Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center

Mohammad Alizadeh, Abdul Kabbani[†], Tom Edsall[⋆], Balaji Prabhakar,
Amin Vahdat[†§], and Masato Yasuda[¶]
Stanford University [†]Google [⋆]Cisco Systems [§]U.C. San Diego [¶]NEC Corporation, Japan

Abstract

Traditional measures of network goodness—goodput, quality of service, fairness—are expressed in terms of bandwidth. Network latency has rarely been a primary concern because delivering the highest level of bandwidth essentially entails driving up latency—at the mean and, especially, at the tail. Recently, however, there has been renewed interest in latency as a primary metric for mainstream applications. In this paper, we present the HULL (High-bandwidth Ultra-Low Latency) architecture to balance two seemingly contradictory goals: near baseline fabric latency and high bandwidth utilization. HULL leaves 'bandwidth headroom' using Phantom Queues that deliver congestion signals before network links are fully utilized and queues form at switches. By capping utilization at less than link capacity, we leave room for latency sensitive traffic to avoid buffering and the associated large delays. At the same time, we use DCTCP, a recently proposed congestion control algorithm, to adaptively respond to congestion and to mitigate the bandwidth penalties which arise from operating in a bufferless fashion. HULL further employs packet pacing to counter burstiness caused by Interrupt Coalescing and Large Send Offloading. Our implementation and simulation results show that by sacrificing a small amount (e.g., 10%) of bandwidth, HULL can dramatically reduce average and tail latencies in the data center.

1 Introduction

For decades, the primary focus of the data networking community has been on improving overall network goodput. The initial shift from circuit switching to packet switching was driven by the bandwidth and hardware inefficiencies of reserving network resources for bursty communication traffic. The Transmission Control Protocol (TCP) [26] was born of the need to avoid bandwidth/congestion collapse in the network and, subsequently, to ensure bandwidth fairness [14, 39, 45] among the flows sharing a network. Discussion to add quality-of-service capability to the Internet resulted in proposals such as RSVP [55], IntServ [10] and DiffServ [35], which again focussed on bandwidth provisioning.

This focus on bandwidth efficiency has been well justified as most Internet applications typically fall into two categories. Throughput-oriented applications, such as file transfer or email, are not sensitive to the delivery times of individual packets. Even the overall completion times of individual operations can vary by multiple integer factors in the interests of increasing overall network throughput. On the other hand, latency-sensitive applications—such as web browsing and remote login—are sensitive to per-packet delivery times. However, these applications have a human in the loop and completion time variations on the order of hundreds of milliseconds or even seconds have been thought to be acceptable, especially in the interests of maintaining high average bandwidth utilization. Hence, we are left with a landscape where the network is not optimized for latency or the predictable delivery of individual packets.

We are motivated by two recent trends that make it feasible and desirable to make low latency communication a primary metric for evaluating next-generation networks.
Data centers. A substantial amount of computing, storage, and communication is shifting to data centers. Within the confines of a single building—characterized by low propagation delays, relatively homogeneous equipment, and a single administrative entity able to modify software protocols and even influence hardware features—delivering predictable low latency appears more tractable than solving the problem in the Internet at large.
Ultra-low latency applications. Several applications and platforms have recently arisen that necessitate very low latency RPCs; for example, high-frequency trading (see [30]), high-performance computing, and RAMCloud [37, 38]. These applications are characterized by a request–response loop involving machines, not humans, and operations involving multiple parallel requests/RPCs

to thousands of servers. Since an operation completes when all of its requests are satisfied, the tail latency of the individual requests are required to be in microseconds rather than in milliseconds to maintain quality of service and throughput targets. As platforms like RAMCloud are integrated into mainstream applications such as social networking, search and e-commerce, they must share the network with throughput-oriented traffic which consistently moves terabytes of data.

There are several points on the path from source to destination at which packets currently experience delay: end-host stacks, NICs (network interface cards), and switches. Techniques like kernel bypass and zero copy [44, 12] are significantly reducing the latency at the end-host and in the NICs; for example, 10Gbps NICs are currently available that achieve less than 1.5μs per-packet latency at the end-host [41].

In this paper, we consider the latency in the network switching nodes. We propose HULL (for High-bandwidth Ultra-Low Latency), an architecture for simultaneously delivering predictable ultra-low latency and high bandwidth utilization in a shared data center fabric. The key challenge is that high bandwidth typically requires significant in-network buffering while predictable, ultra-low latency requires essentially no in-network buffering. Considering that modern data center fabrics can forward full-sized packets in microseconds (1.2μs for 1500 bytes at 10Gbps) and that switching latency at 10Gbps is currently 300–500ns [19, 23], a one-way delivery time of 10μs (over 5 hops) is achievable across a large-scale data center, *if queueing delays can be reduced to zero*. However, given that at least 2MB of on-chip buffering is available in commodity 10Gbps switches [19] and that TCP operating on tail-drop queues attempts to fully utilize available buffers to maximize bandwidth, one-way latencies of up to a few milliseconds are quite possible—and have been observed in production data centers.[1] This is a factor of 1,000 increase from the baseline. Since the performance of parallel, latency-sensitive applications are bound by tail latency, these applications must be provisioned for millisecond delays when, in fact, microsecond delays are achievable.

Our observation is that it is possible to reduce or eliminate network buffering by marking congestion based not on queue occupancy (or saturation) but rather based on the utilization of a link approaching its capacity. In essence, we cap the amount of bandwidth available on a link in exchange for significant reduction in latency.

Our motivation is to trade the resource that is relatively plentiful in modern data centers, i.e., bandwidth, for the resource that is both expensive to deploy and results in substantial latency increase—buffer space. Data center switches usually employ on-chip (SRAM) buffering to keep latency and pin counts low. However, in this mode, even a modest amount of buffering takes about 30% of the die area (directly impacting cost) and is responsible for 30% of the power dissipation. While larger in size, off-chip buffers are both more latency intensive and incur a significantly higher pin count. The references [5, 25] describe the cost of packet buffers in high bandwidth switching/routing platforms in more detail. The above considerations indicate that higher bandwidth switches with more ports could be deployed earlier if fewer chip transistors were committed to buffers.

The implementation of HULL centers around Phantom Queues, a switch mechanism closely related to existing virtual queue-based active queue management schemes [21, 31]. Phantom queues simulate the occupancy of a queue sitting on a link that drains at *less than* the actual link's rate. Standard ECN [40] marking based on the occupancy of these phantom queues is then used to signal end hosts employing DCTCP [3] congestion control to reduce transmission rate.

Through our evaluation we find that a key requirement to make this approach feasible is to employ hardware packet pacing (a feature increasingly available in NICs) to smooth the transmission rate that results from widespread network features such as Large Send Offloading (LSO) and Interrupt Coalescing. We introduce innovative methods for estimating the congestion-friendly transmission rate of the pacer and for adaptively detecting the flows which require pacing. Without pacing, phantom queues would be fooled into regularly marking congestion based on spurious signals causing degradation in throughput, just as spikes in queuing caused by such bursting would hurt latency.

Taken together, we find that these techniques can reduce both average and 99th percentile packet latency by more than a factor of 10 compared to DCTCP and a factor of 40 compared to TCP. For example, in one configuration, the average latency drops from 78μs for DCTCP (329μs for TCP) to 7μs and the 99th percentile drops from 556μs for DCTCP (3961μs for TCP) to 48μs, with a configurable reduction in bandwidth for throughput-oriented applications. A factor of 10 reduction in latency has the potential to substantially increase the amount of work applications such as web search perform—e.g., process 10 times more data with predictable completion times—for end-user requests, though we leave such exploration of end-application benefits for future work.

2 Challenges and design

The goal of the HULL architecture is to *simultaneously deliver near baseline fabric latency and high throughput*. In this section we discuss the challenges involved in

[1] For example, queuing delays in a production cluster for a large-scale web application have been reported to range from $\sim$350μs at the median to over 14ms at the 99th percentile (see Figure 9 in [3]).

achieving this goal. These challenges pertain to correctly *detecting, signaling* and *reacting* to impending congestion. We show how these challenges guide our design decisions in HULL and motivate its three main components: *Phantom queues, DCTCP congestion control, and packet pacing*.

2.1 Phantom queues: Detecting and signaling congestion

The traditional congestion signal in TCP is the drop of packets. TCP increases its congestion window (and transmission rate) until available buffers overflow and packets are dropped. As previously discussed, given the low inherent propagation and switching times in the data center, this tail-drop behavior incurs an unacceptably large queuing latency.

