Overcoming End-to-End Challenges in Reconfigurable Datacenter Networks

Matthew K. Mukerjee, Daehyeok Kim, Srinivasan Seshan
Carnegie Mellon University

Abstract

Increasing pressure for higher throughput, better response times, and lower cost in datacenters have lead to proposals augmenting traditional packet networks with very high bandwidth reconfigurable circuits. These proposals have focused on switch implementation or scheduling, not end-to-end challenges. In this work, we identify three key challenges: 1) rapid bandwidth fluctuation, 2) poor demand estimation, and 3) difficult-to-schedule workloads. Bandwidth fluctuation requires TCP to immediately jump in sending rate for <10 RTTs; poor demand estimation from shallow ToR queues leads to inefficient circuit schedules; and finally, certain application workloads are fundamentally difficult to schedule. To overcome these challenges, we build an open-source reconfigurable datacenter network emulator, Etalon, for public testbeds. We find solutions at different layers: we combat 1) bandwidth fluctuation with dynamic in-network queue resizing to ramp up TCP earlier, 2) poor demand estimation by communicating endhost stack buffer occupancy, leading to more accurate schedules, and 3) difficult-to-schedule workloads by rewriting application logic (e.g., HDFS' write placement algorithm). We conclude that cross-layer optimization is necessary and provides the most benefit at higher layers.

1 Introduction

Modern datacenter (DC) applications have staggering compute and storage requirements, leading to increased pressure for high bandwidth, low latency, high port count, and low cost networks to connect them. Traditional packet switches are hitting CMOS manufacturing limits, unable to simultaneously provide high bandwidth and port counts [29]. Thus, researchers have proposed augmenting DCs with reconfigurable circuit switches (e.g., optical, wireless) that provide high bandwidth between racks on demand [4, 11, 12, 15, 16, 22, 26, 31, 33, 38]. These *reconfigurable DC networks* (*RDCNs*; hybrid circuit + packet networks), however, are less flexible than traditional networks, as adding/removing bandwidth has a non-trivial reconfiguration penalty during which the circuit switch is unavailable.

Prior work has generally focused on two thrusts: switch implementation [11, 12, 15, 16, 22, 26, 31, 33, 38], or scheduling [3, 25, 27]. While important, little focus has been on end-to-end challenges faced by real applications and network stacks. While some end-to-end problems on switches with millisecond-scale reconfiguration have been explored [11, 33], modern μs-scale switches [12, 26, 31] have changed the nature of these problems, as well as the solutions needed.

We identify three such challenges in modern RDCNs:

1. **Rapid bandwidth fluctuation:** while circuits need to be enabled for a long period of time relative to the reconfiguration penalty, circuit uptime may be only a few (e.g., <10) round-trip times (RTTs), causing rapid bandwidth fluctuation. TCP's additive increase is too slow to utilize the large (e.g., 10×) temporary bandwidth increase, leading to low circuit utilization (§4).
2. **Poor demand estimation:** RDCN schedulers require accurate traffic estimation to produce efficient schedules. Prior work suggests Top-of-Rack (ToR) switch queues as a possible source of estimates [11, 25, 31], but we find that these must be shallow to provide low latency, meaning most demand is hidden from the scheduler on endhosts. This makes it difficult to differentiate large and small flows, leading to idle schedules (§5).
3. **Difficult-to-schedule workloads:** certain workloads are fundamentally difficult to schedule on RDCNs efficiently. Workloads with large, all-to-all flows (i.e., lacking skew and sparsity) waste time repeatedly reconfiguring the network (§6).

These challenges arise from broken assumptions made by endhosts about the network: TCP assumes bandwidth does not predictably fluctuate at short timescales, the network stack assumes the network doesn't perform better if it can see demand in advance, and applications assume that all traffic patterns are equally easy to deliver. Either all layers need additional specialization for this new network (cross-layer optimization), or interfaces need to be changed to accommodate different behaviors or expose more info. As one entity controls all endhosts / networks in the DC, cross-layer optimization is the easiest solution.

We use cross-layer optimization to overcome these challenges at the lowest layer possible, providing transparency, less deployment pain, and keeping higher layers general:

1. **Overcoming rapid bandwidth fluctuation with *dynamic in-network buffer resizing*:** Increasing ToR queue sizes in advance of a circuit start gives TCP time to ramp up its sending rate and fill the circuit as soon as it gets it (§4). *Cross-layer: [network understands TCP behavior]*
2. **Overcoming poor demand estimation with *endhost ADUs*:** An interposition library reports data waiting

on endhosts to the scheduler by tracking the sizes of `write()`s / `send()`s in unmodified applications. The scheduler uses these sizes (though not the boundaries) to decide which flows will benefit most from transiting the circuit switch (§5). *Cross-layer: [network understands app behavior]*

3. **Overcoming difficult-to-schedule workloads by *rewriting application logic*:** Modifying applications to introduce skew and sparsity into workloads results in schedules that require less reconfiguration. We demonstrate this with HDFS' write placement algorithm (§6). *Cross-layer: [apps understand network behavior]*

We argue that these solutions operate at the lowest layer possible; dynamic buffer resizing *in-network* is enough to insulate TCP from bandwidth fluctuation, exposing demand (ADUs) in the *endhost stack* is the only way to provide proper estimates without greatly increasing switch queuing delay, and changing *application* behavior is the only way to fundamentally change the workload.

To evaluate the efficacy of these solutions, we design and implement an open-source RDCN emulator, *Etalon*[1], for use on public testbeds (§3). Tests on CloudLab's 40Gbps APT DC cluster [5, 10] show: 1) dynamic buffer resizing can reduce median packet latency by ~36%, 2) ADUs can reduce median flow-completion time by 8× for large flows, and 3) modifying HDFS can reduce tail write times by 9×. We conclude that while cross-layer optimizations involving higher layers are harder to implement (e.g., modifying individual applications), they may provide the greatest benefit.

To summarize, we make three contributions in this work:

1. We analyze three critical end-to-end challenges in RDCNs (rapid bandwidth fluctuation, poor demand estimation, and difficult-to-schedule workloads) caused by erroneous assumptions between layers.
2. We design solutions that require modifications at varying layers (in-network, network stack, application).
3. We design and implement an emulation platform, Etalon, for evaluating RDCNs end-to-end with real applications, finding that solutions at higher layers (while more painful to implement) lead to more benefit.

2 Setting

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 [11, 26, 31, 33] as an illustrative example for the rest of the paper, the results generalize to other reconfigurable technologies (e.g., free-space optics [12, 16], 60GHz wireless [15, 22, 38]). We eschew older millisecond-scale reconfigurable switches [11, 33] for modern μs-scale switches [12, 26, 31], as the nature of end-to-end challenges and solutions differ with timescale.

2.1 Network Model

We consider an RDCN of N racks of M servers, each containing a Top-of-Rack (ToR) switch (Figure 1(a)). ToRs connect racks to a packet network (one or more switches) and a single circuit switch. The packet switches are low bandwidth (e.g., 10Gbps), but can make forwarding decisions for individual packets. The circuit switch is high bandwidth (e.g., 100Gbps), but makes forwarding decisions (i.e., sets up circuits) at much longer timescales to amortize a *reconfiguration penalty*.

We make the pessimistic assumption that during circuit reconfiguration no circuit links can be used, following prior work [26, 27, 31], allowing us to apply our results to a larger set of technologies. The packet switch, however, can be used at all times. Both switches source packets from $N \times N$ virtual output queues (VOQs) on the ToRs. The circuit switch is queue-less; it functions as a crossbar, only allowing configurations that form perfect matchings [3, 11, 26, 27, 31, 33] (i.e., a given sender is connected to exactly one receiver and vice-versa). Thus, at any point in time, the circuit switch may at most drain one VOQ on each ToR, whereas the packet switch may drain multiple VOQs.

