then drains during circuit uptime, as shown by the dotted VOQ lines. The VOQ length is level throughout the packet network period.

Finding the "proper" VOQ size for a hybrid switch is difficult. Common wisdom is to use the bandwidth-delay product (BDP) of the network, but the BDP is different for the packet network and the circuit network: ~2 and ~34 packets, respectively. A time-weighted average based on the schedule suggests ~8 packets may be appropriate. As shown in Figure 8a, however, none of these values provide full circuit utilization: 64+ packets are needed[6]. However, we cannot simply adopt queues this large because of their high latency. In a datacenter, where link lengths are short, queuing delay impacts tail latency, which in turn directly impacts distributed applications [1]. Figure 8c shows 99th percentile tail latency for the various ToR VOQ sizes, measured as each packet enters and leaves the software switch. As expected, the packet and circuit latencies both grow as we increase the buffer size.

We want the best of both worlds: Can we achieve full circuit utilization while simultaneously not incurring a latency penalty? No static buffer configuration achieves this. The following subsections describe two techniques to meet this goal: *dynamic* buffer resizing (Section 5.2) and explicit circuit state feedback to end-hosts (Section 5.3).

[6]Our later experiments show that ~45 packets may suffice.

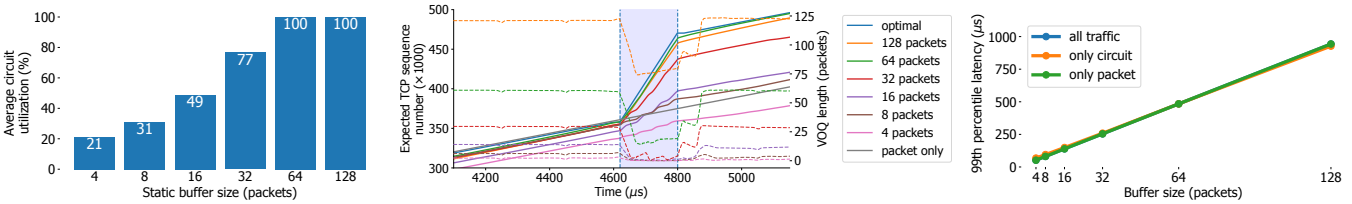
5.2 Dynamic Buffer Resizing

We propose dynamically resizing ToR VOQ capacity to remedy the effects of rapid bandwidth fluctuations on TCP. This is an entirely *in-network* solution, which, to our knowledge, has not been explored in the context of network scheduling. In this section, we focus specifically on loss-based variants of TCP, such as CUBIC.

Our key insight is that bandwidth fluctuation within RDCNs is not arbitrary: It is part of a schedule and is known in advance. With this knowledge, we can align in-network buffer sizes with either the packet switch or the circuit switch in real time. By itself, a packet switch can effectively achieve full throughput with a very small buffer (e.g., 4 packets), whereas large buffers cause queuing delay, as shown in Figure 8c. The previous section demonstrated that a circuit switch needs larger buffers (e.g., 64+ packets) to achieve full utilization due to bandwidth fluctuations. With dynamic buffer resizing, we take a step in the right direction by keeping buffers shallow when the packet switch is in use and deep when a circuit is active. Doing this naïvely (i.e., resize buffers when the circuit comes up) provides little benefit; there is simply not enough time in one day for TCP to grow to fill the circuit link, regardless of how large the buffer. Data needs to be available *immediately* at circuit start (either buffered or via a high TCP sending rate); ramping up *ex post facto* means that circuit time has already been wasted.

Instead, we dynamically resize ToR VOQs for a *(source, destination)* rack pair *in advance* of a circuit starting for that pair. This implies that if the circuit schedule dictates that this rack pair should spend most of the time using the packet switch, then small buffers will be used to avoid incurring additional latency. Increasing a rack pair's VOQ size provides two benefits: 1) Packets build up in the queue and are then drained immediately when the circuit activates, creating a momentary burst of traffic, and 2) When the buffered packets drain at circuit start, they generate a surge of ACKs that increase the sender's congestion window (`cwnd`) and sending rate. Exactly how quickly `cwnd` grows depends on the TCP variant in use.