Active queue management (AQM) schemes [18, 24, 6] aim to proactively signal congestion before buffers overflow. Most of these mechanisms try to regulate the queue around some target occupancy. While these methods can be quite effective in reducing queuing latency, they cannot eliminate it altogether. This is because they must observe a non-zero queue to begin signaling congestion, and sources react to these congestion signals after one RTT of lag, during which time the queue would have built up even further.

This leads to the following observation: Achieving predictable and low fabric latency essentially requires congestion signaling *before* any queueing occurs. That is, achieving the lowest level of queueing latency imposes a fundamental tradeoff with bandwidth—creating a 'bandwidth headroom'. Our experiments (§6) show that bandwidth headroom dramatically reduces average and tail queuing latencies. In particular, the reductions at the high percentiles are significant compared to queue-based AQM schemes.

We propose the Phantom Queue (PQ) as a mechanism for creating bandwidth headroom. A phantom queue is a simulated queue, associated with each switch egress port, that sets ECN [40] marks based on link utilization rather than queue occupancy. The PQ simulates queue buildup for a virtual egress link of a configurable speed, slower than the actual physical link (e.g., running at $\gamma = 95\%$ of the line rate). *The PQ is not really a queue since it does not store packets.* It is simply a counter that is updated while packets exit the link at line rate to determine the queuing that would have been present on the slower virtual link. It then marks ECN for packets that pass through it when the counter (simulated queue) is above a fixed threshold.

The PQ explicitly attempts to set aggregate transmission rates for congestion-controlled flows to be strictly less than the physical link capacity, thereby keeping switch buffers largely unoccupied. This bandwidth headroom allows latency sensitive flows to fly through the network at baseline transmission plus propagation rates.

Remark 1. The idea of using a simulated queue for signaling congestion has been used in Virtual Queue (VQ) AQM schemes [21, 31]. A key distinction is that while a VQ is typically placed in parallel to a real queue in the switch, we propose placing the PQ in *series* with the switch egress port. This change has an important consequence: the PQ can operate independently of the internal architecture of the switch (output-queued, shared memory, combined input-output queued) and its buffer management policies, and, therefore, work with *any* switch. In fact, we implement the PQ external to physical switches as a hardware 'bump on the wire' prototyped on the NetFPGA [33] platform (see §5.1). In general, of course, VQs and PQs can and have been [34] integrated into switching hardware.

2.2 DCTCP: Adaptive reaction to ECN

Standard TCP reacts to ECN marks by cutting the congestion window in half. Without adequate buffering to keep the bottleneck link busy, this conservative back off can result in a severe loss of throughput. For instance, with zero buffering, TCP's rate fluctuates between 50% and 100% of link capacity, achieving an average throughput of only 75% [51]. Therefore, since the PQ aggressively marks packets to keep the buffer occupancy at zero, TCP's back off can be especially detrimental.

To mitigate this problem, we use DCTCP [3], a recent proposal to enhance TCP's reaction to ECN marks. DCTCP employs a fixed marking threshold at the switch queue and attempts to extract information regarding the *extent* of network congestion from the sequence of congestion signals in the ACK train from the receiver. A DCTCP source calculates the fraction of packets containing ECN marks within a given window of ACKs, and reduces its window size in proportion to the fraction of marked packets. Essentially, congestion signals need not result in multiple flows simultaneously backing off drastically and losing throughput. Instead (and ideally), senders can adjust transmission rates to maintain a base, low level of queueing near the marking threshold. In theory, DCTCP can maintain more than 94% throughput even with zero queueing [4]. We refer to [3] for a full description of the algorithm.

2.3 Packet pacing

Bursty transmission occurs for multiple reasons, ranging from TCP artifacts like ACK compression and slow start [27, 56], to various offload features in NICs like Interrupt Coalescing and Large Send Offloading (LSO) designed to reduce CPU utilization [8]. With LSO for instance, hosts transfer large buffers of data to the NIC, leaving specialized hardware to segment the buffer into

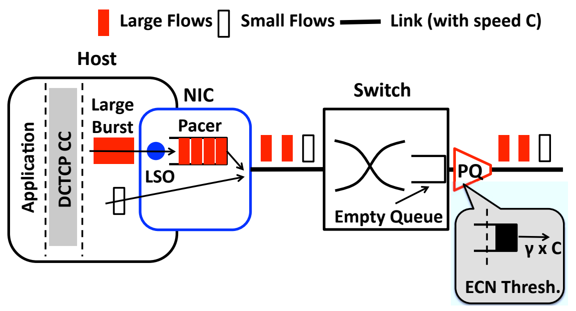


Figure 1: The HULL architecture consists of Phantom queues at switch egress ports and DCTCP congestion control and packet pacers at end-hosts.

individual packets, which then burst out at line rate. As our experiments show (§4.1), Interrupt Coalescing and LSO are required to maintain acceptable CPU overhead at 10Gbps speeds.

Traffic bursts cause temporary increases in queue occupancies. This may not be a big problem if enough buffer space is available to avoid packet drops and if variable latency is not a concern. However, since the PQ aggressively attempts to keep buffers empty (to prevent variable latency), such bursts trigger spurious congestion signals, leading to reduced throughput.

A natural technique for combating the negative effects of bursty transmission sources on network queueing is packet pacing. While earlier work (§7) introduce pacing at various points in the protocol stack, we find that, to be effective, pacing must take place in hardware after the last source of bursty transmission: NIC-based LSO. Ideally, a simple additional mechanism at the NIC itself would pace data transmission. The pacer would likely be implemented as a simple leaky bucket with a configurable exit rate. We describe a hardware design and implementation for pacing in §4.2.

It is paradoxical that the pacer must queue packets at the edge (end-hosts) so that queueing inside the network is reduced. Such edge queueing can actually increase end-to-end latency, offsetting any benefits of reduced in-network queueing. We resolve this paradox by noting that only packets that *belong to large flows* and hence are not sensitive to per-packet delivery times should be paced. Small latency-sensitive flows should not be paced, allowing them to exploit the lowest available fabric latency. We employ a simple adaptive end-host mechanism to determine whether a flow should be subject to pacing. This is inspired by classic work on the UNIX multi-level feedback queue that attempts to classify interactive versus bulk jobs in the operating system [46]. Newly created flows are classified as latency sensitive and initially not subjected to pacing. However, once a flow sees a sufficient number of ECN marks, it is classified as throughput-oriented and paced.

	Throughput	Mean Latency	99th Prctile Latency
TCP	982Mbps	1100.6μs	4308.8μs
DCTCP-30K	975Mbps	153.9μs	305.8μs

Table 1: Baseline throughput and latency for two long-lived flows with TCP and DCTCP-30K (30KB marking threshold).

2.4 The HULL Architecture

The complete High-bandwidth Ultra Low Latency (HULL) architecture is shown in Figure 1. Large flows at the host stack, which runs DCTCP congestion control, send large bursts to the NIC for segmentation via LSO. The Pacer captures the packets of the large flows after segmentation, and spaces them out at the correct transmission rate. The PQ uses ECN marking based on a simulated queue to create bandwidth headroom, limiting the link utilization to some factor, $\gamma < 1$, of the line rate. This ensures that switch queues run (nearly) empty, which enables low latency for small flows.

3 Bandwidth Headroom

This section explores the consequences of creating bandwidth headroom. We illustrate the role of the congestion control protocol in determining the amount of bandwidth headroom by comparing TCP and DCTCP. We then discuss how bandwidth headroom impacts the completion time of large flows.

3.1 Importance of stable rate control

All congestion control algorithms cause fluctuations in rate as they probe for bandwidth and react to delayed congestion signals from the network. The degree of these fluctuations is usually termed ‘stability’ and is an important property of the congestion control feedback system [47, 24]. Typically, some amount of buffering is required to absorb rate variations and avoid throughput loss. Essentially, buffering keeps the bottleneck link busy while sources that have cut their sending rates recover. This is especially problematic in low statistical multiplexing environments, where only a few high speed flows must sustain throughput [5].

Therefore, special care must be taken with the congestion control algorithm if we aim to reduce buffer occupancies to zero. We illustrate this using a simple experiment. We connect three servers to a single switch and initiate two long-lived flows from two of the servers to the third (details regarding our experimental setup can be found in §5). We measure the aggregate throughput and the latency due to queueing at the switch. As a baseline reference, the throughput and latency for standard TCP (with tail-drop), and DCTCP, with the recommended marking threshold of 30KB [3], are given in Table 1. As expected, DCTCP shows an order of magnitude improvement in latency over TCP, because it reacts to queue buildup beyond the marking threshold.

