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, it not clear how best to leverage these optimizations to extract performance. The first contribution of this paper is a detailed empirical study of the impact of different OS and NIC configurations on this four-dimensional trade-off space. We find that enabling certain specific features is crucial for latency, CPU utilization, and throughput. However, substantial flow-level unfairness still remains. The second contribution of this paper is Titan, an extension to the Linux networking stack that systematically addresses unfairness arising in different operating conditions, while minimally impacting CPU utilization, latency, and throughput.

1 Introduction

Many large organizations today operate data centers (DCs) with tens to hundreds of thousands of multi-core servers [29, 27, 12]. 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 future 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 drive the NIC. More recent advances (XPS [34]) allow systematic assignment of these queues to CPU cores to reduce cross-core synchronization for network I/O 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 [14]. Likewise, modern OSes and NIC hardware provide a choice of software queuing logics and configurable queue size limits that improve fairness and avoid bufferbloat [11], leading to lower latencies.

While many tantalizing options exist, it is not clear how to combine them and how they trade off the four key metrics, namely, throughput, latency, CPU utilization, and fairness. Thus, the first contribution of this paper is a novel systematic exploration of the performance trade-offs imposed by different combinations of optimizations and features. 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 NIC queues. We find that the vanilla Linux networking stack is sub-optimal, and that enabling certain specific features (e.g., stochastic queueing, static byte queue limits, increasing interrupt frequency, etc.) consistently outperforms others configurations.

Existing configuration options can optimize throughput, latency, and CPU utilization. But, we found that across almost every configuration there is substantial *un-*

fairness 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 impacts tail flow completions times, which can hurt data center applications [35]. We find that this unfairness 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 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 an entire segment of a flow must be transmitted before 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 enqueued.

Thus, 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 while minimally impacting other metrics. First, Titan uses *dynamic queue assignment* in the OS to evenly distribute flows to queues based on current queue occupancy. Thus, flows never share queues in undersubscribed conditions. Second, Titan adds a new queue weight abstraction to the NIC driver interface and a *dynamic queue weight assignment* mechanism in the kernel, which assigns weights to NIC queues based on current occupancy. In Titan, NICs use deficit round-robin [28] to ensure queues are serviced according to computed weights. Third, Titan adds *dynamic segmentation offload sizing* to dynamically reduce the segment size and hence reduce head-of-line blocking under oversubscription, which balances improvements to fairness against latency and increased CPU utilization.

We implement Titan in Linux, and, using both microbenchmarks on a pair of servers and cluster experiments, we show that Titan greatly reduces the unfairness in throughput across a range of under- and oversubscription conditions and both at short and long timescales. In most cases, there is near zero unfairness, and in the cases where it remains, Titan reduces unfairness by more than 60%. When using transmit packet steering (XPS) to pin queues to cores, Titan adds less than 1ms to 90^{th} percentile tail latency, and generally does not increase CPU utilization. Only at high flow count does the extra segmentation increase CPU utilization, although tail latency is almost halved at the same time. Our cluster experiments show that Titan offers the most fair flow completion times, and decreases flow completion times at the tail (90^th percentile).

In the next section we provide background material on server networking stacks and in Section 3 describe results from a comprehensive study of Linux networking options. Section 4 describes the design of Titan, and Section 5 has information on the implementation. Sections 6 and 7 describe our methodology and evaluation. We follow with a short discussion, related work, and then we conclude.

2 Background

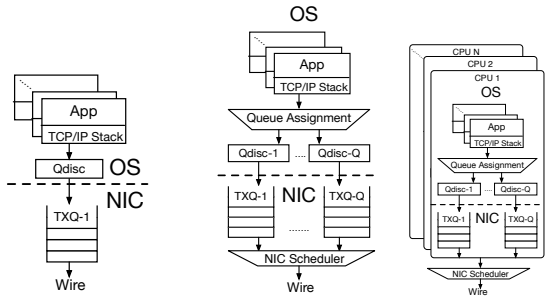
Networking in modern OSes is complex. There are multiple cooperating layers involved, and each layer has its own optimizations and configurations. As a result, it is difficult to know which configuration will provide the best performance. Further, there are multiple different dimensions by which the 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 server-side networking, and Table 1 lists the different configurations.

This section provides important background information on high-performance network stacks. Section 2.1 gives a high-level overview of the different layers involved in networking, introducing three different designs for implementing server-side networking of increasing complexity. Section 2.2 discusses the many important optimizations and design choices made at each layer and their inherent trade-offs.

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 side [19] 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



(a) A single queue (SQ) TX networking subsystem

(b) A multi-queue (MQ) TX networking subsystem

(c) A multicore-partitioned (XPS) multiqueue TX networking subsystem

Figure 1: Different server-side TX networking designs

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) [34].

Single Queue (SQ) In this design, segments from multiple competing applications (and containers and 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 parallel overheads (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 and may be reasonable if it can drive line rate (e.g., single 10Gbps NIC).

Multiqueue (MQ) To avoid SQ's parallel 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, although a single software queue could drive multiple hardware queues. The focus of this paper in on such MQ NICs.