2.2 Computing Schedules

Network scheduling in RDCNs is mapping rack-level demand to a set of circuit configurations (circuit switch port-matchings) with corresponding time durations. Any "left-over" demand is handled by the low-bandwidth packet switch (see Figure 1(b)). Borrowing terminology from prior work [31], we refer to a set of circuit configurations as a *week* of one or more variable-length *days* (individual circuit configurations), each followed by a *night* (down time from reconfiguration). Nights are generally 10-30μs [12, 26, 27, 31]. To allow for 90% link utilization, the average day length must be ≥9× the night length (e.g., 90-270μs). Weeks sufficiently long to amortize schedule computation.

Scheduling is a three-step loop: 1) demand for the next week (some fixed length e.g., 2ms) is estimated (e.g., through ToR VOQ occupancy), 2) an algorithm computes the schedule for the next week, and 3) the schedule is disseminated to the circuit switch. Scheduling algorithms for RDCNs (e.g., Solstice [27] and Eclipse [3]) use skew and sparsity in demand to minimize the number of circuit configurations.

2.3 Schedule Execution

Once a schedule is disseminated to the circuit switch, it runs the circuit configurations for their respective durations (see Figure 1(c)). After reconfiguration, a flow may transition from packet to circuit and vice-versa, but time spent on the

[1] After an optical filter used for solar observation. Source code: forthcoming

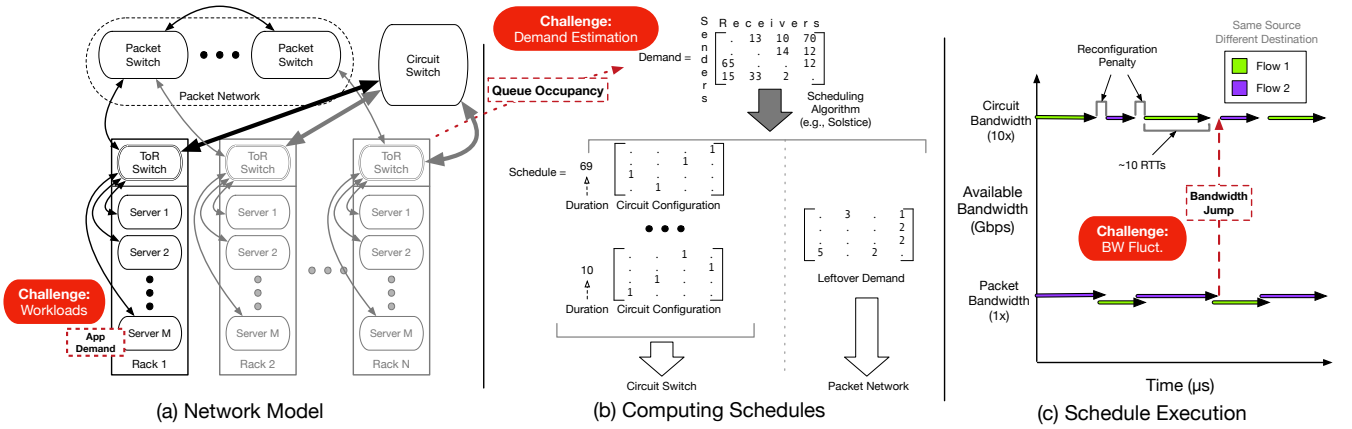


Figure 1: Overview of RDCNs. N racks of M servers connect to a low bandwidth packet network and a high bandwidth circuit switch. Application demand is sent through ToR switches to circuit or packet switches. This network requires scheduling, decomposing demand (e.g., ToR queue occupancy) into explicit circuit schedules and "leftovers" for the packet network. During schedule execution, circuit changes incur a reconfiguration penalty, which downs the circuit switch. Flows must use very brief bandwidth jumps. This model leads to three challenges.

circuit switch is likely only a few RTTs (e.g., <10). Flows must cope with these large (e.g., 10×) bandwidth variations.

2.4 Challenges

Three end-to-end challenges arise naturally (Figure 1):

1. **Rapid bandwidth fluctuation:** can TCP use a 10× increase in bandwidth in so few RTTs?
2. **Poor demand estimation:** efficient schedules require accurate demand estimates. Is ToR queue occupancy enough for scheduling a week (e.g., 2ms)?
3. **Difficult-to-schedule workloads:** schedulers require skew and sparsity for efficiency. Do all DC applications have these characteristics?

3 Etalon

In this section, we present our open-source emulator, Etalon, which measures end-to-end performance of *real applications and endhost stacks* on emulated RDCNs in public testbeds.

3.1 Overview

Figure 2 presents an overview of Etalon. Each of the N physical machines emulates a rack of M servers using Docker containers [7, 30]. Containers are connected to the physical NIC using macvlan [8]. Macvlan virtualizes a NIC into multiple vNICs, connecting them with a lightweight layer-2 software switch. `tc` limits link bandwidths between the containers and the vswitch, emulating a server-to-ToR link. A separate physical machine serves as the emulated hybrid switch, running a software switch (Click [23]) using DPDK [9] to process packets at line rate. ToR VOQs are emulated in the software switch

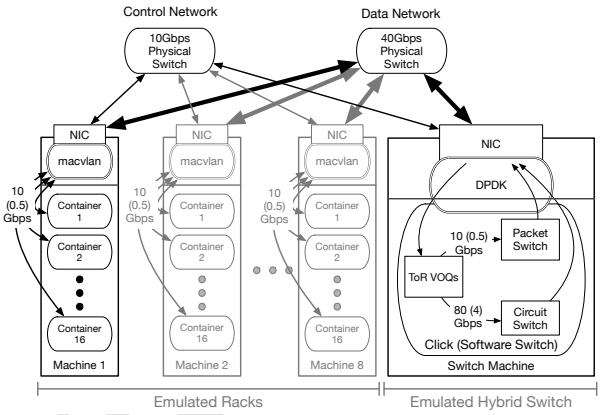


Figure 2: Overview of Etalon emulating 8 racks of 16 servers. Another machine emulates ToR VOQs, circuit & packet switches. Time dilation provides faster links.

for convenience, making circuit and packet link emulation straightforward. We maintain a separate control network for convenience, although it is not strictly necessary.

3.2 Software Switch

Figure 3 shows the software switch's internals. Packets enter the switch via DPDK [9] and are sent to a ToR VOQ based on their *(source, destination)* rack pair. Packets are pulled from each VOQ by the packet switch or circuit switch.