We conduct a series of experiments where we sweep the drain rate of a PQ attached to the congested port. The marking threshold at the PQ is set to 6KB and we also enable our hardware pacing module. The results are shown in Figure 2. Compared to the baseline, a significant latency reduction occurs for both TCP-ECN (TCP with ECN enabled) and DCTCP, when bandwidth headroom is created by the PQ. Also, for both schemes, the throughput is lower than intended by the PQ drain rate. This is because of the rate variations imposed by the congestion control dynamics. However, *TCP-ECN loses considerably more throughput than DCTCP at all PQ drain rates*. The gap between TCP-ECN's throughput and the PQ drain rate is ∼17–26% of the line rate, while it is ∼6–8% for DCTCP, matching theory quite well [4].

3.2 Slowdown due to bandwidth headroom

Bandwidth headroom created by the PQ will inevitably slow down the large flows, which are bandwidth-intensive. An important question is: *How badly will the large flows be affected?*

We answer this question using a simple queuing analysis of the 'slowdown', defined as the ratio of the completion times of a flow with and without the PQ. We find that, somewhat counter-intuitively, the slowdown is not simply determined by the amount of bandwidth sacrificed; it also depends on the traffic load.

Consider the well-known model of a M/G/1-Processor Sharing queue for TCP bandwidth sharing [20, 42]. Flows arrive according to a Poisson process of some rate, λ, and have sizes drawn from a general distribution, S. The flows share a link of capacity C in a fair manner; i.e., if there are n flows in the system, each gets a bandwidth of C/n. We assume the total load $\rho \triangleq \lambda \mathbb{E}(S)/C < 1$, so that the system is stable. A standard result for the M/G/1-PS queue states that in this setting, the average completion time for a flow of size x is given by:

$$FCT_{100\%} = \frac{x}{C(1-\rho)}, \tag{1}$$

where the '100%' indicates that this is the FCT without bandwidth headroom. Now, suppose we only allow the flows to use γC of the capacity. Noting that the load on this slower link is $\tilde{\rho} = \rho/\gamma$, and invoking (1) again, we find that the average completion time is:

$$FCT_{\gamma} = \frac{x}{\gamma C(1-\rho/\gamma)} = \frac{x}{C(\gamma-\rho)}. \tag{2}$$

Hence, dividing (2) by (1), the slowdown caused by the bandwidth headroom is given by:

$$SD \triangleq \frac{FCT_{\gamma}}{FCT_{100\%}} = \frac{1-\rho}{\gamma-\rho}. \tag{3}$$

The interesting fact is that *the slowdown gets worse as the load increases*. This is because giving bandwidth away also increases the effective load ($\hat{\rho} > \rho$). For example, using (3), the slowdown with 20% bandwidth headroom ($\gamma = 0.8$), at load $\rho = 0.2, 0.4, 0.6$ will be 1.33, 1.5, 2, (equivalently: 33%, 50%, 100%) respectively.

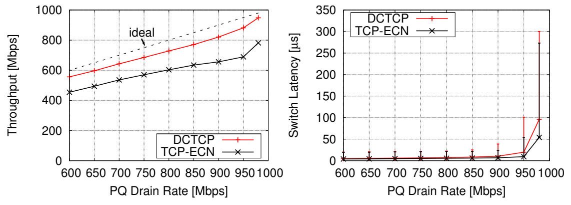


Figure 2: Throughput (left) and average switch latency (right) for TCP-ECN and DCTCP with a PQ, as drain rate varies. The vertical bars in the right plot indicate the 99th percentile.

Our experiments in §6.2 confirm the validity of this model (see Figure 10). This highlights the importance of not giving away too much bandwidth. Fortunately, as we show, *even a small amount of bandwidth headroom (e.g., 10%) provides a dramatic reduction in latency*.

Remark 2. The M/G/1-PS model provides a good approximation for large flows for which TCP has time to converge to the fair bandwidth allocation [20]. It is not, however, a good model for small flows as it does not capture latency. In fact, since the completion time for small flows is mainly determined by the latency, they are not adversely affected by bandwidth headroom (§6).

4 Pacing

4.1 The need for pacing

Modern NICs implement various offload mechanisms to reduce CPU overhead for network communication. These offloads typically result in highly bursty traffic [8]. For example, Interrupt Coalescing is a standard feature which allows the NIC to delay interrupting the CPU and wait for large batches of packets to be processed in one SoftIrq. This disrupts the normal TCP ACK-clocking and leads to many MTUs worth of data being released by TCP in a burst. A further optimization, Large Send Offloading (LSO), allows TCP to send large buffers (currently up to 64KB), delegating the segmentation into MTU-sized packets to the NIC. These packets then burst out of the NIC at line rate.

As later experiments show, this burstiness can be detrimental to network performance. However, *using hardware offloading to reduce the CPU overhead of the network stack is unavoidable as link speeds increase to 10Gbps and beyond.*

We illustrate the need for pacing using a simple experiment. We directly connect two servers with 10Gbps NICs (see §5.2 for testbed details), and enable LSO and Interrupt Coalescing with a MTU of 1500 bytes. We generate a single TCP flow between the two servers, and cap the window size of the flow such that the throughput is ∼1Gbps on average. Figure 3 shows the data and

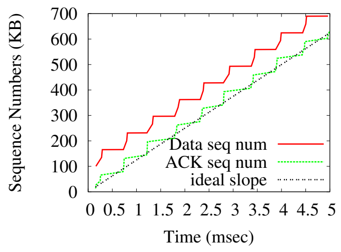


Figure 3: Burstiness with 10Gbps NICs.

Intr. Coalescing	CPU Util. (%)	Thrput (Gbps)	Ack Ratio (KB)
adaptive	37.2	9.5	41.3
rx-frames=128	30.7	9.5	64.0
rx-frames=32	53.2	9.5	16.5
rx-frames=8	75	9.5	12.2
rx-frames=2	98.7	9.3	11.4
rx-frames=0	99	7.7	67.4

Table 2: The effect of Interrupt Coalescing. Note that 'adaptive' is the default setting for Interrupt Coalescing.

ACK sequence numbers within a 5ms window (time and sequence numbers are relative to the origin) compared to the 'ideal slope' for perfectly paced transmission at 1Gbps. The sequence numbers show a step-like behavior that demonstrates the extent of burstiness. Each step, occurring roughly every 0.5ms, corresponds to a back-to-back burst of data packets totaling 65KB. Analyzing the packet trace using `tcpdump` [49], we find that the bursty behavior reflects the batching of ACKs at the receiver: Every 0.5ms, 6 ACKs acknowledging 65KB in total are received within a 24–50μs interval. Whereas, ideally, for a flow at 1Gbps, the ACKs for 65KB should be evenly spread over 520μs.

The batching results from Interrupt Coalescing at the receiver NIC. To study this further, we repeat the experiment with no cap on the window size and different levels of Interrupt Coalescing. We control the extent of Interrupt Coalescing by varying the value of `rx-frames`, a NIC parameter that controls the number of frames between interrupts. Table 2 summarizes the results. We observe a tradeoff between the CPU overhead at the receiver and the average ACK ratio (the number of data bytes acknowledged by one ACK), which is a good proxy for burstiness. Setting `rx-frames` at or below 8 heavily burdens the receiver CPU, but improves the ACK ratio. With Interrupt Coalescing disabled (`rx-frames` = 0), the receiver CPU is saturated and cannot keep up with the load. This further increases the ACK ratio and also causes a 1.8Gbps loss in throughput.

Remark 3. Enabling LSO is also necessary to achieve the 10Gbps line rate. Without LSO, the sender's CPU is saturated and there is close to 3Gbps loss of throughput.

4.2 Hardware Pacer module

The Pacer module inserts suitable spacing between the packets of flows transmitted by the server. We envision the Pacer module operating at the NIC. Pacing in hardware has a number of advantages over software pacing. A hardware pacer can easily support the sub-microsecond scheduling granularity required to pace at 10Gbps rates and beyond. Moreover, unlike pacing in the host stack that typically requires disabling segmentation offload, a hardware module in the NIC is oblivious to server LSO settings since it operates on the outgoing packet stream *after* segmentation takes place.

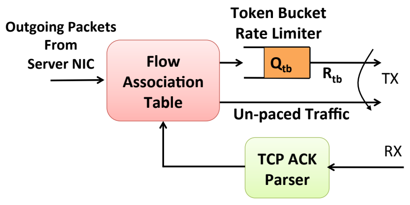


Figure 4: Block diagram of Pacer module.

Figure 4 shows the block diagram of the Pacer module. The Flow Association Table is consulted to check whether an outgoing packet requires pacing (see below). If so, the packet is placed into a token bucket rate limiter with a configurable transmission rate. Otherwise, it bypasses the token bucket and is sent immediately.

