Queues don't matter when you can JUMP them!

Matthew P. Grosvenor   Malte Schwarzkopf   Ionel Gog   Robert N. M. Watson
Andrew W. Moore   Steven Hand[†]   Jon Crowcroft

University of Cambridge Computer Laboratory

[†] now at Google, Inc.

Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. To mitigate network interference, QJUMP applies Internet QoS-inspired techniques to datacenter applications. Each application is assigned to a latency sensitivity level (or class). Packets from higher levels are rate-limited in the end host, but once allowed into the network can "jump-the-queue" over packets from lower levels. In settings with known node counts and link speeds, QJUMP can support service levels ranging from strictly bounded latency (but with low rate) through to line-rate throughput (but with high latency variance).

We have implemented QJUMP as a Linux Traffic Control module. We show that QJUMP achieves bounded latency and reduces in-network interference by up to 300×, outperforming Ethernet Flow Control (802.3x), ECN (WRED) and DCTCP. We also show that QJUMP improves average flow completion times, performing close to or better than DCTCP and pFabric.

1 Introduction

Many datacenter applications are sensitive to tail latencies. Even if as few as one machine in 10,000 is a straggler, up to 18% of requests can experience high latency [13]. This has a tangible impact on user engagement and thus potential revenue [8, 9].

One source of latency tails is *network interference*: congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. For example, Hadoop MapReduce can cause queueing that extends memcached request latency tails by 85 times the interference-free maximum (§2).

If memcached packets can somehow be prioritized to "jump-the-queue" over Hadoop's packets, memcached will no longer experience latency tails due to Hadoop. Of course, multiple instances of memcached may still interfere with *each other*, causing long queues or incast collapse [10]. If each memcached instance can be appropriately rate-limited at the origin, this too can be mitigated.

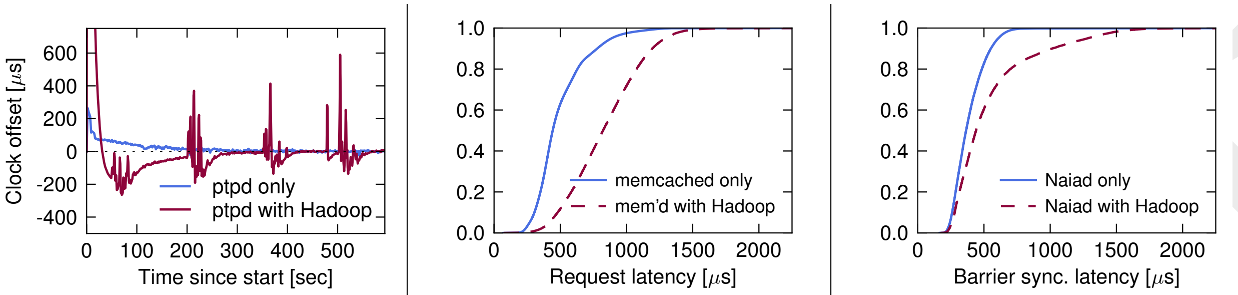
These observations are not new: QoS technologies like DiffServ [7] demonstrated that coarse-grained classification and rate-limiting can be used to control network latencies. Such schemes struggled for widespread deployment, and hence provided limited benefit [12]. However, unlike the Internet, datacenters have well-known network structures (i.e. host counts and link rates), and the bulk of the network is under the control of a single authority. In this environment, we can enforce system-wide policies, and calculate specific rate-limits which take into account worst-case behavior, ultimately allowing us to provide a guaranteed bound on network latency.

QJUMP is implemented via a simple rate-limiting Linux kernel module and application utility. QJUMP has four key features. It:

1. resolves network interference for latency-sensitive applications **without sacrificing utilization** for throughput-intensive applications;
2. offers **bounded latency** to applications requiring low-rate, latency-sensitive messaging (e.g. timing, consensus and network control systems);
3. is simple and **immediately deployable**, requiring no changes to hardware or application code; and
4. **performs close to or better** than competing systems, including ECN, 802.3x, DCTCP and pFabric, but is considerably less complex to understand, develop and deploy.

In this work, we consider only latency tails that result from in-network interference. Other work mitigates host-based sources of latency tails [14, 23, 30, 32, 36].

Please see *http://www.cl.cam.ac.uk/netos/qjump* for full details including the QJUMP source-code. In the electronic version of this paper, most of the figures and tables are clickable with links to a full experimental description and original datasets.



(a) Timeline of PTP synchronization offset. **(b)** CDF of memcached request latencies. **(c)** CDF of Naiad barrier sync. latencies.

Figure 1: Motivating experiments: Hadoop traffic interferes with *(a)* PTPd, *(b)* memcached and *(c)* Naiad traffic.

Setup	50th%	99th%
one host, idle network	85	126µs
two hosts, shared switch	110	130µs
shared source host, shared egress port	228	268µs
shared dest. host, shared ingress port	125	278µs
shared host, shared ingress and egress	221	229µs
two hosts, shared switch queue	**1,920**	**2,100µs**

Table 1: Median and 99th percentile latencies observed as `ping` and `iperf` share various parts of the network.

2 Motivation

We begin by showing that shared switch queues are the primary source of network interference. We then quantify the extent to which network interference impacts application-observable metrics of performance.

2.1 Where does the latency come from?

Network interference may occur at various places on the network path. Applications may share ingress or egress paths in the host, share the same network switch, or share the same queue in the same network switch. To assess the impact of interference in each of these situations, we emulate a latency-sensitive RPC application using `ping` and a throughput-intensive bulk transfer application by running two instances of `iperf`. Table 1 shows the results of arranging `ping` and `iperf` with various degrees of network sharing. Although any sharing situation results in interference, the effect is worst when applications share a congested switch queue.. In this case, the 99th percentile `ping` latency is degraded by over 16× compared to the unshared case.

2.2 How bad is it really?

Different applications use the network in different ways. To demonstrate the degree to which network interference affects different applications, we run three representative latency-sensitive applications (PTPd, memcached and Naiad) on a network shared with Hadoop (details in §6) and measure the effects.

1. Clock Synchronization Precise clock synchronization is important to distributed systems such as Google's Spanner [11]. PTPd offers microsecond-granularity time synchronization from a time server to machines on a local network. In Figure 1a, we show a timeline of PTPd synchronizing a host clock on both an idle network and when sharing the network with Hadoop. In the shared case, Hadoop's shuffle phases causes queueing, which delays PTPd's synchronization packets. This causes PTPd to temporarily fall 200–500µs out of synchronization; 50× worse than on an idle network.

2. Key-value Stores Memcached is a popular in-memory key-value store used by Facebook and others to store small objects for quick retrieval [25]. We benchmark memcached using the `memaslap` load generator[2] and measure the request latency. Figure 1b shows the distribution of request latencies on an idle network and a network shared with Hadoop. With Hadoop running, the 99th percentile request latency degrades by 1.5× from 779µs to 1196µs. Even worse, approximately 1 in 6,000 requests take over 200ms to complete[3], over 85× worse than the maximum latency seen on an idle network.

3. Iterative Data-Flow Naiad is a framework for distributed data-flow computation [24]. In iterative computations, Naiad's performance depends on low-latency state synchronization between worker nodes. To test Naiad's sensitivity to network interference, we execute a barrier synchronization benchmark (provided by the Naiad authors) with and without Hadoop running. Figure 1c shows the distribution of Naiad synchronization latencies in both situations. On an idle network, Naiad takes around 500µs at the 99th percentile to perform a four-way barrier synchronization. With interference, this grows to 1.1–1.5ms, a 2–3× performance degradation.

[2] *http://libmemcached.org*

[3] Likely because packet loss triggers the TCP minRTO timeout.