Packet up link 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 up link enters the packet switch, where it is multiplexed over a packet down link and exits

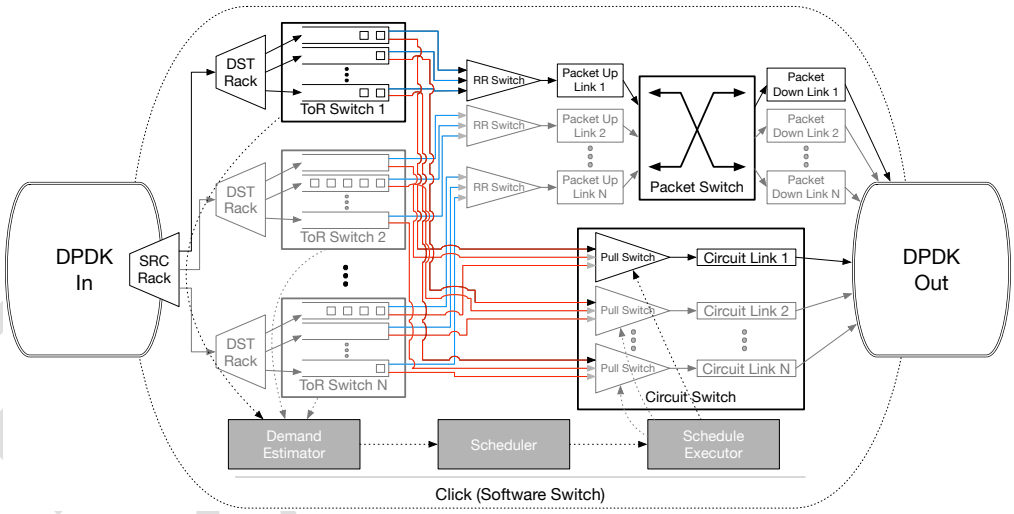


Figure 3: We emulate a circuit and packet switch using Click with DPDK. ToR VOQs are pulled round-robin by the packet switch. Circuit links pull from ToR VOQ based on current pull switch settings. The circuit schedule is computed using ToR VOQ occupancy and is executed by adjusting pull switch inputs.

using DPDK. If a packet would be dropped in the packet switch, it is held at the ToR VOQ (similar to PFC [19]).

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 transit the circuit link, they exit using DPDK.

Our software switch contains three control elements (shown in gray): demand estimator, scheduler, and schedule executor. Demand estimator estimates rack-to-rack demand using ToR VOQ occupancy. The scheduler computes a schedule from this demand, which is run by the schedule executor by modifying the circuit link pull switches' "input" value. Our scheduler element is pluggable; we implement Solstice [27] as an example, but modify its objective to schedule maximal demand within a set window W (like Eclipse [3]), rather than scheduling all demand in unbounded time.

3.3 Time Dilation

As the goal of Etalon is to emulate RDCNs on public testbeds, the software switch machine likely only has one high-speed NIC, yet we wish to emulate a switch with N high-speed ports. We solve this with *time dilation (TD)*. Originally proposed for VMs [13, 14, 32] (and recently containers [24, 36, 37]), TD provides accurate emulation of higher bandwidth links by "slowing down" the rest of the machine. We implement an open-source TD interposition library called LibVirtualTime (LibVT) which applies TD to many common syscalls in unmodified applications. We catch: `clock_gettime()`, `gettimeofday()`, `sleep()`, `usleep()`, `alarm()`, `select()`, `poll()`, and `setitimer()`. Extending LibVT to other calls is trivial. We verify that common network benchmarks (iperf [20], iperf3 [21], netperf [17], sockperf [28], flowgrind [39, 40], ping) perform correctly with TD. We additionally limit CPU time for containers with respect to TD.

3.4 Testbed

All experiments are performed using Etalon on the public APT [10] DC cluster (CloudLab [5]), with 8 "racks" of 16 "hosts" (Figure 2). Each machine has an 8-core 2.1GHz Xeon and 16GB of RAM. Each machine is connected to a 40Gbps data network (with jumbo frames) and a 10Gbps control network, and use TCP Reno.

We use a time dilation factor (TDF) of 20× to emulate an 8-port 10Gbps (0.5Gbps)[2] packet switch and an 8-port 80Gbps (4Gbps) circuit switch. Outside of TD, 0.5Gbps * 8 + 4Gbps * 8 = 36Gbps total traffic, below our 40Gbps physical link speed. Each per-container link is limited to 10Gbps (0.5Gbps). Packet switch up/down links have 5μs (100μs) delay each. The circuit links have 30μs (600μs) delay. While this delay is 3× higher than the packet links, prior work assumes a single circuit switch, requiring long fibers. We perform circuit link delay sensitivity analysis in §4.5.

We emulate a circuit switch with reconfiguration penalty (night length) 20μs (400μs) and run the Solstice [27] scheduler over W = 2ms windows (week length). To avoid out-of-order delivery, we disable the packet switch when the circuit is available for a rack pair. For the rest of the paper all timing values presented will be time dilated.

[2]values in parenthesis represent bandwidth/delay outside TD

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

Table 1: Validating Etalon's timing and throughput.

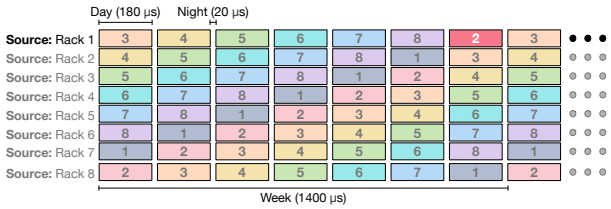


Figure 4: Strobe schedule used in §4 experiments. All *(source, dest.)* pairs of racks can communicate 1/7th of the time. Traffic is generated from rack 1 to rack 2.

3.5 Validation

We validate Etalon using a strobe schedule (Figure 4) of seven days of minimal length (9 * night length, 180μs), 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. We validate timing and bandwidth in Table 1.

4 Overcoming rapid bandwidth fluctuation with dynamic buffer resizing

While circuits need to be enabled for long periods of time relative to the reconfiguration penalty, this may be only a few (e.g., 3) RTTs, causing rapid bandwidth fluctuation. TCP's additive increase requires a longer timescale to make use of the extra (e.g., 8×) bandwidth. While TCP's interaction with RDCNs has been explored previously, it was on millisecond-scale reconfigurable switches (with thousands of RTTs per circuit configuration) [33], or required kernel/NIC modifications to pause/unpause flows according to the schedule [26, 31]. We explore an entirely *in-network* solution, dynamic buffer resizing, which to our knowledge has not been explored in the context of network scheduling.

4.1 Understanding the problem

Modern reconfigurable switches can only be efficient if they make forwarding decisions for hundreds of packets (for 80Gbps links / 9000 byte packets / reconfigured every 180μs) at a time, due to a reconfiguration penalty (e.g., 20μs). To avoid generating schedules with too many configurations (thus many reconfigurations), schedulers like Solstice [27] have a minimum day length parameter (180μs; 9 * night length in Etalon). Thus, in our worst case (minimal length circuit configuration), TCP flows are expected to immediately burst packets at 8× their rate for 3 RTTs when a circuit starts. This is difficult for TCP Reno's one-packet-per-RTT additive increase, as we will show.

We illustrate the problem in an experiment. Using Etalon, we fix a strobe schedule of 7 days, covering all *(source, destination)* rack pairs (except where the source and destination are the same rack), shown in Figure 4. We fix the day length to the minimum, 180μs, thus, our schedule's week repeats every 1400μs. Solstice [27] would produce this schedule for an all-to-all workload (e.g., MapReduce's shuffle [6]). We generate 2-second-long TCP flows using flowgrind [39, 40] from rack 1 to rack 2. A fixed schedule and small flow count is illustrative, but limited. We relax these assumptions in §5.

Figure 5a shows three weeks versus the expected next TCP sequence number. We vary static ToR VOQ sizes from 4 to 128 packets, and plot optimal based link bandwidths. Circuit uptime is shaded. The slope of each line is the flow's bandwidth; the area under the curve is data transmitted. As optimal shows, we expect an 8× bigger slope during circuit uptime. Circuit utilization is how well the slope matches optimal's slope during uptimes.