The key challenges to pacing, especially in a hardware module, are: (i) determining the appropriate pacing rate, and (ii) deciding the flows that require pacing.

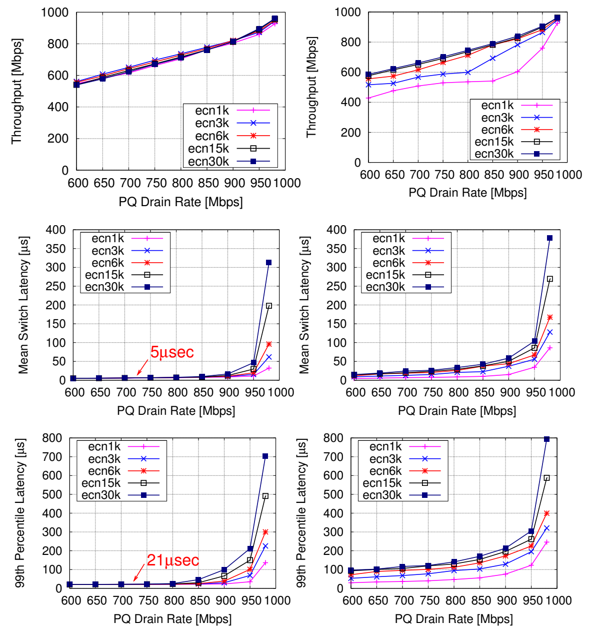
Dynamic pacing rate estimation. The NIC is unaware of the actual sending rate ($Cwnd/RTT$) of TCP sources. Therefore, we use a simple algorithm to estimate the congestion-friendly transmission rate. We assume that over a sufficiently large measurement interval (e.g., a few RTTs) each host's aggregate transmission rate will match the rate imposed by higher-level congestion control protocols such as TCP (or DCTCP). The pacer dynamically measures this rate and appropriately matches the rate of the token bucket. More precisely, every T_r seconds, the Pacer counts the number of bytes it receives from the host, denoted by M_r. It then modifies the rate of the token bucket according to:

$$R_{tb} \leftarrow (1-\eta) \times R_{tb} + \eta \times \frac{M_r}{T_r} + \beta \times Q_{tb}. \qquad (4)$$

The parameters η and β are positive constants and Q_{tb} is the current backlog of the token bucket in bytes. R_{tb} is in bytes per second.

Equation (4) is a first order low-pass filter on the rate samples M_r/T_r. The term $\beta \times Q_{tb}$ is necessary to prevent the Pacer backlog from becoming too large.[2] This is crucial to avoid a large buffer for the token bucket, which adds to the cost of the Pacer, and may also induce significant latency to the paced flows (we explore the latency of the Pacer further in §4.4).

[2] In fact, if the aggregate rate of paced flows is fixed at $R^* = M_r/T_r$, the only fixed point of equation (4) is $R_{tb} = R^*$, and $Q_{tb} = 0$.



(a) With Pacing (b) Without Pacing

Figure 5: Throughput and switch latency as PQ drain rate varies, with and without pacing.

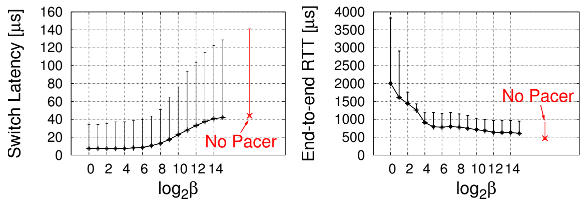


Figure 6: The average switch latency and end-to-end RTT (measured by ping) for paced flows, as β in Eq. (4) is varied.

Which flows need pacing? As previously discussed, only packets belonging to large flows (that are not latency-sensitive) should be paced. Moreover, only those flows that *are causing congestion* require pacing. We employ a simple adaptive mechanism to automatically detect such flows. Newly created flows are initially not paced. For each ACK with the ECN-Echo bit set, the corresponding flow is associated with the pacer with some probability, p_a (e.g., 1/8 in our implementation). This probabilistic sampling ensures that small flows are unlikely to be paced. The association times out after some time, T_i, so that idle flows will eventually be reclassified as latency sensitive.

Remark 4. We have described the pacer module with a single token bucket rate-limiter for simplicity. However, the same design can be used with multiple rate-limiters, allowing more accuracy when pacing multiple flows.

4.3 Effectiveness of the Pacer

We demonstrate the effectiveness of pacing by running an experiment with 2 long-lived DCTCP flows (similar to §3.1), with and without pacing. As before, we sweep the drain rate of the PQ. We also vary the marking threshold at the PQ from 1KB to 30KB. The results are shown in Figure 5. We observe that pacing improves both throughput and latency. The throughput varies nearly linearly with the PQ drain rate when the Pacer is enabled. Without pacing, however, we observe reduced throughput with low marking thresholds. This is because of spurious congestion signals caused by bursty traffic. Also, with pacing, the average and 99th percentile latency plummet with bandwidth headroom, quickly reaching their floor values of 5μs and 21μs respectively. In contrast, the latency decreases much more gradually without pacing, particularly at the 99th percentile.

4.4 The tradeoff between Pacer delay and effectiveness

Pacing, *by definition*, implies delaying the transmission of packets. We find that there is a tradeoff between the delay at the Pacer and how effectively it can pace. This tradeoff is controlled by the parameter β in Equation (4). Higher values of β cause a more aggressive increase in the transmission rate to keep the token bucket backlog, Q_{tb}, small. However, this also means that the Pacer creates more bursty output when a burst of traffic hits the token bucket; basically, the Pacer does 'less pacing'.

The following experiment shows the tradeoff. We start two long-lived DCTCP flows transmiting to a single receiver. The Pacer is enabled and we sweep β over the range $\beta = 2^0$ to $\beta = 2^{14}$ (the rest of the parameters are set as in Table 3). The PQ on the receiver link is configured to drain at 950Mbps and has a marking threshold of 1KB. We measure both the latency across the switch and the end-to-end RTT *for the flows being paced*, which is measured using ping from the senders to the receiver.

Figure 6 shows the results. The Pacer is more effective in reducing switch latency with smaller value of β, but also induces more delay. We observe a sharp increase in Pacer delay for values of β smaller than 2^4 without much gain in switch latency, suggesting the sweet spot for β. Nonetheless, the Pacer does add a few hundreds of microseconds of delay to the paced flows. This underscores the importance of selectively choosing the flows to pace. Only large flows which are throughput-oriented and are not impacted by the increase in delay should be paced. In fact, the throughput (not shown due to lack of space) is also higher with better pacing: Decreasing from ~870Mbps at $\beta = 2^0$ to ~770Mbps at $\beta = 2^{14}$ (and slightly lower without pacing).

5 Experimental Setup

5.1 Implementation

We use the NetFPGA [33] platform to implement the Pacer and PQ modules. NetFPGA is a PCI card with

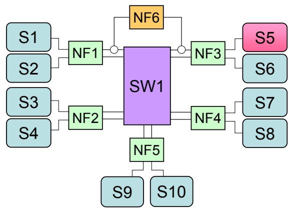


Figure 7: Testbed topology.

four Gigabit Ethernet ports and a Xilinx Virtex-II Pro 125MHz FPGA that has 4MB of SRAM. Each NetFPGA supports two Pacers and two PQs. The complete implementation consumes 160 block RAMs (out of 232, or 68% of the total NetFPGA capacity), and occupies 20,364 slices (86% of the total NetFPGA capacity).

The Pacer module has a single token bucket rate-limiter with a 128KB FIFO queue. The Pacer's transmission rate is controlled as described in §4.2 at a granularity of 0.5Mbps. Tokens of variable size (1-2000 bytes, proportional to the Pacer rate) are added every 16μs and the maximum allowed outstanding tokens (bucket depth) is 3KB. The Flow Association Table can hold 64 entries, identifying flows that require pacing. The Pacer enqueues the packets of these flows in the rate-limiter and forwards the rest in pass-through mode.

The PQ module implements the virtual queue length counter. It is incremented upon receiving a packet and decremented according to the PQ drain rate every 800ns. If the counter exceeds the configured marking threshold, the ECN/CE (Congestion Experienced) bit in the IP header of the incoming packet is set and the checksum value is recalculated. The PQ is completely pass-through and does not queue packets.

We have also introduced modifications to the TCP stack in Linux 2.6.26 for DCTCP, following the algorithm in [3]. Our code is available online at [13].