Figure 1b illustrates MQ in Linux. Note that there is no explicit assignment of queues to cores in this model. Instead, the core that processes a queue is chosen as an ad-hoc side-effect of the OS scheduler and the current interrupt assignment, which may be dynamic. Although this may seem undesirable as it requires locking, it can improve performance during periods of asymmetric load. Computation can be migrated to idle or under-utilized cores [25] at the expense of performance isolation provided by dedicating queues to cores.

Moving to a multiqueue NIC requires that the OS implement some mechanism for assigning traffic to queues; in Linux queue assignment is done by hashing. Also, 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 [16, 24].

Modern NICs typically provide many more hardware queues, and using more queues than cores can be advantageous. For example, the Intel 82599 provides up to 128 queue pairs. However, the driver that we use (`ixgbe`) tries to set the number of queues to be equal to the number of cores by default.

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 [34]) 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.

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 a single interrupt, interrupt affinity (IRQ SMP affinity) must also be configured correctly for XPS to be effective.

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.

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
Interrupt frequency	Avoid wasting CPU polling	Increasing frequencies should increase CPU utilization, reduce latency, and improve fairness
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.

TSO/GSO: *Segmentation offloading* reduces per-packet processing overheads in the OS by processing segments instead of packets, which is important for driving high line-rates. Many NICs are capable of packetizing a TCP segment without CPU involvement, called TCP Segmentation Offloading (TSO). If a NIC supports scatter/gather lists, segmentation can be performed without copying data to linearize the packets in a segment. In Linux, 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.

The trade-off is that TSO/GSO hurts latency and fairness by causing head-of-line (HOL) blocking. Competing traffic must now wait until an entire segment is transmitted.

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 can implement 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 [28] to fairly schedule segments from competing flows regardless of differing segment sizes.

Interrupt Frequency: OSes often use interrupts to schedule network I/O, and NICs can throttle/coalesce interrupts to reduce CPU usage. Increasing interrupt frequency decreases latency at the cost of increasing CPU utilization, but only to a limit: OSes switch to polling when under heavy load to avoid livelock [22], for example, with NAPI in Linux.

TSO Interleaving: Transmitting every TSO segment in every queue to completion can significantly increase latency and harms fairness, even if each queue is serviced equally. Some NICs address this with *TSO interleaving* [16, 24], which sends a single MTU sized packed from each queue round-robin even if TSO segments are enqueued. This can lead to a more fair packet schedule 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 [11], which can hurt CPU utilization, latency, and fairness. TCP Small Queues (TSQ) [32] 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, all segments are freed by the driver only after they have been transmitted (or dropped), so it is easy to use a segment being freed as a signal that it has been transmitted. This has the side effect that the enqueuing of additional data for flows sharing the same queue is likely to be batched.

Hardware queue occupancy limits: Hardware queues are simple FIFOs, so increases in the number of bytes enqueued per-hardware queue translates directly to increase in HOL blocking latency. Byte Queue Limits (BQL) [8] 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) [8], which dynamically adjusts each hardware queue's BQL independently so as to decrease HOL blocking while avoiding starvation.

In summary, there are numerous configurable op-

tions in a multi-queue modern network stack that affect throughput, latency, CPU utilization and fairness. We next empirically study the tradeoffs imposed by these options.

3 Configuration Trade-Off Study

In this section, we study the impact of the aforementioned server configurations on server-side performance. Our experiments use two Linux servers that each have a 4-core/8-thread Intel Xeon E5-1410 CPU, and each server uses an Intel 82599 10 Gbps NIC [16] to communicate with the other across a dedicated TOR switch.

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* by sending 1KB ICMP pings between the two servers and report the 90$^{\text{th}}$ percentile latency. Linux software queues (`Qdiscs`) prioritize ICMP pings above all other traffic, so this experiment studies the latency experienced by *high priority traffic* that may end up sharing hardware queues with other flows.

4. We use a *fairness metric* introduced by Shreedhar and Varghese [28]. For every flow $i \in F$, there is some configurable quantity f_i that expresses i's fair share. In all of our experiments, we assume that 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 [28] no matter the size of the time interval. As mentioned earlier, this fairness metric is important, because even short time-scale unfairness can impact tail flow completion times in data centers [35].

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. 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.

Results and analysis We study server-side network performance in three workload settings: (1) *undersubscribed*, with three active flows whereas our hardware supports 8 queues; (2) *low oversubscribed*, where we have 12 active flows; and, (3) *high oversubscribed* where we have 48 active flows. In #2, even the best allocation of flows to queues will be uneven (some queues end up with 2 flows, whereas others have 1, i.e., 50% fewer). In contrast, #3 reflects a situation where it is possible to achieve an even allocation of flows to queues.

In each case, we consider how the queue configuration (MQ or XPS) works for various choices of the configuration/optimization options in Section 2.2. For simplicity, we look primarily at the MQ queue configuration, as single queues cannot sustain high line rates, and comment briefly on the (small) differences seen with XPS.

