Titan: Fair Packet Scheduling for Commodity Multiqueue NICs

Brent Stephens, Arjun Singhvi, Aditya Akella, Michael Swift
UW-Madison

Abstract

The performance of an OS's networking stack can be measured by its achieved throughput, CPU utilization, latency, and per-flow fairness. To be able to drive increasing line-rates at 10Gbps and beyond, modern OS networking stacks rely on a number of important hardware and software optimizations, including but not limited to using multiple transmit and receive queues and segmentation offloading. Unfortunately, we have observed that these optimizations lead to substantial flow-level unfairness.

We describe Titan, an extension to the Linux networking stack that systematically addresses unfairness arising in different operating conditions. Across both fine and coarse timescales and when NIC queues are undersubscribed and oversubscribed, we find that the Titan can reduce unfairness by 58% or more when compared with the best performing Linux configuration. We also find that improving fairness can lead to a reduction in tail flow completion times for flows in an all-to-all shuffle in a cluster of servers.

1 Introduction

Many large organizations today operate data centers (DCs) with tens to hundreds of thousands of multi-core servers [37, 35, 20]. These servers run a variety of applications with different performance needs, ranging from latency-sensitive applications such as web services, search, and key-value stores, to throughput-sensitive applications such as Web indexing and batch analytics. With the scale and diversity of applications growing, and with applications becoming more performance hungry, data center operators are upgrading server network interfaces (NICs) from 1Gbps to 10Gbps and beyond. At the same time, operators continue to aim for multiplexed use of their servers across multiple applications to ensure optimal utilization of their infrastructure.

The main goal of our work is to understand how we can enable DC applications to drive high-speed server NICs while ensuring key application performance goals are met—i.e., throughput is high and latency is low—and key infrastructure performance objectives are satisfied—i.e., CPU utilization is low and applications share resources fairly.

Modern end-host network stacks offer a variety of optimizations and features to help meet these goals. Foremost, many 10Gbps and faster NICs provide multiple hardware queues to support multi-core systems. Recent advances in the network stack (RPS [7]/RFS [6]/XPS [11]) allow systematic assignment of these queues and the flows using them to CPU cores to reduce cross-core synchronization and improve cache locality. In addition, provisions exist both in hardware and in the operating system for offloading the packetization of TCP segments, which vastly reduces CPU utilization [22]. Likewise, modern OSes and NIC hardware provide a choice of software queuing logics and configurable queue size limits that improve fairness and lower latencies by avoiding bufferbloat [19].

The first contribution of this paper is a systematic exploration of the performance trade-offs imposed by different combinations of optimizations and features for four key metrics, namely, throughput, latency, CPU utilization, and fairness. We study performance under extensive controlled experiments between a pair of multi-core servers with 10G NICs where we vary the level of oversubscription of queues.

We find that existing configuration options can optimize throughput and CPU utilization. But, we found that across almost every configuration there is substantial *unfairness* in the throughput achieved by different flows using the same NIC: some flows may transmit at twice the throughput or higher than others, and this can happen at both fine and coarse time scales. Such unfairness increases tail flow completion times and makes data transfer times harder to predict. We find that this unfair-

ness between flows arises because of three key aspects of today's networking stacks:

Foremost, OSes today use a simple hash-based scheme to assign flows to queues, which can easily lead to hash collisions even when NIC queues are undersubscribed (fewer flows than queues). Even a more optimal flow-to-queue assignment can result in flow imbalance across queues especially under moderate oversubscription (when the number of flows is only slightly larger than the number of queues).

Second, NIC schedulers strive for equal throughput from each transmit queue and thus service packets from queues in a strict round-robin fashion. Flows that share a queue as a result receive only a fraction of the throughput of those that do not. Even over long periods, a flow may receive half its fair-share throughput or less.

Finally, segmentation offload, which is crucial for lowering CPU utilization, exacerbates head-of-line blocking because a large segment of a flow must be transmitted before a segment from a different flow can be transmitted out of the same queue. This becomes acute at high levels of oversubscription, when there may be multiple segments from different flows in each queue. In this case, head-of-line blocking is also exacerbated by the number of queues that are in use. The NIC performs round robin scheduling of packets from different queues, and the OS aims to keep the same number of bytes enqueued in each hardware queue. If a large segment of the same size is in every queue, a newly arrived packet will have to wait for every enqueued segment to be sent before it can be sent, regardless of which queue it uses.

The second contribution of this paper is an extension to the Linux networking stack called *Titan* that incorporates novel ideas to overcome the above fairness issues. First, Titan uses *dynamic queue assignment* (DQA) to evenly distribute flows to queues based on current queue occupancy. This avoids flows sharing queues in undersubscribed conditions. Second, Titan adds a new queue weight abstraction to the NIC driver interface and a *dynamic queue weight assignment* (DQWA) mechanism in the kernel, which assigns weights to NIC queues based on current occupancy. In Titan, NICs use deficit round-robin [36] to ensure queues are serviced according to computed weights. Third, Titan adds *dynamic segmentation offload sizing* (DSOS) to dynamically reduce the segment size and hence reduce head-of-line blocking under over-subscription, which balances improvements to fairness against increased CPU utilization.

We implement Titan in Linux, and, using experiments both without and with network congestion, we show that Titan greatly reduces unfairness in flow throughput across a range of under- and oversubscription conditions and both at short and long timescales. In many cases, there is near zero unfairness, and in the cases where it remains, Titan reduces unfairness by more than 58%. Our experiments on a cluster of servers show that Titan offers the most fair flow completion times and decreases flow completion times at the tail (90^{th} percentile).

Titan can increase CPU utilization and latency. We have designed Titan so as to try to minimize its impact on CPU utilization. In our experiments, Titan with DQA and DQWA often increases CPU utilization by less than 10%, although in the worst case it increases CPU utilization by 17% and 27% with and without pinning queues to cores, respectively. Also, Titan often matches the RTT latency of unmodified Linux with average latencies ranging from 123–660μs. At most, Titan increases latency by 134μs, and DSOS often reduces latency by more than 200μs. Still, latency under load still remains higher than when there is no other traffic using the NIC (32μs).

Current best practices for preventing long-running bulk data transfers from impacting latency sensitive traffic is to isolate different traffic classes in different priorities [26, 20]. Titan is compatible with DCB, so DCB priorities can still be used to isolate latency-sensitive traffic from bulk traffic in Titan. At the NIC level, this is accomplished by allocating dedicated pools of NIC queues for each DCB priority.