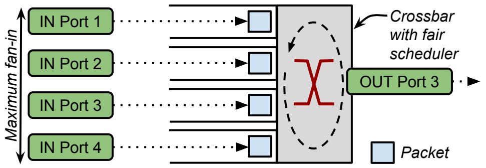


Figure 2: Packets fanning in to a four-port, virtual output queued switch. Output queues shown for port 3 only.

3 QJump System Design

Our exploratory experiments demonstrate that applications are sensitive to network interference, and that network interference occurs primarily as a result of shared switch queues. QJump therefore tackles network interference by reducing switch queueing: essentially, if we can reduce the amount of queueing in the network, then we will also reduce network interference. In the extreme case, if we can place a low, finite bound on queueing, then we can fully control network interference. This idea forms the basis of QJump.

In this section, we derive an intuitive model to place such a bound on queueing in any datacenter network topology. We first consider a single switch case, before extending the model to cover multiple switches. We then relax our model's throughput constraints and quantify the latency variance vs. throughput tradeoff. Finally, we describe how latency-sensitive traffic is allowed to "jump-the-queue" over high-throughput traffic.

Although the model we present is intuitive, it amounts to a simplification of the classic Parekh-Gallager theorem [27, 28]. The theorem shows that end-to-end delay can be bounded in a Weighted Fair Queueing (WFQ) network, provided that the network remains undersubscribed. In the Appendix, we show the relationship between our model and the theorem. In essence, we use the fact that datacenter networks have a well-known structure (unlike the Internet) to simplify the theorem, resulting in the version that we now present.

3.1 Bounded Queues – Bounded Latency

To begin, we assume an initially idle network in which each host is connected by a single link. We also assume that the link rate never decreases from the edge to the core of the network—an assumption that is true in any reasonable datacenter network.

Single Switch Queueing Model Consider the simplified model of a typical virtual-output queued (VOQ) layer 2 switch shown in Figure 2. The figure shows four input ports which are connected to four output ports via a crossbar. Only the output queues for port 3 are shown. One of two scenarios might occur at an instant in time: (*i*) only one input port sends packets to output port 3; or (*ii*) multiple input ports send packets to output port 3.

In the first case, a single sender can communicate with the destination port without queueing. Packets are only delayed by the processing delay across the switch, which is typically less than 0.5μs.

In the second case, packets arrive concurrently and only one packet can exit from the output port at a time. The switch scheduler must share access to this output by serializing the concurrent arrivals. In the worst case, the number of packets that arrive concurrently is equal to the *maximum fan-in* of the switch (see Figure 2), which is the number of input ports on the switch (four in this example). Thus, a packet may have to wait for up to $max\ fan\text{-}in - 1$ packets before it is serviced .

Multi Switch Queueing Model We can easily expand this understanding to cover multi-hop networks by treating the whole network as a single "big switch" (this is a version of the *hose-constraint* [15] model). Since we assume that each host has only one connection to the network, all packets "fanning in" to a host must eventually use this one link. This represents a mandatory serialization point, regardless of the core network topology. Given n hosts in the network, a packet may therefore experience at most $max\ network\ fan\text{-}in - 1 = n - 2 \approx n$ packets worth of delay. Knowing that a packet of size P will take ${}^{P}/_{R}$ seconds to transmit at link-rate R, we can therefore bound the maximum interference delay at:

$$worst\ case\ end\text{-}to\text{-}end\ delay \leq n \times \frac{P}{R} + \varepsilon \qquad (1)$$

where n is the number of hosts, P the maximum packet size (in bits), R is the rate of the slowest link in bits per second and ε is the cumulative processing delay introduced by switch hops.

Network epochs So far, our model assumes that switch queues are initially empty and that the network is undersubscribed. In this case, Equation 1 offers an upper bound on end-to-end network delay. We refer to the result from Equation 1 as a *network epoch*. Intuitively, a network epoch is the maximum time that an idle network will take to service one packet from every sending host, regardless of the source, destination or timing of those packets. If all hosts are rate-limited so that they cannot issue more than one packet per epoch, no permanent queues can build up and the end-to-end network delay bound will be maintained forever.

One problem with a network epoch is that it is a global concept: to maintain it, all hosts must agree on when an epoch begins and when an epoch ends. This requires scheduling and precise timing. If all hosts in the network share a single time source, network epochs can be synchronized. Hardware time-stamped PTP synchronization on modern hardware can be used for micro-second granularity network scheduling [29]. PTP synchronization hardware is not yet ubiquitous. As an alternative, we can allow the network to become *mesochronous*. That is, we require all network epochs in the system to occur at the same *frequency*, but impose no restriction on the *phase* relationship between epochs. In this case, host-based timing is sufficient, so long as drift remains minimal over the sub-millisecond timespan of a network epoch.

This mesochronous relaxation does, however, affect our assumption of an initially idle network. A phase misalignment between hosts (or switches) means that a switch may encounter *two* packets within a host's network epoch: the first packet being issued at the end of an epoch and the second packet issued immediately at the start of the next epoch. The probability of this unfortunate alignment occurring decreases exponentially with scale. With as few as ten machines, the likelihood of waiting behind more than n packets is very small. Nevertheless, to ensure that the latency bound is *guaranteed*, we can accommodate the mesochronous case by doubling our worst-case latency bound. Our network epoch calculation thus becomes:

$$\textit{network epoch} = 2n \times \frac{P}{R} + \varepsilon \tag{2}$$

This is a key property of QJUMP: if we rate-limit all hosts so that they can only issue one packet every network epoch, then no packet will take more than one network epoch to be delivered in the worst case.

3.2 Latency Variance vs. Throughput

Although the equation derived above provides an absolute upper bound on in-network delay, it also has the effect of aggressively restricting throughput. Formulating Equation 2 for throughput, we obtain:

$$\textit{throughput} = \frac{P}{\textit{network epoch}} \approx \frac{R}{2n} \tag{3}$$

That is, as we increase the number of hosts n linearly, we decrease the throughput capacity for each host by a factor of $2n$. For example, with 1,000 hosts and a 10Gb/s edge we obtain an effective throughput of less than 5Mb/s per host. Clearly, this is not ideal.

We can improve this situation by making two observations. First, Equation 2 is pessimistic: it assumes that all hosts transmit to one destination at the worst time, which is unlikely given a realistic network and traffic distribution. Second, some applications (e.g. PTP) are more sensitive to interference than others (e.g. memcached, Naiad) whereas still other applications (e.g. Hadoop) are more sensitive to throughput restrictions.

From the first observation, we can relax the throughput constraints in Equation 2 by assuming that fewer than n hosts send to a single destination at the worst time. For example, if we assume that only 500 of the 1,000 hosts concurrently send to a single destination, then those 500 hosts can send at twice the rate and maintain the same network delay. More generally, we define a scaling factor f so that the assumed number of senders n' is given by:

$$n' = \frac{n}{f} \quad \text{where } 1 \leq f \leq n. \tag{4}$$

Intuitively, f is a "throughput factor": as the value of f grows, so does the amount of bandwidth available.

From the second observation, some (but not all) applications can tolerate some degree of latency variance. For these applications, we aim for a statistical reduction in latency variance. This re-introduces a degree of statistical multiplexing to the network, but one that is more tightly controlled than in current networks. When the the value of f is too optimistic (i.e. the actual number of senders is greater than n'), some queueing may occur, resulting in network interference.

The probability that interference occurs increases with increasing values of f. At the upper bound ($f = n$), latency variance is no worse than in existing networks and full network throughput is available. At the lower bound ($f = 1$), latency is guaranteed, but with much reduced throughput. In essence, f quantifies the latency variance vs. throughput tradeoff.

