A large-scale deployment of DCTCP

Abhishek Dhamija and Balasubramanian Madhavan, *Meta;* Hechao Li, *Netflix;*
Jie Meng, Shrikrishna Khare, and Madhavi Rao, *Meta;* Lawrence Brakmo*;*
Neil Spring, Prashanth Kannan, and Srikanth Sundaresan, *Meta;*
Soudeh Ghorbani, *Meta and Johns Hopkins University*

https://www.usenix.org/conference/nsdi24/presentation/dhamija

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

A large-scale deployment of DCTCP

Operational Systems Track

Abhishek Dhamija[1], Balasubramanian Madhavan[1], Hechao Li[2], Jie Meng[1], Shrikrishna Khare[1], Madhavi Rao[1], Lawrence Brakmo, Neil Spring[1], Prashanth Kannan[1], Srikanth Sundaresan[1], and Soudeh Ghorbani[1,3]

[1]*Meta*, [2]*Netflix*, [3]*Johns Hopkins University*

Abstract

This paper describes the process and operational experiences of deploying the Data Center TCP (DCTCP) protocol in a large-scale data center network. In contrast to legacy congestion control protocols that rely on loss as the primary signal of congestion, DCTCP signals in-network congestion (based on queue occupancy) to senders and adjusts the sending rate proportional to the level of congestion. At the time of our deployment, this protocol was well-studied and fairly established with proven efficiency gains in other networks. As expected, we also observed improved performance, and notably decreased packet losses, compared to legacy protocols in our data centers. Perhaps unexpectedly, however, we faced numerous hurdles in rolling out DCTCP; we chronicle these unexpected challenges, ranging from its unfairness (to other classes of traffic) to implementation bugs. We close by discussing some of the open research questions and challenges.

1 Introduction

Congestion control algorithms (CCAs) modulate traffic entry into the network, seeking high utilization, low latency, and relative fairness by making frequent decisions about how much data to send and when. These decisions are based on congestion signals, and the base signal in many CCAs is the *dropped packet.* CCAs tend to increase the amount of data in flight until the point to induce packet loss. Subsequently, they create queue buildup and increase delay. This impacts latency requirements for our datacenter applications. These workloads generate a large number of small request and response flows across the datacenter that, combined, complete a user-requested computation. For a fast response time, each of these short flows should be completed fast.

A class of CCAs [4, 5, 12, 14, 17, 20, 24] tailored specifically to the requirements of datacenters workloads, leverage a diverse set of congestion signals (notifications from the network, measured delay at endpoints, etc.) to detect and react to imminent congestion faster. As one of the earliest and most mature protocols in this class, Data Center Congestion Control (DCTCP) [4] uses Explicit Congestion Notifications (ECNs) from the switches to adjust the sending rate proportional to the level of congestion. ECN remedies large queue buildups and drops by providing a congestion signal before queues overflow, and DCTCP interprets the fraction of ECN-marked packets to scale its response, avoiding persistent queueing and overflows that lead to loss.

A few features of our network made DCTCP's potential worth considering. First, our top of rack switches had "shallow buffers," providing limited space for queueing. Second, buffer sharing and contention over buffers led to variable and unpredictable queue capacities [13]. Finally, our approach to distributing jobs across datacenters created a set of racks with a mixture of large throughput-heavy flows and small latency-sensitive flows that had to compete for the limited and variable buffer space in the network. DCTCP could moderate large flows' use of switch buffers, providing more isolation between jobs sharing shallow-buffered switches.

Using DCTCP required resolving a challenge: designed specifically for short-RTT datacenter traffic, DCTCP had to be applied exclusively to in-region[1] traffic and not cross-backbone traffic. This translates to a few problems: (a) we had to identify in-region traffic, separate it from cross-region traffic, and negotiate DCTCP only for the former. Doing so in a network with the scale and complexity of ours, without risking a broken network, required a large engineering effort, and (b) the in-region DCTCP and cross-region Cubic traffic had to co-exist and share the network. This required careful parameter tuning and network configuration to strike a balance between various classes of traffic. Although challenging, these were surmountable issues that our engineers could overcome to apply DCTCP only to the in-region traffic alongside the Cubic cross-region traffic.

Lawrence Brakmo and Hechao Li contributed to this work during their time at Meta.

[1]A *region* is a collection of datacenters in roughly the same location.

In addition to fitting our workloads' needs, one specific feature of DCTCP made it an operationally appealing choice: it was a relatively simple, mature, and well-established protocol. When we initiated an effort to change our datacenter congestion control algorithm from Cubic in 2018, DCTCP's design had been published eight years prior (2010), its design was well-understood, it had been added to the Linux kernel for four years (since 2014), and had widespread hardware support for ECN marking. We anticipated a smooth transition to DCTCP. Contrary to our anticipation, however, nearly every point in the end-to-end stack presented a riddle to solve to rollout DCTCP: the kernel had bugs; optimizations such as receive offloading could not always inter-operate with it as DCTCP's smaller congestion windows were not always enough to trigger prompt delivery; some switches dropped ECN-capable packets despite having space to buffer them, resulting in poor application performance; not all switches could consistently and reliably support ECN; we could not change the congestion control of long-running connections in the middle of their data transfer, etc.

In this paper, we share our experience of deploying DCTCP in Meta Datacenters. This project started in earnest in 2018, balancing successful tests with head-scratching problems. We share the stumbling blocks we discovered in the hope of helping researchers consider the *deployability* of new data center congestion control algorithms. In particular, switch and NIC implementations are diverse, vendor specific, not always known to datacenter operators, and evolve frequently as the scale and demand of our networks change. Protocols that are dependent on parameter tuning for optimal performance are hard to deploy and maintain. In a large-scale and diverse network, each congestion control algorithm will coexist with a large and diverse set of protocols. Finally, bugs are inevitable in any large-scale, complex network. Ideally, the congestion control protocols should be equipped with mechanisms to detect and gracefully handle bugs and corner cases. We found every problem in the text that follows to be rich and deep, with quite a few surprises. In the moment, we questioned why is this so hard: switches can mark ECN and the kernel implements DCTCP. What more does there have to be? But reflecting on the experience, by expecting unforeseen trouble, careful deployment and monitoring paid off.

Ours is not the first report on production DCTCP deployment; Judd [15] shared experiences that influenced our design. They showed that DCTCP could be unfair to an established Cubic flow, motivating our study of mark and drop thresholds to keep them fair. He also noted that SYN and SYN/ACK packets should be ECN-capable, despite standards; we were surprised to find the same even after our switch thresholds prevented DCTCP from starving out Cubic. His deployment supported ECN marking only on top-of-rack switches; this became the starting point we focus on in this paper.

We organize the paper by characterizing problems by where they occurred. In the next section, we briefly overview DCTCP. Feel free to skip this section or read Alizadeh et al. [4] instead. Section 3 describes how we chose to enable DCTCP for in-region TCP connections but not cross region ones; focusing on deployment safety. Section 4 describes how to configure switches to mark DCTCP traffic and encourage balance with competing Cubic flows; for some devices in our network, this was unexpectedly elaborate. Section 5 describes what we built to monitor the deployment, looking to confirm that congestion-experienced bits were set, DCTCP was negotiated and not falling back to Reno, switch buffers were less occupied, etc. Section 6 describes the often subtle kernel and driver bugs that ECN marking packets and smaller congestion windows surfaced. Section 7 describes a few extensions we applied to the initial DCTCP deployment, extending where congestion can be instrumented with ECN. We conclude in section 8, discussing lingering problems in congestion and reflecting on how the design of nearly every component in the network influenced this deployment.