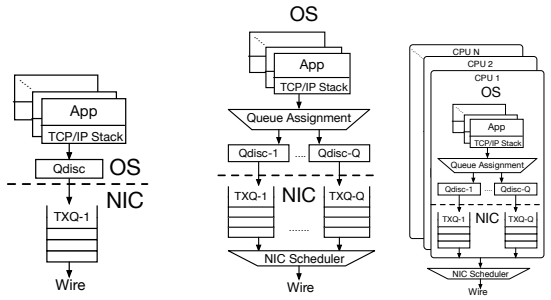
In the next section we provide background material on server networking stacks. Section 3 describes the design of Titan, and Section 4 has information on the implementation. Sections 5 and 6 describe our methodology and evaluation. We follow with related work and then we conclude.

2 Background

Networking in modern OSes is complex. There are multiple cooperating layers involved, and each layer has its own optimizations and configurations. Further, there are multiple different dimensions by which the performance of a server's network stack can be measured, and different configurations have subtle performance trade-offs. Figure 1 shows the different layers involved in a server's network stack (server-side networking), and Table 1 lists the most significant configuration options.

2.1 Server Networking Queue Configurations

We focus on the *transmit* (TX) side of networking because choices made when transmitting segments have a much larger potential to impact fairness: a server has no control over what packets it receives and complete control over what segments it transmits. Although the RX-side of networking is important, TX and RX are largely independent, so recent improvements to the RX



(a) **(b)** **(c)**

Figure 1: Different server-side TX networking designs: (a) Single queue (SQ) TX networking. (b) Multiqueue (MQ) TX networking. (c) Multicore-partitioned (XPS) multiqueue TX networking.

side [25, 18, 30] are complementary to improvements to the TX side.

In an OS, data from application buffers are passed as a segment (smaller than some maximum segment size) through many different layers of the network stack as it travels to the NIC, where it is turned into one or more packets on the wire. Both the design of each layer that touches a segment and the interfaces between them can impact performance.

There are many ways of connecting the layers of a networking stack that differ in the number of *NIC transmit queues* and *the assignment of queues to CPU cores*. Figure 1 illustrates three designs. Figure 1a shows how the OS interfaces with a single queue NIC (SQ). Figures 1b and 1c show two different ways for an OS to interface with a multiqueue NIC. The first (MQ) allows for flows on any core to use any NIC queue. The second partitions queues into pools that are dedicated to different cores, which we refer to by its name in Linux, XPS (transmit packet steering) [11].

Single Queue (SQ): In this design, segments from multiple competing applications (and containers/VMs) destined for the same output device are routed by the TCP/IP stack first to a per-device software queue and then to a per-device hardware queue (Figure 1a). The software queue (`Qdisc` in Linux) may implement any scheduling policy. The hardware transmit queues are simple FIFOs.

On a multicore system, SQ can lead to increased resource contention (locking, cache coherency, *etc.*). Thus, SQ has largely been replaced by designs that use multiple independent software and hardware transmit queues. Nevertheless, SQ offers the OS the most control over packet scheduling because the NIC will transmit packets in the exact order chosen by the OS.

Multiqueue (MQ): To avoid SQ's resource contention overheads, many 10 Gbps and faster NICs provide multiple hardware transmit and receive queues (MQ). Most OSes use multiple partitioned software queues, one for each hardware queue. Figure 1b illustrates MQ in Linux. Note that queues are not pinned to individual cores in this model, although flows may be assigned to queues. This allows computation to be migrated to idle or underutilized cores [32] at the expense of performance isolation provided by dedicating queues to cores. Given a multiqueue NIC, by default, Linux will use MQ.

The driver that we use (`ixgbe`) sets the number of queues to be equal to the number of cores by default. However, modern NICs typically can provide more hardware queues than cores, and using more queues than cores can be advantageous.

Moving to a multiqueue NIC requires that the OS implement some mechanism for assigning traffic to queues. In Linux, queue assignment is determined by RSS hashing for incoming flows and by a per-socket hash for outgoing flows. Because the number of open sockets may be much larger than both the number of NIC queues and the number of simultaneously active sockets, hash collisions would be expected given this approach regardless of the specific hash algorithm that is used.

In MQ, NICs must implement some algorithm for processing traffic from the different queues because they can only send a single packet at a time on the wire. Both the Intel 82599 and Mellanox ConnectX-3 NICs perform round-robin (RR) scheduling across competing queues of the same priority [2, 31]. Because of this, MQ can increase HOL blocking latency. If a multi-packet segment is enqueued in an empty queue, the time to send this entire segment in MQ will be the transfer time in SQ multiplied by the number of active queues. For example, sending a single 64KB segment at 10Gbps line-rate takes 52μs, while sending a 64KB segment from 8 different queues takes 419μs. Further, if all of the queues are full, the queueing latency of the NIC for any new segment is at least equal to the minimum number of bytes enqueued in a queue times the number of queues.

Multicore-Partitioned Multiqueue (XPS): The third networking design partitions NIC queues across the available CPUs, which can reduce or eliminate the inter-core communication performed for network I/O and improve cache locality. This configuration (transmit packet steering or XPS [11]) is particularly important for performance isolation because it ensures VMs/containers on one core do not consume CPU resources on another core to perform I/O. As in MQ, when a core can use multiple queues, hashing is used to pick which queue individual flows are assigned to in Linux.

In Linux, partitioning queues across cores involves significant configuration. XPS assigns NIC TX queues to a pool of CPUs. Because many TX queues can share an interrupt, interrupt affinity must also be configured correctly for XPS to be effective.

Config	Purpose	Expected Impact
Segmentation offloading (TSO/GSO)	Offload or delay segment packetization	Increases to segment sizes should reduce CPU utilization, increase latency, and hurt fairness
Choice of software queue (`Qdisc`)	Optimize for different performance goals	Varies
Assignment of queues to CPU cores (XPS, *etc.*)	Improve locality and performance isolation	Improved assignment should reduce CPU utilization
TCP queue occupancy limits (TCP Small Queues)	Avoid bufferbloat	Decreasing should reduce CPU utilization and latency up to a point of starvation.
Hardware queue occupancy limits (BQL)	Avoid head-of-line (HOL) blocking	Decreasing the byte limit should reduce latency up to a point of starvation. Further decreases should decrease throughput.

Table 1: A table that lists the different server-side network configurations investigated in this study, their purpose, and their expected performance impact.

2.2 Optimizations and Queue Configurations

There are many additional configurations and optimizations that impact network performance. Combined with the above queue configurations, these options induce key trade-offs in terms of latency, throughput, fairness and CPU utilization.

TSO/GSO: *Segmentation offloading* allows the OS to pass segments larger than the MTU through the network stack and down to the NIC. This reduces the number of times the network stack is traversed for a given bytestream. There are many per-segment operations in an OS networking stack, so increasing segment sizes reduces CPU utilization [28].