Our buffer resize function has three parameters: `resize`(s, b, τ), where s and b are the small and large buffer sizes in packets, respectively, and τ is how early a buffer should be resized in advance of the circuit start. For the rest of the paper, we use $s = 16$ and $b = 50$. τ is a trade-off: Resizing too late means low circuit utilization, but resizing too early increases latency. We vary τ in experiments in Section 6.1. While the value of τ impacts the circuit utilization/latency trade-off, we find that waiting to resize the buffer back to s after a circuit stops (and thus retaining large buffers for a short time while the packet network comes back into use) has no benefit. Thus, we always reset buffers back to s immediately after circuit teardown.



(a) Circuit utilization vs. static buffer size.

(b) Expected TCP sequence number vs. time, for various static buffer sizes. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.

(c) 99th percentile tail latency vs. static buffer size.

Figure 8: Tradeoffs of various static VOQ sizes. Larger buffers improve utilization at the cost of latency.

5.3 Incorporating Explicit Network Feedback

Dynamically resizing ToR VOQs is a transparent, in-network technique that raises circuit utilization without harming latency and does not require end-host modifications. However, this approach is intended for loss-based congestion control schemes and does entail additional latency and buffering. In this section, we propose a direct form of circuit state feedback that applies to more TCP variants (e.g., delay and explicit-feedback–based schemes) and ultimately mitigates this latency penalty. Of course, the trade-off is that techniques involving explicit feedback require modifying the TCP congestion control algorithm running at the sender, making them non-transparent and more difficult to deploy.

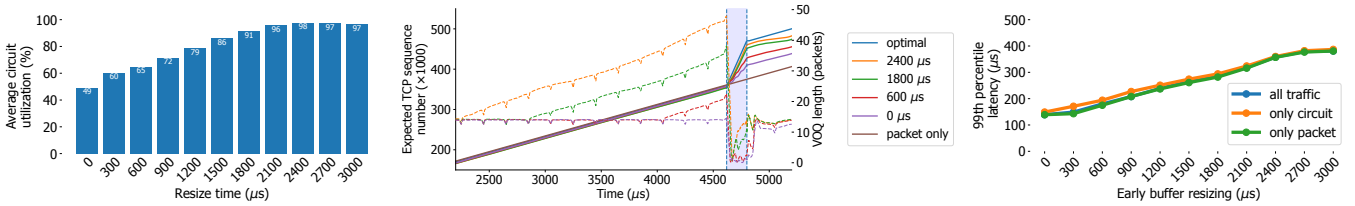
The idea behind explicit network feedback is simple: Notify the sender when a flow is traversing a circuit and should ramp up. However, the delay in this feedback reaching the sender is crucial. Since circuits are live for as few as 3 RTTs, a signal that requires 1 RTT to propagate to the sender (e.g., marking ECN bits of outgoing flows), provides limited benefit. We tighten the feedback loop by instead marking ACKs as they return to the sender. For some rack pair (S, D), we modify our software switch to set the `ECN-echo` (ECE) bit in the TCP headers of ACKs sent by D, if there is currently a circuit enabled from S to D. The hybrid switch examines the circuit state and marks ACKs *after* they traverse the circuit link, so the "freshness" of the feedback signal is equal to the propagation delay of a single packet link between the ToR and the sending end-host, S.

We create a pluggable TCP congestion control module for Linux called reTCP (REconfigurable datacenter network TCP) which looks at this stream of ECE bits, multiplicatively increasing its `cwnd` by $\alpha \geq 1$ on $0 \rightarrow 1$ transitions and decreasing it by $0 \leq \beta \leq 1$ on $1 \rightarrow 0$ transitions. reTCP is an edge detector: It modifies `cwnd` on ECE-bit state *transitions*, not for every packet. We set $\alpha = 2$ and $\beta = 0.5$, based on the results of a parameter sweep. Intuitively, this provides higher circuit utilization because TCP will immediately have a higher sending rate when a circuit starts.