2 Background

Data Center TCP (DCTCP). DCTCP uses Explicit Congestion Notification (ECNs) signals from switches. It uses 2 bits in the IP header for ECN information. If neither of the two bits are set, the flow does not support ECN and switches will not mark the packets. When only one bit is set, the flow supports ECN signals and no congestion has been encountered. Finally, when both bits are set, the flow supports ECN and the packet has encountered congestion. Congestion, in this context, is usually defined as the queue sizes of the switches on the packet's path passing a pre-defined threshold. That is, when a switch receives an ECN-enabled packet, if the queue used to enqueue the packet is larger than some threshold, the switch marks the packet as having experienced congestion. This signal then arrives at the receiver. The receiver notifies the sender by echoing back the congestion signals on the TCP header of the ACK packet.

Pre-DCTCP, senders treat ECN-marked ACKs as packet loss. TCP Reno, for instance, reduces its congestion window (CWND) by 50%. This aggressive throttling can lead to link under-utilization. Another issue is that legacy protocols do not differentiate between short bursts and standing congestion. For example, sub-RTT queue buildups due to microbursts still result in reducing CWND, a suboptimal outcome.

In contrast, DCTCP reduces CWND proportional to the level of congestion by tracking the percentage of bytes per RTT. For example, if 100% of bytes encounter congestion, DCTCP reduces its CWND by 50% but if only 50% of bytes do so, it reduces CWND by 25%. DCTCP also leverages a moving average to avoid overreacting to transient bursts. For example, if 100% of the bytes in an RTT encounter congestion, but there was no congestion in previous RTTs, then the CWND would only be reduced by 1/32 instead of 1/2 [9].

Given the characteristics of our workloads, notably their

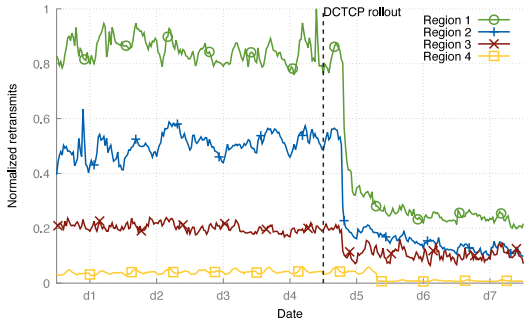

Figure 1: Retransmission rates drop following DCTCP rollout.

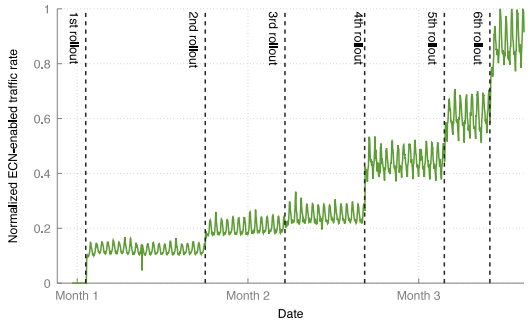

Figure 2: DCTCP's rollout accelerated over time.

burstiness [13, 27], we expected DCTCP's faster reaction to bursts to improve the performance of our applications.

DCTCP helped our network. Encouraged by positive test results, we gradually rolled out DCTCP to each region in the fleet. Overall, we observed improvements in network metrics such as reduced top-of-rack switch congestion discards and queue lengths, resulting in the reduction in the number of retransmitted packets that hosts received. We compared the overall volume of retransmissions for each region after deploying DCTCP and compared it against one week before the transition and observed a reduction of around 75%. Figure 1 shows the change in normalized retransmission rates[2] for four regions a few days before and after the rollout of DCTCP in each region. Note that the reduction in retransmissions after deploying DCTCP is not immediate. In §3.2, we discuss a potential reason (the delay in changing the congestion control algorithm for long-running connections).

Along with retransmissions, we tracked base host metrics like throughput, and the congestion window size, as well as general system state metrics such as CPU and memory utilization. We did not observe any regressions in these metrics. For four regions, we measured the changes in retransmission rates, throughput, the average CWND, and the average smooth RTT (`srtt`) after transitioning to DCTCP. Table 1 reports the results. Note the variance across different metrics and regions, e.g., while `srtt` did not change in Region 4, it did improve in Region 5 albeit not as dramatically as the retransmission rate (7% *vs.* 50%). Moreover, for one of our data-intensive services, we measured the changes in read latency in a region before and after transitioning to DCTCP and observed 38% reduction in the 90^{th} and 99^{th} percentiles of latency (from 65*ms* to 40*ms* and from 130*ms* to 80*ms*, respectively).

Rollout timeline. It should be noted that we did not upgrade all selected regions to DCTCP at the same time; we proceeded gradually over a four-month period while carefully monitoring the impact of the change on our networks (§5). Figure 2 shows the timeline of the per-region rollout, overlapped with the aggregate rate of ECN-enabled packets during this period. The rate of ECN-enabled packets in our networks is a proxy of DCTCP's adoption, as DCTCP traffic is by default ECN-enabled. During this interval, the DCTCP was the only class of ECN-enabled traffic in our networks. To compute the normalized rates in Figure 2, we divided the number of ECN-enabled packets received by all datacenter hosts by the maximum number of ECN-enabled packets received by all hosts once the rollout process was completed. The figure depicts progressively smaller and smaller time gaps between successive rollouts; our deployment process gradually accelerated as our initial deployments helped us identify and resolve the issues that we will discuss in the rest of the paper.

To gauge the impact and effectiveness of DCTCP after the rollout, we temporarily disabled it for a few hours in one region. This resulted in around a 10% drop in throughput and 4.5× increase in retransmissions in that region.

Metric	Region 3	Region 4	Region 5	Region 6
Retrans.	33% ↓	79.5% ↓	50% ↓	73.4% ↓
Throughput	10.3% ↑	4.8% ↑	2% ↑	5.8% ↓
Avg. `srtt`	No change	No change	7% ↓	7% ↓
Avg. `cwnd`	30% ↑	22.3% ↑	20% ↑	18.1% ↑

Table 1: Changes in performance metrics.

3 Enablement

We wanted to enable DCTCP only for in-region connections, which have small enough RTTs that DCTCP is effective. This presented several immediate challenges for us as our available options could not achieve this, e.g., the selective enablement requirement meant that `sysctl` is not viable, while complexity ruled out options such as `setsockopt` and `routes`.

As we explored and experimented with enablement options, we also encountered several other challenges related to kernel features, and obscure DCTCP fallback behavior. We discuss these challenges below.

3.1 Enabling DCTCP for only in-region flows

We discuss the drawbacks of available knobs in greater detail before presenting our approach for resolving this challenge.

[2] *Normalized retransmission rates* are retransmission rate of each region divided by the maximum retransmission rate across all regions.

3.1.1 Potential knobs for changing congestion control

Our goal was to identify knobs to target short-RTT connections and change their congestion control algorithm (CCA). For DCTCP, this should be done before the 3-way handshake since ECN is negotiated during connection establishment. We targeted sub millisecond latency which is typical for traffic within a region. In our data center network, we can use IP addresses as a proxy for RTT, and classify connections created within the region by looking at the source and destination IP addresses. CCAs can be changed in many ways, notably via `sysctl`, `setsockopt`, and `routes` as we discuss next.

`sysctl` `sysctl` is the most common and easiest approach to change CCA. This takes effect for *all new sockets*. This was not ideal for two reasons: (a) we could not easily distinguish between inter- and intra-region traffic and limit the adaptation of DCTCP only to the latter, and (b) the transition process would be slow. Since changes through `sysctl` take effect only on new sockets, listening sockets had to be reset for the passive side of the connection to pick-up this change. At a region-scale, resetting listening sockets requires restarting services which takes days or weeks.