Many NICs are capable of packetizing a TCP segment without CPU involvement, called TCP Segmentation Offloading (TSO). For NICs that do not support TSO, Generic Segmentation Offloading (GSO) provides some of the benefit of TSO without hardware support by passing large segments through the stack and segmenting only just before passing them to the driver.

TSO/GSO hurts latency and fairness by causing HOL blocking. Competing traffic must now wait until an entire segment is transmitted. Further, sending large segments can cause bursts of congestion in the network [24]. To avoid the problems associated with TSO/GSO, Linux does not always send as large of segments as possible. Instead, Linux automatically reduces the size of TSO segments to try to ensure that at least one segment is sent each millisecond [9]. In effect, this causes Linux to use smaller segments on slow networks while still using as large of segments as possible on fast networks. (*e.g.* 10 Gbps and beyond).

Software Queue Discipline: Before segments are passed to a hardware queue, they are processed by a software queue (`Qdisc`). By default, the queuing discipline in Linux is FIFO (`pfifo_fast`), which is sub-optimal for latency and fairness. Linux implements at least two other superior policies: (1) The `prio` policy strictly prioritizes all traffic from a configurable class over all other traffic, improving latency. (2) The `sfq` policy implements Stochastic Fair Queueing (SFQ) using the deficient round robin (DRR) scheduling algorithm [36] to fairly schedule segments from competing flows regardless of differing segment sizes.

TSO Interleaving: Transmitting an entire TSO segment at once for a given queue can significantly increase latency and harm fairness, even if each queue is serviced equally. Some NICs address this with *TSO interleaving* [2, 31], which sends a single MTU sized packet from each queue in round-robin even if TSO segments are enqueued. This can lead to fairer packet scheduling as long as there is only one flow per-queue. HOL blocking can still occur if there are multiple flows in a queue.

TCP Queue Occupancy Limits: Enqueuing too many bytes for a flow into software queues causes bufferbloat [19], which can hurt latency and fairness. TCP Small Queues (TSQ) [10] limits the number of outstanding bytes that a flow may have enqueued in either hardware or software queues to address this problem. Once the limit is reached (256KB by default in Linux), the OS waits for the driver to acknowledges that some segments for that flow have been transmitted before enqueuing more data. As long as more bytes are enqueued per-flow than can be transmitted by the NIC before the next interrupt, TSQ can still drive line-rate while reducing bufferbloat.

In Linux, the enqueueing of additional data for flows sharing a queue in TSQ happens in batches. This is a side-effect of Linux using the freeing of an `skbuff` as a signal that it has been transmitted and `skbuffs` only being freed by the driver in batches in the TX interrupt handler.

Hardware queue occupancy limits: Hardware queues are simple FIFOs, so increasing the bytes enqueued per-hardware queue directly increases HOL blocking latency. Byte Queue Limits (BQL) [1] in Linux limits the total amount of data enqueued in a hardware queue. However, it is important to enqueue at least as many bytes as can be sent before the next TX interrupt, otherwise starvation may ensue. A recent advancement is Dynamic Queue Limits (DQL) [1], which dynamically adjusts each hardware queue's BQL independently so as to decrease HOL blocking while avoiding starvation.

2.3 Configuration Trade-off Study

We studied the impact of the aforementioned configurations on server-side performance (CPU utilization,

Cvanilla:	Default Linux networking stack incurs significant latency and unfairness, regardless of how many NIC queues are used, but has high throughput and low CPU.
C1: No TSQ:	TSQ is an important optimization. Disabling can cause significant latency and unfairness.
C2: Improved software scheduling:	Improving the software scheduler can significantly reduce latency and increase fairness, especially when only a single NIC queue is used. Comes at the cost of CPU utilization.
C3: No BQL:	BQL is an important optimization because disabling it can lead to increased latency and decreased fairness.
C4: 64KB BQL:	Setting BQL too small decreases latency but hurts fairness at long timescales with many flows.
C5: No TSO:	Disabling segmentation offloading hurts every performance metric because CPUs saturate.
C6: 16KB GSO:	Using a smaller GSO size than the default (64KB) improves fairness at short timescales (ms), increases CPU utilization.
Cmax: C2 + 256KB BQL:	Dynamic Queue Limits (DQL) leads to a higher queue limit than necessary to avoid starvation. If BQL is manually set smaller, it is possible to reduce latency and improve fairness.

Table 2: Summary of experimental results for different networking configurations.

throughput, latency, and fairness). Our high-level takeaways are listed in Table 2. These are synthesized from the raw results presented for each combination of workload, queue configuration, and optimization, which we detail in a technical report [40]. Table 3 in Section 6 shows the raw results for default Linux (*Cvanilla*) and the best performing configuration (*Cmax*). These results show that using SFQ for the queuing discipline with TCP small queues enabled and byte queue limits manually set to 256KB tend to out-perform all other combinations across different queue configurations. This is denoted by Cmax, which we henceforth focus on as the baseline best-performing MQ/XPS configuration today.

While we find that using multiqueue NICs can generally offer low CPU utilization and high throughput, we also find that the current Linux networking stack is unable to provide fairness at any time scale across flows at any subscription level. In the undersubscribed case, the central problem with MQ in Linux is the assignment of flows to queues. At low oversubscription, unfairness is uniformly high at short (1ms) and long (1 sec) timescales. We find that this largely occurs because some queues have more flows than others, and flows that share a queue send half as much data as those that do not. At high oversubscription, fairness is uniformly worse, as hashing is not perfect and leads to variable number of flows per queue, and a flow sharing a queue with 9 other flows will send much more slowly than one sharing with 5. However, using the best practices, exemplified particularly by configuration Cmax, can have substantial benefits over vanilla Linux without optimizations (Cvanilla).

2.4 Summary

Multiqueue NICs allow different CPU cores to perform network I/O independently, which is important for reducing the CPU load of network I/O caused by locking and cross-core memory contention. Each core can use independent software queueing disciplines feeding independent hardware queues. Further, TSO reduces CPU utilization by allowing the OS to treat multiple sequential packets as a single large segment. However, as a consequence, a packet scheduler in the NIC is now responsible for deciding which queue is allowed to send packets out on the wire. Because the NIC performs round-robin scheduling across competing hardware queues and TSO segments cause HOL blocking, the NIC will emit an unfair packet schedule when the network load is asymmetrically partitioned across the NIC's hardware queues and when multiple flows share a queue.