Our reTCP implementation is based on TCP New Reno [23] and relies on its congestion control algorithm, in addition to the above-mentioned technique. reTCP also requires a single-line kernel change, as the kernel only passes ECE flags to congestion control modules if ECN is enabled. Enabling ECN shrinks `cwnd` upon receiving an ECE-marked packet (because ECE bits typically convey congestion information), so we leave ECN disabled and instead modify the kernel to always pass the ECE flag to reTCP.

reTCP is beneficial on its own by increasing the TCP sending rate when a circuit begins, but in combination with dynamic buffer resizing, its benefits multiply. As a starting point, consider the period when the packet network is active, operating with static VOQs (not dynamic buffer resizing). The sender ramps up to a steady state that saturates the packet network bandwidth and VOQ capacity. Suppose that we trigger a reTCP `cwnd` increase during this regime. The network would drop the additional packets and the sender would slow back down, with no benefit. However, dynamic buffer resizing is specifically designed to provide extra VOQ capacity to flows before circuit start, but the usefulness of this capacity and for how long it must be available is determined by the sender's ramp-up rate. The two techniques play to each other's strengths: Combining them, dynamic buffer resizing provides more capacity for flows, and then reTCP fills it. When used in this way, we modify reTCP to mark ACKs not at the start of a circuit, but at the start of the dynamic buffer resizing period. By increasing `cwnd` at the same time that the VOQs grow, reTCP quickly fills the larger capacity, dramatically reducing the duration of prebuffering required to achieve full utilization. Bringing the collaboration full-circle, reTCP's ramp-up burst reduces the prebuffering duration, which in turn lessens the time fraction during which the network experiences deep buffers, thus mitigating the ensuing tail latency spike. Section 6.3 visualizes how dynamic buffer resizing and reTCP jump-start the TCP sender just in time for the circuit network.

Instead of communicating circuit state on-demand via ACKs and their ECE bits, an alternative design is to inform end-hosts directly using control plane messages, similar to how the ToRs are notified to increase their VOQ capacity prior to circuit start. This would allow the sender to increase its `cwnd` exactly when desired, instead of after the first marked ACK arrives. However, this requires invasive modifications to the end-hosts, since the sender TCP stack would need to communicate with the central scheduler. Our approach is



(a) Circuit utilization vs. prebuffering duration.

(b) Expected TCP sequence number vs. time, for various prebuffering durations. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.

(c) 99th percentile latency vs. prebuffering duration.

Figure 9: Dynamic buffer resizing improves circuit utilization at the expense of tail latency.

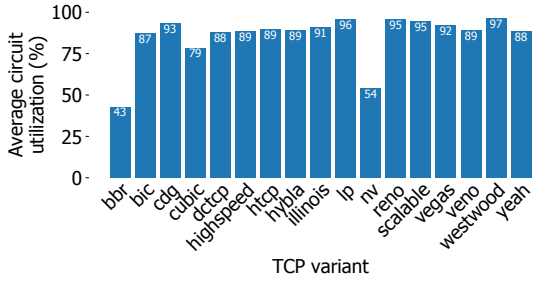


Figure 10: Circuit utilization vs. TCP variant.

less reactive but more practical because it uses existing TCP header fields, modifies only a few lines of code, and avoids distributed control challenges like time synchronization.

6 Evaluation

This section demonstrates that our two proposed techniques, dynamic buffer resizing and explicit circuit state feedback, overcome the challenge of bandwidth fluctuation in RDCNs and enable TCP to take advantage of high-bandwidth circuits when they become available. Additionally, we demonstrate that dynamic buffer resizing provides a general benefit to many TCP variants (beyond CUBIC).