5.2 Testbed

Our testbed consists of 10 servers and 6 NetFPGAs connected to a Broadcom Triumph2 switch as shown in Figure 7. The Triumph2 is an ECN-capable switch with 48 nonblocking 1Gbps ports and 4MB of buffer memory shared across all ports. Each server has 4-core Intel Xeon E5620 2.4GHz CPUs with Hyper-Threading and at least 16GB of RAM. The servers use Intel's 82574L 1GbE Ethernet Controller. Two of the servers, S9 and S10, also have Mellanox ConnectX-2 ENt 10Gbase-T NICs, which were used for the 10Gbps experiments in §4.1.

Each of the NetFPGAs NF1-NF5 implements two Pacers and two PQs: One for each of the two servers and the two switch ports connected to it. All server-to-switch traffic goes through the Pacer module and all switch-to-server traffic goes through the PQ module.

For the majority of the experiments in this paper, we use machine S5 as the receiver, and (a subset of) the rest of the machines as senders which cause congestion at the switch port connected to S5 (via NF3).

Measuring Switch Latency. We have also developed a Latency Measurement Module (LMM) in NetFPGA for sub-microsecond resolution measurement of the latency across the congested switch port. The LMM (NF6 in Figure 7) works as follows: Server S1 generates a 1500 byte ping packet to S5 every 1ms.[3] The ping packets are intercepted and timestamped by the LMM before entering the switch. As a ping packet leaves the switch, it is again intercepted and the previous time-stamp is extracted and subtracted from the current time to calculate the latency.

6 Results

This section presents our experimental and simulation results evaluating HULL. We use micro-benchmarks to compare the latency and throughput performance of HULL with various schemes including TCP with drop-tail, default DCTCP, DCTCP with reduced marking threshold, and TCP with an ideal two-priority QoS scheme (TCP-QoS), where small (latency-sensitive) flows are given strict priority over large flows. We also check scalability of HULL using large-scale ns-2 [36] simulations. We briefly summarize our main findings:

(i) In micro benchmarks with both static and dynamic traffic, we find that HULL significantly reduces average and tail latencies compared to TCP and DCTCP. For example, with dynamic traffic (§6.2) HULL provides a more than 40x reduction in average latency compared to TCP (more than 10x compared to DCTCP), with bigger reductions at the high percentiles. Compared to an optimized DCTCP with low marking threshold and pacing, HULL achieves a 46–58% lower average latency, and a 69–78% lower 99th percentile latency. The bandwidth traded for this latency reduction increases the completion-time of large flows by 17–55%, depending on the load, in good agreement with the theoretical prediction in §3.2.

(ii) HULL achieves comparable latency to TCP-QoS with two priorities, but lower throughput since QoS does not leave bandwidth headroom. Also, unlike TCP-QoS which loses a lot of throughput if buffers are shallow (more than 58% in one experiment), HULL is much less sensitive to the size of switch buffers, as it (mostly) keeps them unoccupied.

(iii) Our large-scale ns-2 simulations confirm that HULL scales to large multi-switch topologies.

Parameter choices. Table 3 gives the baseline parameters used in the testbed experiments (the ns-2 parameters are given in §6.3). The parameters are determined experimentally. Due to space constraints, we omit the details

[3]Note that this adds 12Mbps (1.2%) of throughput overhead.

Phantom Queue	Drain Rate = 950Mbps, Marking Thresh. = 1KB
Pacer	$T_r = 64\mu s$, $\eta = 0.125$, $\beta = 16$, $p_a = 0.125$, $T_i = 10ms$

Table 3: Baseline parameter settings in experiments.

and summarize our main findings.

The PQ parameters are chosen based on experiments with long-lived flows, like that in Figure 5. As can be seen in this figure, the PQ with 950Mbps drain rate and 1KB marking threshold (with pacing) achieves almost the latency floor. An interesting fact is that smaller marking thresholds are required to maintain low latency as the PQ drain rate (γC) increases. This can be seen most visibly in Figure 5(a) for the 99th percentile latency. The reason is that since the input rate into the PQ is limited to the line rate (because it's in series), it takes longer for it to build up as the drain rate increases. Therefore, the marking threshold must also be reduced with increasing drain rate to ensure that the PQ reacts to congestion quickly.

Regarding the Pacer parameters, we find that the speed of the Pacer rate adaptation—determined by T_r/η—needs to be on the order of a few RTTs. This ensures that the aggregate host transmission rate is tracked closely by the Pacer and provides a good estimate of the rate imposed by the higher-layer DCTCP congestion control. The parameter β is chosen as described in §4.4. The parameters p_a and T_i are chosen so that small flows (e.g., smaller than 10KB) are unlikely to be paced. Overall, we do not find the Pacer to be sensitive to these parameters.

6.1 Static Flow Experiments

We begin with an experiment that evaluates throughput and latency in the presence of long-lived flows. We call this the *static* setting since the number of flows is constant during the experiment and the flows always have data to send. Each flow is from a different server sending to the receiver S5 (see Figure 7). We sweep the number of flows from 2 to 8. (Note that at least 2 servers must send concurrently to cause congestion at the switch.)

Schemes. We compare four schemes: (i) standard TCP (with drop-tail), (ii) DCTCP with 30KB marking threshold, (iii) DCTCP with 6KB marking threshold and pacing enabled, and (iv) DCTCP with a PQ (950Mbps with 1KB marking threshold). For schemes (ii) and (iii), ECN marking is enabled at the switch and is based on the physical queue occupancy, while for (iv), marking is only done by the PQ.

Note: The recommended marking threshold for DCTCP at 1Gbps is 30KB [3]. We also lower the marking threshold to 6KB to evaluate how much latency can be improved with pure queue-based congestion signaling. Experiments show that with this low marking threshold, pacing is required to avoid excessive loss in throughput (§6.2.1). Reducing the marking threshold below 6KB severely degrades throughput, even with pacing.

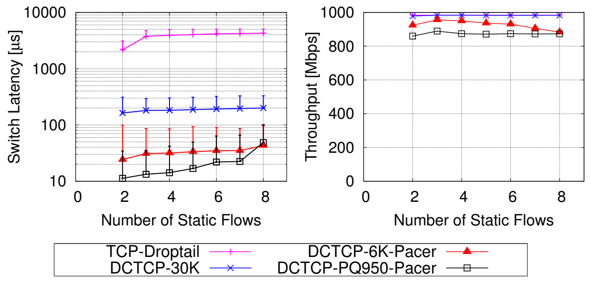


Figure 8: Switch latency (left) and throughput (right) as the number of long-lived flows varies. The latency plot uses a logarithmic scale and the vertical bars indicate the 99th percentile.

Analysis. The results are shown in Figure 8. We observe more than an order of magnitude (about 20x) reduction in average latency with DCTCP compared to TCP. Reducing the marking threshold to 6KB gives a further 6x reduction, bringing the average latency down from $\sim$200μs to $\sim$30μs. When there are a few static flows (e.g., less than 4), the PQ reduces the average latency by another factor 2–3 compared to DCTCP with 6KB marking threshold. Moreover, it also significantly lowers the jitter, achieving a 99th percentile of $\sim$30μs compared to $\sim$100μs for DCTCP-6K-Pacer, and more than 300μs for standard DCTCP. The PQ's lower latency is because of the bandwidth headroom it creates: The throughput for the PQ is about 870Mbps. The 8% loss compared to the PQ's 950Mbps drain rate is due to the rate fluctuations of DCTCP, as explained in §3.1.

Behavior with increasing flows. Figure 8 shows that bandwidth headroom becomes gradually less effective with increasing the number of flows. This is because of the way TCP (and DCTCP) sources increase their window size to probe for additional bandwidth. As is well-known, during Congestion Avoidance, a TCP source increases its window size by one packet every round-trip time. This is equivalent to an increase in sending rate of $1/RTT$ (in pkts/sec) each round-trip-time. Now, with N flows all increasing their rates at this slope, more bandwidth headroom is required to prevent the aggregate rate from exceeding the link capacity and causing queuing. More precisely, because of the one RTT of delay in receiving ECN marks from the PQ, the sources' aggregate rate overshoots the PQ's target drain rate by N/RTT (in pkts/sec). Hence, we require:

$$(1-\gamma)C > \frac{N}{RTT} \Longrightarrow 1-\gamma > \frac{N}{C \times RTT}, \qquad (5)$$

where $C \times RTT$ is the bandwidth-delay product in units of packets. Equation (5) indicates that *the bandwidth headroom required (as a percentage of capacity) to prevent queuing increases with more flows and decreases with larger bandwidth-delay product.*