3 Titan

This section presents the design of Titan, an OS networking stack that that introduces new mechanisms for improving network fairness with multiqueue NICs. To improve fairness, Titan dynamically adapts the behavior of the many different layers of an OS's network stack to changes in network load and adds a new abstraction for programming the packet scheduler of a NIC. Specifically, Titan comprises the following components: Dynamic Queue Assignment (DQA), Dynamic Queue Weight Assignment (DQWA), and Dynamic Segmentation Offload Sizing (DSOS).

Given a fixed number of NIC queues, we target the three behavior modes of behavior we previously described: undersubscribed, low oversubscription, and high oversubcription. Titan is designed to improve server-side networking performance regardless of which mode a server currently is operating in, and the different components of Titan are targeted for improving performance in each of these different regimes. The rest of this section discusses the design of these components.

3.1 Dynamic Queue Assignment (DQA)

When it is possible for a segment to be placed in more than one queue, the OS must implement a queue assignment algorithm. In Linux, a per-socket hash is used to assign segments to queues. Even when there are fewer flows than queues (undersubscribed), hash collisions can lead to unfairness.

Titan uses Dynamic Queue Assignment (DQA) to avoid the problems caused by hash collisions when there are fewer flows than queues. Instead of hashing, DQA chooses the queue for a flow dynamically based on the current state of the software and hardware queues. DQA assigns flows to queues based on queue weights that are internally computed by Titan. In other words, there are

two components to DQA: an algorithm for computing the OS's internal weight for each queue and an algorithm for assigning a segment to a queue based on the current weight of every software/hardware queue that the segment can use.

Queue weight computation: Titan uses the current traffic that is enqueued in a software/hardware queue pair to compute a weight for each queue. We assume that the OS can assign a weight to each network flow based on some high-level policy. Titan dynamically tracks the sum of the weights of the flows sharing the same queue: it updates a queue's weight when a flow is first assigned to a queue and when a TX interrupt frees the last outstanding `skbuff` for the flow.

Queue assignment algorithm: Dynamically tracking queue occupancy can allow a queue assignment algorithm to avoid hash collisions. Our goals in the design of a DQA are to avoid packet reordering and provide accurate assignment without incurring excessive CPU utilization overheads. We use a greedy algorithm to assign flows to queues with the aim of spreading weight evenly across all queues. This algorithm selects the queue with the minimum weight.

The main overhead of our current implementation of DQA is that it reads the weights of every queue a flow may use. XPS reduces this overhead by reducing the number of queue weights that need to be read: if a flow is not allowed to use a queue, DQA will not read its weight. Although not necessary, our current implementation introduces a lock to serialize queue assignment per XPS pool. We are currently investigating using a lock-free priority queue to allow multiple cores to simultaneously perform queue assignment without reading every queue's weight while still avoiding choosing the same queues.

In order to avoid packet reordering, DQA only changes a flow's queue assignment when it has no outstanding bytes enqueued in a software or hardware queue. This also has the added benefit of reducing the CPU overheads of queue assignment because it will be run at most once per TX interrupt/NAPI polling interval and often only once for as long as a flow has backlogged data and is allowed to send by TCP. However, this also implies that unfairness can arise as flows complete because remaining flows are not rebalanced.

3.2 Dynamic Queue Weight Assignment (DQWA)

DQA computes queue weights to perform queue assignment. However, these queue weights are only an OS construct. The NIC does not perform scheduling decisions based on these weights; it services queues based on simple round-robin instead. During periods of oversubscription, this can lead to unfairness.

To solve this problem, Titan modifies NIC drivers to expose a queue weight abstraction whereby higher levels of the network stack can cause the NIC scheduler to service queues in proportion to the OS' weights. This is accomplished by introducing the new `ndo_set_tx_weight` network device operation (NDO) for drivers to implement. The OS calls this function whenever it updates a queue's weight, which allows the NIC driver to dynamically program the NIC scheduler. We call this Dynamic Queue Weight Assignment (DQWA). Although simple, this new function allows the NIC to generate a fair packet schedule provided that the NIC scheduler is capable of being programmed.

The main overhead of DQWA is that each update generates a PCIe write. Like DQA, DQWA weights only need to be changed at most once per TX interrupt/NAPI polling interval. However, if necessary, the number of DQWA updates can also be rate limited.

While not all commodity NICs allow weight setting, it is a small addition to mechanisms already present. A NIC scheduler must implement a scheduling algorithm that provides per-queue fairness even if different sized segments are enqueued. To modify this algorithm to service queues in proportion to different weights is simple; we borrow the classic networking idea of Deficit Round Robin (DRR) scheduling [36]. Specifically, by allocating each queue its own pool of credits that are decreased proportional to the number of bytes sent by the queue, DRR can provide per-queue fairness. Providing an interface to modify the allocation of credits to queues enables the NIC to configure DRR to service queues in proportion to different weights.

We implement the `ndo_set_tx_weight` in the `ixgbe` driver by configuring the NIC scheduler's per-queue DRR credit allocation.

3.3 Dynamic Segmentation Offload Sizing (DSOS)

When segments from competing flows share the same software/hardware queue pair, the size of a GSO segment becomes the minimum unit of fairness. Under periods of heavy oversubscription, the GSO size can become the major limiting factor on fairness because of the HOL blocking problems that large segments cause. Importantly, improving the interleaving of traffic from multiple different flows at finer granularities can also benefit network performance [18].

Currently, the only way to improve the fairness of software scheduling is by reducing the GSO size. However, this only improves fairness when multiple flows share a single queue. Otherwise, TSO interleaving in the NIC provides per-packet fairness independent of the

GSO (TSO) size. Reducing the GSO size when the network queues are not oversubscribed only wastes CPU.

Dynamic Segmentation Offload Sizing (DSOS) enables an OS to reduce GSO sizes for improved fairness under heavy load while avoiding the costs of reducing GSO sizes when NIC queues are not oversubscribed. This provides a better CPU utilization trade-off than was previously available.

In DSOS, packets are segmented from the default GSO size to a smaller segment size before being enqueued in the per-queue software queues only if multiple flows are sharing the same queue. (In our current implementation, re-segmentation happens in all queues as soon as there is oversubscription.) Segmentation in DSOS is identical to the implementation of GSO except that segmentation happens before `Qdisc` instead of after. Because the software queue (`Qdisc`) is responsible for fairly scheduling traffic from different flows, this enables the OS to generate a fair packet schedule while still benefiting from using large segments in the TCP/IP stack. Further, many multiqueue NICs also support passing a single segment as a scatter/gather list of multiple regions in memory. This enables a single large segment to be converted into multiple smaller segments without copying the payload data. If automatic TSO sizing generates segments smaller than the DSOS segment size, then no additional work is done.

