# pFabric: Minimal Near-Optimal Datacenter Transport

Mohammad Alizadeh[†‡], Shuang Yang[†], Milad Sharif[†], Sachin Katti[†], Nick McKeown[†], Balaji Prabhakar[†], and Scott Shenker[§]

[†]Stanford University [‡]Insieme Networks [§]U.C. Berkeley / ICSI
{alizade, shyang, msharif, skatti, nickm, balaji}@stanford.edu
shenker@icsi.berkeley.edu

June 2, 2013

**Abstract**

In this paper we present pFabric, a minimalistic datacenter transport design that provides near theoretically optimal flow completion times even at the 99th percentile for short flows, while still minimizing average flow completion time for long flows. Moreover, pFabric delivers this performance with a very simple design that is based on a key conceptual insight: datacenter transport should decouple flow scheduling from rate control. For flow scheduling, packets carry a single priority number set independently by each flow; switches have very small buffers and implement a very simple priority-based scheduling/dropping mechanism. Rate control is also correspondingly simpler; flows start at line rate and throttle back only under high and persistent packet loss. We provide theoretical intuition and show via extensive simulations that the combination of these two simple mechanisms is sufficient to provide near-optimal performance.

**Categories and Subject Descriptors:** C.2.1 [Computer-Communication Networks]: Network Architecture and Design
**General Terms:** Design, Performance
**Keywords:** Datacenter network, Packet transport, Flow scheduling

## 1 Introduction

Datacenter workloads impose unique and stringent requirements on the transport fabric. Interactive soft real-time workloads such as the ones seen in search, social networking, and retail generate a large number of small requests and responses across the datacenter that are stitched together to perform a user-requested computation (e.g., delivering search results). These applications demand low latency for each of the short request/response flows, since user-perceived performance is dictated by how quickly responses to all (or a large fraction of) the requests are collected and delivered back to the user. However in currently deployed TCP-based fabrics, the latency for these short flows is poor — flow completion times (FCT) can be as high as tens of milliseconds while in theory these flows could complete in 10-20 microseconds. The reason is that these flows often get queued up behind bursts of packets from large flows of co-existing workloads (such as backup, replication, data mining, etc) which significantly increases their completion times.

Motivated by this observation, recent research has proposed new datacenter transport designs that, broadly speaking, use rate control to reduce FCT for short flows. One line of work [3, 4] improves FCT by keeping queues near empty through a variety of mechanisms (adaptive congestion control, ECN-based feedback, pacing, etc) so that latency-sensitive flows see small buffers and consequently small latencies. These *implicit* techniques generally improve FCT for short flows but they can never precisely determine the right flow rates to optimally schedule flows. A second line of work [20, 13] *explicitly* computes and assigns rates from the network to each flow in order to schedule the flows based on their sizes or deadlines. This approach can potentially provide very good performance, but it is rather complex and challenging to implement in practice because accurately computing rates requires detailed flow state at switches and also coordination among switches to identify the bottleneck for each flow and avoid under-utilization (§2).

Our goal in this paper is to design the simplest possible datacenter transport scheme that provides near-optimal flow completion times, even at the $99^{th}$ percentile for latency-sensitive short flows. To this end, we present pFabric,[1] a minimalistic datacenter fabric whose entire design consists of the following:

- End-hosts put a single number in the header of every packet that encodes its priority (e.g., the flow's remaining size, deadline). The priority is set independently by each flow and no coordination is required across flows or hosts to compute it.
- Switches are simple; they have very small buffers (e.g., 36KB per port in our evaluation) and decide which packets to accept into the buffer and which ones to schedule strictly according to the packet's priority number. When a new packet arrives and the buffer is full, if the incoming packet has lower priority than all buffered packets, it is dropped. Else, the lowest priority packet in the buffer is dropped and replaced with the incoming packet. When transmitting, the switch sends the packet with the highest priority. Thus each switch operates independently in a greedy and local fashion.
- Rate control is minimal; all flows start at line-rate and throttle their sending rate only if they see high and persistent loss. Thus rate control is lazy and easy to implement.

pFabric thus requires no flow state or complex rate calculations at the switches, no large switch buffers, no explicit network feedback, and no sophisticated congestion control mechanisms at the end-host. pFabric is a clean-slate design; it requires modifications both at the switches and the end-hosts. We also present a preliminary design for deploying pFabric using existing switches, but a full design for incremental deployment is beyond the scope of this paper.