In what follows, we start by highlighting the main insights from the relative performance of the configurations/optimizations in Section 2.2 that generally hold across the different queue configurations (MQ or XPS) and workload settings. Our high-level take-aways for each optimization are listed in Table 2, and these are synthesized from the raw results presented for each combination of workload, queue configuration, and optimization that we detail in Tables 3 and 5. We note that using SFQ for the queuing discipline, with TCP small queues enabled, byte queue limits manually set to 256KB, and using 20KIPS tend to out-perform all other combinations across different queue configurations. This is denoted by C6 across the different queue configurations, which we henceforth focus on as the baseline best-performing MQ/XPS configuration today.

We now look at the differences across various subscription levels. Looking first at the undersubscribed case, the left side of Table 3 shows that it is possible to achieve good fairness (at both 1ms and 1s), low latency, and low CPU utilization with the MQ configurations C4–C7. However, it is surprising that fairness is not better with so few flows, as the hardware can do TCP interleaving for almost perfect fairness. We found the cause is imperfect hashing: frequently, multiple flows hash to the same queue. While Table 3 shows the average of 5 runs, Table 4 shows the best run, which occurs when flows hash to different queues. In those results, fairness is perfect for almost every configuration and latency is uniformly low. Hence, the central problem with MQ in

C1: Vanilla:	Default Linux networking stack incurs significant latency and unfairness, regardless of how many NIC queues are used, but has high throughput and low CPU.
C2: No TCP Small Queues (TSQ):	TSQ is an important optimization. Disabling can cause significant latency and unfairness.
C3: 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.
C4: No Byte Queue Limit (BQL):	BQL is an important optimization because disabling it can lead to increased latency and decreased fairness.
C5: 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.
C6: 20KIps:	Increasing the interrupt frequency to 20KIps can decrease tail latency and CPU utilization. At higher frequencies NAPI disables interrupts at high frequencies and switches to polling to avoid livelock.
C7: 64KB BQL:	Decreases latency but hurts fairness at long timescales with many flows.
C8: No TSO:	Disabling segmentation offloading hurts every performance metric because CPUs saturate.
C9: 16KB GSO:	Using a smaller GSO size than the default (64KB) improves fairness at short timescales (ms), increases CPU utilization.

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

Config	3 flows, 8 Queue (1 per CPU) TPut (Gbps)	CPU (%)	Latency (ms)	NFM (1ms)	NFM (1s)	Config	12 flows, 8 Queues (1-per CPU) TPut (Gbps)	CPU (%)	Latency (ms)	NFM (1ms)	NFM (1s)
C1: Vanilla	9.4	58	0.377	0.33	0.31	C1:	9.4	50	1.832	1.74	1.22
C2: Vanilla - SmallQs	9.4	48	1.186	0.20	0.16	C2:	9.4	52	1.725	1.87	1.41
C3: Vanilla + SFQ/Prio	9.4	54	0.498	0.36	0.30	C3:	9.4	50	1.194	1.83	1.46
C4: SFQ/Prio - BQL	9.4	50	0.543	0.17	0.15	C4:	9.4	54	2.044	1.88	1.47
C5: C3 + 256KB BQL	9.4	55	0.333	0.47	0.44	C5:	9.4	51	1.242	1.43	1.16
C6: C3 + 256KB BQL + 20KIps	9.4	57	0.388	0.16	0.15	C6:	9.4	44	1.040	1.92	1.51
C7: C3 + 64KB BQL + 20KIps	9.1	50	0.364	0.19	0.03	C7:	9.4	53	0.824	1.83	1.37
C8: C6 + No TSO	6.5	257	0.360	0.88	0.69	C8:	3.6	466	1.144	1.62	1.05
C9: C6 + 16KB GSO	9.4	70	0.578	0.15	0.17	C9:	9.4	58	0.907	1.55	1.45
XPS: C6 + XPS	9.4	63	0.366	0.00	0.01	XPS:	9.4	61	1.547	0.96	0.80
Titan1: DQA	9.4	61	0.268	0.00	0.00	T1:	9.4	43	1.448	1.33	0.84
Titan2: DQA + DQWA	9.4	84	0.300	0.01	0.12	T2:	9.4	89	2.215	1.20	0.14
Titan3: DQA + DQWA + DSOS (16KB)	9.4	88	0.315	0.01	0.11	T3:	9.4	103	3.862	0.32	0.24
TitanXPS1: T1 + XPS	9.4	72	0.506	0.00	0.00	TX1:	9.4	64	1.689	1.48	0.98
TitanXPS2: T2 + XPS	9.4	67	0.712	0.00	0.00	TX2:	9.4	54	2.086	0.64	0.07
TitanXPS3: T3 + XPS	9.4	68	0.710	0.00	0.02	TX3:	9.4	56	2.253	0.31	0.02

Table 3: The performance trade-offs of different OS configurations given 3 and 12 flows spread across 8 cores.

Config	TPut (Gbps)	CPU (%)	Latency (ms)	NFM (1ms)	NFM (1s)
C1:	9.4	52	0.256	0.00	0.02
C2:	9.4	32	0.247	0.00	0.00
C3:	9.4	38	0.270	0.00	0.00
C4:	9.4	44	0.255	0.00	0.00
C5:	9.4	40	0.253	0.00	0.00
C6:	9.4	45	0.262	0.00	0.00
C7:	8.7	38	0.242	0.18	0.04
C8:	5.6	216	0.238	0.41	0.07
C9:	9.4	57	0.236	0.00	0.00

Table 4: The best performing run of five runs given different OS configuration and 3 flows spread across 8 cores and 8 NIC queues. Because hash collisions are possible even when there are 3 flows, looking at the minimum across multiple runs captures performance when there are no hash collisions.