3.3 Jump the Queue with Prioritization

We would like to use multiple values of f concurrently, so that different applications can benefit from the latency variance vs. throughput tradeoff that suits them best. To achieve this, we partition the network so that traffic from latency-sensitive applications (e.g. PTPd, memcached, Naiad) can "jump-the-queue" over traffic from throughput intensive applications (e.g. Hadoop).

Datacenter switches support the IEEE 802.1Q [18] standard which provides eight (0–7) hardware enforced "service classes" or "priorities". Priorities are rarely used in practice because priority selection can become a "race to the top". For example, memcached developers may assume that memcached traffic is the most important and should receive the highest priority. Meanwhile, Hadoop developers may assume that Hadoop traffic is the most important, and should similarly receive the highest priority. Since there is a limited number of priorities, neither can achieve an advantage and prioritization loses its value. QJUMP is different.

QJUMP couples priority values and rate-limits: for each priority, we assign a distinct value of f, with higher priorities receiving *smaller* values. Since a small value of f implies an aggressive rate limit, priorities become useful because they are no longer "free": QJUMP users must choose between low latency variance at low throughput (high priority) and high latency variance at high throughput (low priority).

We call the assignment of an f value to a priority a QJUMP *level*. The latency variance of a given QJUMP level is a function of the sum of the QJUMP levels above it. In Section 5, we discuss various ways of assigning f values to QJUMP levels.

4 QJUMP Implementation

QJUMP has two components: a rate-limiter to provide admission control to the network, and an application utility to configure unmodified applications to use QJUMP levels. In a multi-tenant environment, the rate-limiter is deployed as a component in the hypervisor and QJUMP is configured for the total number of virtual hosts. In a single-authority environment, the rate-limiter is deployed as an addition to the kernel network egress path and QJUMP is configured for the number of physical hosts.

Rate limiting QJUMP differs from many other systems that use rate-limiters. Instead of requiring a rate-limiter for each flow, each host only needs one coarse-grained rate-limiter per QJUMP level. This means that just eight rate-limiters per host are sufficient when using IEEE 802.1Q priorities. As a result, QJUMP rate-limiters can be implemented efficiently in software.

In our prototype, we use the queueing discipline (`qdisc`) mechanism offered by the Linux kernel traffic control (TC) subsystem to rate-limit packets. TC modules do not require kernel modifications and can be inserted and removed at runtime, making them flexible and easy to deploy. We also use Linux's built-in 802.1Q VLAN support to send layer 2 priority-tagged packets.

Listing 1 shows our a custom rate-limiter implementation. To keep the rate-limiter efficient, all operations quantify time in cycles. This requires us to initially convert the network epoch value from seconds into cycles (line 1). We then synthesize a clock from the CPU timestamp counter (`rdtsc`, line 6). This provides us with extremely fine-grained timing for the price of just one instruction on the critical path.

When a new packet arrives at the rate-limiter, it is classified into a QJUMP level using the priority tag found in its `sk_buff` (line 7). Users can set this priority directly in the application code, or assign priorities to unmodified binaries using our application utility. Next, the rate-limiter checks if a new epoch has begun. If so, it issues a fresh allocation of bytes to itself (lines 8–10). It then checks to see if sufficient bytes are remaining to send the packet in this network epoch (line 12). If so, the packet is forwarded to the driver (line 15–16), if not, the packet is dropped (line 13). In practice, packets are rarely dropped because our application utility also resizes socket buffers to apply early back-pressure.

Forwarded packets are mapped onto individual driver queues depending on the priority level. QJUMP therefore prioritizes low-latency traffic in the end-host itself, before packets are issued to the network card.

```
1  long epoch_cycles = to_cycles(network_epoch);
2  long timeout = start_time;
3  long bucket[NUM_QJUMP_LEVELS];
4
5  int qJumpRateLimiter(struct sk_buff* buffer) {
6    long cycles_now = asm("rdtsc"); /* read cycle ctr */
7    int level = buffer->priority;
8    if (cycles_now > timeout) { /* new token alloc? */
9      timeout += epoch_cycles;
10     bucket[level] = tokens[level];
11   }
12   if (buffer->len > bucket[level]) {
13     return DROP; /* tokens for epoch exhausted */
14   }
15   bucket[level] -= buffer->len;
16   sendToHWQueue(buffer, level);
17   return SENT;
18 }
```

Listing 1: QJUMP rate-limiter pseudocode.

Since Equation 2 assumes pessimal conditions, our rate-limiter also tolerates bursts up to the level-specific byte limit per epoch. This makes it compatible with hardware offload techniques such as TSO, LSO or GSO.

On our test machines, we found no measurable effect of the rate-limiter on CPU utilization or throughput. On average it imposes a cost of 35.2 cycles per packet ($\sigma = 18.6$; $99^{\text{th}}\%$ = 69 cycles) on the Linux kernel critical path of ≈8,000 cycles. This amounts to less than 0.5% overhead.

QJUMP Application Utility QJUMP requires that applications (or, specifically, sockets within applications) are assigned to QJUMP levels. This is easily done in application code directly with a `setsockopt()` using the `SO_PRIORITY` option. However, we would also like to support unmodified applications without recompilation. To achieve this, we have implemented a utility that dynamically intercepts socket setup system calls and alters their options. We inject the utility into unmodified executables via the Linux dynamic linker's `LD_PRELOAD` support (a similar technique to OpenOnload [31]).

The utility performs two tasks: (*i*) it configures socket priority values, and (*ii*) it sets socket send buffer sizes. Modifying socket buffer sizes is an optimization to apply early back-pressure to applications. If an application sends more data than its QJUMP level permits, an `ENOBUFS` error is returned rather than packets being dropped. While not strictly required, this optimization brings a significant performance benefit in practice as it helps avoid TCP retransmit timeouts (minRTOs).

5 Configuring QJUMP

A QJUMP deployment requires five parameters to be configured: (*i*) n, the number of hosts; (*ii*) P, the maximum packet size; (*iii*) R, the rate of the slowest edge link; (*iv*) ε, the edge-to-edge cumulative switch processing delay; and (*v*) f_i, the assumed fraction of concurrently transmitting hosts at each level.

Configuring R and ε As the topology of a datacenter network is static, the minimum link speed R and the cumulative switching delay ε do not vary. Typical values are R = 10Gb/s or 40Gb/s and ε = 1μs to 4μs.

Configuring P In §3.1, we defined P as the "maximum packet size". However, it is more correctly defined as the maximum number of bytes that can be issued into the network at the guaranteed latency level in a single network epoch. From Equation 2, the network epoch grows linearly with increasing P, so P should be kept small to keep the network epoch short. However, we also want P to be big enough to be useful. Benson *et al.* found that 30%–50% of packets in many datacenters contain fewer than 256 bytes [5]. This suggests that ≤256B packets are sufficient for some applications. For 1,000 hosts, setting P to 256 bytes results in a worst-case delay of <500μs.

Configuring n This usefulness of QJUMP depends on the size of the latency bound, which scales as a function of n. If all hosts in the network use QJUMP, then n can take a value of between 1,000 and 4,000 hosts and maintain a bound of 100-500μs using small messages of 64–256B. QJUMP can also be configured with n set as a subset of the hosts, provided that the remainder of hosts only use the lowest network priority.

Application-specific knowledge may also be exploited to increase the number of hosts that can participate in a QJUMP network. For example, a distribute/aggregate service may send requests to 10,000 hosts, but can be certain that fewer than 1,000 will respond. In this case, n can still be set to 1,000 hosts, but all 10,000 hosts can use QJUMP with guarantees. Finally, QJUMP scales with the network speed. On a faster network (e.g. a 40Gb/s edge), the same delay can be maintained for larger n (e.g. 16,000).