The key conceptual insight behind our design is the observation that rate control is a poor and ineffective technique for flow scheduling and the mechanisms for the two should be decoupled and designed independently. In pFabric, the priority-based packet scheduling and dropping mechanisms at each switch ensure that it schedules flows in order of their priorities. Further, the local and greedy decisions made by each switch lead to an approximately optimal flow scheduling decision across the entire fabric (§4.3). Once flow scheduling is handled, rate control's only goal is to avoid persistently high packet drop rates. Hence, the rate control design gets correspondingly simpler: start at line rate and throttle only if bandwidth is being wasted due to excessive drops.

We evaluate our design with detailed packet-level simulations in ns2 [14] using two widely used datacenter workloads: one that mimics a web application workload [3] and one that mimics a typical data mining workload [11]. We compare pFabric with four schemes: an ideal scheme which is theoretically the best one could do, the state-of-the-art approach for datacenter transport, PDQ [13], as well as DCTCP [3] and TCP. We find that:

- pFabric achieves near-optimal flow completion times. Further, pFabric delivers this not just at the mean, but also at the $99^{th}$ percentile for short flows at loads as high as 80% of the network fabric capacity. pFabric reduces the FCT for short flows compared to PDQ and DCTCP by more than 40% and 2.5–4× respectively at the mean, and more than 1.5–3× and 3–4× respectively at the 99th percentile.
- With deadline driven workloads, pFabric can support a much larger number of flows with deadlines as well as much tighter deadlines compared to PDQ. For instance, even for deadlines where the slack with respect to the lowest possible FCT is only 25%, pFabric meets the deadline for 99% of the flows (about 2× more than PDQ) at 60% network load.
- If the network designer has detailed knowledge of the flow size distribution in advance and carefully tunes parameters such as the flow size thresholds for each priority queue, minimum buffer per priority queue, etc pFabric can be approximated using existing priority queues in commodity switches. This approach provides good performance too, but we find that it is rather brittle and sensitive to several parameters that change in a datacenter due to flow and user dynamics.

[1]pFabric was first introduced in an earlier paper [5] which sketched a preliminary design and initial simulation results.

## 2 Related Work

Motivated by the shortcomings of TCP, a number of new datacenter transport designs have been proposed in recent years. We briefly contrast our work with the most relevant prior work. As discussed earlier, broadly speaking, the previous efforts all use rate control to reduce flow completion time.

**Implicit rate control:** DCTCP [3] and HULL [4] try to keep the fabric queues small or empty by employing an adaptive congestion control algorithm based on ECN and other mechanisms such as operating the network at slightly less than 100% utilization, packet pacing, etc to appropriately throttle long elephant flows. Consequently, the latency-sensitive flows see small buffers and latencies. $D^2$TCP [17], a recently proposed extension to DCTCP, adds deadline-awareness to DCTCP by modulating the window size based on both deadline information and the extent of congestion. While these schemes generally improve latency, they are fundamentally constrained because they can never precisely estimate the right flow rates to use so as to schedule flows to minimize FCT while ensuring that the network is fully utilized. Furthermore, due to the bursty nature of traffic, keeping network queues empty is challenging and requires carefully designed rate control and hardware packet pacing at the end-hosts and trading off network utilization [4].

**Explicit rate control:** Having recognized the above limitations, subsequent work explicitly assigns a sending rate to each flow in order to schedule flows based on some notion of urgency. The assigned rates are typically computed in the network based on flow deadlines or their estimated completion time. $D^3$ [20] first proposed using deadline information in conjunction with explicit rate control to associate rates to flows. $D^3$ allocates bandwidth on a greedy first-come-first-served basis and does not allow preemptions and has thus been shown to lead to sub-optimal flow scheduling since a near-deadline flow can be blocked waiting for a far-deadline flow that arrived earlier [13].