Config	48 flows, 8 Queues (1-per CPU) TPut (Gbps)	CPU (%)	Latency (ms)	NFM (1ms)	NFM (1s)
C1:	9.4	51	4.329	4.59	2.46
C2:	9.4	55	3.917	5.43	2.35
C3:	9.4	52	1.455	3.12	1.37
C4:	9.4	60	4.673	5.37	2.12
C5:	9.4	54	1.520	3.41	1.55
C6:	9.4	44	1.547	3.41	1.24
C7:	9.4	48	1.045	3.72	2.59
C8:	3.2	754	21.516	3.78	1.37
C9:	9.4	58	1.831	2.89	2.52
XPS:	9.4	65	3.544	3.55	1.17
T1:	9.4	54	1.935	4.08	0.46
T2:	9.3	102	11.173	3.84	1.21
T3:	9.4	111	1.749	1.97	0.63
TX1:	9.4	70	1.675	4.01	1.07
TX2:	9.4	66	1.783	3.78	0.43
TX3:	9.4	90	2.075	1.32	0.07

Table 5: The performance trade-offs of different OS configurations given 48 flows spread across 8 cores.

Linux and undersubscribed flows is the assignment of flows to queues. Only XPS consistently achieves near perfect fairness. This is because there is at most one flow per-core in the undersubscribed experiments. Because each core is dedicated its own queue, XPS avoids hash collisions.

Turning to low oversubscription on the right of Table 3, we see that 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. We also see that latency degrades, because (i) it is more likely that the ICMP packet encounters queued data and has to wait, and (ii) even if the queue is empty, the packet has to wait for many more other queues to be processed by the round-robin NIC scheduler. XPS here has substantially better fairness, but at the cost of increased latency and CPU utilization.

Finally, Table 5 shows performance for high oversubscription, i.e., 48 flows and 8 queues. Focusing on C6, surprisingly, latency is not much worse than 12 flows, likely due to byte queue limits (BQL) limiting the buildup of queued data. Fairness is uniformly worse here, 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. Focusing now on XPS, it looks largely similar to C6, except with somewhat worse latency. This is because XPS reduces the variance in the time it takes to enqueue packets in software and hardware queues. As a result, these queues are more likely to be full and cause some HOL blocking.

In summary, we find that using multiqueue NICs can generally offer low CPU utilization and good latency, but that the current Linux networking stack is unable to provide fairness at any time scale across flows. In the undersubscribed case fairness is relatively good, while for any level of oversubscription it is significant and persists over long timescales. In addition, using the best practices, exemplified particularly by configuration C6, can have substantial benefits over vanilla Linux without optimizations.

4 Titan

This section presents the design of Titan, an OS networking stack that that introduces new abstractions for improving network fairness with multiqueue NICs. These 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. However, as a consequence, a packet switch/scheduler in the NIC is now responsible for deciding which queue is allowed to send packets out on the wire at time. Unless the OS configures the NIC to behave differently, the best that a NIC can do is perform round-robin scheduling across competing hardware queues. As we showed in the previous section, when the network load is asymmetrically partitioned across the NIC's hardware queues, this will lead to an unfair packet schedule.

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 is composed of 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.

4.1 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 that 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 computes the assignment of flows to queues based on queue weights that are internally computed by the OS. 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.

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 uses the sum of the per-flow weights of all of the traffic in a software/hardware queue pair as the OS's internal queue weight. This enables a queue assignment algorithm to improve per-flow fairness. Titan dynamically tracks the sum of the weights of the flows using a 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 segment 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 for each newly arriving flow. The main overhead of this algorithm is that it read the weights of every queue.

In order to avoid packet reordering, DQA only changes a flow's queue assignment when there are 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.

4.2 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 introduces Dynamic Queue Weight Assignment (DQWA), a new abstraction that enables OSes to dynamically program NIC packet schedulers with per-queue weight information. 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. 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. However, the number of updates can be rate limited to reduce overhead.

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 [28]. 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. By providing an interface to modify the allocation of credits to queues, a NIC can configure DRR to service queues in proportion to different weights.

In our implementation, supporting the `ndo_set_tx_weight` in the `ixgbe` driver boils down to configuring the NIC scheduler's per-queue DRR credit allocation.

4.3 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 (FM). 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 have a significantly benefit network performance [10].

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 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 be configured to use reduced 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.) Because the software queue (`Qdisc`) is responsible for fairly scheduling traffic from different flows, this enables the OS to generate a fair packet 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.