Configuring f_i The most complicated parameters to determine are the throughput factors f_i. Fortunately, each value of f_i is easily expressible as a rate-limit (e.g in Mb/s) which makes choosing values relatively intuitive (see §6 for examples). The best value for f_i depends on the desired latency distribution and the workload. The simplest configuration is to use only two QJUMP levels: (*i*) guaranteed latency (f_1 = 1) and (*ii*) maximum throughput (f_7 = n). Alternatively, a set of f_i values can be configured for a known application mix or for a known traffic distribution.

1. Known Application Mix Datacenter application mixes are often known, or information on application profiles can be obtained from users [4, 20, 21]. If application latency and throughput requirements can be estimated or measured, the QJUMP levels can be set to accommodate their needs.[4] In practice, simple benchmarks at different rate limits make it easy to characterize an application. We show an example in §6.5.

2. Known Traffic Distribution While the application mix in large datacenters can be complex, monitoring infrastructure supplies aggregate traffic statistics. An approximate distribution of flow sizes is often available [1, 5, 16]. For a known flow size distribution, f_i values can be configured to partition the traffic according to a desired latency variance vs. throughput distribution. We applied this method on a flow size CDF using a simple spreadsheet. This worked well in our experiments and simulations in §6.4.

6 Evaluation

We evaluate QJUMP both on a small deployment and in simulation. Our evaluation shows that QJUMP:

1. resolves network interference for a collection of real-world datacenter applications (§6.2);
2. outperforms Ethernet Flow Control (802.3x), ECN and DCTCP in our deployment (§6.3);
3. provides excellent flow completion times, close to or better than DCTCP [1] and pFabric [3] (§6.4);
4. is easily configurable, illustrated by examples of methods to determine QJUMP parameters (§6.5).

6.1 Experimental setup

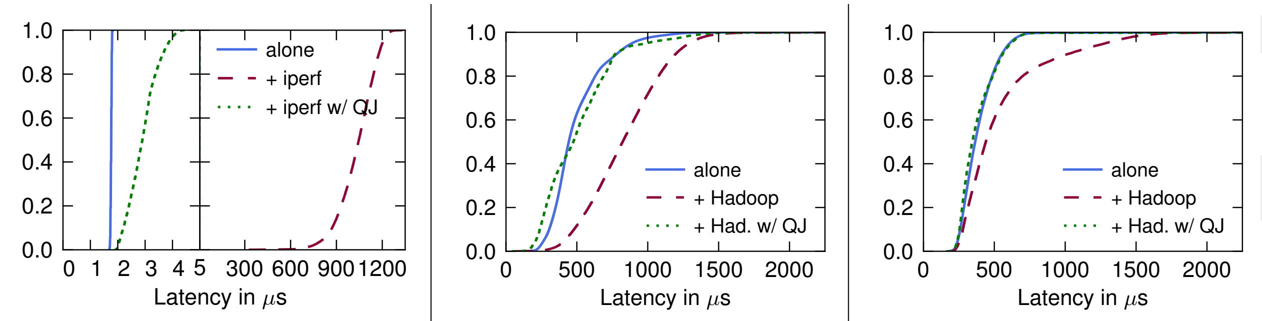
Our physical test-bed comprises an otherwise idle, 12 node cluster of recent AMD Opteron and Intel Xeon-based machines running Ubuntu 14.04 with Linux kernel 3.4.55. Each machine has one two-port 10Gb/s NIC installed. Our network is comprised of four Arista DCS-7124fx switches arranged as per Figure 4. We use `ptpd` v2.1.0 and memcached v1.4.14. We generate load for memcached using `memaslap` from `libmemcached` v1.0.15 running a binary protocol, mixed GET/SET workload of 1 KB requests in TCP mode with 128 concurrent requests. The Naiad experiments use v0.2.3 and the barrier-sync microbenchmark was supplied by the Naiad authors. Hadoop 2.0.0-mr1-cdh4.5.1 is deployed on eight of our twelve nodes, with the HDFS data in `tmpfs` and the replication factor set to six.[5] The Hadoop workload is a natural join between two uniformly randomly generated 512 MB data sets (39M rows each), which produces an output of 29 GB (1.5B rows).

6.2 QJUMP Resolves Network Interference

Our experiments in §2 showed that network interference degrades application performance. We now repeat those experiments with QJUMP enabled and show that QJUMP mitigates the network interference, resulting in near ideal

[4] There may be more applications than QJUMP levels. In this case, some levels will need to be shared between applications.

[5] This simulates the traffic a larger Hadoop cluster would generate.



(a) CDF of `ping` packet latency across a switch. Note the change in x-axis scale.

(b) QJUMP reduces memcached request latency: CDF of 9 million samples.

(c) QJ fixes Naiad barrier synchronization latency: CDF over 10k samples.

Figure 3: Application-level latency experiments: QJUMP (green, dotted line) mitigates the latency tails from Figure 1.

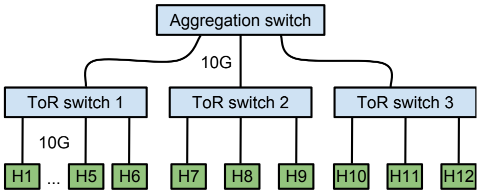


Figure 4: Network topology of our test-bed.

performance. We also show that in a realistic multi-application setting, QJUMP both resolves network interference and outperforms other readily available systems. We execute these experiments on the topology shown in Figure 4.

Low Latency RPC vs. Bulk Transfer Remote Procedure Calls (RPCs) and bulk data transfers represent extreme ends of the latency-bandwidth spectrum. QJUMP resolves network interference at these extremes. As in §2.1, we emulate RPCs and bulk data transfers using `ping` and `iperf` respectively. We measure in-network latency for the `ping` traffic directly using a high resolution Endace DAG capture card and two optical taps on either side of a switch. This verifies that queueing latency at switches is reduced by QJUMP. By setting `ping` to the highest QJUMP level ($f_7 = 1$), we reduce its packets' latency at the switch by over 300× (Figure 3a). The small difference between idle switch latency (1.6μs) and QJUMP latency (2–4μs) arises due a small on-chip FIFO through which the switch must process packets in-order. The switch processing delay, represented as ε in Equation 2, is thus no more than 4μs for each of our switches.

Memcached QJUMP resolves network interference experienced by memcached sharing a network with Hadoop. We show this by repeating the memcached experiments in §2.2. In this experiment, memcached is configured at an intermediate QJUMP level, rate-limited to 5Gb/s (above memcached's maximum throughput; see §6.5). Figure 3b shows the distribution (CDF) of memcached request latencies when running on an idle network, a shared network, and a shared network with QJUMP enabled. With QJUMP enabled, the request latencies are close to the ideal. The median latency improves from 824μs in the shared case to 476μs, a nearly 2× improvement.[6]

Naiad Barrier Synchronization QJUMP also resolves network interference experienced by Naiad [24], a distributed system for executing data parallel dataflow programs. Figure 3c shows the latency distribution of a four-way barrier synchronization in Naiad. On an idle network network, 90% of synchronizations take no more than 600μs. With interfering traffic from Hadoop, this value doubles to 1.2ms. When QJUMP is enabled, however, the distribution closely tracks the uncontended baseline distribution, despite sharing the network with Hadoop. QJUMP here offers a 2–5× improvement in application-level latency.

Multi-application Environment In real-world datacenters, a range of applications with different latency and bandwidth requirements share same infrastructure. QJUMP effectively resolves network interference in these shared, multi-application environments. We consider a datacenter setup with three different applications: `ptpd` for time synchronization, memcached for serving small objects and Hadoop for batch data analysis. Since resolving on-host interference is outside the scope of our work, we avoid sharing hosts between applications in these experiments and share only the network infrastructure.

