Carousel: Scalable Traffic Shaping at End Hosts

Ahmed Saeed[*]
Georgia Institute of Technology

Nandita Dukkipati
Google, Inc.

Vytautas Valancius
Google, Inc.

Vinh The Lam
Google, Inc.

Carlo Contavalli
Google, Inc.

Amin Vahdat
Google, Inc.

ABSTRACT

Traffic shaping, including pacing and rate limiting, is fundamental to the correct and efficient operation of both datacenter and wide area networks. Sample use cases include policy-based bandwidth allocation to flow aggregates, rate-based congestion control algorithms, and packet pacing to avoid bursty transmissions that can overwhelm router buffers. Driven by the need to scale to millions of flows and to apply complex policies, traffic shaping is moving from network switches into the end hosts, typically implemented in software in the kernel networking stack.

In this paper, we show that the performance overhead of end-host traffic shaping is substantial limits overall system scalability as we move to thousands of individual traffic classes per server. Measurements from production servers show that shaping at hosts consumes considerable CPU and memory, unnecessarily drops packets, suffers from head of line blocking and inaccuracy, and does not provide backpressure up the stack. We present Carousel, a framework that scales to tens of thousands of policies and flows per server, built from the synthesis of three key ideas: i) a single queue shaper using time as the basis for releasing packets, ii) fine-grained, just-in-time freeing of resources in higher layers coupled to actual packet departures, and iii) one shaper per CPU core, with lock-free coordination. Our production experience in serving video traffic at a Cloud service provider shows that Carousel shapes traffic accurately while improving overall machine CPU utilization by 8% (an improvement of 20% in the CPU utilization attributed to networking) relative to state-of-art deployments. It also conforms 10 times more accurately to target rates, and consumes two orders of magnitude less memory than existing approaches.

CCS CONCEPTS

• **Networks → Packet scheduling**;

KEYWORDS

Traffic shaping, Rate-limiters, Pacing, Timing Wheel, Backpressure

ACM Reference format:

Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017*, 14 pages.
https://doi.org/10.1145/3098822.3098852

1 INTRODUCTION

Network bandwidth, especially across the WAN, is a constrained resource that is expensive to overprovision [25, 28, 32]. This creates an incentive to shape traffic based on the priority of the application during times of congestion, according to operator policy. Further, as networks run at higher levels of utilization, accurate shaping to a target rate is increasingly important to efficient network operation. Bursty transmissions from a flow's target rate can lead to: i) packet loss, ii) less accurate bandwidth calculations for competing flows, and iii) increasing round trip times. Packet loss reduces goodput and confuses transport protocols attempting to disambiguate fair-share available-capacity signals from bursty traffic sources. One could argue that deep buffers are the solution, but we find that the resulting increased latency leads to poor experience for users. Worse, high latency reduces application performance in common cases where compute is blocked on, for example, the completion of an RPC. Similarly, the performance of consistent storage systems is dependent on network round trip times.

In this paper, we use traffic *shaping* broadly to refer to either pacing or rate limiting, where *pacing* refers to injecting inter-packet gaps to smooth traffic within a single connection, and *rate limiting* refers to enforcing a rate on a flow-aggregate consisting of one or more individual connections.

While traffic shaping has historically targeted wide area networks, two recent trends bring it to the forefront for datacenter communications, which use end-host based shaping. The first trend is the use of fine grained pacing by rate-based congestion control algorithms such as BBR [17] and TIMELY [36]. BBR and TIMELY's use of rate control is motivated by studies that show pacing flows can reduce packet drops for video-serving traffic [23] and for incast communication patterns [36]. Incast arises from common datacenter applications that simultaneously communicate with thousands of servers. Even if receiver bandwidth is allocated perfectly among senders at coarse granularity, simultaneous bursting to NIC line rate from even a small subset of senders can overwhelm the receiver's network capacity.

Furthermore, traffic from end hosts is increasingly bursty, due to heavy batching and aggregation. Techniques such as NIC offloads optimize for server CPU efficiency — e.g., Transmission Segmentation Offload [20] and Generic Receive Offload [3]. Pacing reduces burstiness, which in turn reduces packet drops at shallow-buffered switches [22] and improves network utilization [14]. It is for these reasons that BBR and TIMELY rely on pacing packets as a key technique for precise rate control of thousands of flows per server.

The second trend is the need for network traffic isolation across competing applications or Virtual Machines (VMs) in the Cloud. The emergence of Cloud Computing means that individual servers may host hundreds or perhaps even thousands of VMs. Each virtual

[*] Work done while at Google.

SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4653-5/17/08.
https://doi.org/10.1145/3098822.3098852

endpoint can in turn communicate with thousands of remote VMs, internal services within and between datacenters, and the Internet at large, resulting in millions of flows with overlapping bottlenecks and bandwidth allocation policies. Providers use predictable and scalable bandwidth arbitration systems to assign rates to flow aggregates, which are then enforced at end systems. Such an arbitration can be performed by centralized entities, e.g., Bandwidth Enforcer [32] and SWAN [25], or distributed systems, e.g., EyeQ [30].

For example, consider 500 VMs on a host providing Cloud services, where each VM communicates on average with 50 other virtual endpoints. The provider, with no help from the guest operating system, must isolate these 25K VM-to-endpoint flows, with bandwidth allocated according to some policy. Otherwise, inadequate network isolation increases performance unpredictability and can make Cloud services unavailable [11, 30].

Traditionally, network switches and routers have implemented traffic shaping. However, inside a datacenter, shaping in middleboxes is not an easy option. It is expensive in buffering and latency, and middleboxes lack the necessary state to enforce the right rate. Moreover, shaping in the middle does not help when bottlenecks are at network edges, such as the host network stack, hypervisor, or NIC. Finally, when packet buffering is necessary, it is much cheaper and more scalable to buffer near the application.

Therefore, the need to scale to millions of flows *per server* while applying complex policy means that traffic shaping must largely be implemented in end hosts. The efficiency and effectiveness of this end-host traffic shaping is increasingly critical to the operation of both datacenter and wide area networks. Unfortunately, existing implementations were built for very different requirements, e.g., only for WAN flows, a small number of traffic classes, modest accuracy requirements, and simple policies. Through measurement of video traffic in a large-scale Cloud provider, we show that the performance of end-host traffic shaping is a primary impediment to scaling a virtualized network infrastructure. Existing end-host rate limiting consumes substantial CPU and memory; e.g., shaping in the Linux networking stack use 10% of server CPU to perform pacing (§3); shaping in the hypervisor unnecessarily drops packets, suffers from head of line blocking and inaccuracy, and does not provide backpressure up the stack (§3).

We present the design and implementation of Carousel, an improvement on existing, kernel-based traffic shaping mechanism. Carousel scales to tens of thousands of flows and traffic classes, and supports complex bandwidth-allocation mechanisms for both WAN and datacenter communications. We design Carousel around three core techniques: i) a single queue shaper using time as the basis for scheduling, specifically, Timing Wheel [43], ii) coupling actual packet transmission to the freeing of resources in higher layers, and iii) each shaper runs on a separate CPU core, which could be as few as one CPU. We use lock-free coordination across cores. Our production experience in serving video traffic at a Cloud service provider shows that Carousel shapes traffic effectively while improving the overall machine CPU utilization by 8% relative to state-of-art implementations, and with an improvement of 20% in the CPU utilization attributed to networking. It also conforms 10 times more accurately to target rates, and consumes two orders of magnitude less memory than existing approaches. While we implement Carousel in a software NIC, it is designed for hardware offload. By the novel combination of previously-known techniques, Carousel makes a significant advance on the state-of-the-art shapers in terms of efficiency, accuracy and scalability.

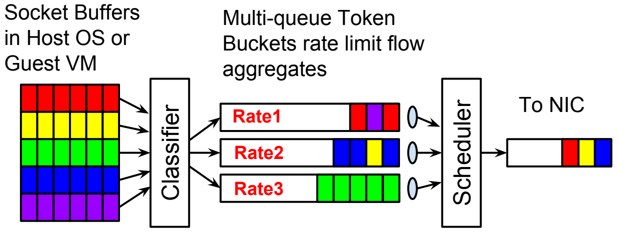

Socket Buffers in Host OS or Guest VM
Multi-queue Token Buckets rate limit flow aggregates
Classifier
Rate1
Rate2
Rate3
Scheduler
To NIC

Figure 1: Token Bucket Architecture: pre-filtering with multiple token bucket queues.

2 TRAFFIC SHAPERS IN PRACTICE

In the previous section we established the need for at-scale shaping at end hosts: first, modern congestion control algorithms, such as BBR and TIMELY, use pacing to smooth bursty video traffic, and to handle large-scale incasts; second, traffic isolation in Cloud is critically dependent on efficient shapers. Before presenting details of Carousel, we first present an overview of the shapers prevalently used in practice.

Nearly all of rate limiting at end hosts is performed in software. Figure 1 shows the typical rate limiter architecture, which we broadly term as *pre-filtering with multiple token bucket queues*. It relies on a *classifier*, *multiple queues*, *token bucket shapers* and/or a *scheduler* processing packets from each queue. The classifier divides packets into different queues, each queue representing a different traffic aggregate.[1] A queue has an associated traffic shaper that paces packets from that queue as necessary. Traffic shapers are synonymous with token buckets or one of its variants. A scheduler services queues in round-robin order or per service-class priorities.