		Switch Latency (μs)			1KB FCT (μs)			10MB FCT (ms)		
		Avg	**90th**	**99th**	**Avg**	**90th**	**99th**	**Avg**	**90th**	**99th**
20% Load	**TCP-DropTail**	111.5	450.3	1224.8	1047	1136	10533	110.2	162.3	349.6
	DCTCP-30K	38.4	176.4	295.2	475	638	2838	**106.8**	**155.2**	**301.7**
	DCTCP-6K-Pacer	6.6	13.0	59.7	389	531	888	111.8	168.5	320.0
	DCTCP-PQ950-Pacer	**2.8**	**7.6**	**18.6**	**380**	**529**	**756**	125.4	188.3	359.9
40% Load	**TCP-DropTail**	329.3	892.7	3960.8	1537	3387	5475	**151.3**	**275.3**	575.0
	DCTCP-30K	78.3	225.0	556.0	495	720	1794	155.1	281.5	**503.3**
	DCTCP-6K-Pacer	15.1	35.5	213.4	403	560	933	168.7	309.3	567.5
	DCTCP-PQ950-Pacer	**7.0**	**13.5**	**48.2**	**382**	**536**	**808**	198.8	370.5	654.7
60% Load	**TCP-DropTail**	720.5	2796.1	4656.1	2103	4423	5425	**250.0**	**514.6**	1007.4
	DCTCP-30K	119.1	247.2	604.9	511	740	1268	267.6	538.4	**907.3**
	DCTCP-6K-Pacer	24.8	52.9	311.7	403	563	923	320.9	632.6	1245.6
	DCTCP-PQ950-Pacer	**13.5**	**29.3**	**99.2**	**386**	**530**	**782**	389.4	801.3	1309.9

Table 4: Baseline dynamic flow experiment. The average, 90th percentile, and 99th percentile switch latency and flow completion times are shown. The results are the average of 10 trials. In each case, the best scheme is shown in red.

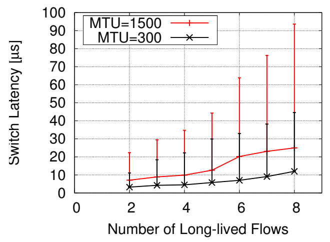


Figure 9: The impact of increasing number of flows on switch latency with PQ, for MTU = 1500 bytes and MTU = 300 bytes.

An important consequence of Equation (5) is that for a fixed RTT, less bandwidth headroom is needed as the link capacity increases (e.g., from 1Gbps to 10Gbps). We demonstrate this using a simple experiment in Figure 9. In the absence of an experimental setup faster than 1Gbps, we emulate what happens at higher link speeds by decreasing the MTU (packet size) to 300 bytes. Since the default MTU is 1500 bytes, this increases the bandwidth-delay product by a factor of 5 in units of packets, effectively emulating a 5Gbps link. As can be seen in the figure, the latency with the PQ is much lower with the smaller packet size at all number of flows. This confirms that the *sensitivity to the number of flows decreases with increasing link speed. Essentially, the same amount of bandwidth headroom is more effective for faster links.*

Remark 5. We found that when the MTU is reduced to 300 bytes, the receiver NIC cannot keep up with the higher packets/sec and starts dropping packets. To avoid this artifact, we had to reduce the PQ drain rate to 800Mbps for the tests in Figure 9.

6.2 Dynamic Flow Experiments

In this section we present the results of a microbenchmark which creates a *dynamic* workload. We develop a simple client/server application to generate traffic based on patterns seen in storage systems like memcached [54]. The client application, running on server S5 (Figure 7) opens 16 permanent TCP connections with each of the other 9 servers. During the test, the client repeatedly chooses a random connection among the pool of connections and makes a request for a file on that connection. The server application responds with the requested file. The requests are generated as a Poisson process in an open loop fashion [43]; that is, new requests are triggered independently of prior outstanding requests. The request rate is chosen such that the average RX throughput at the client is at a desired level of load. For example, if the average file size is 100KB, and the desired load is 40% (400Mbps), the client makes 500 requests per second on average. We conduct experiments at low (20%), medium (40%), and high (60%) levels of load. During the experiments, we measure the switch latency (using the NetFPGA Latency Measurement Module), as well as the application level flow completion times (FCT).

6.2.1 Baseline

For the baseline experiment, we use a workload where 80% of all client requests are for a 1KB file and 20% are for a 10MB file. Of course, this is not meant to be a realistic workload. Rather, it allows a clear comparison of how different schemes impact the small (latency-sensitive) and large (bandwidth-sensitive) flows.

Note: The 1KB flows are just a single packet and can complete in one RTT. Such single packet flows are very common in data center networks; for example, measurements in a production data center of a large cloud service provider have revealed that more than 50% of flows are smaller than 1KB [22].

Table 4 gives the results for the same four schemes that were used in the static flow experiments (§6.1).

Analysis: Switch latency. The switch latency is very high with TCP compared to the other three schemes since it completely fills available buffers. With DCTCP, the latency is 3–6 times lower on average. Reducing the marking threshold to 6KB gives another factor of 5 reduction in average latency. However, some baseline level of queueing delay and significant *jitter* remains, with hundreds of microseconds of latency at the 99th percentile.

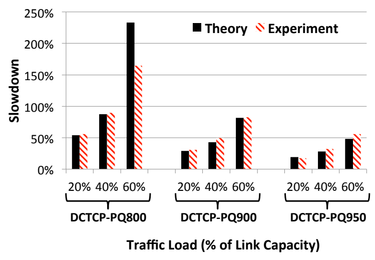


Figure 10: Slowdown for 10MB flows: Theory vs Experiment.

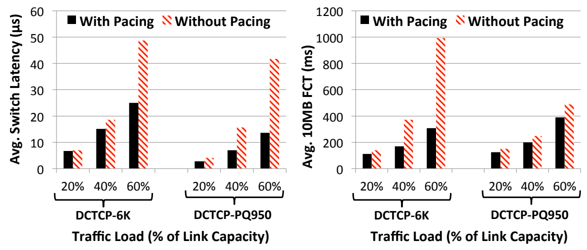


Figure 11: Average switch latency (left) and 10MB FCT (right), with and without pacing.

As explained in §2.1, this is because queue-based congestion signaling (even with pacing) is too late to react; by the time congestion is detected, a queue has already built up and is increasing. The lowest latency is achieved by the PQ which creates bandwidth headroom. Compared to DCTCP-6K-Pacer, the PQ reduces the average latency by 46–58% and the 99th percentile by 69–78%. The latency with the PQ is especially impressive considering just a single 1500 byte packet adds 12μs of queueing latency at 1Gbps.

Analysis: 1KB FCT & End-host latency. The 1KB FCT is rather high for all schemes. It is evident from the switch latency measurements that the high 1KB FCTs are due to the delays incurred by packets at the end-hosts (in the software stack, PCIe bus, and network adapters). The host-side latency is particularly large when the servers are actively transmitting/receiving data at high load. As a reference, the minimum 1KB FCT is about 160μs. Interestingly, the latency at the end-host (and the 1KB FCT) improves with more aggressive signaling of congestion in the network, especially, compared to TCP-DropTail. This suggests that the un-checked increase in window size with TCP-DropTail (and to a lesser extent with DCTCP-30K) causes queuing at both the network *and* the end-hosts. Essentially, flows with large windows deposit large chunks of data into NIC buffers, which adds delay for the small flows.

The main takeaway is that *bandwidth headroom significantly reduces the average and tail switch latency under load, even compared to optimized queue-based AQM with a low marking threshold and pacing. However, to take full advantage of this reduction, the latency of the software stack and network adapters must also improve.*

Analysis: Slowdown for 10MB flows. The bandwidth given away by the PQ increases the flow completion of the 10MB flows, which are throughput-limited. As predicted by the theoretical model in §3.2, the slowdown is worse at higher load. Compared to the lowest achieved value (shown in red in Table 4), with the PQ, the average 10MB FCT is 17% longer at 20% load, 31% longer at 40% load, and 55% longer at 60% load. Figure 10 compares the slowdown predicted by theory with that observed in experiments. The comparison includes the results for PQ with 950Mbps drain rate, given in Table 4, as well as PQ with 900Mbps and 800Mbps for which the detailed numbers are omitted in the interest of space.

The theoretical slowdown is computed using Equation (3). To invoke the equation, we use $\rho = 0.2, 0.4$, and 0.6 corresponding to the load. We also account for the additional throughput loss due to DCTCP rate fluctuations (§3.1), by subtracting 8% from the PQ drain rate to get γ. That is, we use $\gamma = 0.72, 0.82$, and 0.87 corresponding to the drain rates 800, 900, and 950Mbps. Overall, the theory and experiments match very well.

Pacing vs No Pacing. Figure 11 compares the average switch latency and 10MB FCT, with and without pacing. The comparison is shown for DCTCP with the marking threshold at the switch set to 6KB, and for DCTCP with the PQ draining at 950Mbps. In all cases, pacing lowers both the switch latency and the FCT of large flows, improving latency *and* throughput.