`setsockopt` The most granular approach is to call `setsockopt()` per socket. This also provides the most flexibility since it enables running arbitrary logic for each individual connection. On a datacenter scale, `setsockopt` involves building a library that is used by every service and fitting into the continuous integration and delivery schedule of thousands of services which run across the fleet. Although this was conceptually feasible, this tight coupling was operationally not ideal as it would complicate debugging and fault isolation. Plus, it offered little benefit since DCTCP did not require any service-specific information that would justify inter-dependency with services.

`routes` Linux supports customizing TCP parameters per IP destination route. By changing the granularity of `routes`, we could target different scopes, ranging from an entire region to a single host. For enablement, we needed to enumerate all the intra-region route prefixes and create multiple route table rules. This approach was attractive since it had the lowest dependencies across all options. However, it was not *extensible*: it did not support matching based on other criteria such as the port number, the kernel version, and the NIC model that we envisioned using in future designs. Another risk of this approach was modifying the host routing table and depending on aggregatable IP prefixes. Given how routes control reachability in our fleet, these created substantial risks.

In summary, none of the approaches discussed above was adequate for our use case. We had to develop a new approach based on eBPF which we discuss next.

3.1.2 TCP socket hook eBPF

We developed a method based on eBPF to address the challenges discussed above. By providing hookpoints across the kernel stack, eBPF allows us to customize the network stack. Since its introduction in Linux 4.11, `sockops` has provided a way to run eBPF programs during TCP socket events. A major advantage of `sockops` is its flexibility and programmability. It enables running an eBPF program at the start of every connection which can use user-space configuration to select a CCA for every connection. We use the same IP-based classifier to identify connections to enable DCTCP.

By design, `sockops` is restricted to a specific version of `cgroup` (`cgroup-v2`).[3] Alas, in our fleet, we deployed heterogeneous kernel versions with two versions of `cgroup`. Kernel and `cgroup` limitation meant this cannot be used standalone.

To work around the dependence of `sockops` on `cgroup` version, we used *two* BPF programs to enable DCTCP for new in-region TCP connections: (1) On `cgroup-v2` hosts, we attached a per-connection `sockops` BPF program to set DCTCP as the CCA for in-region connections while leaving inter-region connections unchanged, and (2) on `cgroup-v1` hosts, we used `sysctl` to set DCTCP as the default CCA and attached a per-packet Traffic Control (TC) program to clear ECN-related bits on inter-region SYN packets, forcing those connections to fallback to Cubic. This program not only disables DCTCP for inter-region traffic, but also enables DCTCP for connections on pre-existing listening sockets.

3.1.3 Enablement plan

Our intended scope for deploying DCTCP was an instantaneous region-wide enablement, so that we could minimize disruption in the network caused by in-region DCTCP and Cubic flows interacting in the same switch buffer. This was a substantial change that had never been done before in our network and had to be performed safely without causing disruption to services. This was a challenge given the heterogeneity of our fleet (e.g., in terms of *cgroup* and kernel versions). Ensuring safety despite the diversity of the fleet required having visibility and monitoring into enablement knobs to detect problems, and the ability to revert fast if there was a problem. Safety, flexibility, and fail-open characteristics made eBPF a good choice for this problem. Given we had 90% of the fleet already on `cgroup-v2` when we started with this effort, we took on more complexity in the short-term to handle `cgroup-v1`, but over time, we retired the `cgroup-v1` solution and evolved eBPF to be able to customize both the algorithm and deployment capabilities that we had originally envisioned. We explain the two separate eBPF knobs for the two `cgroup` versions next.

3.1.4 `cgroup-v2` and 4.11+ kernels

We attach a per-connection sockops BPF program [8] to the root-cgroup. A sockops program is invoked for different state

[3] Control groups or *cgroups* is a Linux kernel feature that limits, monitors, and isolates the resource usage of a collection of processes.

```
int sockops_program(skops) {
  switch (skops->op) {
    // Before sending SYN on client or SYN-ACK on server
    // if peer in-region request ECN
    case BPF_SOCK_OPS_NEEDS_ECN:
      if(in_region(skops)) {
          skops->reply = 1;
      }
      break;
    // Once connection is established
    // If we see ECN then enable DCTCP, else Cubic
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      if(ecn_enabled(skops)) {
        bpf_setsockopt(skops, TCP_CONGESTION, "dctcp");
      } else {
        bpf_setsockopt(skops, TCP_CONGESTION, "cubic");
      }
  }
}
```

Figure 3: Summary of ECN enablement in cgroup-v2 sockops

transitions in the lifetime of a TCP connection. During specific TCP events, `sockops` eBPF programs can change some TCP parameters using `setsockopt()` calls or influence kernel actions. For example, when client or server have to decide ECN negotiation, eBPF programs registered on this callback can opt-in for ECN negotiation after analyzing the TCP and IP headers in the packet or any user-space map state. Our `cgroup-v2` program uses the source and destination IP address and calculates the scope (rack/cluster/dc/region) of the connection. It uses an IP prefix eBPF map populated with all intra-region IP prefixes. It takes the following actions also summarized in Figure 3.

#1: NEED_ECN: When kernel asks whether a socket needs ECN or not, choose "YES" for in-region traffic and "NO" for inter-region traffic.

#2: CONN_ESTABLISHED: When a connection is established, if it is in-region and ECN enablement succeeded, then call `bpf_setsockopt` to set the socket's CCA algorithm to DCTCP. Inter-region CCA remains unchanged and uses the default specified via `sysctl`.

3.1.5 `cgroup-v1` and older kernels

On hosts with `cgroup-v1` or older kernels where sockops eBPF hookpoint was not available (less than 10% hosts), we still needed a solution to selectively change CCA. We chose a combination of `sysctl` and TC eBPF to solve this. We used `sysctl` to set DCTCP as the default CCA and used a Traffic Control (TC) eBPF program to terminate ECN negotiation for inter-region connections, thereby forcing them to fall back to Cubic as we explain below. Table 2 shows the sequence of events that happen during a 3-way handshake (the blue parts are specific to ECN/DCTCP).

To selectively disable DCTCP for inter-region scope, we use a per-packet TC eBPF program on the server in the ingress direction. TC eBPF programs can access socket buffer (skb) and through this modify packet contents. This program can thus selectively clear all ECN state set by the client (step 1 from Table 2). This is possible because in the ingress direction TC programs are executed before the TCP stack can negotiate ECN. This program uses a similar scope resolver as explained in the sockops section above to detect inter-region clients and match TCP SYN packets with ECN negotiation bits (ECE/CWR bits [6]). If there is a pattern match, this program clears the ECN-related bits in the TCP and IP header of the SYN packets thus causing the ECN negotiation to fail. Without ECN, DCTCP is configured to fallback [7] and this eBPF program exploits this configuration.

Step	Direction	IP flags	TCP Flags
1	Client→Server	ECT(0)	SYN + ECE + CWR
2	Server→Client	ECT(0)	SYN + ACK + ECE
3	Client→Server	ECT(0)	ACK

Table 2: ECN + DCTCP Handshake

Although the Linux DCTCP implementation was configured to fall back to TCP Reno as the CCA when ECN negotiation failed, we did not want this extra algorithm to be added to the set of algorithms in our networks. Recall that the co-existence of Cubic and DCTCP traffic was a major challenge in deploying DCTCP. Adding a new protocol to the set of CCAs would exacerbate the situation and require re-running the entire process of parameter tuning and switch configuration with a new combination of protocols. To avoid this problem, our kernel team altered the DCTCP code [7] to fallback to Cubic instead, a change we did not publish upstream.[4] This was our only custom kernel change for the effort, and we removed this patch by replacing the logic with BPF code that could itself handle the fallback.