6.1 Evaluating Dynamic Buffer Resizing

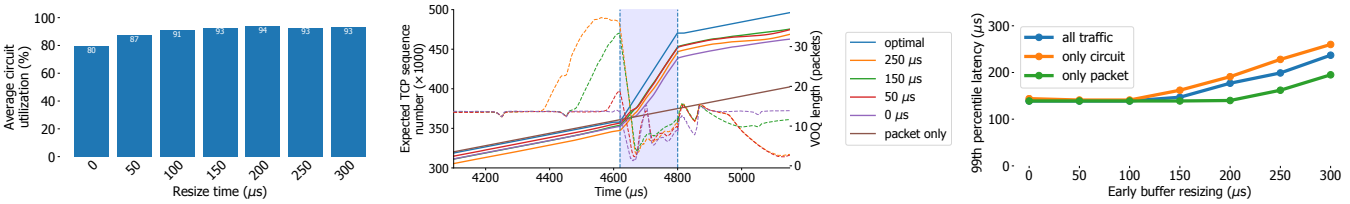
To test the efficacy of dynamic in-network buffer resizing, we repeat the experiments from Section 5.1, but with dynamic instead of static ToR VOQs. Using the schedule and workload described in Section 4.5, we configure the hosts to run TCP CUBIC and vary how early buffer resizing takes place, τ, from 0 μs to 3000 μs, in intervals of 300 μs. Buffers switch between a short capacity of 16 packets and a large capacity of 50 packets. Figure 9a shows average circuit utilization for the various τ. The earlier we resize, the higher utilization flows achieve. With $\tau = 1800$ μs, circuit utilization increases by 1.87× (48.6% → 91.1%), compared to 16-packet static queues.

Figure 9b shows a graph of the expected TCP sequence number and the VOQ length during the lead-up to a circuit day, for selected τ. VOQ length hovers at the small buffer size until dynamic buffering takes effect, then grows steadily, and finally drains sharply when the higher bandwidth circuit activates. For situations that see high utilization, sufficiently many ACKs return and ramp up the sending rate before the VOQs drain completely, which, for TCP CUBIC, is effectively achieved at $\tau = 1800$ μs.

The queue length required for high utilization is a function of the BDPs of the packet (10 Gb/s × 2 × 5 μs = 100 Kb) and circuit (80 Gb/s × 30 μs = 2.4 Mb) networks. These BDPs differ by ~32 9000 B packets. Assuming a few extra packets due to store-and-forward delays, between 35 and 40 packets must be in the VOQs before circuit start to keep the network fully utilized. The matches the growth of the VOQ line for $\tau = 1800$ μs (91% utilization) in Figure 9b.

The remaining question is whether this high utilization comes at the cost of high latency, as it did for static buffers. Median latency does not increase until $\tau = 2700$ μs; we omit the results for brevity. Figure 9c shows how dynamic resizing affects 99th percentile tail latency. Tail latency depends on the length of the VOQs, which increases with the prebuffering duration. The 1.87× circuit utilization increase is paired with a tail latency growth of 2.33× (123 μs → 286 μs). These results can be compared to static VOQs in two ways: (1) Comparing $\tau = 1800$ μs to a static buffer with similar throughput (64 packets), dynamic buffering improves tail latency by 0.59× (484 μs → 286 μs); (2) Comparing to a static buffer with similar tail latency (32 packets), the circuit utilization increases by 1.19× (76.7% → 91.1%). Ultimately, this experiment demonstrates that dynamic buffer resizing has the potential to meet our goal of achieving full circuit utilization with less of an impact on latency than large static buffers, but its latency penalty, especially at the tail, is still unreasonably high for distributed applications.

Given that resizing provides large buffers to flows for a significant amount of time (e.g., 41% of the schedule for $\tau = 1800$ μs) it is surprising that tail latency is not as bad as static buffers for comparable circuit utilization. We can use large buffers for a large fraction of time because when circuits are torn down and flows transition back to the packet network, the resulting bandwidth reduction (8× in these experiments) causes TCP to dramatically scale back its sending rate at



(a) Circuit utilization vs. prebuffering duration.

(b) Expected TCP sequence number vs. time, for various prebuffering durations. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.

(c) 99th percentile latency vs. prebuffering duration.

Figure 11: reTCP achieves high utilization with much lower tail latency than dynamic buffers alone.

the same moment when the VOQs shrink. In effect, switching back to the packet network resets TCP and the networks experiences a period of short queues while TCP recovers.