The most closely related work to pFabric and in fact the state-of-the-art approach in this space is PDQ [13]. PDQ was the first to point out that minimizing FCTs requires preemptive flow scheduling and attempts to approximate the same ideal flow scheduling algorithm as pFabric to minimize average FCT or missed deadlines (§3). However, like $D^3$, PDQ's flow scheduling mechanism is also based on switches assigning rates to individual flows using explicit rate control. In PDQ, on packet departure, the sender attaches a scheduling header to the packet that contains several state variables including the flow's deadline, its expected transmission time, and its current status such as its sending rate and round-trip-time. Each switch then maintains this state for some number of outstanding flows and uses it to decide how much bandwidth to allocate to each flow and which flows to "pause".

PDQ provides good performance but is quite challenging and complex to implement in practice. Since the switches on a flow's path essentially need to agree on the rate that is to be assigned to the flow, PDQ needs to pass around state regarding a flow's rate and which switch (if any) has paused the flow. Further, since switches need to be aware of the active flows passing through them, in PDQ, every flow must begin with a SYN and terminate with a FIN so that switches can perform the required book-keeping. This one extra round-trip of latency on every flow may not be acceptable because most latency sensitive flows in datacenters are small enough to complete in just one RTT.[2] Thus, requiring the network to explicitly and efficiently assign a rate to each flow requires detailed flow state (size, deadline, desired rate, current rate, round-trip time, etc) at switches and also coordination among switches to identify the bottleneck for each flow and avoid under-utilization. This is a major burden, both in terms of communication overhead and requisite state at switches, particularly in the highly dynamic datacenter environment where flows arrive and depart at high rates and the majority of flows last only a few RTTs [11, 6].

**Load balancing:** Finally, there are a number of proposals on efficient load balancing techniques for datacenter fabrics [2, 16, 21, 10]. Better load balancing of course reduces hotspots and thus helps reduce flow completion time, however the techniques and goals are orthogonal and complementary to pFabric.

## 3 Conceptual Model

Our conceptual viewpoint in designing our flow scheduling technique is to abstract out the entire fabric as one giant switch. Specifically, the datacenter fabric typically consists of two or three tiers of switches

[2]In measurements from a production datacenter of a large cloud provider, more than 50% of the flows were observed to be less than 1KB [11] — just a single packet.

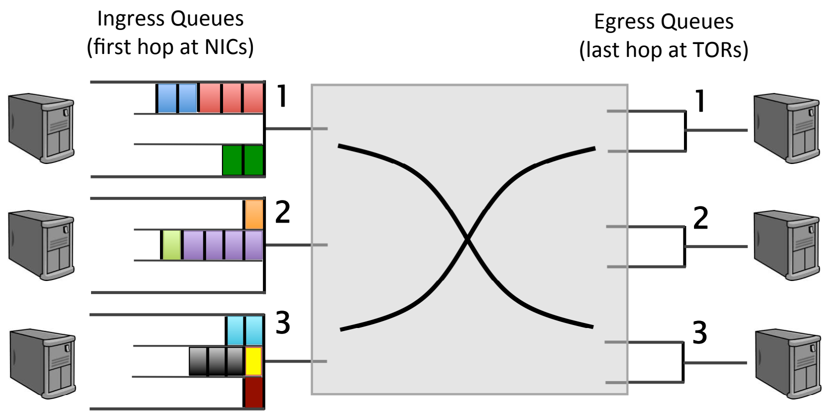


Figure 1: Conceptual view of flow scheduling over a datacenter fabric.

in a Fat-tree or Clos topology [1, 11]. Instead of focusing on the individual switches, the whole fabric can be abstracted as one giant switch that interconnects the servers as shown in Figure 1. The ingress queues into the fabric switch are at the NICs and the egress queues out of the fabric switch are at the last-hop TOR switches. Each ingress port (source NIC) has some flows destined to various egress ports. It is convenient to view these as organized in virtual output queues at the ingress as shown in Figure 1. For example, the red and blue flows at ingress 1 are destined to egress 1, while the green flow is destined to egress 3.