3.2 Long-lived connections

ECN is negotiated at the onset of the connection: this created a challenge for changing the CCA of ongoing connections from Cubic to DCTCP as we could not enable ECN for a flow in the middle of the connection. Unfortunately, many connections in our networks are long, running for days or longer. This slowed down the process of migrating to DCTCP. The naive solutions such as terminating the existing connections and collaborating with service owners to force all services to be restarted were suboptimal due to their complexity and perceived impact on service performance.

We realized that one of our internal disaster recovery tools, Maelstrom, can be repurposed to aid with the CCA upgrade process. Maelstrom [26] is a large-scale disaster recovery system. It provides a traffic management framework with modular primitives that can be composed to safely and efficiently drain the traffic of interdependent services from one or more failing datacenters to the healthy ones. Maelstrom encoded inter-service dependencies and had a safe way to temporarily

[4] There are some drawbacks to including this fallback policy in the mainline kernel, e.g., Cubic is an optional module in Linux, while Reno is not.

drain traffic from a region. We leveraged Maelstrom to run multiple drain exercises to gradually upgrade the connections to DCTCP. Each drain exercise resulted in approximately 50% of the connections to flip to DCTCP resulting in significant coverage for the CCA change.[5]

To enable future upgrades and maintenance, eventually, we enhanced our eBPF framework with a connection iterator which can iterate over all existing connections, upgrade congestion control, and make other transport changes. The ability to trigger this program on-demand and share similar code as sockops simplified CCA's evolvability and maintenance.

4 Switches and buffers

DCTCP relies on network support—switches mark packets with ECN when buffer occupancy exceeds a certain threshold. Most modern switches support ECN, but it quickly became clear that our specific network characteristics made widespread deployment of ECN unfeasible. We list the specific challenges we faced in deploying ECN in our network.

4.1 Switch queues are a scarce resource

DCTCP, as the name suggests, is designed for data center flows with low RTT ($\sim 1ms$). It is not expected to work well with long-distance flows (10s to 100s of *ms*). This is because the ECN signal, based on queue occupancy, is ephemeral; current queue occupancy is meaningless several milliseconds in the future. So, for our cross-region flows, we continue using Cubic, which relies on packet loss as its signal.

It is common that DCTCP and Cubic flows terminate at the same host, a service can talk to other hosts both in-region and cross-region. In such cases, ideally, we would isolate the feedback signal of the two CCAs in the network. For Cubic and DCTCP, isolation is even more important because they rely on two very different signals. ECN-based CCAs aim to keep bottleneck buffer utilization low, while loss-based CCAs drives the buffer to capacity and loss. In switch terms, conceptually, isolation is easy enough to achieve – we just need to put each CCA in its own queue where it gets access to its own allocation of the dynamically shared buffer.

However, switch queues are primarily used for traffic classification in our network; we support several classes of traffic [3], and each class is allocated to a queue. To isolate CCAs from each other, we would need two queues per traffic class. While later generations of switches supported this, initially, our switches did not have enough queues to support this configuration. This meant that we had to put both DCTCP and Cubic in the same queue, and depending on the switch vendor, the switch supported ECN and Drop Tail (or WRED) in the same queue, or only either ECN or Drop Tail (or WRED). Luckily for us, the bottleneck layer in our data center network had the switches that supported both.

[5] Another benefit of Maelstrom was that it could keep the region in the drained state if we observe any CCA-related outage. This enabled us to manually connect to hosts and remediate the issue.

4.2 ToR switches were sufficient for ECN

We analyzed our production network, and found that bottlenecks largely occurred in the ToR switch downlinks [13]–to hosts–or in our backbone WAN network. The latter is out of scope for this work, as DCTCP does not traverse regions, so we focused only on the ToR downlinks.

A natural question is why we saw congestion largely in the ToR downlink. The answer is a combination of factors. First, due to the nature of our hardware, there was very high disparity in link speeds in the ToR downlink compared to the rest of the fabric, making this layer more susceptible to bursty incasts. Secondly, provisioning and topology ensured high cross-section bandwidth between racks. Finally, rack-agnostic job scheduling ensured a high level of heterogeneity in traffic, largely preventing potential hotspots caused by concentrated placement of network-heavy services.

As we mentioned previously, not all layers in our fabric had switches that supported dual mode thresholding to support DCTCP and Cubic. However, since congestion was rarely an issue elsewhere in the network (and therefore buffer utilization not a big enough issue to cause discards), we could effectively ignore them, and focus our attention of getting the thresholds right for the ToR downlink.

4.3 How to set a mark and drop threshold

Our switches had the ability to mark ECN or do Drop-Tail/WRED on a flow depending on ECN-Capable Transport (ECT) marking, so both DCTCP and Cubic would receive the correct signal, but the buffer itself was shared across both classes of flows, resulting in no isolation.

ECN signaling tries to keep DCTCP queues low, but offers no guarantees – it is entirely possible that a burst of DCTCP traffic can occupy a buffer well beyond the ECN threshold. Cubic flows, on the other hand will try to maintain high buffer utilization. This is a challenging scenario: high buffer occupancy with bursty Cubic flows will result in very high ECN marks for a competing DCTCP flow, while high buffer occupancy with bursty DCTCP flows will result in very low available buffer for competing Cubic flows. Complicating the issue further, not all DCTCP flows are bursty (unless within an incast), while even a few long-RTT Cubic flows could be bursty due to the high BDP.

Our hypothesis was that we need to prevent either CCA from entirely taking over the buffer – but we were more concerned about Cubic's ability to do so, so we narrowed down to a solution where we set relatively high ECN thresholds for DCTCP and low droptail thresholds for Cubic. This was a sub-optimal solution for both CCAs, because it diluted ECN for DCTCP and made loss likelier for Cubic, but it solved for

what we thought was the dominant problem in our network – Cubic bursts undermining DCTCP.

Post-deployment experience proved our hypothesis to be not entirely correct: while Cubic indeed could capture a shallow buffer, incast scenarios were far more serious in our network than we originally anticipated, and had to implement further features to control DCTCP burstiness and buffer utilization, particularly for services that had both sets of flows.

4.4 The multi-host NIC

Multi-host NICs use a single connection to a ToR switch to provide connections to the PCI busses of (typically two or four) different hosts. The multi-host (MH) NIC design is efficient in space, power, and hardware, but is a network component not typically modeled in congestion control work.

The MH NIC connects multiple machines to the network and has a buffer (on the order of 0.5 to 1 MB in size), but it does not act as a conventional switch. The MH NIC does not mark ECN mark by default; earlier versions do not even have this feature. The buffer is not partitioned explicitly across hosts or queues, so it does not provide isolation, nor is able to preferentially deliver high priority traffic. It does not have predictable downlink rates; PCI bus rates are split across the hosts, resulting in per-host rates being significantly less than the MH NIC rate. Furthermore, when a host kernel is unable to keep up with interrupts, or is unable to supply free buffers, the delivery speed to an individual host can drop further.

As the MH NIC buffer fills, the NIC can send ethernet pause frames back to the top-of-rack switch, leading to queuing at the switch. This can happen when a single host is the target of incast: the complete bitrate of the NIC can be applied to deliver the burst of traffic from switch to NIC, reaching a bottleneck at delivering the data to the memory of the destination host. This delivery can also be slowed enough to send pause frames when a host is processing difficult-to-accept data such as small frames from many different connections not amenable to offloaded reassembly.

Unmodified, the ECN marking threshold at the ToR means that the effective queue backlog to the host is the ToR ECN threshold *plus* the size of the NIC buffer. This creates an effectively far-too-high ECN threshold, resulting in persistent unfairness, poor performance, and even packet loss of DCTCP traffic as the congestion window overshoots the target.