However, a 2.33× tail latency increase is unacceptable for many latency-sensitive applications. We have not yet met our goal of achieving the "best of both worlds" of high throughput without the corresponding latency penalty. Section 6.3 demonstrates how reTCP completes the picture, achieving high utilization with only a 1.20× tail latency increase.

6.2 Benefits for all TCP Variants

In Section 3.2.4, we demonstrated that the problem of bandwidth fluctuation impacts many TCP variants we tested. Returning to the experiment in Figure 4, we evaluate the variants' performance with dynamic buffer resizing, with $\tau = 1200$ μs. Figure 10 shows the utilization achieved by the 17 variants. The variants experience an average utilization improvement of 36%, compared to static 16-packet buffers, without any modifications to the end-hosts. Resizing earlier (e.g., $\tau = 1800$ μs) yields still-higher utilization, but we present $\tau = 1200$ μs to better illustrate how the TCP variants respond differently. BBR [5] and TCP NV [3], being delay-based protocols that seek to keep buffer occupancy low, naturally do not take advantage of dynamic buffer resizing and realize only 43% and 54% utilization, respectively. Note that DCTCP [1] relies on explicit congestion feedback from network switches in the form of a stream of ECN marks set if adding a packet to a switch queue would cause the queue's capacity to pass a threshold. To accurately support DCTCP in the Etalon emulator, we modify the hybrid switch to mark packets in this way, at 10 packets when using small queues (16 packets) and 31 packets when using large queues (50 packets).

6.3 Evaluating Explicit Network Feedback

Conveying circuit state information to end-hosts provides an explicit signal that TCP can use to adapt to bandwidth fluctuations in RDCNs. In isolation, reTCP yields higher average circuit utilization than static buffers alone, with an average improvement of 2.65% across the static buffer capacities in Figure 8a. An improvement of 6% with 32-packet buffers is the most significant. For 64 and 128–packet queues, both static buffers and reTCP achieve full utilization. We omit the graphs for brevity.

When used in combination with dynamic buffer resizing, reTCP achieves high circuit utilization with a shorter prebuffering duration and, in turn, lower tail latency. Figure 11 repeats the experiments from Section 6.1, but with both dynamic buffer resizing and reTCP, and instead varies τ from 0 μs to 300 μs, in intervals of 50 μs. Note that the x-axis range in Figure 11b is different than in Figure 9b to better visualize the range of τ values during which reTCP ramps up. Figure 11a shows that the two techniques working together achieve a 1.91× (48.7% → 92.7%) circuit utilization improvement compared to 16-packet static buffers, but with lower τ than dynamic buffers alone: at $\tau = 150$ μs instead of $\tau = 1800$ μs. The impact of this order-of-magnitude decrease in prebuffering duration manifests itself in lower tail latency in Figure 11c: an only 1.20× increase (123 μs → 147 μs), compared to 2.33× for dynamic buffer resizing alone.

Comparing Figures 9b and 11b, the steady VOQ growth is replaced by a jump as `cwnd` doubles, sending more packets into the network. At $\tau = 150$ μs, the sender injects sufficiently many packets to grow the VOQ large enough to saturate the higher BDP of the circuit network, when it becomes active. The rate of growth is quick: It keeps the required prebuffering duration short, which reduces tail latency. Overall, dynamic buffer resizing and reTCP overcome the challenge of bandwidth fluctuation in RDCNs, increasing circuit utilization by 1.91× with an only 1.20× tail latency penalty.

6.4 Limitations

Increasing the VOQ size does not, on its own, lead to higher circuit utilization. The TCP sender must ramp up to fill the additional capacity. Section 6.1 shows that this ramp-up may take milliseconds, posing a challenge for schedules that allocate circuits between some rack pairs frequently: The circuit downtime period may be too short to support this lengthy ramp-up. E.g., the time between circuits for a 3-rack cluster with a strobe schedule (week length = 400 μs) is 220 μs, far lower than the $\tau = 1800$ μs deemed necessary by Section 6.1. Investigating this case revealed that as the prebuffer start time

approaches the end of the previous circuit, the residual high sending rate quickly fills the VOQs, causing the network to enter a mode similar to using large static buffers, with high tail latency. reTCP ameliorates this problem.