Remark 6. Most data center networks operate at loads less than 30% [9], so a load of 60% with Poisson/bursty traffic is highly unlikely—the performance degradation would be too severe. The results at 20% and 40% load are more indicative of actual performance.

6.2.2 Comparison with two-priority QoS scheme

Ethernet and IP provide multiple Quality of Service (QoS) priorities. One method for meeting the objective of ultra-low latency and high bandwidth is to use two priorities: an absolute priority for the flows which require very low latency and a lower priority for the bandwidth-intensive elastic flows. While this method has the potential to provide ideal performance, it may be impractical (and is not commonly deployed) because applications do not segregate latency-sensitive short flows and bandwidth-intensive large flows dynamically. Indeed, application developers do not typically consider priority classes for network transfers. It is more common to assign an entire application to a priority and use priorities to segregate *applications*.

Despite this, we now compare HULL with TCP using an ideal two-priority QoS scheme for benchmarking purposes. We repeat the baseline experiment from the previous section, but for QoS, we modify our application to classify the 1KB flows as 'high-priority' using the Type

		Switch Latency (μs)			1KB FCT (μs)			10MB FCT (ms)		
		Avg	**90th**	**99th**	**Avg**	**90th**	**99th**	**Avg**	**90th**	**99th**
Large Buffer	**TCP-QoS**	**6.4**	17.2	**20.2**	565	570	2308	**152.6**	**275.4**	**585.1**
	DCTCP-PQ950-Pacer	6.9	**13.5**	47.8	**381**	**538**	**810**	199.0	370.2	658.9
Small Buffer	**TCP-QoS**	5.3	15.5	**19.7**	**371**	**519**	**729**	362.3	811.9	1924.3
	DCTCP-PQ950-Pacer	**5.0**	**13.2**	35.1	378	521	759	**199.4**	**367.0**	**654.9**

Table 5: HULL vs QoS with two priorities. The switch buffer is configured to use either Dynamic buffer allocation (Large Buffer) or a fixed buffer of 10pkts = 15KB (Small Buffer). In all tests, the total load is 40%. In the case of QoS, the switch latency is that of the high priority queue. The results are the average of 10 trials.

of Service (TOS) field in the IP header. The switch uses a separate queue for these flows which is given strict priority over the other, 'best-effort', traffic.

Scenarios. We consider two settings for the switch buffer size: (i) Dynamic buffer allocation (the default settings in the switch), and (ii) a fixed buffer of 10 packets (15KB) per priority. Note that in the latter setting TCP-QoS gets 30KB in total, whereas HULL gets just 15KB since it always uses only one priority. The second setting is used to evaluate how switches with very shallow buffers impact performance, since dynamic buffer allocation allows a congested port to grab up to $\sim$700KB of the total 4MB of buffer in the switch).

Analysis: HULL vs QoS. Table 5 gives the results. We make three main observations:

(i) HULL and QoS achieve roughly the same average switch latency. HULL is slightly better at the 90th percentile, but worse at the 99th percentile.

(ii) When the switch buffer is large, TCP-QoS achieves a better FCT for the 10MB flows than HULL as it does not sacrifice any throughput. However, with small buffers, there is about a 2.4x increase in the FCT with TCP-QoS (equivalent to a 58% reduction in throughput). HULL achieves basically the same throughput in both cases because it does not need the buffers in the first place.

(iii) In the large buffer setting, the 1KB flows complete significantly faster with HULL—more than 33% faster on average and 65% faster at the 99th percentile. This is because the best-effort flows (which have large window sizes) interfere with high-priority flows at the end-hosts, similar to what was observed for TCP-DropTail in the baseline experiment (Table 4). This shows that all points of contention (including the end-hosts, PCIe bus, and NICs) must respect priorities for QoS to be effective.

Overall, this experiment shows that *HULL achieves nearly as low a latency as the ideal QoS scheme, but gets lower throughput. Also, unlike QoS, HULL can cope with switches with very shallow buffers because it avoids queue buildup altogether.*

6.3 Large-scale ns-2 Simulation

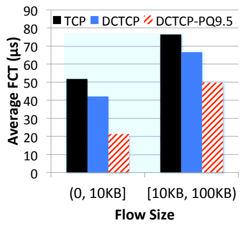
Due to the small size of our testbed, we cannot verify in hardware that HULL scales to the multi-switch topologies common in data centers. Therefore, we complement our hardware evaluation with large-scale ns-2 simulations targeting a multi-switch topology and workload.

Topology. We simulate a three-layered fat-tree topology based on scalable data center architectures recently proposed [2, 22]. The network consists of 56 8-port switches that connect 192 servers organized in 8 pods. There is a 3:1 over-subscription at the top-of-the-rack (TOR) level. The switches have 250 packets worth of buffering. All links are 10Gbps and have 200ns of delay, with 1μs of additional delay at the end-hosts. This, along with the fact that ns-2 simulates *store-and-forward* switches, implies that the end-to-end round-trip latency for a 1500 byte packet and a 40 byte ACK across the entire network (6 hops) is 11.8μs.

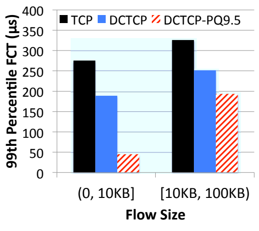
Routing. We use standard Equal-Cost Multi-Path (ECMP) [7] routing. Basically, ECMP hashes each flow to one of the shortest paths between the source and destination nodes. All packets of the flow take the same path. This avoids the case where packet reordering is misinterpreted as a sign of congestion by TCP (or DCTCP).

Workload. The workload is generated similarly to our dynamic flow experiments in hardware. We open permanent connections between each pair of servers. Flows arrive according to a Poisson process and are sent from a random source to a random destination server. The flow sizes are drawn from a Pareto distribution with shape parameter 1.05 and mean 100KB. This distribution creates a heavy-tailed workload where the majority of flows are small, but the majority of traffic is from large flows, as is commonly observed in real networks: 95% of the flows are less than 100KB and contribute a little over 25% of all data bytes; while 0.03% of the flows that are larger than 10MB contribute over 40% of all bytes.

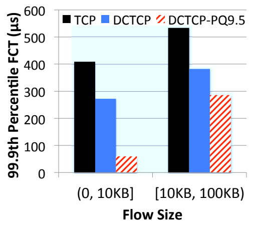
Simulation settings. We compare standard TCP, DCTCP, and DCTCP with a PQ draining at 9.5Gbps (HULL). The Pacer is also enabled for HULL, with parameters: $T_r = 7.2\mu$s, $\eta = 0.125$, $\beta = 375$, $p_a = 0.125$, and $T_i = 1$ms. The changes to the parameters compared to the ones we used for the hardware Pacer (Table 3) are because of the difference in link speed (10Gbps vs 1Gbps) and the much lower RTT in the simulations. We set the flow arrival rate so the load at the server-to-TOR links is 15% (We have also run many simulations with other levels of load, with qualitatively similar results).



(a) Average



(b) 99th Percentile



(c) 99.9th Percentile

Figure 12: Average and high percentile FCT for small flows.

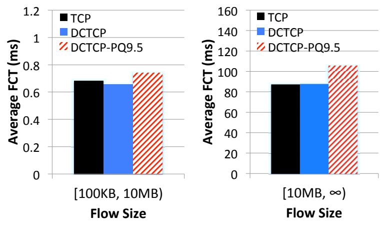


Figure 13: Average FCT for large flows.

All simulations last for at least 5 million flows.

Note: Because the topology has 3:1 over-subscription at the TOR, the load is approximately 3 times higher at the TOR-to-Agg and Agg-to-Core links. A precise calculation shows that the load is 43.8% at the TOR-to-Agg links, and 39.6% at the Agg-to-Core links.

Analysis: Small flows. Figure 12 shows the average, 99th percentile, and 99.9th percentile of the FCT for small flows. The statistics are shown separately for flows smaller than 10KB, and flows with size between 10KB and 100KB. We observe a significant improvement in the FCT of these flows with HULL; especially for flows smaller than 10KB, there is more than 50% reduction in the average FCT compared to DCTCP, and more than 80% reduction at the 99th and 99.9th percentiles.

It is important to note that the 20μs average FCT for flows smaller than 10KB achieved by HULL is near ideal given that the simulation uses store-and-forward switching. In fact, the average size for flows smaller than 10KB is 6.8KB. Because of store-and-forward, the FCT for a flow of this size is at least $\sim$16.2μs. This implies that with HULL, across the 5 switches end-to-end between source and destination, there are, *in total*, only 3 packets being queued on average (each adding 1.2μs of delay).