The effectively larger buffer for a host is not as severe a problem for Cubic alone: it appears as a single large buffer shared across the hosts. However, a single host's traffic can still dominate the buffer, and since only the switch has the ability to prioritize traffic, unfairness between buffer-hungry services and services that need highly-reliable delivery led to shifting the buffer to the switch via per-host queuing.

On racks with this type of MH NIC, the ToR switch creates a separate, rate-limited queue for each downstream host. This rate limit is based on the host's "share" of the NIC bandwidth, i.e., 1/2 or 1/4. This queue is then configured with ECN mark and drop thresholds from Section 4.3. This queue-per-host design does not remedy all interactions between different jobs sharing the same NIC: for example, a slow kernel processing small packets can still fill the NIC buffer, leading to some pause frames to the ToR, but while the link is unpaused, the ToR can round-robin among the other hosts to limit performance degradation. Important for DCTCP however is that in the expected case, packets are marked when the effective queue, entirely on the ToR, reaches the intended threshold.

This queue-per-host feature played an important role in getting performance and fairness out of DCTCP on MH NIC systems. Testing focused on fairness and performance of flows to individual hosts; the performance of concurrent, production-like traffic to different hosts on the MH NIC was not easily observed. We were fortunate that the queue-per-host feature in our ToRs was rolled out in time for DCTCP deployment.

4.4.1 Database clients in particular

Here is a specific example of how queue-per-host became necessary to deploy DCTCP. During our first region rollout, it turned out that many in-region connections *establishments* were timing out. After one second, the Database client code making this connection would time out and report an error. Packet loss and retransmissions overall were down, utilization up; the typical signals of network performance looked good.

We have a set of tools that instrument retransmissions generated by the Linux kernel using eBPF "tracepoints" and "kprobes." While the kernel's built-in counters can track how often retransmissions happen, with eBPF we can classify what generated the retransmission (timeout? duplicate ACK?) what was retransmitted (a SYN? SYN/ACK? a tail loss probe?), and which services were the endpoints of these retransmissions. We observed both SYN retransmissions toward the Database server and SYN/ACKs in return.

The issue was that while SYNs and data packets were marked as being ECN-capable, potentially being marked, the listening socket did not mark the SYN/ACK as being ECN-capable, directing it into the Cubic connection packet drop profile. With DCTCP acting aggressively due to the too-high effective threshold, DCTCP traffic from—potentially from other hosts in the MH NIC—would fill the space between its mark threshold and the Cubic drop threshold, causing the "Cubic" seeming SYN/ACK to be discarded.

We were fortunate that this problem affected this Database client, which had a hard-coded application level timeout just long enough that SYNs and SYN/ACKs would be retransmitted. With that information, it was easy to find endpoints that saw SYN/ACK loss. Many other services abandon connections after a short timeout, and thus do not retransmit SYN packets; such services also reuse connections, making them less sensitive to problems in the three way handshake.

Enabling the queue-per-host feature immediately solved

this problem, though we also prepared BPF filters that would look at each packet to ensure that DCTCP-negotiating SYN/ACK packets would be ECT marked, just in case.

4.5 Experience with different switch ASICs

We discussed in §4.3 how we tuned our ECN and Droptail thresholds for the ToR layer. At the time of initial deployment, the vast majority of ToR switches had the same ASIC, which we refer to as Asic-A1. However, our production network continued to evolve, with newer ASICs from the same manufacturer (Asic-A2), and a new ASIC from a new vendor (Asic-B). There are two major issues to consider about switch ASICs when we rely on them for congestion marking: buffer sizing and ECN implementation.

In our case, initially, the newer ASICs, Asic-A2 and Asic-B inherited the thresholds that we developed for Asic-A1. Asic-A2 had the same architecture as Asic-A1, though it had 4x the buffers. With host NIC speeds also evolving, the newer switches also served faster NICs (2-4x). Put together, although the original thresholds were not always optimal, they still resulted in reasonably good performance.

However, Asic-B had a fundamentally different approach to buffer design and management making our thresholds behave differently from Asic-A family. For service operators, ToR placement is considered to be transparent: however, with Asic-B, performance was potentially now dependent on the ToR hardware; we explain the differences in ASIC architecture that proved consequential for threshold tuning next.

Queue management. Asic-B used separate Virtual Output Queues (VOQs) for ECN-Capable Transport (ECT) traffic and non-ECT traffic, which facilitated isolating these two classes of traffic, even when they are destined to the same host. For Asic-B, we could separate out DCTCP and Cubic buffer threshold tuning as two independent problems. However, this raised unexpected, somewhat intractable issues due to how the shared buffer was allocated in Asic-B.

Buffer allocation. Asic-B's shared buffer space is divided into separate "slices", and buffer thresholds are applied *independently* in each slice. This is in contrast to Asic-A family where the shared buffer pool is broken down into ingress traffic managers (ITMs), but the buffer threshold for a specific VOQ is applied as the sum of buffer to that VOQ across all the ITMs. This distinction was particularly important for any traffic pattern more than a single flow: any such traffic could end up consuming effectively more buffer space on Asic-B–across all slices–compared to Asic-A family for the same buffer thresholds. Needless to say, naively reducing thresholds by a factor of number of slices was not an option because that would affect individual flows that get mapped to a single slice when we did not have incast.

Quantized thresholds. Further complicating threshold deployment, the Asic-B architecture used "quantized" regions, resulting in a small number of actual threshold values, which meant that it did not support arbitrary thresholds, and any configured threshold between two quantized values would be applied on the lower value. For example, if the quantized regions are at 100KB and 200KB, the configuration would accept 120KB, but it would actually apply the threshold at 100KB. This quantized buffer management reduced the effective parameter space; although this had the potential to simplify search, it also reduced tuning flexibility.

Quantized drop probabilities. Similar to thresholds, Asic-B also has quantized drop probabilities (important for schemes such as Drop Tail and WRED. This made it hard to model WRED's performance in our fleet as it was unclear if WRED would perform consistently for different sets of flows with the same thresholds and drop probabilities.

Drop decisions. Although all our ASICs have a notion of dynamically shared buffers, Asic-A family and Asic-B use very different logic to share available buffer. The Asic-A family use the α parameter to share available buffer, while Asic-B uses a function based on the total buffer use, the VOQ size, and the delay experienced by the last packet sent from the VOQ. These values were quantized and used to index into a lookup table. Compared to the Asic-A family's single-parameter shared-buffer model, this mechanism is substantially more complicated, with many knobs to tune. This was further compounded by the fact that the value of these knobs were not always known to us.

These differences are irreconcileable—it is impossible to guarantee that a specific traffic pattern will see the same buffer and marking or drop probabilities across all platforms, or even just across Asic-A2 and Asic-B, which we consider to be generationally equivalent. Even with only a single parameter to tune—ECN—the above challenges underscore the difficulty and complexity of parameter tuning, particularly for more complex parameter-sensitive protocols, in an increasingly heterogeneous network similar to ours.

5 Visibility for Operations

We approached visibility from two perspectives. First, we looked at metrics from each of the layers in the network—from switches and their counters through to services and their query times. Second, we looked at covering different time scales—fine-grained debugging at RTT scale, looking at packet traces, out to long term trends in network metrics such as retransmissions. Visibility and monitoring are important components to any deployment effort. We needed to ensure that our regular monitoring systems were enhanced to account for CCA where possible, and DCTCP-specific counters, such as packet counts with ECT/CE bits set.

5.1 eBPF for monitoring

We used eBPF-based instrumentation extensively to monitor the DCTCP deployment. Similar to our enablement efforts, we found that kernel-maintained counters are not always enough: they do not separate bytes sent with DCTCP and with Cubic, or in-region and cross-region traffic. Similarly, counters of retransmissions have major limitations, as we describe later.