7 Related Work

Research into RDCN design [6,16,20,25,26,32,38,47,51,57] and scheduling [2,36,39] has yet to examine TCP-related challenges for modern RDCNs. c-Through [51] proposes resizing end-host network buffers, but for the purpose of traffic batching, which is necessary to ensure that senders have enough data to fill a circuit when it becomes available. This is only an issue for circuits with long (e.g., millisecond-scale) uptimes, not the microsecond-scale technologies we address in this paper. Other work avoids the bandwidth fluctuation problem by segregating traffic to use either the packet or circuit networks exclusively [16, 38]. Our techniques simplify the network design by not segregating traffic.

Many of the TCP variants in Section 3.2 are tailored for high-bandwidth networks, but they assume that the bandwidth does not change on short timescales. Prior work [52] has examined how TCP reacts to bandwidth fluctuations in wireless networks, but these networks are fundamentally different than RDCNs. Wireless networks have lower bandwidths and BDPs than RDCNs, and unlike in wireless networks, bandwidth fluctuations in RDCNs are not random: They are part of a schedule that is known in advance. This insight enables proactive techniques like dynamic VOQ resizing.

8 Conclusion

With new advances in RDCN technology comes the need to reexamine the protocols running on top of these networks. This paper proposes two techniques to adapt TCP to the rapid bandwidth fluctuation inherent in microsecond-scale reconfigurable datacenter networks: 1) In the network, we transparently resize ToR VOQ buffers prior to circuit activation to help TCP ramp up, but at the cost of higher tail latency; 2) Involving the end-hosts opens the door for high utilization without a latency penalty by incorporating explicit circuit state signals sent from the hybrid switch. Etalon provides opportunities for future work, e.g., exploring multicast-enabled optical circuit switching (e.g., Blast [53]), providing a cross-cutting evaluation of different RDCN designs, and investigating challenges in future sub-μs RDCNs. While this paper focuses on TCP specifically, our investigation into implicit and explicit techniques for adapting the sending rate to the state of the network applies to congestion control in general. We believe that our experiences speak to the need for an end-to-end evaluation of future RDCN designs.

Acknowledgements We would like to thank the anonymous reviewers and our shepherd, Manya Ghobadi, for their helpful feedback. This work was supported in part by NSF grant CNS-1565343.

References

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

[2] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 44, pages 75–88. ACM, 2016.

[3] Lawrence Brakmo. Tcp-nv: Congestion avoidance for data centers. Linux Plumbers Conference 2010, 2010.

[4] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, October 1994.

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[6] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, and Xitao Wen. OSA: An Optical Switching Architecture for Data Center Networks and Unprecedented Flexibility. In *Proc. USENIX NSDI*, April 2012.

[7] Click. FromDPDKDevice element documentation. https://github.com/kohler/click/wiki/FromDPDKDevice, 2020.

[8] Click. ToDPDKDevice element documentation. https://github.com/kohler/click/wiki/ToDPDKDevice, 2020.

[9] CloudLab. CloudLab. https://www.cloudlab.us/, 2018.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Docker. Docker. https://www.docker.com/, 2018.

[12] Docker. Get started with Macvlan network driver. https://docs.docker.com/engine/userguide/networking/get-started-macvlan/, 2018.

[13] DPDK. DPDK. https://dpdk.org, 2018.

[14] Emulab. APT cluster. https://www.aptlab.net/, 2018.

[15] Etalon. Etalon: A reconfigurable datacenter network (rdcn) emulator. https://github.com/mukerjee/etalon, 2020.

[16] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, August 2010.

[17] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(04):397–413, jul 1993.

[18] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 https://rfc-editor.org/rfc/rfc3649.txt, December 2003.