4 Implementation

We implemented Titan in Linux 4.4.6 and modified Intel's out-of-tree `ixgbe-4.4.6` release [4] to support the new `ndo_set_tx_weight` NDO. We were able to implement this new NDO in this driver from the public hardware datasheets [2]. In a similar spirit, Titan is open source and available at https://github.com/bestephe/titan.

There is one major limitation in our current `ixgbe` driver implementation. We were only able to program the packet scheduler on the Intel 82599 NIC when it was configured in VMDq mode. As a side-effect, this causes the NIC to hash received packets (received side steering, or RSS) to only four RX queues. This effectively decreases the NIC's RX buffering capacity, so enabling this configuration can increase the number of packet drops. To try to mitigate the impact of reducing the receive buffering capacity of the NIC, we modified the `ixgbe-4.4.6` driver to enable a feature of the 82599 NIC that immediately triggers an interrupt when the number of available RX descriptors drops below a threshold.

During development, we found a problem with the standard Linux software queue scheduler. Linux tries to dequeue packets from software queues in a batch and enqueue them in their corresponding hardware queue whenever a segment is sent from any TCP flow. When multiple ACKs are received in a single interrupt, multiple TCP flows may try to create new `skbuffs` and enqueue them. If no bytes are enqueued in the software queues for two flows, and then ACKs for both flows arrive, the second flow will not have a chance to enqueue new `skbuffs` in the software queues before packets are dequeued from the software queue until the hardware queue is filled up to the BQL limit. In general, sending segments to the NIC as soon as the first TCP flow sends a segment may cause later TCP flows to miss an opportunity to send, leading to unfairness.

In Titan, we improve fairness with *TCP Xmit Batching*. With this mechanism, all of the TCP flows that enqueue segments at the same time in TSQ are allowed to enqueue packets into their respective software queues before *any* packets are dequeued from software queues and enqueued in the hardware queues. This is accomplished by changing the per-CPU TSQ tasklet in Linux so enqueuing a segment returns a pointer to a `Qdisc`. Packets are dequeued from the returned `Qdiscs` only after all pending segments have been enqueued.

5 Methodology

To evaluate Titan, we perform experiments by sending data between two servers and within a cluster of servers.

In the two server experiments, we use a cluster of three servers connected to a dedicated TOR switch via 10 Gbps Ethernet cables. One server is a source, another a sink, and the third server is for monitoring. The switch is a Broadcom BCM956846K-02. The first and second server are the traffic source and sink respectively. Both of these servers have a 4-core/8-thread Intel Xeon E5-1410 CPU, 24GB of memory, and connect to the TOR with Intel 82599 10 Gbps NICs [2]. We configure the switch to use port mirroring to direct all traffic sent by the first server to the third server. To monitor traffic, this server uses an Intel NetEffect NE020 NIC [5], which provides packet timestamps accurate to the microsecond.

We perform two types of two server experiments. First, we generate traffic using at most one `iperf3` [3] client per core pinned to different CPUs. Each client only uses a single thread. Because the fairness problems only arise when load is asymmetric, we distribute the flows across cores such that half of the cores have twice as many active flows as the other half of the cores. To measure latency, we use `sockperf` [8]. To measure CPU utilization, we use `dstat`. To avoid impacting CPU utilization by measuring latency, we measure latency and CPU utilization in separate experiments. Second, we use YCSB [12] to request both small and large values from memcached from different threads. We perform all of

the two server experiments with the NIC configured in VMDq mode.

In the cluster workloads, we use a cluster of 24 servers on CloudLab. Each of the servers has 2 10-core Intel E5-2660 v2 CPUs and 256GB of memory. All the servers connect to a Dell Networking S6000 switch via Intel 82599 NICs. Inspired by shuffle workloads used in prior work [13, 33, 22], we have all 24 servers simultaneously open a connection to every other server and send 1GB. We measure flow completion times. Because `iperf3` opens up additional control connections that can impact performance, we use a custom application to transfer data in this workload.

We compare Titan against two base configurations: Cvanilla, which is the default Linux configuration, and Cmax, which uses the MQ configuration system with a GSO size of 64KB, a TCP small queues limit of 256KB, and byte queue limits manually set to 256KB. In Cmax, interrupt coalescing on the NIC is also configured so that the NIC will use an interrupt interval of 50μs. In other words, the NIC will wait at least 50μs after raising an interrupt before it will be raised again. In the 2 server experiments, the traffic sink always uses configuration Cmax. Large receive offload (LRO) is disabled in all of the experiments because it can increase latency. We perform all experiments 10 times and report the average.

6 Evaluation

First, we evaluate the performance impact of individual components of Titan in the absence of any network congestion. Second, we evaluate Titan on a cluster of servers. In summary, we find that Titan is able to improve fairness on multiqueue NICs while only having a small impact on other metrics.

We study the following four metrics:

1. We measure *CPU utilization* as the sum percent of the time each core was not idle during a one second interval, summed across all cores and averaged across the duration of the experiment.
2. We measure *network throughput* as the total number of bytes that were sent per second across all flows, averaged across the duration of the experiment.
3. We measure *latency* with `sockperf` and report average latency. When we configure Linux software queues (`Qdiscs`), we prioritize the port used by `sockperf` above all other traffic.
4. We use a *normalized fairness metric* inspired by Shreedhar and Varghese [36]. For every flow $i \in F$, there is some configurable quantity f_i that expresses i's fair share. In all of our experiments, f_i is 1. If $sent_i(t_1,t_2)$ is the total number of bytes sent by flow i in the interval (t_1,t_2), then the fairness metric FM is as follows:

$$FM(t_1,t_2) = max\{i,j \in F | sent_i(t_1,t_2)/f_i - sent_j(t_1,t_2)/f_j\}$$

In other words, the fairness metric $FM(t_1,t_2)$ is the instantaneous *worst case difference* in the normalized bytes sent by any two competing flows over the time interval. Ideally, the fairness metric should be a small constant no matter the size of the time interval [36].

For our experiments, we do not report this ideal FM but instead use normalized fairness *NFM(τ)*, which is the fairness metric FM over all intervals of duration τ, normalized to the fair share of data for a flow in the interval.

$$NFM(\tau) = FM(\tau) * \frac{line\_rate * \tau^{-1}}{\sum_{j \in F} f_j}$$

For example, with 10 flows, a flow's fair share of a 10 Gbps link over 1 second is 128MB; if the highest FM over a 1-second interval is 64 MB, then NFM is 0.5. Note that NFM can exceed 1 when some flows get much higher performance than others.