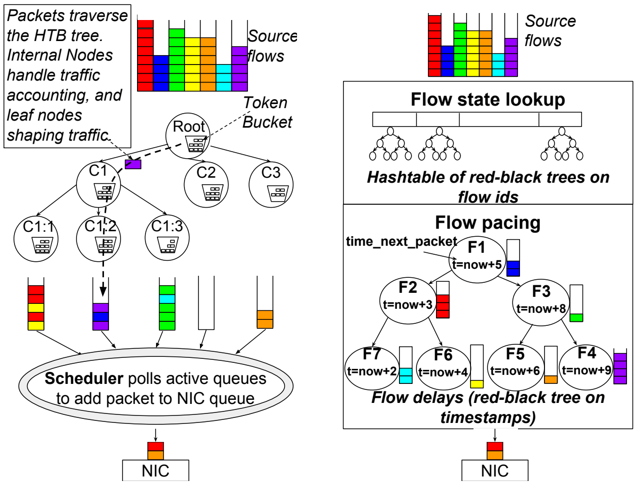
This design avoids head of line blocking, through a separate queue per traffic aggregate: when a token bucket delays a packet, all packets in the same queue will be delayed, but not packets in other queues. Other mechanisms, such as TCP Small Queues (TSQ) [4] in Linux, provide backpressure and reduce drops in the network stack.

Below, we describe three commonly used shapers in end-host stacks, hypervisors and NICs: 1) Policers; 2) Hierarchical Token Bucket; and 3) FQ/pacing. They are all based on per-traffic aggregate queues and a token bucket associated with each queue for rate limiting.

2.1 Policers

Policers are the simplest way to enforce rates. Policer implementations use token buckets with zero buffering per queue. A token bucket policer consists of a counter representing the number of currently available tokens. Sending a packet requires checking the request against the available tokens in the queue. The policer drops the packet if too few tokens are available. Otherwise, the policer forwards the packet,

[1] Traffic aggregate refers to a collection of individual TCP flows being grouped on a shaping policy.


Packets traverse the HTB tree. Internal Nodes handle traffic accounting, and leaf nodes shaping traffic.
Source flows
Token Bucket
Root
C1
C2
C3
C1:1
C1:2
C1:3
Scheduler polls active queues to add packet to NIC queue
NIC
Source flows
Flow state lookup
Hashtable of red-black trees on flow ids
Flow pacing
time_next_packet
F1 t=now+5
F2 t=now+3
F3 t=now+8
F7 t=now+2
F6 t=now+4
F5 t=now+6
F4 t=now+9
Flow delays (red-black tree on timestamps)
NIC

(a) HTB **(b) FQ/pacing**

Figure 2: Architecture of Linux rate limiters.

reducing the available tokens accordingly. The counter is refilled according to a target *rate* and capped by a maximum *burst*. The rate represents the average target rate and the burst size represents the tolerance for jitter, or for short-term deviation from the target rate conformance.

2.2 Hierarchical Token Bucket (HTB)

Unlike Policers that have zero queueing, traffic shapers buffer packets waiting for tokens, rather than drop them. In practice, related token buckets are grouped in a complex structure to support advanced traffic management schemes. Hierarchical Token Bucket (HTB) [2] in the Linux kernel Queueing Discipline (Qdisc) uses a tree organization of shapers as shown in Figure 2a. HTB classifies incoming packets into one of several traffic classes, each associated with a token bucket shaper at a leaf in the tree. The hierarchy in HTB is for borrowing between leaf nodes with a common parent to allow for work-conserving scheduling.

The *host enforcer* in the Bandwidth Enforcer (BwE) [32] system is a large-scale deployment of HTB. Each HTB leaf class has a one-to-one mapping to a specific BwE task flow group – typically an aggregate of multiple TCP flows. The number of HTB leaf classes is directly proportional to the number of QoS classes × destination clusters × number of tasks.

To avoid unbounded queue growth, shapers should be used with a mechanism to backpressure the traffic source. The simplest form of backpressure is to drop packets, however, drops are also a coarse signal to a transport like TCP. Linux employs more fine grained backpressure. For example, HTB Qdisc leverages TCP Small Queues (TSQ) [4] to limit two outstanding packets for a single TCP flow within the TCP/IP stack, waiting for actual NIC transmission before enqueuing further packets from the same flow. [2]. In the presence of flow aggregates, the number of enqueued packets equals the number of individual TCP flows × TSQ limit. If HTB queues are full, subsequent arriving packets will be dropped.

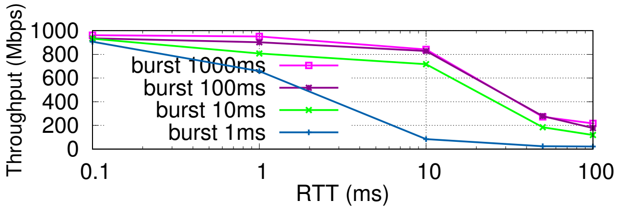


Figure 3: Policing performance for target throughput 1Gbps with different burst sizes. Policers exhibit poor rate conformance, especially at high RTT. The different lines in the plot correspond to the configured burst size in the token bucket.

2.3 FQ/pacing

FQ/pacing [5] is a Linux Qdisc used for packet pacing along with fair queueing for egress TCP and UDP flows. Figure 2b shows the structure of FQ/pacing. The FQ scheduler tracks per-flow state in an array of Red-Black (RB) trees indexed on flow hash IDs. A deficit round robin (DRR) scheduler [41] fetches outgoing packets from the active flows. A garbage collector deletes inactive flows.

Maintaining per-flow state provides the opportunity to support egress pacing of the active flows. In particular, a TCP connection sends at a rate approximately equal to cwnd/RTT, where cwnd is the congestion window and RTT is the round-trip delay. Since cwnd and RTT are only best-effort estimates, the Linux TCP stack conservatively sets the flow pacing rate to 2 × cwnd/RTT [6]. FQ/pacing enforces the pacing rate via a leaky bucket queue [8], where sending timestamps are computed from packet length and current pacing rate. Flows with next-packet timestamps far into the future are kept in a separate RB tree indexed on those timestamps. We note that pacing is enforced at the granularity of TCP Segmentation Offload (TSO) segments.

The Linux TCP stack further employs the pacing rate to automatically size packets meant for TCP Segmentation Offloading. The goal is to have at least one TSO packet every 1ms, to trigger more frequent packet sends, and hence better acknowledgment clocking and fewer microbursts for flows with low rates. TSO autosizing, FQ, and pacing together achieve traffic shaping in the Linux stack [18].

In practice, FQ/pacing and HTB are used either individually or in tandem, each for its specific purpose. HTB rate limits flow aggregates, but scales poorly with the number of rate limited aggregates and the packet rate (§3). FQ/pacing is used for pacing individual TCP connections, but this solution does not support flow aggregates. We use Policers in hypervisor switch deployments where HTB or FQ/pacing are too CPU intensive to be useful.

3 THE COST OF SHAPING

Traffic shaping is a requirement for the efficient and correct operation of production networks, but uses CPU cycles and memory on datacenter servers that can otherwise be used for applications. In this section, we quantify these tradeoffs at scale in a large Cloud service.

Policiers: Among the three shapers, Policers are the simplest and cheapest. They have low memory overhead since they are bufferless, and they have small CPU overhead because there is no need to schedule or manage queues. However, Policers have poor conformance to the target rate. Figure 3 shows that as round-trip time (RTT) increases,

[2] The limit is controlled via a knob whose default value of 128KB yields two full-sized 64KB TSO segments.

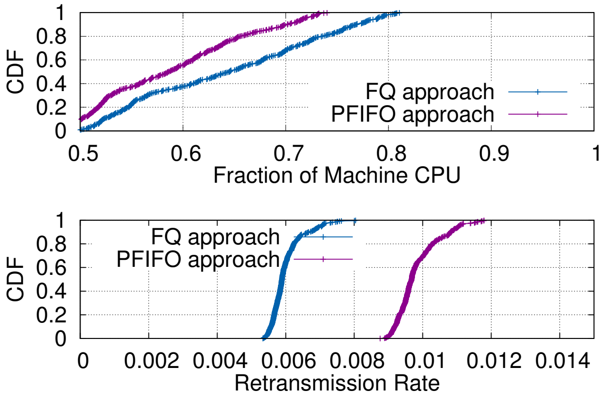


Figure 4: Impact of FQ/pacing on a popular video service. Retransmission rate on paced connections is 40% lower than that on non-paced connections (lower plot). Pacing consumes 10% of total machine CPU at the median and the tail (top plot). The retransmission rates and CPU usage are recorded over 30 second time intervals over the experiment period.

Policers deviate by 10x from the target rate for two reasons: first by dropping non-conformant packets, Policers lose the opportunity to schedule them at a future time, and resort to emulating a target rate with on/off behavior. Second, Policers trigger poor behavior from TCP, where even modest packet loss leads to low throughput [21] and wasted upstream work.

To operate over a large bandwidth-delay product, Policers require a large configured burst size, e.g., on the order of the flow round-trip time. However, large bursts are undesirable, due to poor rate conformance at small time scales, and tail dropping at downstream switches with shallow buffers. Even with a burst size as large as one second, Figure 3 shows that the achieved rate can be 5x below the target rate, for an RTT of 100ms using TCP CUBIC.

Pacing: FQ/pacing provides good rate conformance, deviating at most 6% from the target rate (§7), but at the expense of CPU cost. To understand the impact on packet loss and CPU usage, we ran an experiment on production video servers where we turn off FQ/pacing on 10 servers, each serving 37Gbps at peak across tens of thousands flows. We instead configured a simple PFIFO queueing discipline typically used as the default low overhead Qdisc [1]. We ensured that the experiment and baseline servers have the same machine configuration, are subject to similar traffic load, including being under the same load balancers, and ran the experiments simultaneously for one week.

Figure 4 compares the retransmission rate and CPU utilization with and without FQ/pacing (10 servers in each setting). The upside of pacing is that the retransmission rate on paced connections is 40% lower than those on non-paced connections. The loss-rate reduction comes from reducing self-inflicted losses, e.g., TSO or large congestion windows. The flip side is that pacing consumes 10% of total CPU on the machine at the median and the tail (90th and 99th percentiles); e.g., on a 64 core server, we could save up to 6 cores through more efficient rate limiting.