Figure 5 *(top)* shows a timeline of average request latencies (over a 1ms window) for memcached and synchronization offsets for `ptpd`, each running alone on an otherwise idle network. Figure 5 *(middle)*, shows the two applications sharing the network with Hadoop. In this

[6] The distributions for the idle network and the QJUMP case do not completely agree due of randomness in the load generated.

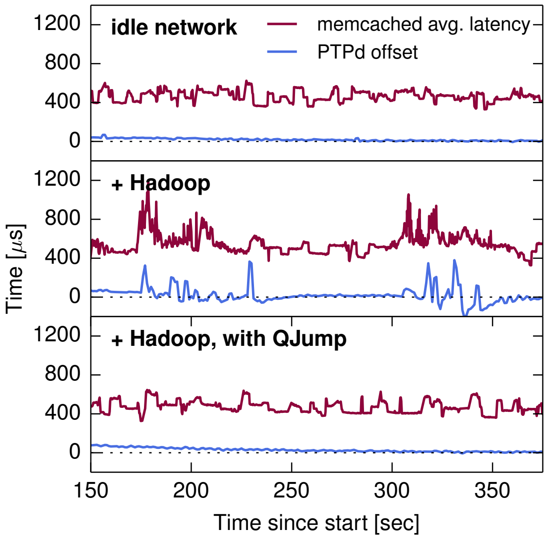


Figure 5: PTPd and memcached in isolation *(top)*, with interfering traffic from Hadoop *(middle)* and with the interference mitigated by QJUMP *(bottom)*.

case, average latencies increase for both applications and visible latency spikes (corresponding to Hadoop's shuffle phases) emerge. With QJUMP deployed, we assign ptpd to $f_7 = 1$, Hadoop to $f_0 = n = 12$ and memcached to $T_5 = 5\text{Gb/s} \implies f_5 = 6$ (see §6.5). The three applications now co-exist without interference (Figure 5 *(bottom)*). Hadoop's performance is not noticeably affected by QJUMP, as we will further show in §6.3.

Distributed Atomic Commit One of QJUMP's unique features is its guaranteed latency level (described in §3.1). Bounded latency enables interesting new designs for datacenter coordination software such as SDN control planes, fast failure detection and distributed consensus systems. To demonstrate the usefulness of QJUMP's bounded latency level, we built a simple distributed two-phase atomic-commit (2PC) application.

The application communicates over TCP or over UDP with explicit acknowledgements and retransmissions. Since QJUMP offers reliable delivery, the coordinator can send its messages by UDP broadcast when QJUMP is enabled. This optimization yields a $\approx$30% throughput improvement over both TCP and UDP.

In Figure 6, we show the request rate for one coordinator and seven servers as a function of network interference. Interference is created with two traffic generators: on that generates a constant 10Gb/s of UDP traffic and another that sends fixed-size bursts followed by a 25ms pause. We report interference as the ratio of the burst size to the internal switch buffer size. Beyond a ratio of 200%, permanent queues build up in the switch. At this point the impact of retransmissions degrades throughput of the UDP and TCP implementations to 20% of the 10,000 requests/sec observed on an idle network. By contrast, the UDP-over-QJUMP implementation does not degrade as its messages "jump the queue". At high interference ratios ($>$200%), two-phase commit over QJUMP achieves 6.5× the throughput of standard TCP or UDP. Furthermore, QJUMP's reliable delivery and low latency enable very aggressive timeouts to be used for failure detection. Our 2PC system detects component failure within two network epochs ($\approx$40μs on our network), far faster than typical failure detection timeouts (e.g. 150 ms in RAMCloud [26, §4.6]).

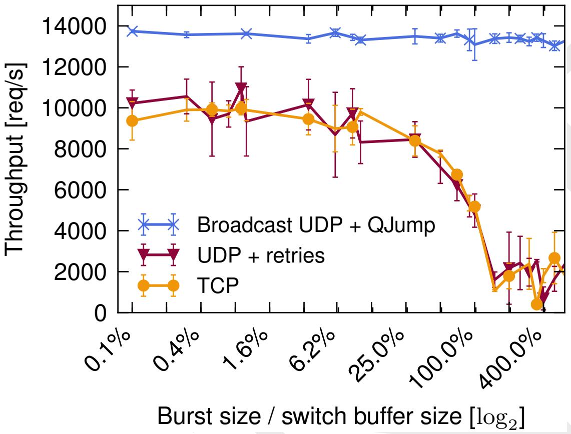


Figure 6: QJUMP offers constant two-phase commit throughput even at high levels of network interference.

6.3 QJUMP Outperforms Alternatives

Several readily deployable congestion control schemes exist, including Ethernet Flow Control (802.1x), Explicit Congestion Notifications (ECN) and Data Center TCP (DCTCP). We repeat the multi-application experiment described in §6.2 and show that QJUMP exhibits better interference control than other schemes.

Since interference is transient in these experiments, we measure the degree to which it affects applications using the root mean square (RMS) of each application-specific metric.[7] For Hadoop, PTPd and memcached, the metrics are job runtime, synchronization offset and request latency, respectively. Figure 7 shows six cases: an ideal case, a contended case and one for each of the four schemes used to mitigate network interference. All cases are normalized to the ideal case, which has each application running alone on an idle network. We discuss each result in turn.

Ethernet Flow Control Like QJUMP, Ethernet Flow Control is a data link layer congestion control mechanism. Hosts and switches issue special *pause* messages

[7] RMS is a statistical measure of the *magnitude* of a varying quantity [6, p. 64]. This is not the same as the root mean square error (RMSE), which measures prediction accuracy.

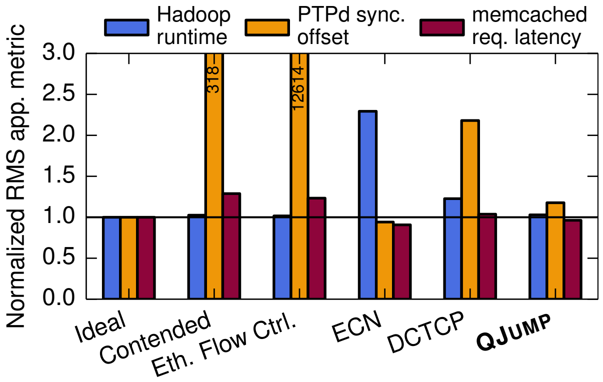


Figure 7: QJUMP comes closest to ideal performance for all of Hadoop, PTPd and memcached.

when their queues are nearly full, alerting senders to slow down. Figure 7 shows that Ethernet Flow Control has a limited positive influence on memcached, but increases the RMS for PTPd. Hadoop's performance remains unaffected.

Early Congestion Notification (ECN) ECN is a network layer mechanism in which switches indicate queueing to end hosts by marking TCP packets. Our Arista 7050 switch implements ECN with Weighted Random Early Detection (WRED). The effectiveness of WRED depends on an administrator correctly configuring upper and lower *marking thresholds*. We investigated ten different marking thresholds pairs, ranging between [5, 10] and [2560, 5120] ([upper, lower], in packets). None of these settings achieve ideal performance for all three applications, but the best compromise was [40, 80]. With this configuration, ECN very effectively resolves the interference experienced by PTPd and memcached. However, this comes at the expense of increased Hadoop runtimes.

Datacenter TCP (DCTCP) DCTCP uses the rate at which ECN markings are received to build an estimate of network congestion. It applies this to a new TCP congestion avoidance algorithm to achieve lower queueing delays [1]. We configured DCTCP with the recommended ECN marking thresholds of [65, 65]. Figure 7 shows that DCTCP reduces the variance in PTPd synchronization and memcached latency compared to the contended case. However, this comes at an increase in Hadoop job runtimes, as Hadoop's bulk data transfers are affected by DCTCP's congestion avoidance.