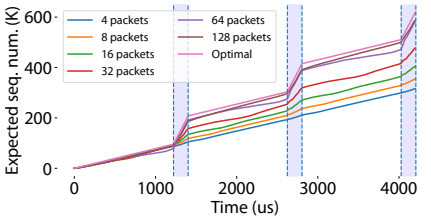
For small buffers, circuit utilization is low, while large buffers fare better. Figure 6a illustrates this concisely, showing circuit utilization directly. While TCP grows at one-packet-per-RTT regardless of buffer size, larger switch buffers, allow flows to have a "backlog" of packets queued up, draining during circuit uptime.

Finding the "proper" buffer size for a hybrid switch is difficult; common wisdom is to use the bandwidth-delay product (BDP), but the BDP is different for each switch (~4 packets and ~32 packets). Using a weighted sum based on the schedule gives ~8 packets. As seen, none of these values provide maximum circuit utilization; 64+ packets are needed. Queues this large have high latency. Figure 7a shows median latency, measured per-packet entering and leaving the software switch. We see that the packet switch latency grows sharply (as we already past the BDP for the packet switch), becoming high compared to latency over the circuit switch.

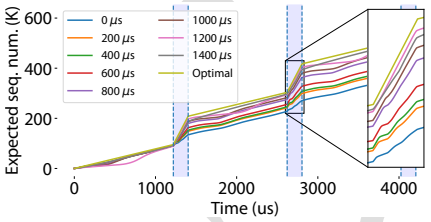
We want the best of both worlds: can we minimize latency and simultaneously maximizing our circuit utilization? No static buffer setting achieves this. We propose an entirely in-network solution to overcome this, dynamic buffer resizing.

4.2 Dynamic buffer resizing

We propose dynamically resizing ToR VOQ capacity to fix the effects of rapid bandwidth fluctuation on TCP. The key insight is bandwidth fluctuation within RDCNs is not arbitrary; it is part of a schedule and is known in advance.

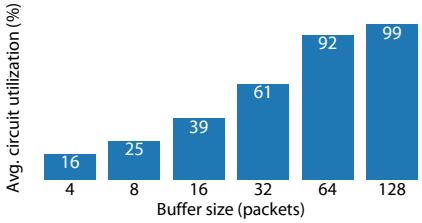


(a) Static buffer sizing

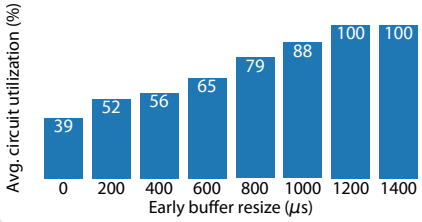


(b) Dynamic buffer resizing

Figure 5: Sequence plots for (a) buffer sizes / (b) how early buffers are resized. Circuit use is shaded.

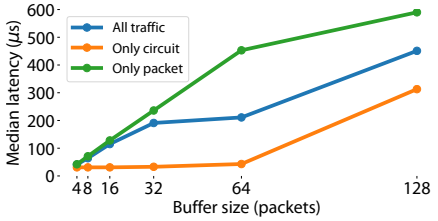


(a) Static buffer sizing

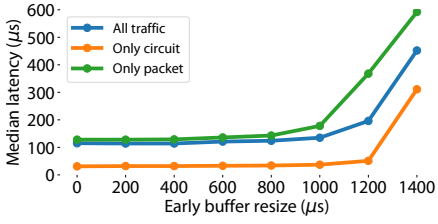


(b) Dynamic buffer resizing

Figure 6: Circuit utilization as a function of (a) buffer size / (b) how early buffers are resized.



(a) Static buffer sizing



(b) Dynamic buffer resizing

Figure 7: Median latency for (a) buffer size / (b) how early buffers are resized, split per switch.

With this knowledge, we can align buffer sizes with the packet switch or circuit switch in real time. By itself, a packet switch can effectively achieve full throughput with a very small buffer (e.g., 4 packets), but large buffers cause queuing delay (Figure 7a). And, by itself, a circuit switch needs larger buffers (e.g., 32+) to achieve full utilization, but incurs no queuing delay up to that point.

With dynamic buffer resizing we try to get the "best of both worlds" by keeping buffers small when the packet switch is in use and big when the circuit is in use. Doing this naively (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 you make the buffer. Data needs to be available *immediately* at circuit start (either buffered or via a high TCP send rate); ramping up post facto means circuit time is already wasted.

Instead, we dynamically resize ToR VOQs for a *(source, destination)* rack pair *in advance* of a circuit starting for that pair. In scenarios where this pair spends most time using the packet switch, small buffers are used to avoid additional latency. In advance of getting a circuit, this pair's VOQ size increases, providing time to 1) queue up packets, and 2) ramp up TCP flows, leading to better circuit utilization.

Our buffer resize function has three parameters:

$$\texttt{resize}(s, b, \tau)$$

where s and b are the small and large buffer sizes in packets, 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 $b = 128$ to slightly favor throughput, although (4, 64) would be reasonable to slightly favor latency in Etalon. τ is a tradeoff; resizing too late means low circuit utilization, but resizing too early increases latency. We vary τ in experiments below. While the value of τ impacts the circuit utilization/latency tradeoff, we find that waiting to resize the buffer back to s after a circuit stops has no benefit, thus we always set buffers back to s immediately after circuit teardown.

4.3 Experiments

To test the efficacy of dynamic in-network buffer resizing, we repeat the experiments from §4.1 (strobe schedule, TCP flows between racks 1 and 2), but with resizing.

We vary τ from 0 to 1400μs at intervals of 200μs. These values correspond to the length of a day + night in our schedule (Figure 4). Thus, at $\tau = 0\mu$s buffers are always size $s = 16$, and at $\tau = 1400\mu$s buffers are always sized $b = 128$.

Figure 5b shows a sequence plot for the next expected TCP sequence number for the various τ and a calculated optimal based on link rates. Circuit uptimes are shaded. The earlier we resize, the higher bandwidth obtained, as expected.

Recall that the benefits of resizing come from 1) packet buildup in queues and 2) TCP ramp up. Both are illustrated in the inset in Figure 5b: when the circuit comes up, there is an initial region with high slope as built-up packets drain from the queue. Then there is a plateau as TCP waits for ACKs from these packets before sending more data (recall we disable the packet switch a night in advance of the circuit start to avoid out-of-order packets). The higher the slope after the plateau, the more TCP ramped up before the circuit start. We see both an increase in the length of the initial burst (how many packets built up) and the following slope

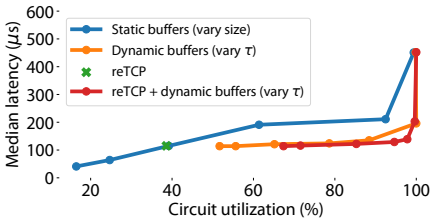


Figure 8: Comparing throughput to median latency for various configurations.

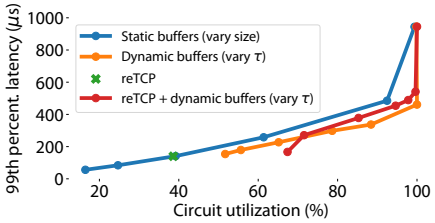


Figure 9: Comparing throughput to 99th percentile latency for various configurations.

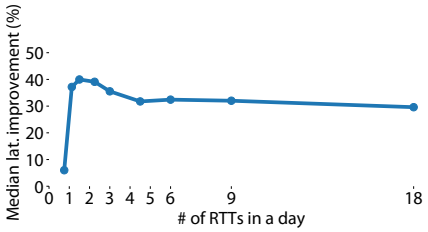


Figure 10: Improvement in latency for same circuit util. for dynamic v. static, varying RTTs in day.

as we move to earlier resize times. Achieving line rate requires the initial queue-draining burst to be long enough to overcome the length of the plateau, and then TCP's sending rate equaling the link rate, effectively achieved by $\tau = 1400$.