To demonstrate that the cost of pacing is not restricted to FQ/pacing Qdisc or its specific implementation, we conduct an experiment with QUIC [34] running over UDP traffic. QUIC is a transport protocol designed to improve performance for HTTPS traffic. It is globally deployed at Google on thousands of servers and is used to serve YouTube video traffic. QUIC runs in user space with a packet pacing implementation that performs MTU-sized pacing independent from the kernel's FQ/pacing. We experimented with QUIC's pacing turned ON and OFF. We obtained similar results with QUIC/UDP experiments as we obtained with the kernel's FQ/pacing. Figure 5 shows the CPU consumption of a process generating approximately 8Gbps QUIC traffic at peak. With pacing enabled, process CPU utilization at the 99th percentile jumped from 0.15 to 0.2 of machine CPU – a 30% increase, due to arming and serving timers for each QUIC session. Pacing lowered retransmission rates from 2.6% to 1.6%.

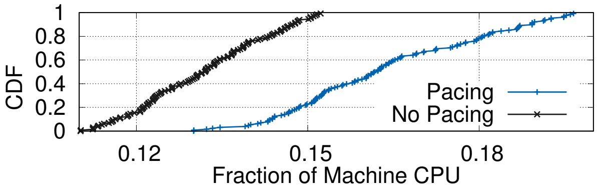


Figure 5: Pacing impact on a process generating QUIC traffic.

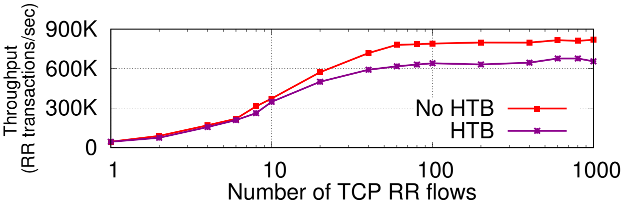
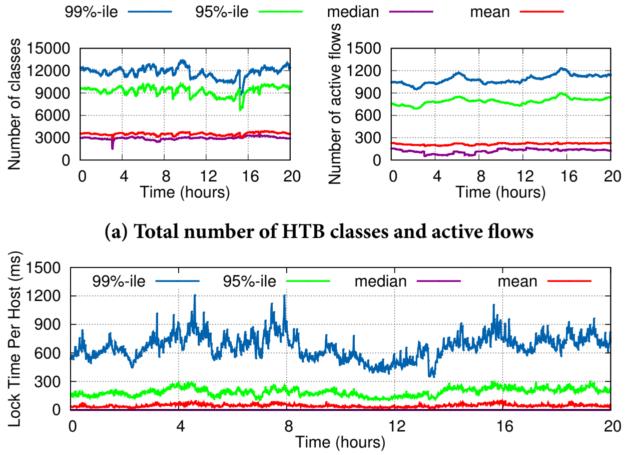


Figure 6: TCP-RR rate with and HTB. HTB is 30% lower due to locking overhead.

HTB: Like FQ/pacing, HTB can also provide good quality rate conformance, with a deviation of 5% from target rate (§7), albeit at the cost of high CPU consumption. CPU usage of HTB grows linearly with the packets per second (PPS) rate. Figure 6 shows an experiment with Netperf TCP-RR, request-response ping-pong traffic of 1 Byte in each direction, running with and without HTB. This experiment measures the overhead of HTB's shaping architecture. We chose the 1-byte TCP-RR because it mostly avoids overheads other than those specifically related to packets per second. In production workloads, it is desirable for shapers (and more generally networking stacks) to achieve high PPS rates while using minimal CPU. As the offered PPS increases in Figure 6 (via the increase in the number of TCP-RR connections on the x-axis), HTB saturates the machine CPU at 600K transactions/sec. Without HTB, the saturation rate is 33% higher at 800K transactions/sec.

The main reason for HTB's CPU overhead is the global Qdisc lock acquired on every packet enqueue. Contention for the lock grows with PPS. [15] details the locking overhead in Linux Qdisc, and the increasing problems posed as links scale to 100Gbps. Figure 7 shows HTB statistics from one of our busiest production clusters over a 24-hour period. Acquiring locks in HTB can take up to 1s at the 99^{th} percentile, directly impacting CPU usage, rate conformance, and tail latency. The number of HTB classes at any instant on a server is tens



(a) Total number of HTB classes and active flows

(b) HTB lock acquire time

Figure 7: HTB statistics over one day from a busy production cluster.

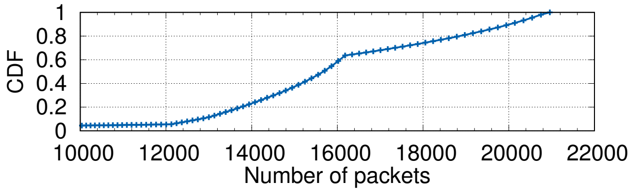


Figure 8: Buffered packets of a VM at a shaper in hypervisor. The bump in the curve is because there are two internal limits on the number of buffered packets: 16K is a local per-VM limit, and 21K is at the global level for all VMs. There are packet drops when the 16K limit is exceeded, which impacts the slope of the curve.

of thousands at the 90th percentile, with 1000-2000 actively rate limited.

Memory: There is also a memory cost related to traffic shaping: the memory required to queue packets at the shaper, and the memory to maintain the data structures implementing the shaper. Without backpressure to higher level transports like TCP, the memory required to queue packets grows with the number of flows × congestion window, and reaches a point where packets need to be dropped. Figure 8 shows the number of packets at a rate limiter in the absence of backpressure for a Cloud VM that has a single flow being shaped at 2Gbps over a 50ms RTT path. In the 99th percentile, the queue of the shaper has 21 thousand MTU size packets (or 32 MB of memory) because of CUBIC TCP's large congestion window, adding 120ms of latency from the additional buffering. In the presence of backpressure (§5), there will be at most two TSO size worth of packets or 85 MTU sized packets[3] (or 128KB of memory), i.e., two orders of magnitude less memory.

A second memory cost results from maintaining the shaper's data structures. The challenge is partitioning and constraining resources such that there are no drops during normal operation; e.g., queues in HTB classes can be configured to grow up to 16K packets, and FQ/pacing has a global limit of 10K packets and a per queue limit of 1000 packets, after which the packets are dropped even if there is plenty of global memory available. With poorly tuned configurations, packets within a queue may be dropped even while there is plenty of room in the global pool.

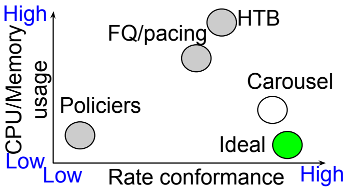


Figure 9: Policers have low CPU usage but poor rate conformance. HTB and FQ/pacing have good shaping properties but a high CPU cost.

Summary: The problem with HTB and FQ/pacing is more than just accuracy or timer granularities. The CPU inefficiency of HTB and FQ/pacing is intrinsic to these mechanisms, and not the result of poor implementations. The architecture of these shapers is based on synchronization across multiple queues and cores, which increases the CPU cost:

1) The token bucket architecture of FQ/pacing and HTB necessitates multiple queues, with one queue per rate limit to avoid head of line blocking. The cost of maintaining these queues grows, sometimes super-linearly, with the number of such queues. CPU costs stem from multiple factors, especially the need to poll queues for packets to process. Commonly, schedulers either use simple round-robin algorithms, where the cost of polling is linear in the number of queues, or more complex algorithms tracking active queues.

2) Synchronization across multiple cores: Additionally, the cost on multi-CPU systems is dominated by locking and/or contention overhead when sharing queues and associated rate limiters between CPUs. HTB and FQ/pacing acquire a global lock on a per-packet basis, whose contention gets worse as the number of CPUs increases.

Solving these problems requires a fundamentally different approach, such as the one we describe in the following sections. Today, practitioners have a difficult choice amongst rate limiters: Policers have high CPU/memory efficiency but unacceptable shaping performance. HTB and FQ/pacing have good shaping properties, but fairly high CPU and memory costs. Figure 9 summarizes this tradeoff. 10% CPU overhead from accurate shaping can be worth the additional network efficiency, especially for WAN transfers. Now, the question becomes whether we can get the same capability with substantially less CPU overhead.

4 CAROUSEL DESIGN PRINCIPLES

Our work begins with the observation that a unified, accurate, and CPU-efficient traffic shaping mechanism can be used in a variety of important settings. Our design principles follow naturally from the following requirements:

1) Work compatibly with higher level congestion control mechanisms such as TCP.

a)**Pace packets correctly**: avoid bursts and unnecessary delays.

b) **Provide backpressure and avoid packet drops**: delaying packets because of rate limiting should quickly result in slowing down the end application. Each additional packet generated by the application will otherwise need to be buffered or dropped in the shaper, wasting memory and CPU to either queue or regenerate the packet. Additionally, loss-based congestion control algorithms often react to

[3] We assume maximum TSO size of 64KB and MTU size of 1.5KB.

packet loss by significantly slowing down the corresponding flows, requiring more time to reach the original bandwidth.

c) **Avoid head of line blocking**: shaping packets belonging to a traffic aggregate should not delay packets from other aggregates.

2) Use CPU and memory efficiently: Rate limiting should consume a very small portion of the server CPU and memory. As implied by §3, this means avoiding the processing overhead of multiple queues, as well as the associated synchronization costs across CPUs in multi-core systems.

We find that these requirements are satisfied by a shaper based on three simple tenets:

1. Single Queue Shaping (§5.1): Relying on a single queue to shape packets alleviates the inefficiency and overhead with multi-queue systems which use token buckets, because these require a queue per rate limit. We employ a single queue indexed by time, e.g., a Calendar Queue [16] or Timing Wheel [43]. We insert all packets from different flows into the same queue and extract them based on their transmission schedule. We set a send-time timestamp for each packet, based on a combination of its flow's rate limit, pacing rate, and bandwidth sharing policy.

2. Deferred Completions (§5.2): The sender must limit the number of packets in flight per flow, potentially using existing mechanisms such as TCP Small Queues or the congestion window. Consequently, there must be a mechanism for the network stack to ask the sender to queue more packets for a specific flow, e.g., completions or acknowledgments. With appropriate backpressure such as *Deferred Completions* described in §5.2, we reduce both HoL blocking and memory pressure. Deferred Completions is a generalization of the layer 4 mechanism of TCP Small Queues to be a more general layer 3 software switch mechanism in a hypervisor/container software switch.