5 Implementation

We implemented Titan on top of Linux 4.4.6 and modified Intel's out-of-tree `ixgbe-4.4.6` release [15] to

support the new `ndo_set_tx_weight` NDO. The authors of this paper have no special affiliation with Intel. We were able to implement this new NDO in this driver from the public hardware datasheets [16]. In a similar spirit, Titan is open source. The source code is 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 only hash received packets (received side steering, or RSS) to only four RX queues. This effectively decreases the NICs RX buffering capacity, so enabling this configuration can increase the number of packet drops.

During development, we found problems 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. If a TCP flow that will be processed later does not currently have any segments in the software queue, it will 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 in a batch are allowed to enqueue packets into their respective software queues before *any* packets are dequeued from software queues and enqueued in the hardware queues.

6 Methodology

To evaluate Titan, we perform both micro- and macrobenchmarks. In the microbenchmarks, we send traffic between two hosts across an otherwise uncongested switch. In the macrobenchmarks, we send traffic between up to 24 different servers connected by two switches. This section discusses the methodology used in these experiments in more detail.

In the microbenchmarks, we use a cluster of three servers connected to a dedicated TOR switch via 10 Gbps Ethernet cables. 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 [16]. The third server is used to monitor the packet schedule emitted by server A. This is because the Intel 82599 does not provide accurate per-packet timestamps: all packets delivered in the same interrupt appear to have arrived at the same time. Because of this, computing the fairness of the packet schedule at the traffic sink may be highly inaccurate. Instead, we configure the switch to use port mirroring to direct all traffic sent by the first server to the monitoring machine. This server uses an Intel NetEffect NE020 NIC [2], which provides packet timestamps accurate to the microsecond and is sufficient for measuring per-packet fairness at 10 Gbps line-rates.

We generate traffic in the microbenchmarks using multiple parallel `iperf3` [1] clients pinned to different CPUs. 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 send 1 KB pings between the two servers. To measure CPU utilization, we use `dstat`.

In the cluster workloads, we use a cluster of 24 servers on CloudLab. To evaluate Titan at scale, we carry out experiments on 24 physical servers. Each of the servers has 2 14-core/28-thread Intel Xeon E5-2683 v3 CPUs and 256GB of memory. Each server connects to one of two different interconnected switches via Intel 82599 NICs. The first switch is a Dell Networking Z9100, and the second is a Dell Networking S6000. As a result of job placement in CloudLab, 20 of the hosts are attached to the S6000 switch. In effect, our topology is an asymmetric dumbbell. Inspired by shuffle workloads used in prior work [3, 26, 14], 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 write our own application to transfer data in this workload.

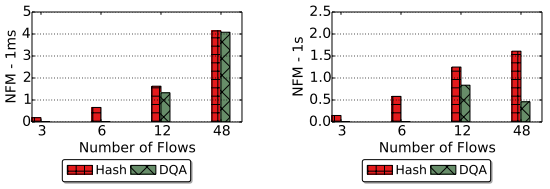
The base system uses the MQ configuration system with a GSO size of 64KB, TCP small queues enabled, byte queue limits manually set to 256KB, and 20KIPS. We perform all experiments 5 times and report the average.

7 Evaluation

We perform two different types of experiments to evaluate Titan. First, we run microbenchmarks to evaluate the performance impact of individual components of Titan in the absence of any of any network congestion. Second, we evaluate the impact of Titan on the network performance of a cluster of servers. In summary, we find that Titan is able to improve fairness on multiqueue NICs without harming other performance metrics.

7.1 Microbenchmarks

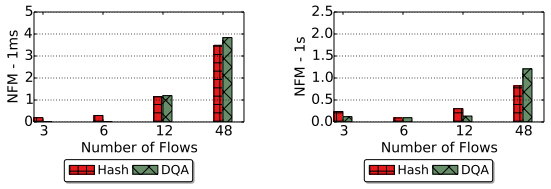
There are multiple complementary components to Titan, and we evaluate the impact of individual components on performance in the absence of network congestion. To avoid congestion, we measure the network fairness of competing flows that originate from the same server and are sent to the same destination. Tables 3 and 5



(a) Fairness Metric - 1ms (KB) **(b)** Fairness Metric - 1s (MB)

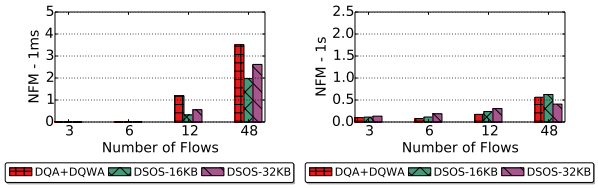
Figure 2: The importance of DQA to fairness. The Y-axis in these figures is NFM, the normalized fairness metric.

(bottom rows) show the performance of different components of Titan for each metric. The expected benefit of Titan is improved fairness, without hurting CPU utilization, throughput, or latency. These results show that this is the case, at least when XPS is used. If XPS is not configured, then a drawback of Titan is that it can increase CPU utilization.



(a) Fairness Metric - 1ms (KB) **(b)** Fairness Metric - 1s (MB)

Figure 3: The impact of dynamic queue weights (DQWA) on fairness.



(a) Fairness Metric - 1ms (KB) **(b)** Fairness Metric - 1s (MB)

Figure 4: The impact of dynamic segmentation (DSOS) on fairness.