Figure 6b show circuit utilization for the various τ. As expected from the sequence plot, circuit utilization increases rather dramatically when compared to a static buffer size of 16, eventually achieving full utilization for large τ.

The remaining question is whether this utilization comes at the cost of high latency, as it did for static buffers. Figure 7b shows how dynamic resizing affects median latency (we show 99th percentile latency in §4.4). Surprisingly, median latency is relatively flat until $\tau > 1000\mu$s. More concisely, 5/7ths of a week (1000 / 1400μs) can be spent with large buffers with negligible impact on median latency (even for packets sent over the packet switch), while doubling circuit utilization. **Takeaway:** comparing the results for $\tau = 1000\mu$s to a static buffer with similar throughput (64 packets), **dynamic buffering improves median latency by ~36%**, or for the same latency as a static buffer with 16 packets **increases circuit utilization by ~2×**.

4.4 Incorporating explicit network feedback

Dynamically resizing in-network buffers raised circuit utilization without raising latency (for $\tau \leq 1000\mu$s) and did not require endhost modification. Next we remove that limitation to see if incorporating explicit network feedback about circuit state into TCP offers additional improvement.

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.

We create a pluggable TCP congestion control module (reconfigurable DC network TCP: reTCP) which looks at this stream of ECE bits, multiplicatively increasing it's cwnd by $\alpha \geq 1$ on $0 \rightarrow 1$ transitions and decreasing by $0 \leq \beta \leq 1$ on $1 \rightarrow 0$ transitions. We set $\alpha = 2$ and $\beta = 0.5$ empirically. Intuitively, this provides higher circuit utilization; TCP will immediately have higher sending rate on circuit start.

reTCP additionally requires a single line kernel change, as the kernel only passes ECE flags to congestion control modules if ECN is enabled on the system. Enabling ECN lowers cwnd upon receiving an ECE marked packet, so we modify the kernel to pass the ECE flag when ECN is disabled.

Results: Figure 8 shows the tradeoff of circuit utilization versus median latency for various static buffer sizes, various τ values (from above) for dynamic buffer resizing, reTCP, and reTCP + dynamic buffers with various τ values. These mechanism are beneficial when they are below and to the right of the "static buffer" curve.

Dynamic buffer resizing is an improvement over static buffers, but interestingly reTCP does not help unless paired with dynamic buffer resizing. Buffer resizing helps because it provides both a backlog of queued packets at circuit start and a higher sending rate for TCP. reTCP without resizing just provides the latter. Combining them helps, resulting in a higher TCP sending rate than dynamic buffers alone. This allows reTCP + dynamic buffers to eek out slightly more circuit utilization for the same τ values.

Figure 9 shows the same results for 99th percentile latency. Given that resizing provides large buffers to flows for a significant amount of time (e.g., 5/7ths of the schedule for $\tau = 1000\mu$s) it is surprising that tail latency is not worse than static buffers for comparable circuit utilization. Since some portion of time still uses small buffers (e.g., 2/7ths) even for large τ, this may be enough to keep low tail latency. Curiously, reTCP + dynamic buffers mostly does worse than just dynamic buffering in terms of tail latency. This is likely due to the fact that the jumps in cwnd caused by reTCP can sometimes occur when cwnd is already very large or very small leading to more variability, increasing tail latency.

4.5 Delay sensitivity analysis

Finally, we analyze how circuit link propagation delay affects the usefulness of dynamic buffer resizing. We use 30μs in previous experiments (§3). Varying link delay with fixed day length changes how many RTTs are in a day. In our experiments so far, 30μs delay and 180μs days gives 3 RTTs

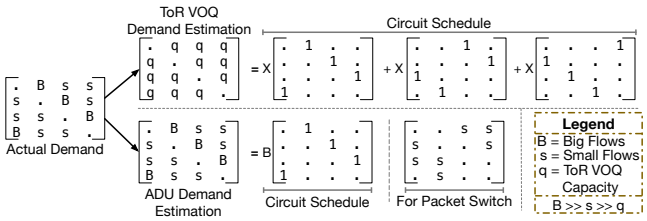


Figure 11: The problem of poor demand estimation: big and small flows can't be differentiated in shallow ToR VOQs, leading to long schedules with much idle time. Communicating endhost ADUs solves this.

per day. How does changing link delay (and therefore # RTTs per day) affect the improvements in median latency (for the same circuit utilization) that we saw for dynamic buffer resizing (with $\tau = 1000\mu$s) compared to static buffers?

Figure 10 shows the improvement in median latency as a function of RTTs per day. We see that increasing the number of RTTs (decreasing link delay) makes early resizing is less important, as there is now more time for TCP to ramp up after the circuit starts. Decreasing the number of RTTs (increasing link delay) makes early buffer resizing more important, as there is less chance to grow TCP during the day. Once days become shorter than an RTT, little improvement can be made.

5 Overcoming poor demand estimation with ADUs

Poor demand estimation leads to inefficient schedules (i.e., unnecessary reconfigurations or circuits that sit idle due to insufficient demand), an issue pointed out in scheduling work [3, 25] that is difficult to correct in-network. We argue that this must be solved at endhosts and demonstrate its impact on real flows over modern switches and schedulers.

5.1 Understanding problems with demand estimation

The quality of the schedules produced by an RDCN scheduler is largely based on the accuracy of the demand estimate. If demand estimation is limited by the size of shallow ToR VOQs (e.g., 16 packets as we suggest in §4), then the scheduler cannot tell which flow are large (and should be scheduled on the circuit switch) and which flows are small (and should be scheduled on the packet switch). Figure 11 illustrates this; ToR VOQ-based estimation produces long schedules requiring more reconfiguration penalties and idle times where the scheduler assumed small flows were large.

Communicating the size of application data units (ADUs) sitting in endhost buffers eliminates this problem. We argue that proper demand estimation must be done in the endhost stack because scheduler overhead is amortized by scheduling a week of demand (3ms in ReacToR [26]; order .1-1ms in Eclipse [3]; 2ms here) at once. Gathering 2ms of traffic at 80Gbps would require per-port buffers for ~2000 (9000 byte) packets, which is likely impractical on a switch and would (greatly) increase packet latency. Using endhost information requires no switch modifications nor inflates latency.

5.2 Using ADUs

We gather ADU information by creating an interposition library (libADU) that sends flow five-tuples (addresses, protocol, ports) and ADU sizes to the demand estimator on the software switch, using TCP over our control network (see Figure 2). LibADU interposes on `write` and `send` to get ADU sizes, and `shutdown` and `close` to indicate when outstanding demand for this flow should be removed. We modify the demand estimator (Figure 3) to keep track of outstanding demand per-flow to ease cleanup on flow shutdown. Alternative solutions could track outstanding demand at the rack level, but would either need to tolerate potential inaccuracies after shutdowns or require more complex cleanup (e.g., timeouts). We additionally modify ToR VOQs to count bytes in app data seen (with care given to avoid double-counting TCP retransmits). While individual ADU sizes are communicated to the scheduler, the scheduler we use (Solstice [27]) does not take advantage of knowing the boundaries between ADUs.