3. Silos of one shaper per-core (§5.3): Siloing the single queue shaper to a core alleviates the CPU inefficiency that results from locking and synchronization. We scale to multi-core systems by using independent single-queue shapers, one shaper per CPU. Note that a system can have as few as one shaper assigned to a specific CPU, i.e. not every CPU requires a shaper. To implement shared rates across cores we use a re-balancing algorithm that periodically redistributes the rates across CPUs.

Figure 10 illustrates the architecture using Timing Wheel and Deferred Completions. Deferred Completions addresses the first of the requirements on working compatibly with congestion control. Deferred Completions allows the shaper to slow down the source, and hence avoids unnecessary drops and mitigates head of line blocking. Additionally, Deferred Completions also allows the shaper to buffer fewer packets per flow, reducing memory requirements. The use of a single Timing Wheel per core makes the system CPU-efficient, thus addressing the second requirement.

Carousel is a more accurate and CPU-efficient than state-of-art token-bucket shapers. Carousel's architecture of a single queue per core, coupled with Deferred Completions can be implemented anywhere below the transport, in either software or NIC hardware.

To demonstrate the benefits of our approach, we explore shaping egress traffic, in §5 using Deferred Completions for backpressure, and shaping ingress incast traffic at the receiver, in §6 using ACKs deferral for backpressure.

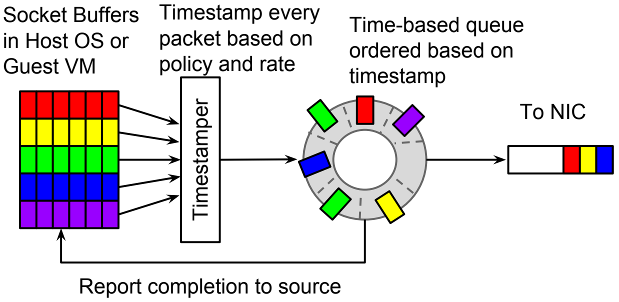

Socket Buffers in Host OS or Guest VM
Timestamp every packet based on policy and rate
Time-based queue ordered based on timestamp
To NIC
Timestamper
Report completion to source

Figure 10: Carousel architecture.

5 THE CAROUSEL SYSTEM

Carousel shaping consists of three sequential steps for each packet. First, the network stack computes a timestamp for the packet. Second, the stack enqueues the packet in a queue indexed by these timestamps. Finally, when the packet's transmission deadline passes, the stack dequeues the packet and delivers a completion event, which can then transmit more packets.

We implement Carousel in a software NIC in our servers. Software NICs dedicate one or more CPU cores for packet processing, as inSoftNIC and FlexNIC [24, 31]. Our software NIC operates in busy-wait polling mode. Polling allows us to check packet timestamps at high frequency, enabling accurate conformance with the packets' schedules. Polling also assists in CPU-efficient shaping, by alleviating the need for locking, and for timers for releasing packets.

We note that Software NICs are not a requirement for Carousel, and it is possible to implement Carousel in the kernel. However, an implementation in the kernel would not be a simple modification of one of the existing Queuing Disciplines [10], and would require re-engineering the kernel's Qdisc path. This is because the kernel is structured around parallelism at the packet level, with per-packet lock acquisitions, which does not scale efficiently to high packet rates. Our approach is to replicate the shaping-related data structure per core, and to achieve parallelism at the level of flows, not at the level of a shared shaper across cores. Since Software NICs already uses dedicated cores, it was easy to rapidly develop and test a new shaper like Carousel, which is a testament to their value.

5.1 Single Queue Shaping

We examine the two steps of emulating the behavior of multiqueue architecture using a single time-indexed queue while improving CPU efficiency.

5.1.1 Timestamp Calculation. Timestamp calculation is the first step of traffic shaping in Carousel. Timestamp of a packet determines its *release time*, i.e., the time at which packet should be transmitted to the wire. A single packet can be controlled by multiple shaping policies, e.g., transport protocol pacing, per-flow rate limit, and aggregate limit for the flow's destination. In typical settings, these policies are enforced independently, each at the point where the policy is implemented in the networking stack. Carousel requires that at each point a packet is timestamped with transmission time according to the rate at that point rather than enforcing the rate there. Hence, these policies are applied sequentially as the packet trickles down the layers of the networking stack. For example, the TCP stack first timestamps each packet with a release time to introduce inter-packet gaps within a connection (pacing); subsequently, different rate limiters in the packet's path modify the packet's timestamp to conform to one or more rates corresponding to different traffic aggregates.

The timestamping algorithm operates in two steps: 1) timestamp calculation based on individual policies, such as pacing and rate limiting, and 2) consolidation of the different timestamps, so that the packet's final transmission time adheres to the overall desired behavior. The packet's final timestamp can be viewed as the *Earliest Release Time* (ERT) of the packet to the wire.

Setting Timestamps: Both pacing and rate limiting use the same technique for calculating timestamps. Timestamps can be set as either relative timestamps, i.e., transmit after a certain interval relative to *now*, or the absolute wall time timestamps, i.e., transmit at a certain time. In our implementation, we use the latter. Both pacing and rate limiting force a packet to adhere to a certain rate: 2×cwnd/RTT for pacing, and a target rate for rate limiting. In our implementation, the timestamp for pacing is set by sender's TCP, and that for target rate is set by a module in the Software NIC, which receives external rates from centralized entities. Policy i enforces a rate R_i and keeps track of the latest timestamp it creates, LTS_i. The timestamp for the j^{th} packet going through the i^{th} policy, P_j^i is then calculated as: $LTS_i' = LTS_i + len(P_j^i)/R_i$. Timestamps are thus advanced on every packet. Note that packets through a policy can belong to the same flow, e.g., in the case of pacing, or multiple flows, e.g., in case of rate limiting flow aggregates. Packets that are neither paced nor rate limited by any policy carry a timestamp of zero.

Consolidating Timestamps: Consolidation is straightforward, since larger timestamps represent smaller target rates. In the course of a packet's traversal through multiple policies, we choose the largest of timestamps, which avoids violating any of the policies associated with the packet. Note that the effective final calculation of a flow's rate is equivalent to a series of token buckets, as the slowest token bucket ends up dominating and slowing a flow to its rate. We also note that Carousel is not a generic scheduler. Hence, this consolidation approach will not work for all scheduling policies (further discussed in Section 8).

5.1.2 Single Time-indexed Queue.

One of the main components of Carousel is Timing Wheel [43], an efficient queue slotted by time and a special case of the more generic Calendar Queue [16]. The key difference is that Calendar Queue has O(1) amortized insertion and extraction but can be O(N) in skewed cases. The constant for O(1) in Timing Wheel is smaller than that of Calendar Queue. A Calendar Queue would provide exact packet ordering based on timestamps even within a time slot, a property that we don't strictly require for shaping purposes.

A Timing Wheel is an array of lists where each entry is a timestamped packet. The array represents the time slots from *now* till the preconfigured time *horizon*, where each slot represents a certain time range of size g_{min} within the overall horizon, i.e., the minimum time granularity. The array is a circular representation of time, as once a slot becomes older than *now* and all its elements are dequeued, the slot is updated to represent *now* + *horizon*. The number of slots in the array is calculated as $\frac{horizon}{g_{min}}$. Furthermore, we allow extracting packets every g_{min}. Within the g_{min} period, packets are extracted in FIFO order. The Timing Wheel is suitable as a single time-indexed queue operated by Carousel. We seek to achieve line rate for the aggregate of tens of thousands of flows. Carousel relies on a single queue to maintain line rate i.e. supporting tens of thousands of entries with minimal enqueue and dequeue overhead. The Timing Wheel's O(1) operations makes it a perfect design choice.

Algorithm 1 Timing Wheel implementation.

```
procedure INSERT(PACKET, TS)
    $Ts = Ts/Granularity$
    if $Ts <= FrontTimestamp$ then
        $Ts = FrontTimestamp$
    else if $Ts > FrontTimestamp + NumberOfSlots - 1$ then
        $Ts = FrontTimestamp + NumberOfSlots - 1$
    end if
    $TW[Ts\%NumberOfSlots].append(Packet)$
end procedure
procedure EXTRACT($now$)
    $now = now/Granularity$
    while $now >= FrontTimestamp$ do
        if $TW[now\%NumberOfSlots].empty()$ then
            $FrontTimestamp+ = Granularity$
        else
            return $TW[now\%NumberOfSlots].PopFront()$
        end if
    end while
    return Null
end procedure
```

With the Timing Wheel, Carousel keeps track of a variable *now* representing current time. Packets with timestamps older than *now* do not need to be queued and should be sent immediately. Furthermore, Carousel allows for configuring *horizon*, the maximum between *now* and the furthermost time that a packet can be queued upto. In particular, if r_{min} represents the minimum supported rate, l_{max} is the maximum number of packets in the queue belonging to the same aggregate that is limited to r_{min}, then the following relation holds: $horizon = \frac{l_{max}}{r_{min}}$ seconds. As Carousel is implemented within a busy polling system, the Software NIC can visit Carousel with a maximum frequency of f_{max}, due to having to execute other functions associated with a NIC. This means that it is unnecessary to have time slots in the queue representing a range of time smaller than $\frac{1}{f_{max}}$, as this granularity will not be supported. Hence, the minimum granularity of time slots is $g_{min} = \frac{1}{f_{max}}$.