QJUMP Figure 7 shows that QJUMP achieves the best results. The variance in Hadoop, PTPd and memcached performance is close to (Hadoop, PTPd) or slightly better than (memcached) in the uncontended ideal case.

6.4 QJUMP Improves Flow Completion

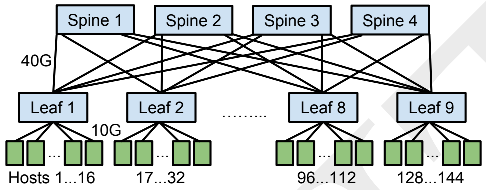


Figure 8: 144 node leaf-spine topology used for simulation experiments.

In addition to resolving network interference, QJUMP also provides excellent overall average and 99th percentile flow completion times (FCTs). Although QJUMP specifically optimizes tail latencies for small flows (at the expense of larger flows), doing so imposes a natural order on the network. This results in a surprisingly good overall network schedule with a generally positive impact on flow completion times.

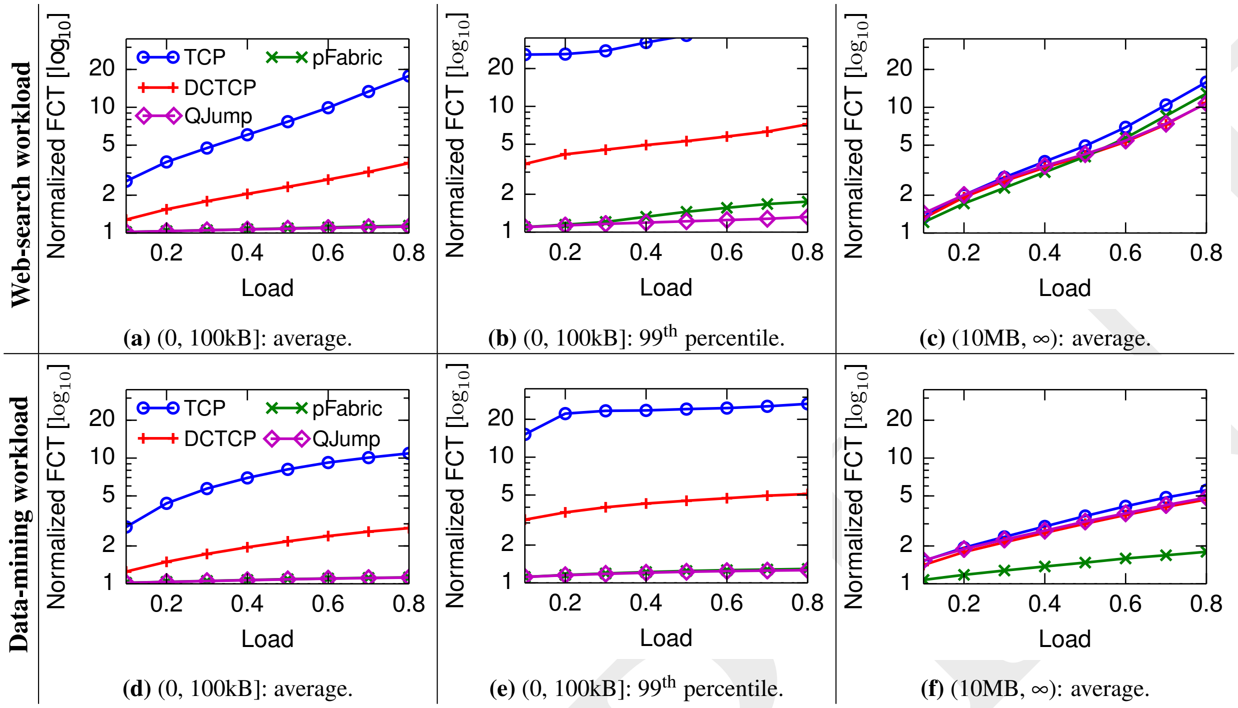
The pFabric architecture has been shown to schedule flows close to optimally [3]. Therefore, we compare QJUMP against pFabric to assess the quality of the network schedule it imposes. pFabric "is a clean-slate design [that] requires modifications both at the switches and the end-hosts" [3, §1] and is therefore only available in simulation. By contrast, QJUMP is far simpler and readily deployable, but applies rigid, global rate limits.

We compare QJUMP against a TCP baseline, DCTCP and pFabric by extending an `ns2` simulation provided by the authors of pFabric. This replicates the leaf-spine network topology used to evaluate pFabric (see Figure 8). We also run the same workloads derived from web search [1, §2.2] and data mining [16, §3.1] clusters in Microsoft datacenters, and show matching graphs in Figure 9.[8] As in pFabric, we normalize flows to their ideal flow completion time on an idle network.

Figure 9 reports the average and 99th percentile normalized FCTs for small flows (0kB, 100kB] and the average FCTs for large flows (10MB, ∞). For both workloads, QJUMP is configured with $P = 9$kB, $n = 144$, and $\{f_0 ... f_7\} = \{144, 100, 20, 10, 5, 3, 2, 1\}$. We chose this configuration based on the distribution of flow sizes in the web search workload. However, in practice it worked well for both workloads.

Despite its simplicity, QJUMP performs very well. As expected, it works best on short flows: on both workloads, QJUMP achieves average and 99th percentile FCTs close to or better than pFabric's. On the web-search workload, QJUMP beats pFabric by a margin of up to 32% at the 99th percentile (Fig. 9b). For larger flows, the results are mixed. On the web search workload, QJUMP

[8] An extended set of graphs for both of the workloads is available at *http://www.cl.cam.ac.uk/netos/qjump/sims.html*.



(a) (0, 100kB]: average.

(b) (0, 100kB]: 99th percentile.

(c) (10MB, ∞): average.

(d) (0, 100kB]: average.

(e) (0, 100kB]: 99th percentile.

(f) (10MB, ∞): average.

Figure 9: Normalized flow completion times in a 144-host simulation (1 is ideal): QJUMP outperforms TCP, DCTCP and pFabric for small flows. N.B.: log-scale y-axis; QJUMP and pFabric overlap in (a), (d) and (e).

outperforms pFabric by up to 20% at high load, but loses to pFabric by 15% at low load (Fig. 9c). On the data mining workload, QJUMP's average FCTs are between 30% and 63% worse than pFabric's (Fig. 9f).

In the data-mining workload, 85% of all *flows* transfer fewer than 100kB, but over 80% of the *bytes* are transferred in flows of greater than 100MB (less than 15% of the total flows). QJUMP's short epoch intervals cannot sense the difference between large flows, so it does not apply any rate-limiting (scheduling) to them. This results in sub-optimal behavior. A combined approach where QJUMP regulates interactions between large flows and small flows, while DCTCP regulates the interactions between different large flows might improve this.

6.5 QJUMP Configuration

As described in §5, QJUMP levels can be determined in several ways. One approach is to tune the levels to a specific mix of applications. For some applications, it is clear that they perform best at guaranteed latency (e.g. ptpd at $f_7 = 1$) or high rate (e.g. Hadoop at $f_0 = n$). For others, their performance at different throughput factors is less straightforward. Memcached is an example of such an application. It needs low request latency variance as well as reasonable request throughput. Figure 10 shows memcached's request throughput and latency as a function of rate-limiting. Peak throughput is reached at a rate allocation of around 5Gb/s. At the same point, the request latency also stabilizes. Hence, a rate-limit of 5Gb/s gives the best tradeoff for memcached. This point has the strongest interference control possible without throughput restrictions. To convert this to a throughput factor, we get $f_i = \frac{nT_i}{R}$ by rearranging Equation 2 for f_i. On our test-bed ($n = 12$ at $R =$10Gb/s), $T_i =$5Gb/s yields a throughput factor of $f = 6$. We can therefore choose a QJUMP level for memcached (e.g. f_4) and set it to a throughput factor ≥6.