6.1 Two Server Performance

There are multiple complementary components to Titan, and we evaluate the impact of individual components on performance in the absence of network congestion. Table 3 shows the performance of different components of Titan for each metric. The expected benefit of Titan is improved fairness, but it is possible for Titan to hurt throughput, latency, or CPU utilization. These results show that Titan is able to significantly improve fairness often without hurting throughput and latency and with a small increase in CPU utilization (often $< 10\%$)

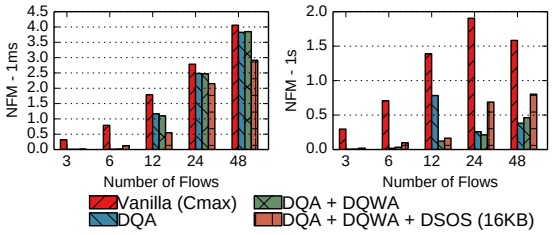
Dynamic Queue Assignment: DQA ensures that when there are fewer flows than queues, each flow is assigned its own queue. The Cmax (hashing) and DQA results in Figure 2 shows the fairness differences between using hashing and DQA for assigning flows to queues given 8 hardware queues and a variable number of flows. We report *NFM*, the normalized fairness metric.

With hashing, fairness is good with 3 flows as there are few collisions. However, with more flows, the unfairness of hashing is high at short and long timescales because there are often hash collisions. Unfairness is bad because of HOL blocking while waiting for GSO/TSO-size segments and hashing leading to uneven numbers of flows per queue.

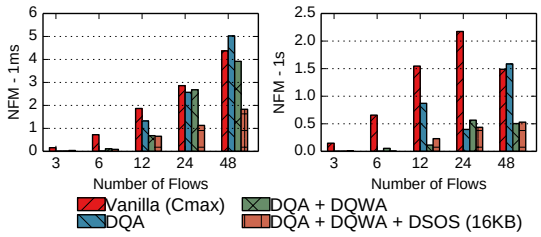
In contrast, with DQA there is no unfairness in the undersubscribed case, as DQA always assigns every flow its own queue. In the low oversubscription case of 12 flows, there is also unfairness because some flows must

Config	3 flows, 8 Queue (1 per CPU) TPut (Gbps)	CPU (%)	Latency (μs)	NFM (1ms)	NFM (1s)	Config	12 flows, 8 Queues (1-per CPU) TPut (Gbps)	CPU (%)	Latency (μs)	NFM (1ms)	NFM (1s)
Cvanilla	9.4	64	298	0.33	0.31	Cv:	9.4	58	912	1.83	1.23
Cmax: SFQ/Prio + 256KB BQL	9.4	72	125	0.16	0.15	Cmax:	9.4	55	912	1.79	1.39
Titan1: DQA	9.4	78	123	0.00	0.00	T1:	9.4	66	657	1.17	0.78
Titan2: DQA + DQWA	9.4	77	124	0.00	0.01	T2:	9.4	70	516	1.10	0.12
Titan3: DQA + DQWA + DSOS (16KB)	9.4	82	180	0.02	0.02	T3:	9.4	96	395	0.55	0.17
XPS: Cmax + XPS	9.4	54	130	0.16	0.15	XPS:	9.4	55	526	1.87	1.55
TitanXPS1: DQA	9.4	55	126	0.00	0.00	TX1:	9.4	49	660	1.32	0.87
TitanXPS2: DQA + DQWA	9.4	57	121	0.01	0.00	TX2:	9.4	50	505	0.68	0.11
TitanXPS3: DQA + DQWA + DSOS (16KB)	9.4	65	128	0.04	0.01	TX3:	9.4	59	269	0.66	0.23
	48 flows, 8 Queue (1 per CPU)						192 flows, 8 Queues (1-per CPU)				
Cvanilla	9.4	72	2019	5.01	1.95	Cv:	9.4	98	3881	15	3.32
Cmax: SFQ/Prio + 256KB BQL	9.4	83	653	4.06	1.58	Cmax:	9.1	109	604	6.93	1.39
Titan1: DQA	9.4	89	660	3.83	0.38	T1:	9.5	118	554	8.35	0.54
Titan2: DQA + DQWA	9.4	87	585	3.85	0.46	T2:	9.5	103	509	8.42	0.49
Titan3: DQA + DQWA + DSOS (16KB)	9.3	103	285	2.92	0.80	T3:	9.4	113	342	3.50	0.80
XPS: Cmax + XPS	9.4	53	639	4.37	1.49	XPS:	9.5	119	517	10	2.66
TitanXPS1: DQA	9.4	61	660	5.02	1.58	TX1:	9.5	113	552	8.46	0.76
TitanXPS2: DQA + DQWA	9.4	62	606	3.92	0.50	TX2:	9.5	123	519	8.28	0.57
TitanXPS3: DQA + DQWA + DSOS (16KB)	9.4	76	333	1.83	0.53	TX3:	9.4	138	300	3.50	0.81

Table 3: The performance of different OS configurations given 3, 12, 48, and 192 flows spread across 8 cores.



(a) No XPS



(b) XPS

Figure 2: The impact of the individual aspects of Titan on short-term and long-term fairness.

share queues, and without DQWA to program weights in the NIC, all queues are serviced equally. With 48 flows, DQA has low unfairness over long timescales because it will place exactly 6 flows in each queue.

Dynamic Queue Weight Assignment: DQWA enables an OS to pass queue weights, in this case the number of flows, to the NIC so that queues with more flows receive more service. Figure 2 shows the fairness of the DQA queue assignment algorithms when DQWA is enabled. These results show that over short timescales, DQWA has little impact as it takes time for queue weights to fix transient unfairness, and in highly oversubscribed cases HOL blocking is the major cause of unfairness. Over longer timescales, DQWA improves the fairness at low levels of oversubscription because the NIC is able to give more service to queues with more flows. At high levels of oversubscription, DQA is able to evenly distribute flow weights across queues, so DQWA is not able to further improve fairness.

We note that DQA is a software-only solution that has the largest impact in undersubcribed cases and helps at both short and long timescales. DQWA helps most in (i) oversubscribed cases and (ii) over longer timescales. In addition, DQWA requires hardware support that, while minimal, may not be present in all NICs. Also, we evaluated DQWA with hashing instead of DQA, and we found that DQWA also improves fairness without DQA.

Dynamic Segmentation Offload Sizing: DSOS addresses HOL blocking by reducing segment size from the default 64KB to a smaller size dynamically under oversubscription. We compare DQA and DQWA with and without DSOS for 16KB DSOS segment sizes. Figure 2 shows that DSOS improves fairness at the 1ms timescale. In the 3 and 6 flow cases there is no oversubscription, so DSOS leaves the GSO size at 64KB. For 12, 24, and 48 flows, though, DSOS reduces the segment size to reduce HOL blocking. At short timescales, this improves fairness. Over longer timescales, DSOS can slightly hurt fairness. This is because DSOS can increase CPU utilization.