An example configuration of the Timing Wheel is for a minimum rate of 1.5Mbps for 1500B packet size, slot granularity of 8us, and a time horizon of 4 seconds. This configuration yields 500K slots, calculated as $\frac{horizon}{g_{min}}$. The minimum rate limit or pacing rate supported is one packet sent per time horizon, which is 1.5Mbps for 1500B packet size. The maximum supported per-flow rate is 1.5Gbps. This cap on maximum rate is due to the minimum supported gap between two packets of 8μs and the packet size of 1500B. Higher rates can be supported by either increasing the size of packets scheduled within a slot (e.g., scheduling a 4KB packet per slot increases the maximum supported rate to 4Gbps), or by decreasing the slot granularity.

We note that Token Bucket rate limiters have an explicit configuration of *burst* size that determines the largest line rate burst for a flow. Such a burst setting can be realized in Carousel via timestamping such that no more than a *burst* of packets are transmitted at line rate. For example, a train of consecutive packets of a total size of *burst* can be configured to be transmitted at the same time or within the same timeslot. This will create a burst of that size.

Algorithm 1 presents a concrete implementation of a Timing Wheel. There are only two operations: `Insert` places a packet at a slot determined by the timestamp and `ExtractNext` retrieves the first enqueued packet with a timestamp older than the current time, *now*. This allows the Timing Wheel to act as a FIFO if all packets have a timestamp of zero. All packets with timestamp older than *now* are inserted in the slot with the smallest time.

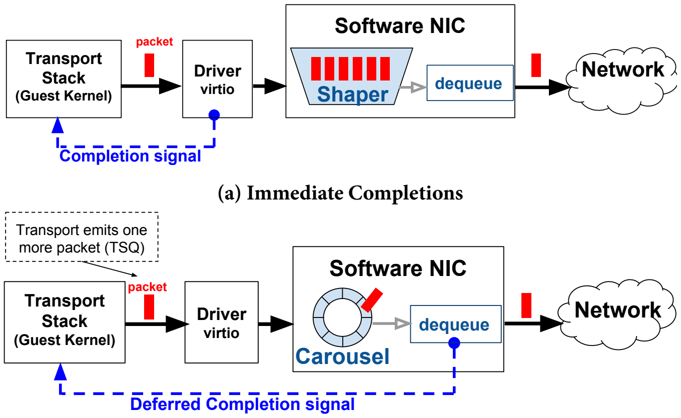
We have two options for packets with a timestamp beyond the horizon. In the first option, the packets are inserted in the end slot that represents the timing wheel horizon. This approach is desirable when only flow pacing is the goal as it doesn't drop packets, but instead results in a temporary rate overshoot because of bunched packets at the horizon. The second approach is to drop packets beyond the horizon, which is desired when imposing a hard rate limit on flow, and an overshoot is undesirable.

A straightforward Timing Wheel implementation supports O(1) insertion and extraction where a list representing a slot is a dynamic list, e.g., `std::list` in C++. Memory is allocated and deallocated on every insertion and deletion within the dynamic list. Such memory management introduces significant cost per packet. Hence, we introduce a *Global Pool* of memory: a free list of nodes to hold packet references. Timing Wheel allocates from the free list when packets are enqueued. To further increase efficiency, each free list node can hold references to *n* packets. Preallocated nodes are assigned to different time slots based on need. This eliminates the cost of allocation and deallocation of memory per packet, and amortizes the cost of moving nodes around over *n* packets. Freed nodes return to the global pool for reuse by other slots based on need. §7 quantifies the Timing Wheel performance.

5.2 Deferred Completions

Carousel provides backpressure to higher layer transport without drops, by bounding the number of enqueued packets per flow. Otherwise, the queue can grow substantially, causing packet drops and head of line blocking. Thus, we need mechanisms in the software NIC or the hypervisor to properly signal between Carousel and the source. We study two such approaches in detail: *Deferred Completions* and delay-based congestion control.

Completion is a signal reported from the driver to the transport layer in the kernel signaling that a packet left the networking stack (i.e. *completed*). A completion allows the kernel stack to send more packets to the NIC. The number of packets sent by the stack to the NIC is transport dependent, e.g., in the case of TCP, TSQ [4] ensures that the number of packets outstanding between the stack and the generation of a completion is at most two segments. Unreliable protocols like UDP could in theory generate an arbitrary number of packets. In practice, however, UDP sockets are limited to 128KB of outstanding send data.


Transport Stack (Guest Kernel)
packet
Driver virtio
Software NIC
Shaper
dequeue
Network
Completion signal
Transport emits one more packet (TSQ)
Transport Stack (Guest Kernel)
packet
Driver virtio
Software NIC
Carousel
dequeue
Network
Deferred Completion signal

(a) Immediate Completions

(b) Deferred Completions

Figure 11: Illustration of Completion signaling. In (a), a large backlog can build up to substantially overwhelm the buffer, leading to drops and head-of-line blocking. In (b), only two packets are allowed in the shaper and once a packet is released another is added. Indexing packets based on time allows for interleaving packets.

The idea behind Deferred Completions, as shown in Figure 11, is *holding* the completion signal until Carousel transmits the packet to the wire, preventing the stack from enqueueing more packets to the NIC or the hypervisor. In Figure 11a, the driver in the guest stack generates completion signals at enqueue time, thus overwhelming the Timing Wheel with as much as a congestion window's worth of packets per TCP flow, leading to drops past the Timing Wheel horizon. In Figure 11b, Carousel in Software NIC returns the completion signal to the guest stack only after packet is dequeued from the Timing Wheel, and thus strictly bounds the number of packets in the shaper, since the transport stack relies on completion signals to enqueue more packets to Qdiscs.

Implementing Deferred Completions requires modifying the way completions are currently implemented in the driver. Completions are currently delivered by the driver in the order in which packets arrive at the NIC, regardless of their application or flow. Carousel can transmit packets in an order different from their arrival because of shaping. This means that a flow could have already transmitted packets to the wire, but be blocked because the transport layer has not received its completion yet. Consider a scenario of two competing flows, where the rate limit of one flow is half of the other. Both flows deliver packets to the shaper at the same arrival rate. However, packets of the faster flow will be transmitted at a higher rate. This means that packets from the faster flow can arrive at the shaper later than the packets from the slow flow and yet be transmitted first. The delivery rate of completion signals will need to match the packet transmission rate, or else the fast flow will be unnecessarily backpressured and slowed down. Ordered completion delivery can cause head of line blocking, hence we implemented out-of-order completions in the driver of the guest kernel.

We implement out-of-order completions by tracking of all the packets still held by Carousel, in a hashmap in the guest kernel driver. On reception of a packet from guest transport stack, the Software NIC enqueues the reference to that packet in the hashmap. When Carousel releases the packet to the network, the Software NIC generates a completion signal, which is delivered to the guest transport stack and allows removal of the packet from the hashmap. The removal of the packet from the hashmap is propagated up in the stack, allowing TSQ to enqueue another packet [19]. This requires changing both the driver and its interface in the guest kernel, i.e., virtio [40].

Carousel is implemented within a Software NIC that relies on zero-copy to reduce memory overhead within the NIC. This becomes especially important in the presence of Carousel, because when the Software NIC shapes traffic, it handles more packets as it would without any shaping. Hence, the Timing Wheel carries references (pointers) to packets and not the actual packets themselves. The memory cost incurred is thus roughly the cost of the number of references held in Carousel, while the packet memory is fully attributed to the sending application until the packet is sent to the wire (completed). In practice, this means that for every one million of packets outstanding in the Timing Wheel our software NIC needs to allocate only in the order of 8MB of RAM. The combination of Deferred Completions and holding packet references provides backpressure while using only a small memory footprint.

Delay-based Congestion Control (CC) is an alternative form of backpresssure to Deferred Completions. Delay-based CC senses the delay introduced in the shaper, and modulates TCP's congestion window to a smaller value as the RTT increases. The congestion window serves as a limit on the number of packets in Timing Wheel. We note that delay-based CC should discount the pacing component of the delay when computing the RTT. Operating with an RTT that includes pacing delay introduces undesirable positive feedback loops that reduce throughput. On the other hand, for Retransmission Timeout (RTO) computations, it is crucial that pacing delay be included, so as to avoid premature firing of the RTO timer. §7 provides a comparison across backpressure mechanisms.

5.3 Scaling Carousel with multiple cores

So far we have described the use of a single Timing Wheel per CPU core to shape traffic. Using a single shaper in multi-core systems incurs contention costs that would make the solution impractical. It is easy to scale Carousel by using independent Timing Wheels, one per CPU. For our approach, each TCP connection is hashed to a single Timing Wheel. Using one shaper per core suffice for pacing or rate limiting individual flows, and allows lock-free queuing of packets to a NIC instance on a specific core.

However, this simple approach does not work for the case when we want to shape multiple TCP connections by a single aggregate rate, because the individual TCP connections land on different cores (through hashing) and hence are shaped by different Timing Wheels. We resolve this with two components:

1) NBA: Our implementation uses a *NIC-level bandwidth allocator* (NBA) that accepts a total rate limit for flow-aggregates and redistributes each rate limit periodically to individual per-core Timestampers. NBA uses a simple water-filling algorithm to achieve work-conservation and max-min fairness. The individual Timestampers provide NBA with up-to-date flow usage data as a sequence of (byte count, timestamp of the byte count measurement) pairs; rate assignments from NBA to Timestampers are in bytes per second. The water-filling algorithm ensures that if flows on one core are using less than their fair share, then the extra is given to flows on the remaining cores. If flows on any core have zero demand, then the central algorithm still reserves 1% of the aggregate rate, to provide headroom for ramping up.

2) Communication between NBA and per-core Timestampers: Usage updates by Timestampers and the dissemination of rates by NBA are performed periodically in a lazy manner. We chose lazy updates to avoid the overhead of locking the data path with every update. The frequency of updates is a tradeoff between stability and fast convergence of computed rates; we currently use ≈100ms.

6 CAROUSEL AT THE RECEIVER

We apply the central tenets discussed in §4 to show how a receiver can shape flows on the ingress side. Our goal is to allow overwhelmed receivers to handle incast traffic through a fair division of bandwidth amongst flows [12]. The direct way of shaping ingress traffic to queue incoming data packets, which can consume considerable memory. Instead, we shape the acknowledgements reported to the sender. Controlling the rate of acknowledged data allows for fine grained control over the sender's rate.