Each TCP connection has a field that stores the CCA for that connection. This allows us to track, using our fbflow [25] packet sampling implementation, the specific CCA that governed that transmission. With this, we can quickly confirm that in-region flows are using DCTCP when we expect them to, and that the overall bit rate of in-region traffic is about the same before and after enablement.

We also instrument and log retransmissions with an eBPF-based system that traps calls to the `tcp_retransmit_skb` function, and annotates the retransmission event with the type (timeout, fast, syn, and synack), CCA, information about the endpoints and the services involved. The CCA field doesn't always have a well defined answer, since a SYN packet can be retransmitted before ECN capability has been negotiated.

5.2 The puzzle of more retransmissions

We observed unexpected increases in retransmissions, both in kernel netstat counters (RetransSegs), and in our eBPF-based pipeline. Packet discard counters at switches were down; so why would the kernel need to retransmit more often?

This increase in retransmissions turned out to be a result of tail loss probes (TLP) [11]. TLP is a means of guarding a TCP connection from a packet loss that otherwise needs a complete RTO to recover. The sender eagerly retransmits as soon as an ACK is overdue, in order to repair the missing packet, to receive an ACK that identifies the missing packet, or to confirm that the original was delivered.

The algorithm for deciding when to send a TLP implemented in Linux is to set a timer after each transmission, set to expire after two times the RTT plus two "jiffies" (i.e., milliseconds when the constant HZ is 1000). We observed that DCTCP reduced queueing and RTTs; on hosts that were somewhat busy and needed a couple milliseconds to answer a query, this reduction was enough to shift the connection from not seeing a TLP (a 3ms RTT would lead to an 8ms TLP timer, and 8ms was plenty to generate the response) to seeing TLPs frequently (a 0ms RTT leads to a 2ms TLP timer). Although there is a counter of transmitted TLPs, it includes both new data and retransmissions, since both can be used in a TLP; the count of necessary retransmissions is not easily recovered.

We adjusted our instrumentation to identify this class of retransmission, allowing us to largely ignore them to focus instead on other retransmissions when debugging performance. TLPs may be wasteful in the common case of a DCTCP connection, where losses are infrequent and RTTs short, but we have not yet experimented with disabling it; the overhead of sending the TLP is low and it may help in certain situations.

5.3 Metrics we monitored for sanity checks

We also monitored existing network metrics that tell us the network state of the fleet. These included metrics from the hosts (e.g., throughput, socket counts, RTOs, TCP memory, CPU utilization), from switches (link utilizations, buffer utilization, congestion discards, queue lengths). To this existing set of metrics, we also added ECT and CE marked packets. These data provided us with baseline assurances that DCTCP was not unnecessarily throttling links, and that it was indeed reducing buffer utilization and packet discards in switches, and that the ECN signaling was working as expected.

We also focused on service monitoring identifying top services in the region of rollout and proactively alerting their oncalls to the rollout. In addition we worked on aggregating metrics to allow both problem identification and the ability to dig into them. The aggregations allowed users to go from a single host to a service to the entire region to see what the scope of an anomaly was and vice-versa from a region level anomaly to a host facing the issue.

Another effort we undertook was to identify if the network improvements were attributable only to DCTCP rollout or some other parallel network change caused by say a higher surge or users in the region. To perform this we created a background signal using all the non-rollout regions and compared it with the signal from the rollout one. We were able to ascertain a statistically significant correlation between the rollout of DCTCP and the improvements in the metrics.

5.4 Metrics that helped us troubleshoot issues

Troubleshooting performance is important—whenever a change of this magnitude is made to the network, any performance degradation seen by services are attributed to the network, whether deserving or not. In such scenarios, the ability to confirm that the network is at fault, or not, can make the difference between a successful versus an unsuccessful deployment. There were several metrics that helped us troubleshoot issues, and blame our rollout as appropriate. We list two in this section.

Connection set up failures: For the Database issue we discussed in 4.4.1, we noticed that connection set up failures spiked at the time of the rollout. This metric was a fleet-wide existing counter; it eventually led us to the root cause, when we saw with the retransmissions data (5.1) that SYNs were affected. Ultimately, we needed `tcpdump` to identify that the ECT bits were missing in the SYNACK.

Hardware tagging: During the initial wave of rollouts, we started noticing that a particular service in a few regions were seeing increased fast retransmissions and timeouts. This led

us to initially wonder if the service was bursty, and whether DCTCP was unable to handle the bursty traffic. However, NIC vendor tagging in our retransmissions data isolated the issue to only one particular vendor, and service tagging in the same dataset told us that other services were affected as well, just not to the same extent. We also built tools for burst visibility that helped us root cause the issue to a driver bug.

The general takeaway is that in a vast and heterogeneous network such as ours, we need extensive monitoring of a variety of network metrics. Even if most of them are not used day-to-day, or point to symptoms rather than the cause, they help isolate the issue and focus the effort to root cause, saving hours or more of engineer time.

6 Kernel and driver trouble

DCTCP exposes a set of interactions with features in the kernel and NIC driver that can lead to undesirably poor or uneven performance. At a high level, the smaller congestion windows of DCTCP mean that CPU-efficiency techniques for segmentation offload ("TSO") and reassembly offload ("GRO") behave a little differently, perhaps adding delay waiting for more data to work with, or simply requiring more operations to send the same number of packets.

Others have noticed performance issues that result from interactions between the kernel's typical use of Cubic, with large windows and large backoff, and DCTCP. For example, Misund [23] notes an interaction between DCTCP, proportional rate reduction [10] and segmentation offload.

6.1 Delayed ACKs

DCTCP appeared in the Linux kernel in 2014 [7], with substantial fixes to delayed ACK handling in 2018. The central bug was that when the sender has a congestion window of 1, the receiver did (but should not) delay its ACK [9]. The delayed ack timeout was 40*ms* by default, resulting in connection having a CWND of 1 stalling that long. We had to backport this change to a significant set of hosts running an older kernel. Although we try to upgrade to the newest kernels whenever possible, sometimes there are specific regressions or driver issues that give older kernels extended life.

6.2 GRO creates unfairness

In addition to issues with delayed acks, a certain vendor NIC delayed delivering packets that it expected to be able to reassemble, and this delay led to wild imbalance in throughput in small scale testing. In particular, an established test flow would reach 91% of link rate, while a second flow would only get 2%. Of course, the fraction of packets being ECN marked was comparable, so one would expect the two flows to converge as they would on a different NIC. With much testing, sending small RPCs that were not delayed, and `tcpdump` at both ends, we found that the NIC was applying the following rule for its GRO. The NIC would deliver if it could reassemble ten packets, if it saw a push bit, or after 1 millisecond.

In practice, this rule meant that flows with a CWND below ten packets would see an extra millisecond added to their delay, and flows having larger windows would not. This GRO rule probably didn't affect long RTT flows, where the 1ms timer was relatively small. But inside the same datacenter, this is much larger than the base RTT. This difference in effective RTT reinforced the unfair distribution of bandwidth between them. To fix, we had to disable hardware GRO for this NIC. There were other alternatives (e.g., to force setting push bits on segments that would not otherwise merit them), but the complexity did not seem worth it.

6.3 New eBPF

In Section 3, we described how BPF provided our best means to express policy about which connections should use DCTCP. However, this left us some additional problems.

First, we had to fix issues with getting ECT marked on SYN/ACKs based on the decision to use DCTCP. This was possible (the decision is made by the BPF code before the SYN/ACK is sent) but was not the default behavior.