In this context, transport over the datacenter fabric can essentially be thought of as scheduling flows over the backplane of a giant switch. The problem is to find the best schedule to minimize the average FCT (or maximize the number of deadlines met). Since datacenter workloads are dominated by large numbers of short flows, minimizing average FCT will ensure that the short, high-priority flows see very low latency.

**Optimal flow scheduling:** The optimal algorithm for minimizing average FCT when scheduling over a single link is the *Shortest Remaining Processing Time (SRPT)* policy which always schedules the flow that has the least work remaining. However, we are not scheduling over a single link but rather over an entire fabric with a set of links connecting the ingress and egress queues. Unfortunately, a simple universally optimal policy does not exist for simultaneously scheduling multiple links. In fact, even under the simplifying assumption that the fabric core can sustain 100% throughput and that only the ingress and egress access links are potential bottlenecks, the scheduling problem for minimizing the average FCT is equivalent to the NP-hard *sum-multicoloring problem* [8]. Fortunately, a simple greedy algorithm is theoretically guaranteed to provide near-ideal performance. This *Ideal* algorithm schedules flows across the fabric in non-decreasing order of the *remaining flow size* and in a maximal manner such that at any time a flow is blocked if and only if either its ingress port or its egress port is busy serving a different flow with less data remaining. The pseudo code is provided in Algorithm 1. This algorithm has been theoretically proven to provide at least a 2-approximation to the optimal average FCT [8]. In practice we find that the actual performance is even closer to optimal (§5). The takeaway is that *the greedy scheduler in Algorithm 1 that prioritizes small flows over large flows end-to-end across the fabric can provide near-ideal average FCT*.

It is important to note that the Ideal algorithm is not plagued by the inefficiencies that inevitably occur in an actual datacenter transport design. It does not have rate control dynamics, buffering (and its associate delays), packet drops, retransmissions, or inefficiency due to imperfect load-balancing. It only captures one thing: the (best-case) delays associated with flows contending for bandwidth at the ingress and egress fabric ports. Consequently, the performance of this algorithm for a given workload serves as benchmark to evaluate *any* scheme that aims to minimize flow completion times. The key contribution of this paper is to show that a very simple distributed transport design can approximate the performance of the Ideal algorithm with remarkable fidelity.

**Remark 1.** For simplicity, the above discussion assumed that all the edge links run at the same speed, though the Ideal algorithm can easily be generalized. See Hong *et al.* [13] for more details.

**Algorithm 1** Ideal flow scheduling algorithm.

**Input:** $\mathcal{F}$ = List of active flows with their ingress and egress port and remaining size. The algorithm is run each time $\mathcal{F}$ changes (a flow arrives or departs).

**Output:** $S$ = Set of flows to schedule (at this time).

```
1: $S \leftarrow \emptyset$
2: $ingressBusy[1..N] \leftarrow FALSE$
3: $egressBusy[1..N] \leftarrow FALSE$
4: for each flow $f \in \mathcal{F}$, in increasing order of remaining size do
5:    if $ingressBusy[f.ingressPort] == FALSE$ and
      $egressBusy[f.egressPort] == FALSE$ then
6:       $S.insert(f)$
7:       $ingressBusy[f.ingressPort] \leftarrow TRUE$
8:       $egressBusy[f.egressPort] \leftarrow TRUE$
9:    end if
10: end for
11: return $S$.
```

## 4 Design

pFabric's key design insight is a principled decoupling of flow scheduling from rate control. This leads to a simple switch-based technique that takes care of flow scheduling and consequently also simplifies rate control. In this section we describe pFabric's switch and rate controller designs. We explain why pFabric's simple mechanisms are sufficient for near-ideal flow scheduling and discuss some practical aspects regarding its implementation.

**Packet priorities:** In pFabric, each packet carries a single number in its header that encodes its priority. The packet priority can represent different things depending on the scheduling objective. For instance, to approximate the Ideal algorithm (Algorithm 1) and minimize average FCT (our main focus in this paper), we would ideally set the priority to be the remaining flow size when the packet is transmitted. For traffic with deadlines, to maximize the number of deadlines met, we would set the priority to be the deadline itself quantized in some unit of time. Other simplifications such as using absolute flow size instead of remaining flow size are also possible (§4.4). Similar to prior work [20, 13], we assume that the required information (e.g., flow size or deadline) is available at the *transport layer* which then sets the packet priorities.