XPS: So far, our evaluation has focused our discussion on the multiqueue NIC configuration (MQ). Transmit packet steering (XPS; Section 2.1) assigns pools of queues to pools of CPUs and behaves differently than MQ. To understand these differences, Figure 2 also shows the fairness of Titan when XPS is configured. For the most part, this figure shows that XPS has little impact

on network fairness in Titan.

The biggest change in Figure 2 is that XPS improves the fairness of DSOS (with both DQA and DQWA enabled) at short timescales during oversubscription. When there are 48 flows, using a 16KB dynamic segment size with XPS almost halves NFM at short time scales. The reason for this is because XPS reduces the CPU overheads of DSOS (Table 3). This is because XPS improves cache locality.

CPU Utilization, Throughput and Latency: While the goal of Titan is improved fairness, it must not come at the cost of increased CPU utilization, decreased throughput, or increased latency. Tables 3 compares the performance of Titan with Cvanilla and Cmax.

At all subscription levels, throughput is almost always identical with Titan and standard Linux networking options. Similarly, CPU utilization is slightly higher with Titan. It must do more work for queue assignment and weight-setting. During oversubscription, DSOS must segment and process smaller segments. Fortunately, enabling XPS reduces the CPU utilization of all of the features of Titan.

Regardless of the subscription level, Titan can increase latency. In the absence of any other traffic, the average baseline latency we observed is 32μs. In the presence of bulk transfers, the minimum average latency we observe is 121μs, and the highest average latency we observe is 3.9ms. This high latency is because the HOL blocking latency of the NIC (for a given priority) is at least equal to the minimum number of bytes enqueued in any queue multiplied by the number of active queues. Although we find that latency in general is high, we observe that Titan does not significantly hurt latency. The latency of Titan is often near that of Cmax, and at most Titan increases latency by 134μs. When NIC queues are oversubscribed, we observe that DSOS can reduce latency by over 200μs. Further, we also looked at tail latency and found that the 90th percentile latency for Titan is never more than 200μs higher than the average.

Currently, the best practice for addressing this problem is to use DCB priorities to isolate high priority traffic onto independent pools of NIC queues that are serviced with higher priority by the NIC hardware. Traffic in one DCB priority is not able to increase the latency of traffic in a higher DCB priority.

In summary, we find that overall Titan greatly improves fairness across a wide range of subscription levels, often at no or negligible throughput or latency overheads. Titan can cause a small increase in CPU utilization, often less than 10%. At most, this increase is 17% and 27% with and without XPS, respectively.

Finally, we have also performed experiments to evaluate the impact of Titan on average and tail request completion times in memcached. These experiments use

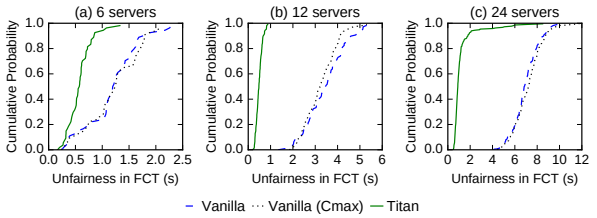


Figure 3: The impact of Titan on fairness on a cluster of servers performing a shuffle.

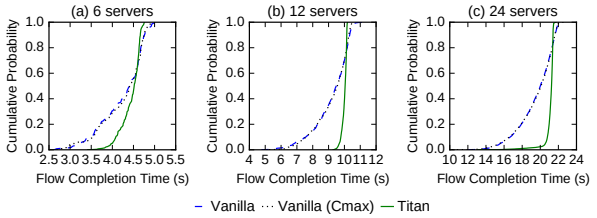


Figure 4: The impact of Titan on flow completion times (FCT) on a cluster of servers performing a shuffle.

YCSB with 7 request threads, 6 of which request 512KB values, while the remaining thread requests small objects (2–64KB). We find that Titan is able to reduce the average and 99th percentile completion times for the small objects by 3.2–10.6% and 7.3–32%, respectively. This is because Titan is able to avoid HOL blocking latency through dynamic queue assignment.

6.2 Cluster Performance

In order to evaluate the cluster performance of Titan, we measure the impact of improving the fairness of the packet stream emitted by a server when there is network congestion and when there are more communicating servers. To do so, we perform an all-to-all shuffle for different cluster sizes where each server simultaneously opens connections to every other server and transfers 1GB of data. This workload is inspired by the shuffle phase of Map/Reduce jobs.

Figure 3 shows the impact of Titan on network performance in a cluster of 6, 12, and 24 servers. We plot a CDF of the difference in the completion time of the earliest completing flow and that of the last completing flow. First, Figure 3 confirms that without Titan flow fairness is a problem in a cluster of servers. Both the default Linux configuration (Cvanilla) and an optimized Linux configuration (Cmax) behave similarly and show substantial variation in completion times. In contrast, with Titan unfairness substantially improves at all three subscription levels and is consistently much better than Cvanilla and Cmax.

Further, we find that Titan is not only able to improve fairness, but that improving fairness also reduces the tail

flow completion times (>80th percentile) for the flows in the shuffle as well. To show why, Figure 4 shows a CDF of the flow completion times across all the flows in the shuffle for different cluster sizes. This figure shows that Titan provides more consistent flow completion times. Because of this, the fastest flows (<20th percentile) in Cvanilla and Cmax complete faster. However, this comes at the expense of tail flow completion times. Figure 4 shows that Titan can reduce the tail of the flow completion time distribution (>80th percentile).

Finally, for this test, DQA (without DQWA or DSOS) is enough to get most of the fairness benefit of Titan. At small cluster sizes, we found that DQWA can still further improve fairness. Unfortunately, we discovered that configuring our NICs into VMDq mode reduces RX buffering capacity and hurts completion times. Because our implementation of DQWA requires VMDq mode to program queue weights, we cannot evaluate DQWA's benefit for large clusters.

7 Related Work

Titan is closely related to SENIC [31] and Silo [23][1]. SENIC argues that NICs in the future will be able to provide enough queues such that two flows will never have to share the same queue. In contrast, Silo builds a system for fairly scheduling traffic from competing VMs using a single transmit queue (SQ) because of the control it gives to the OS. Titan introduces a middle ground that can achieve some of the benefits of both designs.

Many projects in addition to Silo have used the SQ model. In particular, the SQ model is popular for emulating new hardware features not yet provided by the underlying hardware [31, 21, 25]. This is because it provides the OS with the most control over packet scheduling.