5.3 Experiments

Traffic generation: we test the impact of ADU-based demand estimation by using flowgrind [39, 40] to generate a workload of big and small flows. Big flows consist of 1,000 to 10,000 packets, chosen uniformly at random; small flows have 10 to 100. All packets are 9000 bytes. For each flow, we select a start time such that given their start times and sizes, all big flows from a source rack use 1/3 of the circuit bandwidth, and all small flows from a source rack use 1/3 of the packet bandwidth. *(source, destination)* rack pairs are chosen for the big flows so that they form a ring; the remaining racks send small flows. We run this workload for 2 seconds (after which we kill any unfinished flows), resulting in ~10,000 small flows and ~1,000 big flows.

Results: we compare ToR VOQ estimation to ADUs as well as the single "fixed schedule" that our scheduler would output with perfect demand estimation (i.e., send the single ring of big flows over the circuit switch). We show separate flow-completion time (FCT) CDFs for small and big flows in Figures 12 and 13, respectively. We find that for small flows, there is not much room to do better, as they are already fairly well serviced by the packet switch, although ADU-based estimation does cut 99% latency by half. For large flows, however, the median (99%) FCT is cut by almost 8 (11)×. This is due to

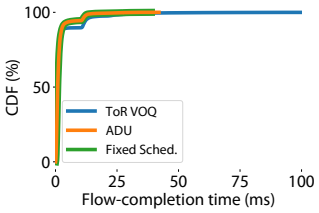


Figure 12: Small flow flow-completion time (1 ring).

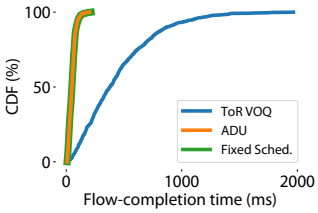


Figure 13: Big flow flow-completion time (1 ring).

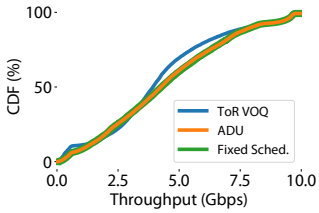


Figure 14: Small flow throughput (1 ring).

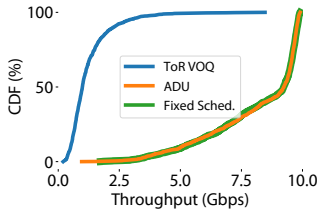


Figure 15: Big flow throughput (1 ring).

	ToR VOQ	VOQ + Resize	VOQ + reTCP	ADU	ADU + Resize	ADU + reTCP	Fixed Schedule	Packet (10Gbps)	Packet (20Gbps)	Packet (40Gbps)	Packet (80Gbps)
Median small FCT (ms)	1.52	1.57	1.60	1.51	1.58	1.59	1.49	2.19	1.05	0.71	0.65
99th small FCT (ms)	56.39	76.54	69.86	65.71	79.07	77.80	64.99	36.34	22.25	14.50	5.20
Median big FCT (ms)	1162.04	737.64	660.19	644.01	**110.61**	**102.11**	724.08	1680.35	1025.92	**46.95**	**45.30**
99th big FCT (ms)	1988.84	1917.55	1869.50	1815.74	**432.50**	**406.44**	1860.53	1998.27	1944.73	**134.29**	**114.77**

Table 2: Median and 99th percentile flow-completion time for small and big flows (2 ring workload).

the smaller number of configurations per schedule, meaning fewer reconfiguration penalties/idle small-flow circuits (see Figure 11). We also find that ADU-based estimation tightly hugs the fixed schedule, implying that for this workload, ADUs provide optimal results. We show throughput for small and big flows in Figures 14 and 15, respectively. While the trends are the same, small flow throughput shows two peaks for ToR VOQ, likely due to the separation of small flows sent over the packet switch and those accidentally sent over the circuit switch. **Takeaway: ADU-based estimation improves median (tail) big FCT by 8 (11)**×.

5.4 Cumulative results

Next we test the techniques from this section (ToR VOQ, ADU, Fixed) together with the techniques from the last section (dynamic buffer resizing and reTCP + resize, both with $\tau = 1000\mu s$). We also try various higher bandwidth traditional packet networks (10, 20, 40, 80Gbps) without circuit switches. We generate the same workload as above, but create two rings of big flows instead of one. Two rings are more difficult to schedule than one ring, requiring two circuit configurations per week (and thus more reconfigurations). We update our "fixed schedule" to include both configurations.

Table 2 shows the median and 99th percentile FCT for small and big flows. As with the prior experiment, we find that for small flows, having better schedule estimates or mechanisms to increase circuit utilization does not provide much benefit, as these flows are generally better served by the packet switch. Interestingly, all of our mechanisms (and the fixed schedule) increase the small tail FCT compared to ToR VOQ, likely because ToR VOQ accidentally sends some small flows over the circuit switch, giving them more bandwidth than they would get if properly scheduled.

Small flow FCTs on packet switches are even smaller, mainly due to increasing capacity as well as their finer-grained per-packet forwarding decisions. Interestingly, while the 10Gbps packet switch's median small FCT is worse then all other scenarios, its tail small FCT is better than VOQ. This is likely due to large flows backing off more (from loss) than they do on RDCNs, due to the lack of circuit ups/downs.

By accidentally scheduling small flows on the circuit, ToR VOQ starves big flows as we see in its FCT. VOQ+Resize, VOQ+reTCP, ADU, and Fixed Schedule reduce the median (by ~1.5–1.8×), but not tail FCTs. For VOQ+Resize & VOQ+reTCP this is due to inefficient schedules computed from inaccurate VOQ-based demand estimation. For ADU & Fixed Schedule this is due to small static switch buffers that (while providing low per-packet latency) cannot provide the required high throughput to big flows.

We see a big drop (highlighted) in median and tail FCT for big flows with ADU+Resize & ADU+reTCP, ~10-11× over ToR VOQ. ADU+Resize & ADU+reTCP take advantage of ADU-based estimation leading to a repeating two configuration schedule serving only the big flows in this two ring workload; buffer resizing only happens to the big flow VOQs because of this schedule, and VOQs are resized early enough (given $\tau = 1000\mu s$ and week length $2000\mu s$) to pin them to the large size. TCP takes advantage of the permanently large buffers in this workload to provide big flows much higher throughput than with either optimization on its own, greatly lowering FCT.

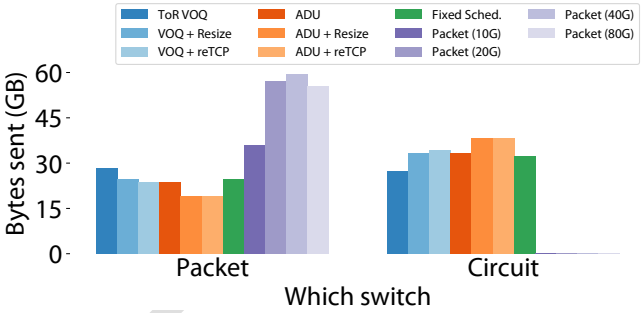


Figure 16: Packet and circuit utilization for various methods (2 ring workload). High circuit utilization is key to low flow-completion times in RDCNs.

Replacing an RDCN with a higher-bandwidth traditional packet network (without a circuit switch) can eek out another 2 (3.5) × improvement (highlighted) over our best optimizations in median (tail) big FCT. This additional improvement is due to its finer-grained per-packet forwarding decisions. These improvements require a 4× jump in capacity over our RDCN packet switch (10Gbps to 40Gbps). CMOS manufacturing limits, however, are making it increasingly difficult to make these capacity jumps for high-port count packet switches [29], making RDCNs more attractive.