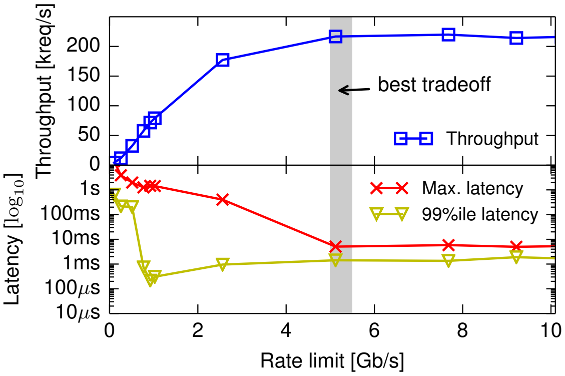


Figure 10: memcached throughput (top) and latency (bottom, $\log_{10}$) as a function of the QJUMP rate limit.

QJUMP offers a bounded latency level at throughput factor f_7. At this level, all packets admitted into the net-

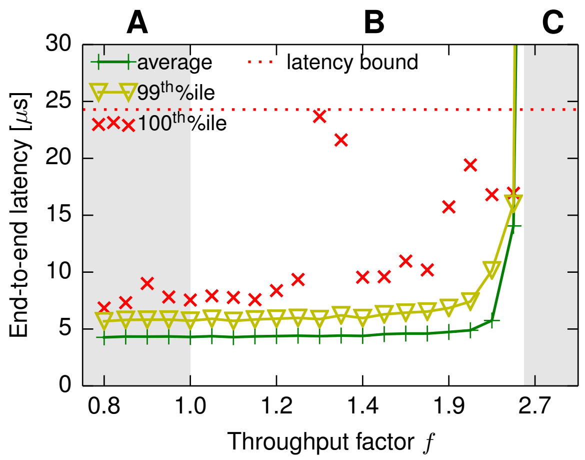


Figure 11: Latency bound validation: 60 host fan-in of f_7 and f_0 traffic; 100 million samples per data point.

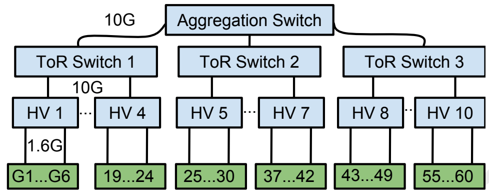


Figure 12: Latency bound validation topology: 10 hypervisors (HV) and 60 guests (G1..60) and 120 apps.

work must reach the destination by the end of the network epoch (§3.1). We now show that our model and the derived configuration perform correctly. To do this, we perform a scale-up emulation using a 60-host virtualized topology running on ten physical machines (see Figure 12). In this topology, each machine runs a "hypervisor" (Linux kernel) with a 10Gb/s uplink to the network. Each hypervisor runs six "guests" (processes) each with a 1.6Gb/s network connection. We provision QJump for the number of guests and run two applications on each guest: (*i*) a coordination service that generates one 256 byte packet per network epoch at the highest QJump level, and (*ii*) a bulk sender that issues 1500 byte packets as fast as possible at the lowest QJump level. All coordination requests are sent to a single destination.

Figure 11 shows the latency distribution of coordination packets as a function of the throughput factor at the highest QJump level, f_7. If the f_7 is set to less than 1.0 (region **A**), the latency bound is met (as we would expect). In region **B**, where f_7 is between 1.0 and 2.7, transient queueing affects some packets—as evident from the 100th percentile outliers—but all requests make it within the latency bound. Beyond $f_7 = 2.7$ (region **C**), permanent queueing occurs.

This experiment offers two further insights about QJump's rate-limiting: (*i*) at throughput factors near 1.0, the latency bound is usually still met, and (*ii*) via rate-limiting, QJump prevents latency-sensitive applications from interfering with their *own* traffic.

7 Related Work

Network congestion in datacenter networks is an active research area. Table 2 compares the properties of recent systems, including those we already compared against in §6.3 and §6.4. We categorize systems as *deployable* if they function on commodity hardware, unmodified transport protocols and unmodified application source code.

Fastpass [29] employs a global arbiter that times the admission of packets into the network and routes them. While Fastpass eliminates in-network queueing, requests for allocation must queue at the centralized arbiter.

EyeQ [22] primarily aims for bandwidth partitioning, although it also reduces latency tails. It, however, requires a full-bisection bandwidth network and a kernel patch in addition to a TC module.

Deadline Aware TCP (D^{2}TCP) [33] extends DCTCP's window adjustment algorithm with the notion of flow deadlines, scheduling flows with earlier deadlines first. Like DCTCP, D^{2}TCP requires switches supporting ECN;[9] it also requires inter-switch coordination, kernel and application modifications.

HULL combines DCTCP's congestion avoidance applied on network links' utilization (rather than queue length) with a special packet-pacing NIC [2]. Its rate-limiting is applied in reaction to ECN-marked packets.

D^3 [35] allocates bandwidth on a first-come-first-serve basis. It requires special switch and NIC hardware and modifies transport protocols.

PDQ uses Earliest Deadline First (EDF) scheduling to prioritize straggler flows, but requires coordination across switches and application changes.

DeTail [37] and pFabric [3] pre-emptively schedule flows using packet forwarding priorities in switches. DeTail also addresses load imbalance caused by poor flow hashing. Flow priorities are explicitly specified by modified applications (DeTail) or computed from the remaining flow duration (pFabric). However, both systems require special switch hardware: pFabric uses very short queues and 64-bit priority tags, and DeTail coordinates flows' rates via special "pause" and "unpause" messages.

SILO [21] employs a similar reasoning to QJump to estimate expected queue lengths. It places VMs according to traffic descriptions to limit queueing and paces hosts using "null" packets.

TDMA Ethernet [34] trades bandwidth for reduced queueing by time dividing network access, but requires invasive kernel changes and centralized coordination.

[9] Only one in five 10Gb/s switches we looked at supports ECN.

		Commodity	Unmodified			Coord.	Flow	Bounded	Imple-
	System	hardware	protocols	OS kernel	apps.	free	deadlines	latency	mented
Deployable	Pause frames	✓	✓	✓	✓	✓	✗	✗	✓‡
	ECN	✓*, ECN	✓	✓	✓	✓	✗	✗	✓‡
	DCTCP [1]	✓*, ECN	✓*	✗	✓	✓	✗	✗	✓‡
	Fastpass [29]	✓	✓	✓, module	✓	✗	✗	✗	✓‡
	EyeQ [22]	✓*, ECN	✓	✗	✓	✗	✗	✗	✓‡
	QJUMP	✓	✓	✓, module	✓	✓	✓*	✓	✓‡
Not deployable	D^2TCP [33]	✓*, ECN	✓*	✗	✗	✗*	✓	✗	✓
	HULL [2]	✗	✓*	✗	✓	✓	✗	✗	✓*
	D^3 [35]	✗	✗	✗	✗	✓	✓	✗	✗*, softw.
	PDQ [17]	✗	✗	✗	✗	✗	✓	✗	✗
	pFabric [3]	✗	✗	✗	✓	✓	✓*	✗	✗
	DeTail [37]	✗	✓	✓	✗	✗*	✗	✗	✗*, softw.
	Silo [21]	✓	✓	✗	✓*	✗*	✓*, SLAs	✗	✓
	TDMA Eth. [34]	✓*	✓*	✗	✓*	✗	✗	✓	✓

Table 2: Comparison of related systems. *with caveats, see text; ‡implementation publicly available.