The ingress shaping algorithm modifies the acknowledgment sequence numbers in packets' headers, originally calculated by TCP, to acknowledge packets at the configured target rate. We apply Carousel as an extension of DCTCP without modifying its behavior. The algorithm keeps track of a per flow Last Acked Sequence Number (SN_a), Latest Ack Time (T_a), and Last Received Sequence Number (SN_r). The first two variables keep track of the sequence number of last-acknowledged bytes and the time that acknowledgement was sent. For an outgoing packet from the receiver to the sender, we check the acknowledgement number in the packet. We use that to update SN_r. Then, we change that number based on the following formula: $NewAckSeqNumber = \min((now - T_a) \times Rate + SN_a, SN_r)$. We update T_a to now and SN_a to $NewAckSeqNumber$, if SN_a is smaller than $NewAckSeqNumber$. Acknowledgments are generated by Carousel when no packet has been generated by the upper layer for the time $\frac{MTU}{Rate}$. The variable $Rate$ is set either through a bandwidth allocation system, or it is set as the total available bandwidth divided by the number of flows.

7 EVALUATION

We evaluate Carousel in small-scale microbenchmarks where all shaped traffic is between machines within the same rack, and in production experiments on machines serving video content to tens of thousands of flows per server. We compare the overhead of Carousel to HTB and FQ/pacing in similar settings and demonstrate that Carousel is 8% more efficient in overall machine CPU utilization (20% more efficient in CPU utilization attributable to networking), and 6% better in terms of rate conformance, while maintaining similar or lower levels of TCP retransmission rates. Note that the video serving system is just for the convenience of production experiments in this section; the same shaping mechanisms are also deployed in our network operations such as enforcing Bandwidth Enforcer (BwE) rates.

7.1 Microbenchmark

Experiments setup: We conduct experiments for egress traffic shaping between two servers sharing the same top of rack switch. The

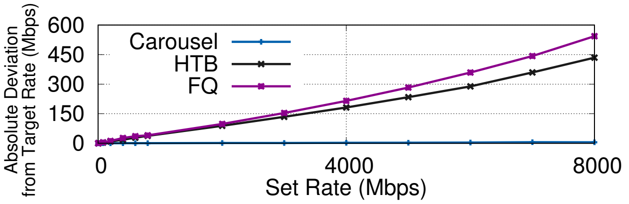


Figure 12: Comparison between Carousel, HTB, and FQ in their rate conformance for a single flow.

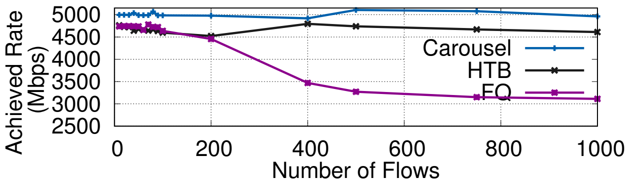


Figure 13: Comparison between Carousel, HTB, and HTB/FQ showing the effect of the number of flows load on their rate conformance to a target rate of 5 Gbps.

servers are equipped with software NIC on which we implemented Carousel to shape traffic. For baseline, servers run Linux with HTB configured with 16K packets per queue; FQ/pacing is configured with a global limit of 10K packets and a flow limit of 1000 packets. These settings follow from best practices. We generate traffic with `neper` [7], a network performance measurement tool that allows for generating large volumes of traffic with up to thousands of flows per server. We vary RTT using `netem`. All reported results are for an emulated RTT of 35ms. We found the impact of RTT on Carousel to be negligible. Experiments are run for 30 seconds each and enough number of runs to make the standard deviation 1% or smaller. Unless otherwise mentioned, the Timing Wheel granularity is two microseconds with a horizon of two seconds.

7.1.1 Rate Conformance. We measure rate conformance as the absolute deviation from the target rate. This metric represents the deviation in bytes per second due to shaping errors. We measure absolute deviation from target rate by collecting achieved rate samples once every 100 milliseconds. We collect samples at the output of the shaper, i.e., on the sender, as we want to focus on the shaper behavior and avoid factoring in network impact on traffic. Samples are collected by using `tcpdump` where throughput is calculated by measuring the amount of bytes sent every interval of 100 milliseconds.

We compare Carousel's performance to HTB and FQ/pacing: 1) pacing is performed by FQ/pacing on a per-flow basis where a maximum pacing rate is configured in TCP using `SO_MAX_PACING_RATE` [5], and 2) rate limiting where HTB enforces rate on a flow aggregate.

Change target rate. We investigate rate conformance for a single TCP connection. As the target rate is varied, we find that HTB and FQ have a consistent 5% and 6% deviation between the target and achieved rates (Figure 12). This is due to their reliance on timers which fire at lower rates than needed for accuracy at high rates. For instance, HTB's timer fires once every 10ms. Carousel relies on accurate scheduling of packets in busy-wait polling Software NIC, by placing them in a Timing Wheel slot that transmits a packet within a slot size of its scheduled time.

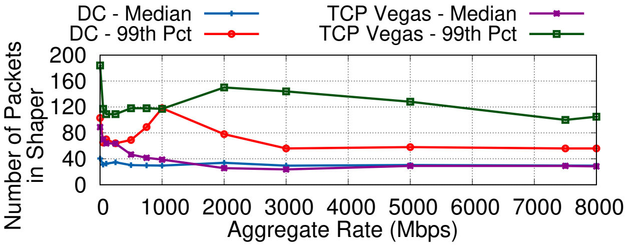


Figure 14: A comparison of Deferred Completion (DC) and delay-based congestion control (TCP Vegas) backpressure, for fifteen flows when varying the aggregate rate limits.

Vary the number of flows. We vary the number of flows in our experiments between one and 10K, typical for user facing servers. Figure 13 shows the effect of varying the number of flows until 1000 flows. Carousel is not impacted as it relies on the Timing Wheel with O(1) insertion and extraction times. HTB maintains a constant deviation of 5% from the target rate. However, FQ/pacing conformity significantly drops beyond 200 flows. This is due to the limited number of per-queue and global packet limits. However, increasing this number significantly increases its CPU overhead due to its reliance on RB-trees for maintaining flows and packets schedules [5].

7.1.2 Memory Efficiency. We compare memory efficiency of Deferred Completions and delay-based CC as backpressure schemes (§5.2). We use the metric of number of packets held in the shaper, which directly reflects the memory allocation expected from the shaper along with the possibility of packet drops as memory demand exceeds allocation. We collect samples of the number of packets every time a new packet is extracted from the shaper.

Change target rate. Figure 14 shows that with Deferred Completions, the average of number of packets in the Carousel is not affected by changes in the target rate. While delay-based CC maintains similar average number of packets like in Deferred Completions, it has twice the number of packets in the 99th percentile. This is because delay variations can grow the congestion window in Vegas to large values, while Deferred Completions maintains a strict bound regardless of the variations introduced by congestion control. The bump in 99th percentile at 1Gbps rate for Deferred Completions results from an interaction of TSQ and TSO autosizing. Deferred Completions behavior is consistent regardless of the congestion control algorithm used in TCP. Our experiments for Deferred Completions used TCP Cubic in Linux.

Vary number of flows. In Figure 15, we compare the number of packets held in the shaper for a fixed target rate while varying the number of flows. In the absence of Deferred Completions, the flows continually send until all of the 70K memory in the shaper is filled. For Deferred Completions, the number of packets held in the shaper is deterministic as approximately twice the number of active flows. This is also reflected in the 99th percentile of the number of packets held. For delay-based CC, the median number of packets is higher than the 99th percentile behavior of Deferred Completion. Furthermore, the 99th percentile of delay-based CC is less predictable than that of the

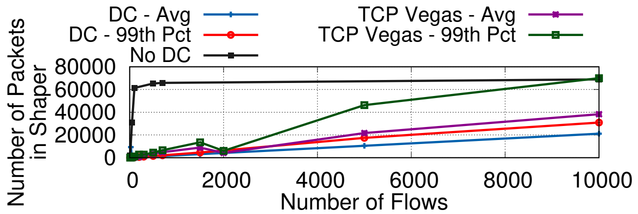


Figure 15: A comparison of Deferred Completion (DC) and delay-based congestion control (TCP Vegas) backpressure, while varying the number of flows with an aggregate rate limit of 5Gbps.

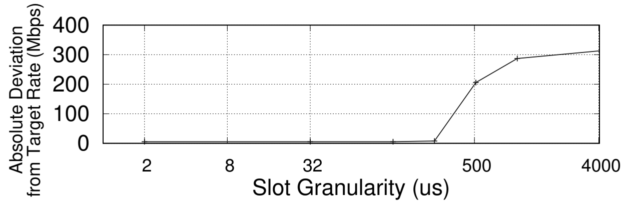


Figure 16: Rate conformance of Carousel for target rate of 5Gbps for different slot granularities.

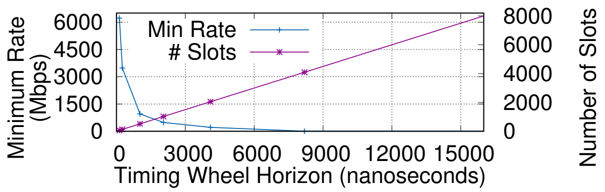


Figure 17: An analytical behavior of the minimum supported rate for a specific Timing Wheel horizon along with the required number of slots for a slot size of two microseconds.

99th percentile of Deferred Completions because it can be affected by variations in RTT. We note that we didn't observe any significant difference in rate conformance between both approaches. We find that Deferred Completion is the better backpressure approach because of its predictable behavior.

There is no hard limit on the number of individual connections or flow aggregates that Carousel can handle. Carousel operates on the references (pointers) to packets, which are small. Surely, we have to allocate memory to hold the references, but in practice this memory is a small fraction of memory available on the machine.