Dynamic Queue Assignment DQA ensures that when there are fewer flows than queues, each flow is assigned its own queue. 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*, normalized fairness metric described in Section 3.

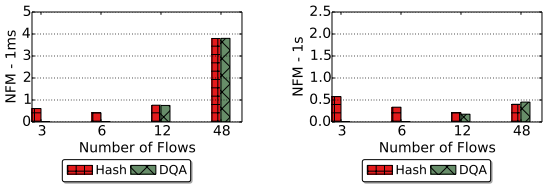
With hashing and 3 flows, fairness is good as there are few collisions. With 6 flows, though, the unfairness of hashing is very high because there are often hash collisions where multiple flows share a queue and achieve half the throughput of the flows that do not share queues. With low oversubscription (12 flows), fairness is again bad because some flows share queues and some do not. In the highly oversubscribed (48 flows) case over short timescales, unfairness is bad because of HOL blocking while waiting for GSO/TSO-size segments. Over long timescales shown in Figure 2b there is persistent unfairness due 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 share queues, and without DQWA to program weights in the NIC, all queues are serviced equally. With 48 flows over 1ms, there is still unfairness from HOL blocking. Over long timescales, though, with 48 flows DQA has low unfairness because it will place exactly 6 flows in each queue, so each flow receives equal treatment and HOL blocking is averaged out.

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 3 shows the fairness of both the hashing and 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 substantially improves the fairness of both hashing and DQA when there are more than a small number of flows because the NIC is able to give more service to queues with more flows. In the highly oversubscribed case, hashing with DQWA outperforms DQA + DQWA by a small margin. This is because it takes time to update queue weights, and updates to the OS's weights are batched due to interrupts and NAPI. DQA introduces more churn in queue weights, which can (momentarily) hurt fairness by increasing the discrepancy between OS and NIC weights.

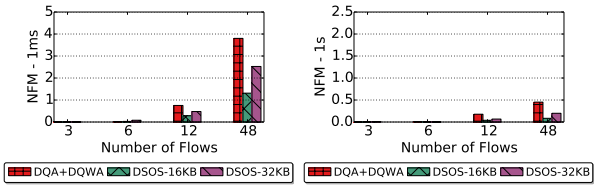
We note that DQA is a software-only solution that has the largest impact in undersubcribed cases and helps at both short and long timescales. DQWA helps most in (i) oversubscribed cases and (ii) over longer timescales. In addition, DQWA requires hardware support that, while minimal, may not be present in all NICs.

Dynamic Segmentation Offload Sizing DQA improves fairness during undersubscription and DQWA improves fairness during oversubscription, but head-of-line



(a) Fairness Metric - 1ms (KB) **(b)** Fairness Metric - 1s (MB)

Figure 5: The impact of dynamic queue weights (DQWA) on fairness when XPS is enabled.



(a) Fairness Metric - 1ms (KB) **(b)** Fairness Metric - 1s (MB)

Figure 6: The impact of dynamic segmentation (DSOS) on fairness when XPS is enabled.

blocking still occurs over short timescales. DSOS addresses this 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 two DSOS segment sizes: 16KB and 32KB.

Figure 4a shows the 1ms fairness of DSOS for a range of flow sizes. In the 3 and 6 flow cases there is no oversubscription, so DSOS leaves the GSO size at 64KB. For 12 and 48 flows, though, DSOS dynamically reduces the segment size to reduce HOL blocking and drops the NFM substantially. The reduction is greater with 12 flows, as there are fewer flows sharing each queue and hence less blocking. Over the longer timescale shown in Figure 4b, DSOS has little impact as expected.

XPS So far, our evaluation has focused on the multi-queue 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 5 and Figure 6 show the fairness of Titan with DQA+DQWA and DSOS when XPS is configured (Figures 3 and 4 with XPS). Section 3 found that XPS has a positive impact on performance during undersubscription and low oversubscription but did not improve performance in heavy oversubscription. These figures show that XPS in Titan has a positive impact on performance in both oversubscription cases. XPS can reduce CPU utilization and improve fairness. When XPS, DQA, DQWA, and DSOS are combined together, it is possible to improve fairness by an order of magnitude.

The main difference between Figure 5 and Figure 3 is that XPS is more fair in the highly oversubscribed case over 1 sec timescales. XPS increases the likelihood that a flow has packets enqueued in the software queue because it reduces the overheads of enqueuing packets into a software queue by improving locality. This reduces queue weight churn and improves DQWA performance.

Figure 6, which considers DSOS with XPS (with both DQA and DQWA enabled), shows that XPS significantly improves fairness when under heavy oversubscription. When there are 48 flows, using a 16KB dynamic segment size with XPS almost halves NFM at short time scales and reduces NFM by even more at long timescales. The reason for this is because XPS reduces the CPU overheads of DSOS (Tables 3 and 5).

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. The last rows of Tables 3 and 5 in Section 3 show the performance of Titan on 3, 12, and 48 flows.

Across all experiments, throughput is identical with Titan and standard Linux networking options. 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. Our implementation of DQWA can also increase CPU utilization because it limits the NIC to using 4 RX queues. Fortunately, enabling XPS significantly reduces the CPU utilization of all of the features of Titan.