To better understand the results, Figure 16 shows packet switch and circuit switch utilization for each scenario. High circuit utilization is a key factor for low flow-completion time in RDCNs. We see that dynamic buffer resizing and reTCP help move significant traffic to the circuit switch, both for VOQ- and ADU-based estimation. Using a fixed schedule does similarly, but requires knowledge of the demand a priori. A 10G packet switch by itself on this workload is so overloaded that it can not send all bytes within the 2 second time limit, ending with outstanding data.

6 Overcoming difficult-to-schedule workloads with application-specific changes

Workloads that are not *skewed* (easily separable into big flows for circuit switch and small flows for packet switch) or *sparse* (few pairs of hosts communicate) are fundamentally difficult to schedule on RDCNs [27], as they require many disruptive switch reconfigurations. The techniques from §4 can help work around this, but it would be better not to need them in the first place. An ideal workload would require a single circuit schedule; in this case, there are no switch reconfigurations and techniques like dynamic buffer resizing and reTCP are not necessary because TCP would consistently see a large circuit buffer. Unfortunately, many existing distributed data center application workloads are not as skewed or sparse as we would like. However, in many cases, this is not a fundamental property of the application; their workloads are uniform simply because there was no reason not to be.

In preparation for running on a RDCN, a few simple application-layer modifications can greatly increase performance by introducing skew and sparsity. Applications amenable to this are those whose communication patterns are flexible rather than completely prescribed—that is, a source sending a large flow has multiple choices of recipient. Introducing skew does not require nodes to have very detailed knowledge of the network topology, simply which *circuit group* (e.g., rack) other nodes belongs to. With this information, the nodes in a circuit group should aim to minimize the number of other circuit groups to which they collectively send large flows. For example, in a ring, which can be serviced with a single circuit configuration, each group sends to exactly one other group. In the rest of this section, we use replica selection in HDFS [2] as a running example.

6.1 HDFS write placement difficulties and solutions

Original workload: HDFS is a distributed file system made up of racks of datanodes (DNs) and a namenode (NN) which coordinates access. Data is generally replicated on three different servers. In the current HDFS write placement algorithm (see Figure 17), when a client writes to HDFS, the client first contacts the NN and is told where to place (blocks of) the file. If the client happens to be a DN, then the client's machine will be the first replica (otherwise a random DN is chosen). Second and third replicas are picked in a different, randomly chosen rack. Different clients in the same rack may be instructed to write to different racks (as shown); the same is true for different files on the same client or blocks within the same file. This results in all-to-all workloads that require many circuit configurations to properly schedule, incurring more reconfiguration penalties.

Modified workload (reHDFS): we create a modified HDFS write placement algorithm for the NN, reHDFS (reconfigurable DC network HDFS), that introduces skew and sparsity by selecting replica racks as a function of the source rack rather than randomly (Figure 18). For source rack i, the NN selects $(i + 1)$ mod N as the replica rack. This ensures that each circuit group (rack i) only sends to one other circuit group (rack $i + 1$). By choosing replicas in this way, the workload shifts from all-to-all to a single ring, which can be scheduled in a single circuit configuration. Concerns that this may cause availability issues during simultaneous rack failures can be addressed by changing this mapping function (e.g., to $i + 2$ etc.) on short timescales (e.g., seconds).

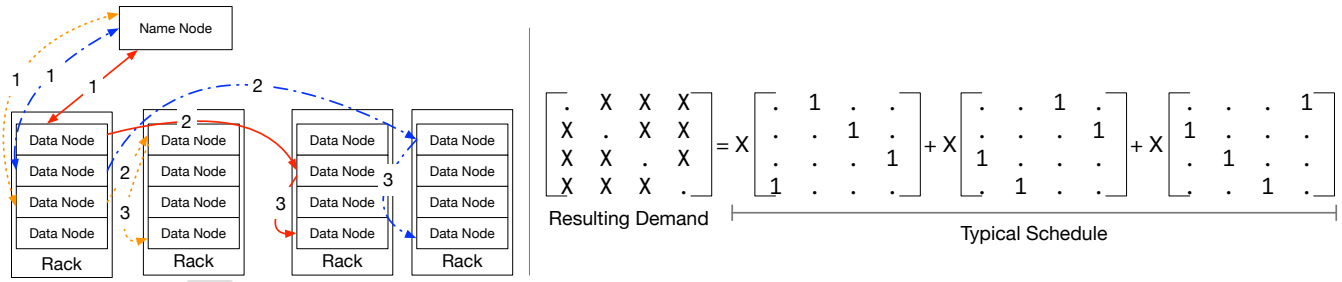


Figure 17: HDFS' default write placement algorithm. Data nodes communicate with the name node to learn where to write data replicas (1). The are told to write to themselves and two nodes in another randomly selected rack (2, 3). This results in all-to-all traffic that is difficult to schedule.

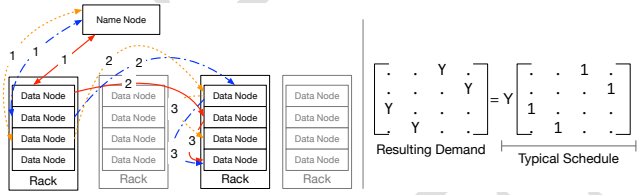


Figure 18: Our modified HDFS write placement algorithm, reHDFS. Replica rack selection is a function of the source rack, rather than random. This results in a single ring of demand that is easy to schedule.

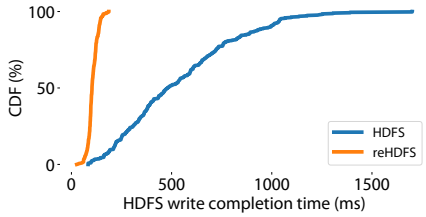


Figure 19: CDF of HDFS write completion time for HDFS and reHDFS for DFSIO benchmark.

6.2 Experiments

Methodology: We test reHDFS's modified write placement algorithm using an industry standard benchmark, DFSIO, in Intel HiBench [18]. DFSIO runs a Hadoop [1] (MapReduce [6]) job in which each of J mappers writes a file of X bytes to HDFS in parallel and then send statistics to a single reducer to output. We use $J = 64$ mappers to write 64 $X = 400MB$ files (25GBs of inter-rack traffic). This kind of workload might be seen during logging in big data applications or in scientific computing.

As we focus on improving network performance in our algorithm, we configure HDFS to use a 10GB RAM disk per host as its backing store to avoid being bottlenecked by slow mechanical disks. Running multiple virtualized Hadoop stacks (Hadoop, YARN, HDFS, JVM) per physical machine is difficult given the limited RAM (16GB) in our testbed machines. Instead, we run 8 mappers per physical machine (one per CPU core), sharing the underlying Hadoop stack to avoid this overhead. While this means tasks no longer have individual server-to-ToR link rate limiting, the lack of memory pressure makes this worthwhile. Machines with more RAM could avoid this limitation.

Results: Figure 19 shows the CDF of HDFS write time, for both HDFS and reHDFS. reHDFS reduces the median write time by ~4.7×, the 99th by ~7.6×, and the max by ~9×. Additionally, we reduce the job completion time for the DFSIO MapReduce task by 47%. **Takeaway: reHDFS improves write latency up to 9×.**

Next, we look at how various combinations of current and previous optimization (buffer resizing and reTCP + resizing, with $\tau = 1000\mu s$) perform under DFSIO. Combinations involving ADUs are less interesting in this workload due to the lack of many small flows. We verify this for some combinations and leave them out for space.