7.1.3 Impact of Timing Wheel Parameters. In §5, we choose the Timing Wheel due to it is high efficiency and predictable behavior. We study the effect of its different parameters on the behavior of Carousel.

Slot Granularity. Slot granularity controls the burst size and also determines the smallest supported gap between packets which affects the maximum rate that Carousel can pace at. Furthermore, for a certain granularity to be supported, the Timing Wheel has to be dequeued at least that rate. We investigate the effect of different granularities on rate conformance. Figure 16 shows that a granularity of 8-16μs suffices to maintain a small deviation from the target rate. We also find that rate conformity at larger slot granularities such as 4ms deviates by 300 Mbps which is comparable to the deviation of HTB and FQ at the same target rate as shown in Figure 12.

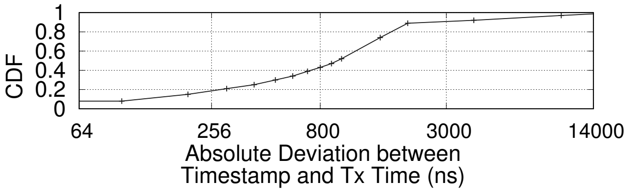


Figure 18: A CDF of deviation between a packet timestamp and transmission time.

Number of Packets in Shaper	1000	4000	32000	256000	20000000
STDList (ns per packet)	22	21	21	21	22
Global Pool (ns per packet)	12	11	11	11	11

Table 1: Overhead per packet of different Timing Wheel implementations.

Timing Wheel Horizon. The horizon represents the maximum delay a packet can encounter in shaper (hence minimum supported rate), and is the product of slot granularity and number of slots. Figure 17, shows the minimum rate and the the number of slots needed for different horizon values, all for a granularity of two microseconds. Note that results here assume that only one flow is shaped by Carousel which means that the horizon needs to be large enough to accommodate the time gap between only two packets.

Deviation between Timestamp and Transmission Time. Figure 18 shows a CDF of the difference between the scheduled transmission time of the packet and the actual transmission time of the packet in nanoseconds. This metric represents the accuracy of the Timing Wheel in how much it adheres to the schedules. Over 90% of packets have deviations smaller than the slot granularity of two microseconds, which occurs due to the rounding down error when mapping packets with schedules calculated in nanoseconds to slots of 2μs. Deviations that exceed a slot size due to excessive queuing delay between the timestamper and the shaper are rare.

CPU Overhead of Timing Wheel. For a system attempting to achieve line rate, every nanosecond counts for processing each packet. The Timing Wheel implementation is the most resource consuming component in the system as the rest of the operations merely change parts of the packet metadata. We explore two parameters that can affect the delay per packet in the Timing Wheel implementation: 1) the impact of the data structures used for representing a slot, and 2) the number of packets held in the Timing Wheel. Table 1 shows the impact of both parameters. It's clear that because of the O(1) insertion and extraction overhead, the Timing Wheel is not affected by the number of packets or flows. The factor affecting the delay per packet is the overhead of insertion and extraction of packets from each slot. We find that implementing a slot data structure that avoids allocating memory for every insertion, i.e., Global Pool (described in §5.1.2), reduces the overhead of C++ `std::list` by 50%.

7.1.4 Carousel Impact at the Receiver. We evaluate the receiver-based shaper in a scenario of 100:1 incast: 10 machines with 10 connections each send incast traffic to one machine; all machines are located in the same rack. Unloaded network RTT is ≈10us, packet size is 1500 Bytes, and experiment is run for 100 seconds. Senders use

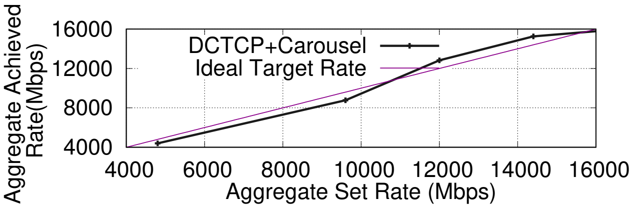


Figure 19: Comparing receiver side rate limiting with target rate.

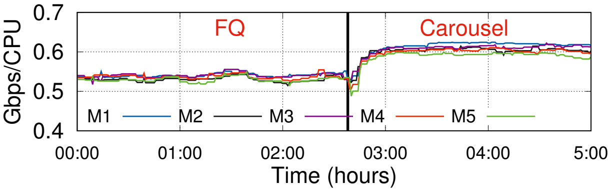


Figure 20: Immediate improvement in CPU efficiency after switching to Carousel in all five machines in a US west coast site.

DCTCP and Carousel to limit the throughput of individual incast flows. Figure 19 shows the rate conformance of Carousel for varying aggregate rates enforced at the receiver. The deviation of aggregate achieved rate is within 1% of the target rate.

7.2 Production Experience

To evaluate Carousel with real production loads, we choose 25 servers in five geographically diverse locations. The servers are deployed in two US locations (East and West Coast), and three European locations (Western and Eastern Europe). They are serving streaming video traffic from a service provider to end clients. Each server generates up to 38Gbps of traffic serving as many as 50,000 concurrently active sessions. For comparison, the servers support both conventional FQ/pacing in Linux kernel and Carousel implemented on software NIC like SoftNIC [24].

The metrics we use are: 1) retransmission rates, 2) server CPU efficiency, and 3) software NIC efficiency. To measure server CPU efficiency, we use Gbps/CPU metric which is the total amount of egress traffic divided by the total number of CPUs on a server and then divided by their utilization. For example, if a server sends 18 Gbps with 72 CPUs that are 50% utilized on average, it is said to have $\frac{18}{72*0.5}$ = 0.5Gbps/CPU efficiency. We present only the data for periods of use when server CPUs are more than 50% utilized (approximately 6 hours every day). Peak periods is when efficiency is the most important because server capacity must be provisioned for such peaks. Gbps/CPU is better than direct CPU load for measuring CPU efficiency. This is because our production system routes new client requests to the least loaded machines to maintain high CPU utilization at all machines. This load-balancing behavior means that more efficient machines will receive more client requests.

CPU efficiency. In the interest of space, we do not show results from all sites. Figure 20 shows the impact of Carousel on Gbps/CPU metric on five machines in the West Coast site. Figure 21 compares two machines one using Carousel and the other using FQ/pacing. We pick these two machines specifically as they have similar CPU utilization during our measurement period. This provides a fair comparison for the Gbps/CPU by unifying the denominator which focuses on how efficiently the CPU is used to push data. At 50^{th} and 90^{th} percentiles, Carousel is 6.4% and 8.2% more efficient (Gbps/CPU) in serving traffic. On 72-CPU machines this translates to saving an average of 4.6 CPUs per machine. Considering that networking accounts for ≈40% of the machine CPU, the savings are 16% and 20% in the median and tail for CPU attributable to networking. The two machines exhibit similar retransmission rate which shows that using Carousel maintains pacing value while significantly reducing its cost.

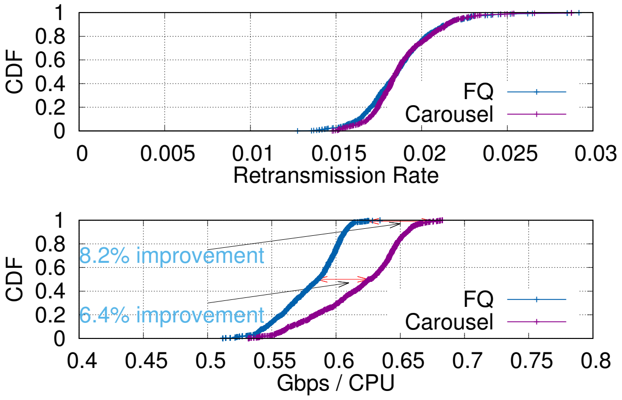


Figure 21: Comparison between FQ and Carousel for two machines serving large volumes of video traffic exhibiting similar CPU loads showing that Carousel can push more traffic for the same CPU utilization.

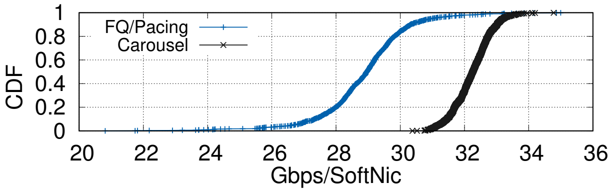


Figure 22: Software NIC efficiency comparison with pacing enforced in kernel vs pacing enforced in software NIC. Bandwidth here is normalized to a unit of software NIC self-reported utilization.

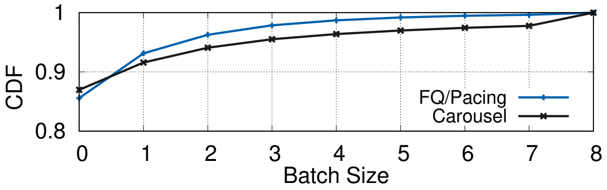


Figure 23: Cumulative batch sizes (in TSO packets) that software NIC receives from kernel when pacing is enforced in kernel vs when pacing is enforced in software NIC.

Carousel on Software NIC. We examine the overhead of Carousel on the software NIC. Surprisingly, we find that moving pacing to software NIC in fact *improves* its performance. The software NIC operates in spin-polling mode, consuming 100% of fixed number of CPUs assigned to it. The NIC reports the CPU cycles it spent in a spin loop and whether it did any useful work in that particular loop (i.e. received a batch that has at least one packet, or there was a packet emitted by timing wheel); the CPU cycles spent in useful loops divided by the total number of CPU cycles in all loops is the Software NIC utilization level. To measure software NIC efficiency, we normalize observed throughput by software NIC self-reported utilization, thus deriving Gbps/SoftNIC metric. Figure 22 shows Gbps/SoftNIC metric for pacing with FQ/pacing and Carousel. We see that moving pacing to the software NIC improves its efficiency by 12% (32.2 Gbps vs 28.8 Gbps per software NIC). The improved efficiency is from larger batching from kernel TCP stack to software NIC, because the pacing is moved from kernel to the NIC. Figure 23 shows cumulative batch sizes, including empty batches, for machines serving production traffic with pacing enforcement in kernel and pacing enforcement in software NIC. On average, non-empty packet batch sizes increased by 57% (from an average batch size of 2.1 to an average batch size of 3.3).