Analysis: Large flows. Figure 13 shows the average FCT for flows between 100KB and 10MB in size, and for those that are larger than 10MB. As expected, these flows are slower with HULL: up to 24% slower for flows larger than 10MB, which is approximately the slowdown predicted by theory (§3.2) at this fairly high level of load.

Overall, the ns-2 simulations confirm that bandwidth headroom created by HULL is effective in large multi-switch networks and can significantly reduce latency and jitter for small flows.

7 Related Work

AQM: AQM has been an active area of research ever since RED [18] was proposed. Subsequent work [32, 17, 24] refined the concept and introduced enhancements for stability and fairness. While these schemes reduce queueing delay relative to tail-drop queues, they still induce too large a queueing latency from the ultra-low latency perspective of this paper because they are, fundamentally, queue-based congestion signaling mechanisms. Virtual-queue based algorithms [21, 31] consider signaling congestion based on link utilization. The Phantom Queue is inspired by this work with the difference that PQs are not *adjacent* to physical switch queues. Instead, they operate on network links (in series with switch ports). This makes the PQ completely agnostic to the internal switch architecture and allows it to be deployed with any switch as a 'bump-on-the-wire'.

Transport Layer: Layer 3 research relevant to our ultra-low latency objective includes protocols and algorithms which introduce various changes to TCP or are TCP substitutes. Explicit feedback schemes like XCP [29] and RCP [16] can perform quite well at keeping low queueing but require major features that do not exist in switches or protocols today. Within the TCP family, variants like Vegas [11], CTCP [48] and FAST [53] attempt to control congestion by reacting to increasing RTTs due to queuing delay. By design, such algorithms require the queue to build up to a certain level and, therefore, do not provide ultra-low latency.

Pacing: TCP pacing was suggested for alleviating burstiness due to ACK compression [56]. Support for pacing has not been unanimous. For instance, Aggarwal *et al.* [1] show that paced flows are negatively impacted when competing with non-paced flows because their packets are deliberately delayed. With increasing line rates and the adoption of TCP segmentation offloading, the impact of burstiness has been getting worse [8] and the need for pacing is becoming more evident. One way pacing has been implemented is using software timers to determine when a packet should be transmitted [28, 52, 15]. However, when the ticks of the pacer are very frequent as happens at high line rates, such software-based schemes often lack access to accurate timers or heavily burden the CPU with significant interrupt processing. Further, a software pacer prior to the NIC cannot offset the effects of offloading functions like LSO which occur in the NIC.

8 Final Remarks

In this paper we presented a framework to deliver baseline fabric latency for latency-sensitive applications while simultaneously supporting high fabric goodput for bandwidth-sensitive applications. Through a combination of Phantom Queues to run the network with near zero queueing, adaptive response to ECN marks using DCTCP, and packet pacing to smooth bursts induced by hardware offload mechanisms like LSO, we showed a factor of up to 10-40 reduction in average and tail latency with a configurable sacrifice of overall fabric throughput. Our work makes another case for a time when aggregate bandwidth may no longer be the ultimate evaluation criterion for large-scale fabrics, but a tool in support of other high-level goals such as predictable low latency.

There are two aspects which warrant further investigation. First, it is useful to evaluate HULL on a larger multi-switch testbed and with more diverse workloads. Second, it is important to quantify the buffering requirements for incast-like communication patterns [50] with our approach.

Acknowledgments

We thank our shepherd, Xiaowei Yang, and the anonymous reviewers for their valuable feedback and suggestions. Mohammad Alizadeh is supported by the Caroline and Fabian Pease Stanford Graduate Fellowship.

References

[1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the Performance of TCP Pacing. In *Proc. of INFOCOM*, 2000.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM*, 2008.

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

[4] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of DCTCP: stability, convergence, and fairness. In *Proc. of SIGMETRICS*, 2011.

[5] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *Proc. of SIGCOMM*, 2004.

[6] S. Athuraliya, S. Low, V. Li, and Q. Yin. REM: active queue management. *Network, IEEE*, 15(3):48 –53, May 2001.

[7] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

[8] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental study of router buffer sizing. In *Proc. of IMC*, 2008.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[10] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview, 1994.

[11] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. *Proc. of SIGCOMM*, 1994.

[12] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun. Remote Direct Memory Access over the Converged Enhanced Ethernet Fabric: Evaluating the Options. In *Proc. of HOTI*, 2009.

[13] DCTCP Linux kernel patch. http://www.stanford.edu/~alizade/Site/DCTCP.html.

[14] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of SIGCOMM*, pages 1–12, 1989.

[15] D.Lacamera. TCP Pacing Linux Implementation. http://danielinux.net/index.php/TCP_Pacing.

[16] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor sharing flows in the internet. In *IWQoS*, 2005.

[17] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. Netw.*, August 2002.

[18] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4), 1993.

[19] Fulcrum FM4000 Series Ethernet Switch. http://www.fulcrummicro.com/documents/products/FM4000_Product_Brief.pdf.

[20] S. B. Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical bandwidth sharing: a study of congestion at flow level. In *Proc. of SIGCOMM*, SIGCOMM '01, pages 111–122, 2001.

[21] R. J. Gibbens and F. Kelly. Distributed connection acceptance control for a connectionless network. In *Proc. of the Int'l. Teletraffic Congress*, 1999.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proc. of SIGCOMM*, 2009.

[23] Arista 7100 Series Switches. http://www.aristanetworks.com/en/products/7100series.

[24] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. On designing improved controllers for AQM routers supporting TCP flows. In *Proc. of INFOCOM*, 2001.

[25] S. Iyer, R. Kompella, and N. McKeown. Designing packet buffers for router linecards. *IEEE/ACM Trans. Netw.*, 16(3):705 –717, june 2008.

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

[27] H. Jiang and C. Dovrolis. Why is the internet traffic bursty in short time scales. In *In Sigmetrics*, pages 241–252. ACM Press, 2005.

[28] H. Kamezawa, M. Nakamura, J. Tamatsukuri, N. Aoshima, M. Inaba, and K. Hiraki. Inter-Layer Coordination for Parallel TCP Streams on Long Fat Pipe Networks. In *Proc. of SC*, 2004.

[29] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proc. of SIGCOMM*, 2002.

[30] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proc. of SIGCOMM*, pages 255–266, 2009.

[31] S. S. Kunniyur and R. Srikant. An adaptive virtual queue (AVQ) algorithm for active queue management. *IEEE/ACM Trans. Netw.*, April 2004.

[32] D. Lin and R. Morris. Dynamics of random early detection. In *Proc. of SIGCOMM*, 1997.

[33] The NetFPGA Project. http://netfpga.org.

[34] Cisco Nexus 5548P Switch. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/ps11215/white_paper_c11-622479.pdf.

[35] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers.

[36] The Network Simulator NS-2. http://www.isi.edu/nsnam/ns/.

[37] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of SOSP'11*, 2011.

[38] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54:121–130, July 2011.

[39] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1:344–357, June 1993.

[40] K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168: the addition of explicit congestion notification (ECN) to IP.

[41] ConnectX-2 EN with RDMA over Ethernet. http://www.mellanox.com/related-docs/prod_software/ConnectX-2_RDMA_RoCE.pdf.

[42] J. W. Roberts. A survey on statistical bandwidth sharing. *Comput. Netw.*, 45:319–332, June 2004.

[43] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06*, pages 18–18, Berkeley, CA, USA, 2006.

[44] P. Shivam, P. Wyckoff, and D. Panda. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proc. of ACM/IEEE conference on Supercomputing*, 2001.

[45] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proc. of SIGCOMM*, pages 231–242, 1995.

[46] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[47] R. Srikant. *The Mathematics of Internet Congestion Control (Systems and Control: Foundations and Applications)*. 2004.

[48] K. Tan and J. Song. A compound TCP approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, 2006.

[49] The tcpdump official website. http://www.tcpdump.org.

[50] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proc. of SIGCOMM*, 2009.

[51] A. Vishwanath, V. Sivaraman, and M. Thottan. Perspectives on router buffer sizing: recent results and open problems. *Proc. of SIGCOMM*, 39, 2009.

[52] V. Visweswaraiah and J. Heidemann. Improving restart of idle tcp connections, 1997.

[53] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw.*, 2006.

[54] memcached - a distributed memory object caching system. http://memcached.org.

[55] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: a new resource reservation protocol. *Communications Magazine, IEEE*, 40(5):116 –127, May 2002.

[56] L. Zhang, S. Shenker, and D. D. Clark. Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. *Proc. of SIGCOMM*, 1991.