Second, to give more flexibility in how DCTCP adapts to different signals, we reimplemented DCTCP in BPF. New features in Linux allow BPF-based congestion control, and we can use the same logic as before to attach a BPF congestion control algorithm to a new socket. However, we also want to be able to upgrade the BPF-based congestion control algorithm "on the fly," replacing the algorithm used by an existing connection. Although it isn't practical to "upgrade" Cubic to DCTCP (if ECN wasn't negotiated, the signal won't be there), replacing one "version" of DCTCP with another allows us to keep fewer versions in use. The key feature here is "bpf-iter," which allows running a loop over all sockets in the system. With this loop, we can replace the congestion control algorithm on every active socket. This is far better than alternatives (drain a datacenter, terminate connections, or wait until all the old connections disappear).

Implementing eBPF CCA. We leverage `struct-ops` [19], an eBPF interface to implement DCTCP through specific kernel function pointers, to create an eBPF program that provides a `tcp-congestion-ops` structure [1] implementation to the TCP subsystem. This capability allows us to manage CCAs similar to all the other eBPF programs we already manage in the fleet. Our DCTCP eBPF implementation closely matches the kernel eBPF example [18].

Managing eBPF CCAs. We built *NetEdit* [16], an agent that orchestrates the composition, deployment and life-cycle management of network eBPF programs across our fleet of servers. NetEdit supports implementation, experimentation, testing and rollout of custom CCAs. This allows us to select

fleetwide defaults for different RTT and also run active experiments at desired scales (specific services or data center). We push a new version of NetEdit almost every week. This allows fast iteration on CCA changes.

7 Ongoing Work, Limitations, Enhancements

This paper up to this point has been primarily about enabling DCTCP and making it work for in-region traffic, and enabling ECN on ToR-switch downlinks. This was a large first step. In this section, we describe some of the follow-on steps: enhancements we made to signaling, and CCA development based on our experience with the deployment. We also list the limitations of DCTCP, in particular for our traffic, and ongoing and future work to mitigate the limitations.

7.1 ECN marking on other hotspots

Our focus on marking from the ToR down ignored all the other links in the network maximizing benefit by targeting where most congestion occurs: most services overload their inbound network connection. However, there were a few cases where the ToR downlink was not the major bottleneck.

The first instance was when the ToR saturated its uplinks; we saw this situation in cases where there were several write-heavy services concentrated on racks, or when the rack did not have its entire capacity available due to maintenance. For such cases, we enabled ECN on the ToR link uplinks

The second instance was when ToR downlink congestion bled over up into the fabric: this happened when there were several read-heavy services concentrated on a rack, with their incoming traffic bursts synchronized at millisecond timescales. This resulted in high contention in both the ToR buffer, as well as the fabric switch immediately uplink of the ToR. To mitigate these cases, we deployed ECN on the fabric switch down links. For both this case as well as the ToR uplinks, we reused the original ECN/DropTail thresholds, which worked well enough. We saw a reduction in uplink queue length and buffer watermark on the switches as well as incoming retransmissions on the hosts.

The third—and surprising—instance was when we saw significant packet discards on the host NIC. This happened on the newer generation faster NICs; our hypothesis is that the host CPU is unable to keep up with faster bursts, resulting in the local NIC buffer overflowing. ECN on the NIC buffer is available to us on a subset of our vendor NICs, however, only one vendor implementation allows us to turn on this feature without rebooting the NIC. None of the vendors provided the means to tune the threshold—in fact, even the marking thresholds are not public information. These limitations meant that we could not deploy NIC ECN marking; however, limited testing showed promise in reducing NIC drops when we enabled it on one vendor NIC with just the default threshold.

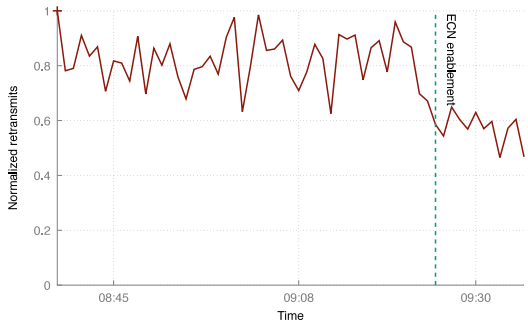

Figure 4: ECN marking on NICs reduces retransmissions.

We tested this feature on one host that was experiencing substantial NIC buffer drops. Turning on this feature reduced incoming retransmissions by about 30% (Figure 4). As host NICs become faster, challenging the CPU's ability to keep up with the incoming traffic, we expect that host-network congestion will become a larger problem for us to resolve.

7.2 Limitations of DCTCP

DCTCP does not solve all congestion issues. Figure 1 showed that DCTCP significantly reduced retransmission rates; it also shows that the gains are uneven across regions, and there is a significant amount of retransmissions that DCTCP is unable to solve. Some of it was due to other hotspots in the network and NIC that we observed and in some cases fixed (Section 7.1). A significant amount, however, is caused by traffic and network characteristics, which DCTCP, in its current form, is unable to solve. Our traffic is characterized by *short* and *heavy* incast bursts [13]. Short bursts do not allow the incast senders to converge, resulting in high buffer usage even with ECN. Heavy of the incast means that a CWND of 1, the lowest DCTCP can maintain, is too high. Other work has tackled high-degree incast [2, 14, 17, 21]; they require enhanced hardware support and a new networking stack. Our bursts can also synchronize across hosts in a ToR: with our ToR buffers being shallow and dynamically shared, synchronized bursts result in contention and high buffer pressure on the ToR, meaning that bursts can receive variable buffer allocation depending on the degree of synchronization.

Although DCTCP by itself is unable to solve the above challenges, we are working on complementary systems and better CCAs that can. One area of ongoing work is receiver-based flow control to help senders converge faster and more reliably [22]; this has resulted in significant wins. Buffer tuning to manage contention is a promising area of research as well. Ultimately though, ECN is a coarse signal which has its limitations; we are working on using delay-based signals that offer more responsive congestion control.

Jumbo frames. Jumbo frames are more efficient on the network; however, enabling Jumbo faced challenges to enabling DCTCP—different kernels and NICs support larger packets

differently; there are bugs in the handling of, say, large TCP fast open SYN packets or in handling a mixed configuration of MTU sizes on MH NICs. We note that open questions exist on how jumbo and non-jumbo frames interact, and whether our empirically derived thresholds need revisiting as frames are larger. We leave that for future work.

7.3 DCTCP implemented in BPF

We reimplemented DCTCP using the recent "bpf-cc" system, a task made a bit easier by the software that used BPF to enable DCTCP. This alternate implementation of the same CCA allows us to make modifications both minor—such as experimenting with different parameters—and major—using the established framework of DCTCP's response to ECN as a foundation, but building-in a response to delay or loss that differs slightly for our environment. Finally, our BPF implementation of DCTCP allows us to log internal CCA state (e.g., to log cwnd every RTT) rather than infer it from packet capture analysis.

We ran extensive tests comparing in-BPF to in-kernel implementations of DCTCP, using long flows as well as small and large RPCs. CPU use at sender and receiver was about the same (perhaps dominated by other parts of packet processing). Both implementations were fair to each other, getting roughly the same throughput alone and in contention.

8 Takeaways and Conclusion

We leave the community with a few observations about *deploying* CCAs in a large scale production network, which we hope will influence CCA research.

Deploying a CCA in production is not a flip of a switch. Safely and incrementally deploying changes leads to a transition period where there is a mixture of CCAs in use. This is not just due to hosts and network devices that have not picked up the changes, but also due to existing connections that have not (and might not be able to) flip over. The resultant transition period could vary depending on the complexity of the switch as well as the nature of the connections / traffic. This means that we have to consider performance during transition as well—if the stable states has excellent performance, but the (long) transition period could have significantly degraded performance, the switch will be more complex. Much of CCA research focuses on the stable state after the transition; insights into how the transition period could affect performance would be immensely useful to plan the deployment.