8 DISCUSSION

Application of Carousel in Hardware NICs: Some hardware devices, such as Intel NICs [26, 27], provide support for rate limiting. The common approach requires one hardware queue per rate limiter, where the hardware queue is bound to one or more transmit queues in the host OS. This approach heads in the direction of increasing transmit queues in hardware which does not scale because of silicon constraints. Hence, hardware rate limiters are restricted for use in custom applications, rather than as a normal offload that the OS can use.

With Carousel, the single queue approach is a sharp departure from the trend of needing more number of hardware queues, and can be implemented follows: 1) hardware is configured with one transmit queue per CPU to avoid contention between the OS and the NIC; 2) Classification and rates on the hardware are configured via an API, 3) hardware is able to consume input packets, generate per-packet timestamps based on configured rates, and enqueue them to Timing Wheel, 4) dequeue packets that are due, 5) return the completion event relative to the packet sent.

Limitations: Carousel does not serve as a generic packet scheduler that can realize arbitrary packet schedules because the Timing Wheel does not support changing timestamps of already enqueued packets. Some schedules such as Weighted Fair Queueing are supportable with existing O(1) Enqueue / Dequeue operations, while preemptive schedules such as strict priorities are not supportable.

A second challenge stems from the use of a *single* queue shaper with timestamping of packets. The challenge is how to make such timestamping consistent when the timestamps are not set in one centralized place. An approach is to standardize all sources to set timestamps in nanoseconds (or microseconds) from the Unix epoch time. Other challenges include handling timestamps from unsynchronized CPU clocks, and timestamps that may be of different granularities.

9 RELATED WORK

Improving shaping efficiency: SENIC and vShaper introduce hybrid software/hardware shapers to reduce CPU overhead. SENIC allows the host's CPU to classify and enqueue packets while the NIC is responsible for pulling packets when it is time to transmit them [38]. vShaper allows for sharing queues across different flows by estimating the share each will take based on their demand [33]. In contrast, Carousel relies on a single queue that can be implemented efficiently in either hardware or software. Pacing was first suggested to have a per-flow timer which made pacing unattractive due to the excessive CPU overhead [14]. FQ/pacing in the OS (§5) uses only one timer and still imposes non-trivial CPU overhead (§3). By relying on time-based queue, Carousel implements shaping more efficiently and accurately as packets are extracted based on their release times rather than spending CPU cycles checking different queues for packets that are due.

New data structures for packet scheduling: Recently new data structures in literature implement scheduling at line rate while saving on memory and CPU. UPS presents Least Slack Time First-based packet scheduling algorithm for routers [35]. The Push-In First-Out Queue (PIFO) [42] is a priority queue that allows packets to be enqueued into an arbitrary location in the queue (thus enabling programmable packet scheduling), but only dequeued from the head. Timing Wheel is also a particular kind of PIFO. PIFO and UPS focus on scheduling in switches which require work-conserving algorithms; Carousel focuses on pacing and rate-limiting at end hosts which are non work-conserving. Carousel is similar to VirtualClock [44] seminal work in its reliance on transmission time to control traffic. However, the focus in VirtualClock is on setting timestamps for specific rates that can be specified by a network controller. Carousel presents a data structure that can efficiently implement such techniques, and extends timestamping to apply it for pacing and rate limiting. Furthermore, Carousel relies on Deferred Completions to limit queue length and avoid harmful interaction amongst flows.

Consumers of shaping: Recent congestion control and bandwidth allocation systems [13, 30, 32, 37, 39] include an implementation for shaping, e.g., FQ/pacing for BBR [17], priority queue in TIMELY [36], hardware token bucket queue in HULL [9], injecting void packets in Silo [29], and HTB [2] for BwE [32]. Most proposed techniques rely on variations of token buckets to enforce rates. Carousel proposes a system that combines both pacing and rate limiting to reduce memory and CPU overhead. TIMELY and HULL both strongly motivate hardware implementation of pacers due to the overhead of software pacing. However, TIMELY presents a software pacer that works on a per flow basis. This requires keeping tack of flow information at the pacer which can be a large overhead in the data path. Carousel requires each packet to only be annotated with a timestamp which reduces the pacer overhead as the notion of flows can be kept where it is only required, e.g., TCP layer.

10 CONCLUSION

Even though traffic shaping is fundamental to the correct and efficient operation of datacenters and WANs, deployment at scale has been difficult due to prohibitive CPU and memory overheads. We showed in this paper that two techniques can help overcome scaling and efficiency concerns: a single time-indexed queue per CPU core for packets, and management of higher-layer resources with Deferred Completions. Scaling is further helped by a multicore-aware design. Carousel is able to individually shape tens of thousands of flows on a single server with modest resource consumption.

We believe the most exciting consequence of our work will be the creation of novel policies for pacing, bandwidth allocation, handling incast, and mitigation against DoS attacks. No longer are we restricted to cherry-picking handful of flows to shape. We are confident that our results with Carousel make a strong technical case for networking stacks and NIC vendors to invest in the basic abstractions of Carousel.

11 ACKNOWLEDGMENTS

We thank Ken Durden and Luigi Rizzo for their direct contributions to this work. We thank Yuchung Cheng, Brutus Curtis, Eric Dumazet, Van Jacobson and Erik Rubow for their insightful discussions; Marco Paganini for assisting in several rounds of Carousel experiments on Bandaid servers; Ian Swett and Jeremy Dorfman for conducting pacing experiments with QUIC; David Wetherall and Daniel Eisenbud for their ongoing support of congestion control work. We thank our technical reviewers, Rebecca Isaacs, Jeff Mogul, Dina Papagiannaki and Parthasarathy Ranganathan for providing useful feedback. We thank our shepherd, Andreas Haeberlen, and anonymous SIGCOMM reviewers for their detailed and excellent feedback.

REFERENCES

[1] 2002. pfifo-tc: PFIFO Qdisc. (2002). https://linux.die.net/man/8/tc-pfifo/.

[2] 2003. Linux Hierarchical Token Bucket. (2003). http://luxik.cdi.cz/ devik/qos/htb/.

[3] 2008. net: Generic receive offload. (2008). https://lwn.net/Articles/358910/.

[4] 2012. tcp: TCP Small Queues. (2012). https://lwn.net/Articles/506237/.

[5] 2013. pkt_sched: fq: Fair Queue packet scheduler. (2013). https://lwn.net/Articles/564825/.

[6] 2013. tcp: TSO packets automatic sizing. (2013). https://lwn.net/Articles/564979/.

[7] 2016. neper: a Linux networking performance tool. (2016). https://github.com/google/neper.

[8] 2017. Leaky bucket as a queue. (2017). https://en.wikipedia.org/wiki/Leaky_bucket.

[9] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI '12*. 253–266.

[10] W. Almesberger, J. H. Salim, and A. Kuznetsov. 1999. Differentiated services on Linux. In *GLOBECOM '99*. 831–836 vol. 1b.

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (2010), 50–58.

[12] Wei Bai, Kai Chen, Haitao Wu, Wuwei Lan, and Yangming Zhao. 2014. PAC: taming TCP Incast congestion using proactive ACK control. In *ICNP '14*. 385–396.

[13] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 242–253.

[14] Neda Beheshti, Yashar Ganjali, Monia Ghobadi, Nick McKeown, and Geoff Salmon. 2008. Experimental Study of Router Buffer Sizing. In *IMC '08*. 197–210.

[15] Jesper Dangaard Brouer. 2015. Network stack challenges at increasing speeds: The 100Gbit/s challenge. LinuxCon North America. (2015). http://events.linuxfoundation.org/sites/events/files/slides/net_stack_challenges_100G_1.pdf

[16] Randy Brown. 1988. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Commun. ACM* 31, 10 (1988), 1220–1227.

[17] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Van Jacobson, and Soheil Yeganeh. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* (2016), 50:20–50:53.

[18] Yuchung Cheng and Neal Cardwell. 2016. Making Linux TCP Fast. In *Netdev Conference.*

[19] David D Clark. 1985. The structuring of systems using upcalls. In *SOSP '85*. 171–180.

[20] Glenn William Connery, W Paul Sherer, Gary Jaszewski, and James S Binder. 1999. Offload of TCP segmentation to a smart adapter. US Patent 5,937,169. (August 1999).

[21] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. 2016. An Internet-Wide Analysis of Traffic Policing. In *SIGCOMM '16*. 468–482.

[22] Yashar Ganjali and Nick McKeown. 2006. Update on Buffer Sizing in Internet Routers. *SIGCOMM Comput. Commun. Rev.* (2006), 67–70.

[23] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. 2012. Trickle: Rate Limiting YouTube Video Streaming. In *USENIX ATC '12*. 191–196.

[24] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware.* Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley.

[25] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (2013), 15–26.

[26] Intel Networking Division. 2016. 82599 10 GbE Controller Datasheet. (2016).

[27] Intel Networking Division. 2016. Ethernet Switch FM10000. (2016).

[28] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM '13*. 3–14.

[29] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM '15*. 435–448.

[30] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI '13*. 297–311.

[31] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *ASPLOS '16*. 67–81.

[32] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM '15*. 1–14.

[33] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. 2013. Virtualizing Traffic Shapers for Practical Resource Allocation. In *5th USENIX Workshop on Hot Topics in Cloud Computing.*

[34] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM '17.*

[35] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *NSDI '16*. 501–521.

[36] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM '15*. 537–550.

[37] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 351–362.

[38] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI '14*.

[39] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. In *SIGCOMM '07*. 337–348.

[40] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[41] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM '95*. 375–385.

[42] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM '16*. 44–57.

[43] G. Varghese and T. Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. 25–38.

[44] L. Zhang. 1990. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. In *SIGCOMM '90*. 19–29.