Figure 20 shows the 99th percentile HDFS write times for various combinations of previous optimizations. Without application-layer changes, dynamic buffer resizing by itself (§4) provides a modest 20% improvement (similar to our previous results with a more controlled workload). Modifying HDFS' workload to increase skew and sparsity, however, yields a drastic improvement: reHDFS improves 99th percentile latency by 87% compared to HDFS and 84% compared to HDFS with dynamic buffer resizing. Adding buffer resizing to reHDFS gives only an additional 3% improvement over vanilla HDFS because the modified workload is close to ideal; applications that cannot reduce their circuit group fan-out as much as reHDFS will benefit more from buffer resizing and reTCP.

Figure 21 shows the aggregate HDFS write throughput for various combinations of previous optimizations, with a dashed line indicating maximum cluster bandwidth. Again,

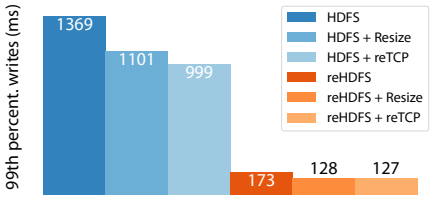


Figure 20: 99th percentile HDFS write times for DFSIO benchmark.

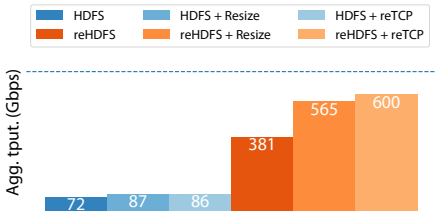


Figure 21: Agg. HDFS write throughput. Dashed line is cluster capacity.

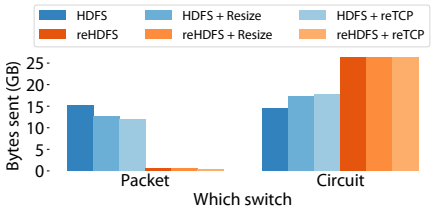


Figure 22: Packet and circuit utilization for DFSIO benchmark.

reHDFS provides the largest benefit (~5×), with others similar to the previous graphs (~1.6×). Interestingly, reHDFS + reTCP gets close to the maximum cluster capacity.

Figure 22 shows packet switch and circuit switch utilization for various combinations of previous optimizations, measured in the software switch. While the results are again similar, it is interesting to note how little data is sent over the packet switch for reHDFS-based optimizations, implying that reHDFS provides an astoundingly easier workload to schedule over RDCNs than vanilla HDFS. Finally, summing the bytes sent over both switches, we find that for HDFS-based methods send roughly 30GB of data, while reHDFS-based methods send about 27GB. Setting aside the 25GB written to HDFS in both cases, we are left with a 60% reduction in the rest of the data transfered (e.g,. MapReduce results shuffle traffic, HDFS name node traffic, etc.).

Though we have focused on only one application in this section, our reHDFS results suggest a few key points that likely generalize:

1. As expected, application workload has a big impact on RDCN performance.
2. There are real world applications that can be easily modified for RDCNs.
3. Though application-agnostic techniques like dynamic buffer resizing and reTCP are useful, modifying workloads to be more skewed and sparse can offer an order of magnitude more improvement, meaning application-specific changes are worth the extra effort.

7 Related work

While there has been much work on RDCN design [4, 11, 12, 15, 16, 22, 26, 31, 33, 38] and scheduling [3, 25, 27], however, no prior work has specifically focused on the end-to-end challenges in this paper. Some do briefly touch on them:

Rapid bandwidth fluctuation: prior work briefly explores TCP ramp-up issues, but either does so on millisecond-scale reconfigurable switches with thousand RTT-long days (removing the issue) [33], or avoid sending packets over the packet switch unless absolutely necessary (using a modified endhost kernel pause protocol) [26]. We only require in-network changes to ramp up TCP, and allow flows to use the packet switch when a circuit is down.

Poor demand estimation: most RDCN designs briefly touch on demand estimation [3, 25–27, 33], but little work has looked at how demand error affect scheduling. Albedo [25] and Eclipse [3] propose *indirect routing*, but note it's lackluster performance. We argue that proper estimation is the solution and must involve the endhost stack.

Difficult-to-schedule workloads: While some work has hinted at opportunities for cross-layer application/network co-scheduling [22, 33], works with concrete examples have focused on making the network "application-aware" [25, 34, 35] (e.g., using MapReduce's job tracker for demand estimation), rather than making applications "network-aware" (i.e., modifying applications to suit the network), as we propose.

8 Conclusion

Recent work has shown the increasing need for augmenting traditional packet networks with reconfigurable circuit technologies in DCs [12, 22, 26, 27, 29, 31]. We show the existence of three key end-to-end challenges in such networks (RDCNs): 1) rapid bandwidth fluctuation, 2) poor demand estimation, and 3) difficult-to-schedule workloads. We overcome bandwidth fluctuation with dynamic in-network resizing of ToR queues, poor demand estimation by communicating endhost ADUs, and difficult-to-schedule workloads by rewriting application logic (e.g., HDFS' write placement algorithm). We design and implement an open-source RDCN emulator Etalon for use on public testbeds, and evaluate the nature of these challenges and the efficacy of our solutions. We find that the cross-layer optimizations we propose are necessary, and that they provide the most benefit at higher layers. Etalon provides additional opportunities to explore in future work, e.g., utilizing ADU boundaries in scheduling, exploring multicast-enabled optical circuit switching (e.g., Blast [35]), or, more importantly, providing a cross-cutting evaluation across different RDCN designs, or an investigation of the challenges faced in future sub-μs RDCNs. We believe our experience speaks to the need for end-to-end system evaluations in future DC designs.

References

[1] Apache. 2018. Hadoop. (2018). https://hortonworks.com/apache/hadoop/

[2] Apache. 2018. Hadoop HDFS. (2018). https://hortonworks.com/apache/hdfs/

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

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

[5] CloudLab. 2018. CloudLab. (2018). https://www.cloudlab.us/

[6] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[7] Docker. 2018. Docker. (2018). https://www.docker.com/

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

[9] DPDK. 2018. DPDK. (2018). https://dpdk.org

[10] Emulab. 2018. APT cluster. (2018). https://www.aptlab.net/

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

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

[13] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. 2008. DieCast: Testing distributed systems with an accurate scale model. (2008).

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

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

[16] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. 2014. FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-space Optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/2619239.2626328

[17] HewlettPackard. 2018. netperf. (2018). https://github.com/HewlettPackard/netperf

[18] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 41–51.

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

[20] iperf. 2018. iperf. (2018). https://iperf.fr/

[21] iperf3. 2018. iperf3. (2018). https://iperf.fr/

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

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

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

[25] Conglong Li, Matthew K Mukerjee, David G Andersen, Srinivasan Seshan, Michael Kaminsky, George Porter, and Alex C Snoeren. 2017. Using Indirect Routing to Recover from Network Traffic Scheduling Estimation Error. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 13–24.

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

[27] 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. 2015. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 41.

[28] Mellanox. 2018. sockperf. (2018). https://github.com/Mellanox/sockperf

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

[30] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

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

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

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

[34] Guohui Wang, TS Ng, and Anees Shaikh. 2012. Programming your network at run-time for big data applications. In *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 103–108.

[35] Yiting Xia, TS Eugene Ng, and Xiaoye Steven Sun. [n. d.]. Blast: Accelerating high-performance data analytics applications by optical multicast. In *Proc. IEEE INFOCOM*.

[36] Jiaqi Yan and Dong Jin. 2015. 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*. ACM, 235–246.

[37] Jiaqi Yan and Dong Jin. 2015. 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*. ACM, 27.

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

[39] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. [n. d.]. Flowgrind-a new performance measurement tool. In *IEEE GLOBECOM*.

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