8 Discussion and Future Work

It would be ideal if applications were automatically classified into QJUMP levels. This requires overcoming a few challenges. First, the rate-limiter needs to be extended to calculate an estimate of instantaneous throughput for each application. Second, applications that exceed their throughput allocation must be moved to a lower QJUMP level, while applications that underutilize their allocation must be lifted to a higher QJUMP level. Third, some applications (e.g. Naiad) have latency-sensitive control traffic as well as throughput-intensive traffic that must be treated separately [19]. We leave this to future work.

9 Conclusion

QJUMP applies QoS-inspired concepts to datacenter applications to mitigate network interference. It offers multiple QJUMP levels with different latency variance vs. throughput tradeoffs, including bounded latency (at low rate) and full utilization (at high latency variance). In an extensive evaluation, we have demonstrated that QJUMP attains near-ideal performance for real applications and good flow completion times in simulations. Source code and data sets are available from *http://goo.gl/q1lpFC*.

Acknowledgements

We would like to thank Alex Ho and John Peach from Arista for arranging 10G switches for us. We would also like to thank Simon Peter, Srinivasan Keshav, Marwan Fayed, Rolf Neugebauer, Tim Harris, Antony Rowstron, Matthew Huxtable, Jeff Mogul and our anonymous reviewers for their valuable feedback. Thanks also go to our shepherd Jeff Dean. This work was supported by a Google Fellowship, EPSRC INTERNET Project EP/H040536/1, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

Appendix

The Parekh-Gallager theorem [27, 28] shows that Weighted Fair Queueing (WFQ) achieves a worst case delay bound given by the equation

$$\textit{end to end delay} \leq \frac{\sigma}{g} + \sum_{i=1}^{K-1} \frac{P}{g_i} + \sum_{i=1}^{K} \frac{P}{r_i}, \qquad (5)$$

where all sources are governed by a leaky bucket abstraction with rate ρ and burst size σ, packets have a maximum size P and pass through K switches. For each switch i, there is a total rate r_i of which each connection (host) receives a rate g_i. g is the minimum of all g_i. It is assumed that $\rho \leq g$, i.e. the network is underutilized.

The final term in the equation adjusts for the difference between PGPS and GPS (Generalized Processor Sharing) for a non-idle network. Since we assume an idle network in our model (3.1), Equation 5 simplifies to

$$\textit{end to end delay} \leq \frac{\sigma}{g} + \sum_{i=1}^{K-1} \frac{P}{g_i} \qquad (6)$$

If we assume that all hosts are given a fair share of the network—i.e. Fair Queueing rather than WFQ—then,

$$g_i = \frac{r_i}{n} \qquad (7)$$

where n is the number of hosts. Therefore the g (the minimum g_i) dominates. Since we assume an idle network, the remaining terms sum to zero. For a maximum burst size $\rho = P$, the equation therefore simplifies to

$$\textit{end to end delay} \leq \frac{P}{g} = n \times \frac{P}{R} \qquad (8)$$

which is equivalent to the equation derived in Equation 1 (§3.1). The Parekh-Gallager theorem does not take into account the switch processing delay ε, since it is negligible compared to the end-to-end delay on the Internet.

References

[1] 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 SIGCOMM* (2010), pp. 63–74.

[2] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of NSDI* (2012), pp. 253–266.

[3] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of SIGCOMM* (2013), pp. 435–446.

[4] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Towards predictable datacenter networks. In *Proceedings of SIGCOMM* (2011), pp. 242–253.

[5] Benson, T., Akella, A., and Maltz, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of IMC* (2010), pp. 267–280.

[6] Bissell, C., and Chapman, D. *Digital Signal Transmission*. Cambridge University Press, 1992.

[7] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. RFC 2475: An architecture for differentiated services, Dec. 1998. Status: Proposed Standard.

[8] Brutlag, J. Speed Matters for Google Web Search. Tech. rep., Google. Available at: http://goo.gl/1qF8xt; accessed 24/09/2014.

[9] Card, S. K., Robertson, G. G., and Mackinlay, J. D. The information visualizer, an information workspace. In *Proceedings of CHI* (1991), pp. 181–186.

[10] Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of WREN* (2009), pp. 73–82.

[11] Corbett, J., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., et al. Spanner: Google's Globally-Distributed Database. In *Proceedings of OSDI* (2012), pp. 251–264.

[12] Crowcroft, J., Hand, S., Mortier, R., Roscoe, T., and Warfield, A. QoS's Downfall: At the bottom, or not at all! In *Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS* (2003).

[13] Dean, J., and Barroso, L. A. The Tail at Scale: Managing Latency Variability in Large-Scale Online Services. *Commun. ACM 56*, 2 (Feb. 2013).

[14] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward Predictable Performance in Software Packet-processing Platforms. In *Proceedings of NSDI* (2012), pp. 141–154.

[15] Duffield, N. G., Goyal, P., Greenberg, A., Mishra, P., Ramakrishnan, K. K., and van der Merwe, J. E. A Flexible Model for Resource Management in Virtual Private Networks. In *Proceedings of SIGCOMM* (1999), pp. 95–108.

[16] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: a scalable and flexible data center network. In *Proceedings of SIGCOMM* (2009), pp. 51–62.

[17] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of SIGCOMM* (2012), pp. 127–138.

[18] IEEE. Standard for local and metropolitan area networks, Virtual Bridged Local Area Networks. *IEEE Std. 802.11Q-2005* (2005).

[19] Isaacs, R. Tuning the performance of Naiad. Part 1: the network. Big Data at SVC blog, http://bit.ly/1gl5Cjk; accessed 25/09/2014.

[20] Jalaparti, V., Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Bridging the tenant-provider gap in cloud services. In *Proceedings of SoCC* (2012), pp. 10:1–10:14.

[21] Jang, K., et al. Silo: Predictable Message Completion Time in the Cloud. Tech. rep., Microsoft Research, 2013. MSR-TR-2013-95.

[22] Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Greenberg, A., and Kim, C. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of NSDI* (2013), pp. 297–311.

[23] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of EuroSys* (2014), pp. 4:1–4:14.

[24] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A Timely Dataflow System. In *Proceedings of SOSP* (2013), pp. 439–455.

[25] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *Proceedings of NSDI* (2013), pp. 385–398.

[26] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast Crash Recovery in RAMCloud. In *Proceedings of SOSP* (2011), pp. 29–41.

[27] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Netw. 1*, 3 (June 1993), 344–357.

[28] Parekh, A. K., and Gallagher, R. G. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Trans. Netw. 2*, 2 (Apr. 1994), 137–150.

[29] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A centralized ”zero-queue” datacenter network. In *Proceedings of SIGCOMM* (2014), pp. 307–318.

[30] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. In *Proceedings of OSDI 14* (Oct. 2014), pp. 1–16.

[31] Solarflare Communications Inc. www.openonload.org, November 2012.

[32] Tang, L., Mars, J., Vachharajani, N., Hundt, R., and Soffa, M.-L. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of ISCA* (2011).

[33] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware Datacenter TCP (D2TCP). *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 115–126.

[34] Vattikonda, B. C., Porter, G., Vahdat, A., and Snoeren, A. C. Practical TDMA for Datacenter Ethernet. In *Proceedings of Eurosys* (2012), pp. 225–238.

[35] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings SIGCOMM* (2011), pp. 50–61.

[36] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M. Bobtail: Avoiding long tails in the cloud. In *Proceedings of NSDI* (2013), pp. 329–342.

[37] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. Detail: Reducing the flow completion time tail in datacenter networks. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 139–150.