Recall that latency is measured by periodically sending a 1KB ICMP ping between the servers and that we report the 90th percentile latency seen. In the undersubscribed case, Titan improves latency as every flow receives its own queue. When oversubscribed, though, Titan can increase latency. In the 12-flow case, this occurs for an unexpected reason. With hashing and 8 hardware queues, there are often idle queues to which the ICMP packets used for latency measurements can be enqueued and sent at low latency. With DQA, all queues are occupied, so the ICMP packets must wait behind TCP data. At 48 flows, Titan without DSOS has high latency again because it keeps all queues equally filled. With DSOS, though, Titan achieves latency similar to the best Linux configurations. With high flow counts causing HOL blocking, using smaller 16KB segments reduces tail latency from over 10ms to under 2ms.

Summary Comparing Titan to optimized Linux with XPS (rows XPS and TitanXPS3 in Tables 3 and 5), we find that overall Titan greatly improves fairness, often at

no or negligible CPU overhead and a minor (< 1ms) increase in tail latency. In the undersubscribed case, Titan and optimized Linux have similar fairness and CPU utilization, and Titan adds only 400μs to 90th tail latency. At low oversubscription, Titan's unfairness is 67% lower at short timescales and 97% lower at longer timescales. Titan's tail latency here is 2.3ms vs 1.5ms for Linux, and CPU utilization drops slightly. At high oversubscription, fairness improvements are similar to low oversubscription, but tail latency improves from 3.5ms to 2.1ms. CPU utilization rises here from 65% to 90% due to extra segmentation from DSOS. Without DSOS, CPU utilization is unchanged, latency is even lower, abut short-term fairness is no better than Linux.

7.2 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.

Figure 7 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 given a cluster of servers that simultaneously open connections to every other server and transfer 1GB of data. Figure 7 shows that Titan has a significant impact on the network fairness of a cluster of servers regardless of the level of oversubscription experienced by the cluster. Titan consistently provides the most fair flow completion times. For this test, DQA (without DQWA or DSOS) is enough to get most of the fairness benefit of Titan. .

In Figure 7, Vanilla shows the performance of Linux without any configuration and optimized Vanilla shows the best performing configuration from Section 3 (C6). Surprisingly, optimized Vanilla leads to more unfairness than Vanilla. This occurs because optimized Vanilla causes the kernel to enter a different mode of operation. In Vanilla, software queues may become starved. When this happens, TCP small queues is the dominant influence on performance. TCP small queues batches TCP transmissions, which can improve fairness. In contrast, software and hardware queues are less frequently empty in optimized Vanilla, so the fairness problems that can be caused by NIC schedulers have a larger impact.

However, this does not mean that optimized Vanilla is not a good configuration choice. To show why, Figure 8 shows a CDF of the flow completion times across all the flows when 6 servers perform all-to-all communication. This figure shows that optimized Vanilla consistently provides faster flow completion times than default Vanilla. However, the variance in flow completion times also increases. In contrast, Titan provides more consistent flow completion times and decreases flow completion times at the tail of the distribution (>90th percentile). Titan also improves fairness.

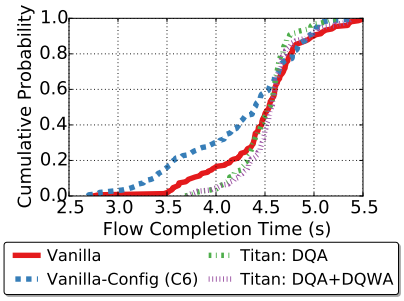


Figure 8: Flow completion times given a cluster of 6 servers performing all-to-all communication

8 Data Center Bridging

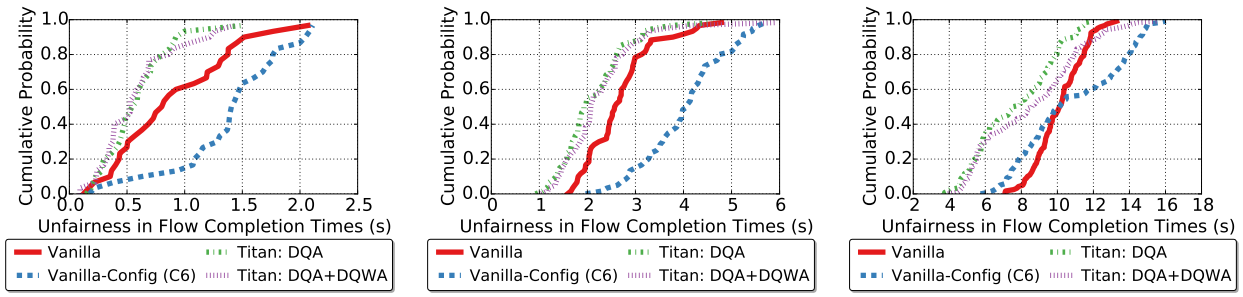
Data center bridging (DCB) [9] is an important standard for defining multiple network traffic classes (priorities). It is implemented by many NICs and switches and can be used to improve network performance. As argued next, DCB and Titan offer complementary benefits.