Similar to Titan, PSPAT [34] performs per-packet scheduling in a dedicated kernel thread that is separated from applications and device drivers with two sets of lock-free queues. Making per-packet scheduling decisions in PSPAT instead of per-segment decisions in Titan can significantly improve fairness and latency, and Titan can cause PCIe contention that is avoided in PSPAT by only issuing PCIe writes from a single core. If PSPAT were extended to use multiple independent scheduling threads to drive independent NIC queues, then programming the NIC scheduler with DQWA in Titan would be complementary.

There has been recent work on building networks that provide programmable packet scheduling [38, 29, 16], allowing flows to fairly compete [15, 41, 39], and performing traffic engineering in the network [13, 22, 17, 33, 14, 18]. Titan is motivated by similar concerns and is complementary. If the packet schedule emitted by a server is not fair, then the end-server can become the main limiting factor on fairness, not the network. Thus, Titan can improve the efficacy of the aforementioned techniques.

Affinity-Accept [30] improves connection locality on multicore processors, and Fastsocket [27] improves the multicore scalability of the Linux stack when a server handles many short-lived network connections. Titan is complementary to both of these designs. Titan benefits from their improvements in connection setup, while these designs can benefit from improved flow fairness in Titan.

[1]The Titan Missile Museum is located in a silo. We imagine it is scenic.

8 Conclusions

With increasing datacenter (DC) server line rates it becomes important to understand how best to ensure that DC applications can saturate high speed links, while also ensuring low latency, low CPU utilization, and per-flow fairness. While modern NICs and OS's support a variety of interesting features, it is unclear how best to use them towards meeting these goals. Using an extensive measurement study, we find that certain multi-queue NIC configurations are crucial to ensuring good latency, throughput and CPU utilization, but substantial unfairness remains. To this end, we designed Titan, an extension to the Linux network stack that incorporates three main ideas – dynamic queue assignment, dynamic queue weights, and dynamic segmentation resizing. Our evaluation using both experiments between two servers on an uncongested network and between a cluster of servers shows that Titan can reduce unfairness across a range of conditions while minimally impacting the other metrics.

Titan is complementary with a variety of other DC host networking optimizations, such as DCB and receive-side network optimizations. Titan's sender-side fairness guarantees are crucial to ensure the efficacy of in-network fair-sharing mechanisms. Finally, the three main ideas in Titan can be employed alongside other systems, e.g., those for DC-wide traffic scheduling and other existing systems optimized for short-lived connections.

9 Acknowledgements

We would like to thank our shepherd Michio Honda and the anonymous reviewers for their help and insightful feedback. This work is supported by the National Science Foundation grants CNS-1654843 and CNS-1551745.

References

[1] bql: Byte queue limits. https://lwn.net/Articles/469652/.

[2] Intel 82599 10 GbE controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[3] iperf3: Documentation. http://software.es.net/iperf/.

[4] ixgbe-4.4.6. https://sourceforge.net/projects/e1000/files/ixgbe%20stable/4.4.6/.

[5] Neteffect server cluster adapters. http://www.intel.com/content/dam/doc/product-brief/neteffect-server-cluster-adapter-brief.pdf.

[6] rfs: Receive flow steering. http://lwn.net/Articles/381955/.

[7] rps: Receive packet steering. http://lwn.net/Articles/361440/.

[8] sockperf: Network benchmarking utility. https://github.com/Mellanox/sockperf.

[9] tcp: TSO packets automatic sizing. https://lwn.net/Articles/564979/.

[10] tsq: Tcp small queues. https://lwn.net/Articles/506237/.

[11] xps: Transmit packet steering. https://lwn.net/Articles/412062/.

[12] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/wiki.

[13] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010), NSDI '10.

[14] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication* (2014), SIGCOMM '14.

[15] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication* (2010), SIGCOMM '10.

[16] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the 2013 ACM Conference on Special Interest Group on Data Communication* (2013), SIGCOMM '13.

[17] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. MicroTE: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh ACM Conference on Emerging Networking Experiments and Technologies* (2011), CoNEXT '11.

[18] GENG, Y., JEYAKUMAR, V., KABBANI, A., AND ALIZADEH, M. Juggler: A practical reordering resilient network stack for datacenters. In *Proceedings of the Eleventh ACM European Conference on Computer Systems* (2016), EuroSys '16.

[19] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark buffers in the internet. *Queue 9*, 11 (Nov. 2011), 40:40–40:54.

[20] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[21] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[22] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J. B., AND AKELLA, A. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[23] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[24] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., , AND PORTER, G. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (2013), CoNEXT '13.

[25] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with FlexNIC. In *Proceedings of the Twenty-First ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16.

[26] KUMAR, A., JAIN, S., NAIK, U., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[27] LIN, X., CHEN, Y., LI, X., MAO, J., HE, J., XU, W., AND SHI, Y. Scalable kernel TCP design and implementation for short-lived connections. In *Proceedings of the Twenty-First ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16.

[28] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *Proceedings of the 2008 USENIX Conference on Annual Technical Conference* (2008), USENIX ATC '08.

[29] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (2015), HotNets-XIV.

[30] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the Seventh ACM European Conference on Computer Systems* (2012), EuroSys '12.

[31] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for end-host rate limiting. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI '14.

[32] RAM, K. K., COX, A. L., CHADHA, M., AND RIXNER, S. Hyper-switch: A scalable software virtual switching architecture. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC '13.

[33] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication* (2014), SIGCOMM '14.

[34] RIZZO, L., VALENTE, P., LETTIERI, G., AND MAFFIONE, V. PSPAT: Software packet scheduling at hardware speed. http://info.iet.unipi.it/~luigi/pspat/. Preprint; accessed May 31 2017.

[35] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[36] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proceedings of the 1995 ACM Conference on Special Interest Group on Data Communication* (1995), SIGCOMM '95.

[37] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15.

[38] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication* (2016), SIGCOMM '16.

[39] STEPHENS, B., COX, A. L., SINGLA, A., CARTER, J. B., DIXON, C., AND FELTER, W. Practical DCB for improved data center networks. In *Proceedings of the 33rd Annual IEEE International Conference on Computer Communications* (2014), INFOCOM '14.

[40] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue NICs. Tech. Rep. TR1840, University of Winconsin-Madison, Department of Computer Sciences, February 2017. http://digital.library.wisc.edu/1793/75739.

[41] VAMANAN, B., HASAN, J., AND VIJAYKUMAR, T. Deadline-aware datacenter tcp (D2TCP). In *Proceedings of the 2012 ACM Conference on Special Interest Group on Data Communication* (2012), SIGCOMM '12.