Data centers are complex and heterogeneous. CCAs must be simple, and forgiving. The mixture of hardware (NIC vendors, NIC speeds, switch vendors, switch ASICs, queues), software (OS kernels, driver versions), and applications (bursty, variable RTT, latency/throughput/tail sensitive), and the combinations thereof can be daunting—making any planned deployment a logistical challenge. While it may be impossible to account for every eventuality prior to deployment (indeed, we discovered a good number of issues only in retrospect), simpler CCAs can be easier to reason about and plan for. This means that new requirements either in the network (ECN, network telemetry) or in the host (hardware timestamps) need to be as minimal and simple as possible. The *tuning* of the new features must also be as forgiving as possible. Much work has gone into identifying ideal ECN thresholds—however, those assume ideal cases where there is no sharing with other CCAs, and the threshold search space is continuous, and not quantized. We were unable to deploy those ideal thresholds, instead having to do extensive testing to find "good-enough" thresholds that resulted in reasonable, though not ideal performance. A CCA that relies heavily on tuned parameters without graceful degradation is harder to deploy successfully in a large scale data center.

Expecting the unexpected. Sometimes long-deployed (and forgotten) configurations or optimizations can be exposed with new CCAs. An ideal CWND size that is only large enough to make full theoretical use of the network link, for example, might not be large enough to trigger the NIC to deliver a reassembled collection of packets, resulting in increased latency, or worse, breaking fairness. Simplicity of CCAs can also reduce probability of bad interactions with other components such as host and NIC optimizations, but it may not be possible to account for every eventuality.

Hotspots may occur in unexpected places. CCAs must have good fallbacks. CCAs moderate how concurrent flows share a *known* bottlenecked resource, but the location of the bottleneck (in-network, host-side, multi-host NICs) is not necessarily clear. A bottleneck in an unexpected location, which is not amenable to deploying the signal that the CCA relies on can be problematic. For example, when we found NIC bottlenecks, we realized that we could not deploy ECN there; therefore packet loss in NICs continued to occur, with DCTCP reacting suboptimally to such losses, being designed to respond optimally to ECN and not loss.

Ultimately, a CCA that might work well analytically and in simulation might not work well in practice—we hope that our experience guides researchers avoid common pitfalls, and design CCAs with an eye towards real-world deployability. Our experience with DCTCP has also guided our own evaluation of the potential of more advanced CCAs with reliance on wider set of signals: in-network telemetry, fine grained hardware timestamps, or early congestion signaling from switches.

Acknowledgments. This paper presents the work of several teams at Meta involved in successfully testing and deploying DCTCP. We thank Andrey Ignatov, Igor Pozgaj, James Zeng, Kiran Palan, Luwei Cheng, Mario Sanchez, Martin Lau, Nivin Lawrence, Omar Baldonado, Petr Lapukhov, Rajiv Krishnamurthy, Rob Sherwood, Rohit Puri, Russell Cloran, Sankaralingam Panneerselvam, Srikrishna Gopu, Takshak Chahande and the NSDI reviewers for their insightful feedback.

References

[1] tcp-congestion-ops. https://elixir.bootlin.com/linux/v5.5/source/include/net/tcp.h#L1043. [Online; accessed 25-February-2024].

[2] ABDOUS, S., SHARAFZADEH, E., AND GHORBANI, S. Burst-tolerant datacenter networks with Vertigo. In *CoNEXT* (2021).

[3] AHUJA, S. S., DANGUI, V., PATIL, K., SOMASUNDARAM, M., GUPTA, V., SANCHEZ, M., YAN, G., NOORMOHAMMADPOUR, M., RAZMJOO, A., SMITH, G., ET AL. Network entitlement: contract-based network sharing with agility and SLO guarantees. In *SIGCOMM* (2022).

[4] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *SIGCOMM* (2010).

[5] ARSLAN, S., LI, Y., KUMAR, G., AND DUKKIPATI, N. Bolt:sub-rtt congestion control for ultra-low latency. In *NSDI* (2023).

[6] BIRO, R., VAN KEMPEN, F. N., EVANS, M., MINYARD, C., LA ROCHE, F., HENDRICK, C., TORVALDS, L., COX, A., DILLON, M., GULBRANDSEN, A., AND CWIK, J. tcp_output.c. https://elixir.bootlin.com/linux/latest/source/net/ipv4/tcp_output.c#L339. [Online; accessed 3-September-2023].

[7] BORKMANN, D., WESTPHAL, F., AND JUDD, G. tcp_dctcp.c. https://elixir.bootlin.com/linux/latest/source/net/ipv4/tcp_dctcp.c#L100. [Online; accessed 3-September-2023].

[8] BRAKMO, L. bpf: BPF support for socket ops. https://lwn.net/Articles/725722/, 2017. [Online; accessed 3-September-2023].

[9] BRAKMO, L., BURKOV, B., LECLERCQ, G., AND MUGAN, M. Experiences evaluating DCTCP. In *Linux Plumbers Conference* (2018).

[10] DUKKIPATI, N., MATHIS, M., CHENG, Y., AND GHOBADI, M. Proportional rate reduction for TCP. In *SIGCOMM* (2011).

[11] FLACH, T., DUKKIPATI, N., TERZIS, A., RAGHAVAN, B., CARDWELL, N., CHENG, Y., JAIN, A., HAO, S., KATZ-BASSETT, E., AND GOVINDAN, R. Reducing web latency: the virtue of gentle aggression. In *SIGCOMM* (2013).

[12] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT* (2015).

[13] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A microscopic view of bursts, buffer contention, and loss in data centers. In *IMC* (2022).

[14] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM* (2017).

[15] JUDD, G. Attaining the promise and avoiding the pitfalls of TCP in the datacenter. In *NSDI* (2015).

[16] KANNAN, P., AND GUPTA, P. Netedit: Fine-grained network tuning at scale. https://atscaleconference.com/videos/netedit-fine-grained-network-tuning-at-scale-prashant 2022. [Online; accessed 25-February-2024].

[17] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., ET AL. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM* (2020).

[18] LAU, M. K. bpf: Add bpf-dctcp example. https://lore.kernel.org/all/20191221062620.1184118-1-kafai@fb.com/, 2019. [Online; accessed 25-February-2024].

[19] LAU, M. K. Introduce BPF STRUCT-OPS. https://lore.kernel.org/bpf/20200109003453.3854769-1-kafai@fb.com/, 2020. [Online; accessed 25-February-2024].

[20] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., ET AL. HPCC: High precision congestion control. In *SIGCOMM* (2019).

[21] LIU, K., TIAN, C., WANG, Q., ZHENG, H., YU, P., SUN, W., XU, Y., MENG, K., HAN, L., FU, J., ET AL. Floodgate: Taming incast in datacenter networks. In *CoNEXT* (2021).

[22] MADHAVAN, B., AND DHAMIJA, A. Tackling dc congestion and bursts. https://atscaleconference.com/videos/netedit-fine-grained-network-tuning-at-scale-prashant 2022. [Online; accessed 25-February-2024].

[23] MISUND, J., AND BRISCOE, B. Disentangling flaws in linux DCTCP. *arXiv preprint arXiv:2211.07581* (2022).

[24] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM* (2018).

[25] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[26] VEERARAGHAVAN, K., MEZA, J., MICHELSON, S., PANNEERSELVAM, S., GYORI, A., CHOU, D., MARGULIS, S., OBENSHAIN, D., PADMANABHA, S., SHAH, A., ET AL. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *OSDI* (2018).

[27] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-resolution measurement of data center microbursts. In *IMC* (2017).