DCB provides up to 8 different network priorities which can be used to isolate different classes of network traffic. DCB-enabled NICs and switches must implement a packet scheduler such that higher priority traffic is scheduled before lower priority traffic, subject to per-priority maximum utilization constraints that are statically configured [31].

Given DCB, an OS can ensure that different segments receive different service from the packet schedulers implemented in both the NICs and switches. In order to implement DCB, multiqueue NICs simply provide the OS with more transmit queues to avoid parallel overheads, with each transmit queue being assigned to one of the 8 static priorities. For example, the Intel 82599 NIC can export at many as 128 queues when DCB is enabled [16].

This is one of the reasons why Titan can help even with DCB: If DCB is enabled, flows in the same DCB priority but spread across the different NIC queues assigned to the priority will still compete unfairly. Titan directly addresses this problem. Additionally, DCB addresses problems not solved by Titan. Section 3 and 7 show that using multiqueue NICs can lead to significant HOL blocking latency, even with Titan. DCB provides a means for solving this problem. If traffic from different classes are isolated in different queues, then bulk traffic will not be able to hurt the latency of high priority traffic.

We are currently extending Titan to support DCB.



(a) Network fairness given a cluster of 6 servers performing all-to-all communication

(b) Network fairness given a cluster of 12 servers performing all-to-all communication

(c) Network fairness given a cluster of 24 servers performing all-to-all communication

Figure 7: The impact of Titan on flow completion times on a cluster of servers.

9 Related Work

Titan is most closely related to SENIC [24] and Silo [17][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 [24, 13, 19]. This is because it provides the OS with the most control over packet scheduling; but, SQ cannot drive future high link rates.

There has been recent work on building networks that provide programmable packet scheduling [30, 21, 6], allowing flows to fairly compete [5, 33, 31], and performing traffic engineering in the network [3, 14, 7, 26, 4, 10]. Titan is motivated by similar concerns 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 afore-mentioned techniques.

Juggler [10] and Presto [14] have designed RX network stacks that are resilient to packet reordering. Such stacks are complementary to Titan. Importantly, they can improve Titan as they enable new queue assignment algorithms that are not required to avoid packet reordering.

Fastsocket [20] observes that the Linux stack has multicore scalability problems when a server handles many short-lived network connections due to cross-core contention. To address this problem, the authors modify kernel data structures so as to allow each core to independently processing incoming traffic. This design requires XPS, and can benefit from DQWA when the per-core load is asymmetric.

Bullet trains [18] found that the packet stream emitted by NICs is very bursty when considered at 10–100 microsecond timescales. The improvements introduced in Titan reduce the per-flow burst length when there are multiple active flows.

Fastpass [23] aims to globally schedule all network traffic in a cluster. In order to scale, Fastpass uses an independent queue per-core. This can lead to increased skew in the packet schedule depending on the behavior of the NIC scheduler. Titan's scheduling can help mitigate the skew.

10 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 microbenchmarks and cluster experiments 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

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

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.

References

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

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

[3] 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* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 19–19.

[4] 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 SIGCOMM* (2014), SIGCOMM '14, pp. 503–514.

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

[6] 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 ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13, pp. 435–446.

[7] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies* (2011), CoNEXT '11, pp. 8:1–8:12.

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

[9] DATA CENTER BRIDGING TASK GROUP. http://www.ieee802.org/1/pages/dcbridges.html.

[10] GENG, Y., JEYAKUMAR, V., KABBANI, A., AND ALIZADEH, M. Juggler: A practical reordering resilient network stack for datacenters. In *EuroSys* (2016), ACM.

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

[12] 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), pp. 139–152.

[13] 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.

[14] 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 (SIGCOMM)* (2015), pp. 465–478.

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

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

[17] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *SIGCOMM* (2015).

[18] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., , AND PORTER, G. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *CoNEXT* (2013), ACM.

[19] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with FlexNIC. In *ASPLOS* (2016).

[20] 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 *ASPLOS* (2016), ACM.

[21] 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.

[22] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15*, 3 (Aug. 1997).

[23] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM 2014* (Chicago, IL, August 2014).

[24] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for end-host rate limiting. In *NSDI* (2014).

[25] RAM, K. K., COX, A. L., CHADHA, M., AND RIXNER, S. Hyper-switch: A scalable software virtual switching architecture. In *USENIX ATC* (2013), USENIX.

[26] 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 SIGCOMM* (2014), SIGCOMM '14, pp. 407–418.

[27] 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), pp. 123–137.

[28] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *SIGCOMM* (1995), ACM.

[29] 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), pp. 183–197.

[30] 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 Conference on ACM SIGCOMM 2016 Conference* (2016), SIGCOMM '16, pp. 44–57.

[31] STEPHENS, B., COX, A. L., SINGLA, A., CARTER, J. B., DIXON, C., AND FELTER, W. Practical DCB for improved data center networks. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014* (2014), pp. 1824–1832.

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

[33] VAMANAN, B., HASAN, J., AND VIJAYKUMAR, T. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, pp. 115–126.

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

[35] ZATS, D., DAS, T., MOHAN, P., BORTHAKUR, D., AND KATZ, R. Detail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, pp. 139–150.