[19] Klaus-Tycho Foerster, Manya Ghobadi, and Stefan Schmid. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 89–96, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 216–229. ACM, 2016.

[21] Diwaker Gupta, Kashi V. Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex C. Snoeren, and Geoffrey M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS)*, 29(2):4:1–4:48, May 2011.

[22] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.

[23] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 https://rfc-editor.org/rfc/rfc6582.txt, April 2012.

[24] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[25] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proc. ACM SIGCOMM*, August 2011.

[26] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 319–330, New York, NY, USA, 2014. ACM.

[27] HewlettPackard. netperf. https://github.com/HewlettPackard/netperf, 2018.

[28] IEEE. 802.1Qbb – Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/, 2018.

[29] iperf. iperf. https://iperf.fr/, 2018.

[30] iperf3. iperf3. https://iperf.fr/, 2018.

[31] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988.

[32] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways To De-Congest Data Center Networks. In *Proc. ACM HotNets-VIII*, October 2009.

[33] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 89–102, New York, NY, USA, 2002. Association for Computing Machinery.

[34] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[35] Jereme Lamps, David M Nicol, and Matthew Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 179–186. ACM, 2014.

[36] Conglong Li, Matthew K Mukerjee, David G Andersen, Srinivasan Seshan, Michael Kaminsky, George Porter, and Alex C Snoeren. Using indirect routing to recover from network traffic scheduling estimation error. In

Proceedings of the Symposium on Architectures for Networking and Communications Systems, pages 13–24. IEEE Press, 2017.

[37] libVT. libVT: user space virtual time library. https://github.com/mukerjee/libvt, 2020.

[38] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *Proc. USENIX NSDI*, April 2014.

[39] He Liu, Matthew K Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M Voelker, David G Andersen, Michael Kaminsky, et al. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 41. ACM, 2015.

[40] Shao Liu, Tamer Baundefinedar, and R. Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*, valuetools '06, page 55–es, New York, NY, USA, 2006. Association for Computing Machinery.

[41] Mellanox. sockperf. https://github.com/Mellanox/sockperf, 2018.

[42] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: a scalable, low-complexity, optical datacenter network. In *Proc. ACM SIGCOMM*, August 2017.

[43] William M Mellette, Alex C Snoeren, and George Porter. P-fattree: A multi-channel datacenter network topology. In *HotNets*, pages 78–84, 2016.

[44] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[45] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of ACM SIGCOMM 2015*, SIGCOMM '15. ACM, 2015.

[46] ping. ping(8) - linux man page. https://linux.die.net/man/8/ping, 2020.

[47] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *Proc. ACM SIGCOMM*, August 2013.

[48] RPyC. RPyC - transparent, symmetric distributed computing. https://rpyc.readthedocs.io/en/latest/, 2020.

[49] tc. tc(8) - linux man page. https://linux.die.net/man/8/tc, 2020.

[50] Kashi Venkatesh Vishwanath, Diwaker Gupta, Amin Vahdat, and Ken Yocum. Modelnet: Towards a datacenter emulation environment. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 81–82. IEEE, 2009.

[51] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time Optics in Data Centers. In *Proc. ACM SIGCOMM*, August 2010.

[52] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, NSDI '13. USENIX, 2013.

[53] Yiting Xia, TS Eugene Ng, and Xiaoye Steven Sun. Blast: Accelerating high-performance data analytics applications by optical multicast. In *Proc. IEEE INFOCOM*, pages 1930–1938. IEEE, 2015.

[54] Lisong Xu, K. Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2514 – 2524 vol.4. IEEE, 04 2004.

[55] Jiaqi Yan and Dong Jin. A virtual time system for linux-container-based emulation of software-defined networks. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 235–246. ACM, 2015.

[56] Jiaqi Yan and Dong Jin. Vt-mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 27. ACM, 2015.

[57] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *Proc. ACM SIGCOMM*, August 2012.

[58] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. Flowgrind-a new performance measurement tool. In *IEEE GLOBECOM 2010*, pages 1–6. IEEE, 2010.

[59] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. flowgrind. http://www.flowgrind.net/, 2018.