### 4.1 Switch Design

The pFabric switch uses two simple and local mechanisms:

- **Priority scheduling:** Whenever a port is idle, the packet with the highest priority buffered at the port is dequeued and sent out.
- **Priority dropping:** Whenever a packet arrives to a port with a full buffer, if it has priority less than or equal to the lowest priority packet in the buffer, it is dropped. Otherwise, the packet with the lowest priority is dropped to make room for the new packet.

**Data structures:** The switch maintains two data structures. One is the queue of actual packets which is maintained in RAM. Second, is another queue that mirrors the packet queue, but only holds packet *metadata*: a flow-id (5-tuple or a hash of the 5-tuple) and the packet's priority number. This is maintained in flops so that we can get fast simultaneous access. pFabric switches have very small queues; typically less than two bandwidth-delay products (~36KB or 24 full-sized packets in our simulations). Traditionally, datacenter switches use nearly 10–30× more buffering per port.

**Dequeue:** For dequeueing, we first find the highest priority packet by using a binary tree of comparators that operate hierarchically on the metadata queue on the priority field. If there are $N$ packets, this operation takes $\log_2(N)$ cycles. At this point, we could simply send this highest priority packet out, however this can lead to starvation for some packets when a flow's priority increases over time. To see

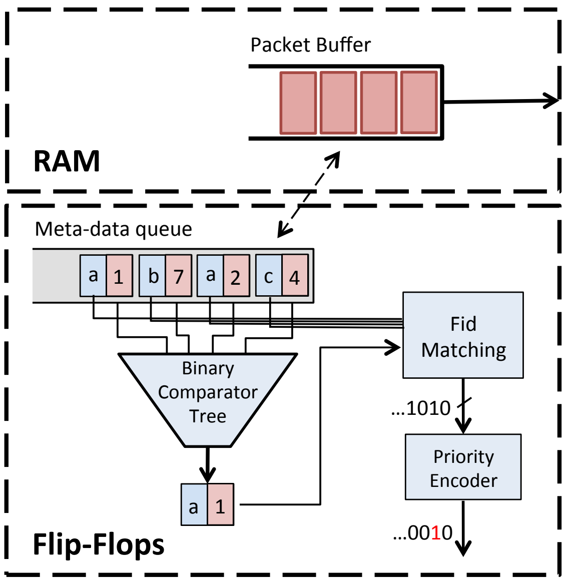


Figure 2: For dequeueing, the switch finds the earliest packet from the flow with the highest priority and sends it out. In the above example, even though the last packet $(\mathbf{a}, \mathbf{1})$ has the highest priority, the second packet in the queue which belongs to the same flow (a) is sent out because it arrived earlier.

how, assume the priority is set to be the remaining flow size and consider the flow to which the highest priority packet belongs. Since packets that are transmitted earlier have lower priority than packets that are transmitted later (because they have relatively higher remaining flow sizes in their priority fields), if the flow has multiple packets waiting in the queue, the highest priority packet among them is likely to have arrived later than the others. If we send out packets purely in order of their priority, then this can lead to situations where packets that arrived earlier might never get serviced since more packets from that flow keep arriving.

To tackle this problem, we implement a technique we christen *starvation prevention* where we dequeue the earliest packet from the flow that has the highest priority packet in the queue. Since packets are queued in the order they arrive, that is simply the earliest packet in the queue that has the same flow-id as the packet with the highest priority. Hence in the second step we perform a parallelized bitwise compare on this flow-id for all the packets in the meta-data queue. The output of this compare operation is a bit-map with a 1 wherever there is a match and 0 otherwise. We pick the packet corresponding to the earliest 1 in the bit vector by using a priority encoder and transmit it. Figure 2 demonstrates the dequeuing algorithm as discussed above.

**Enqueue:** For enqueuing, if the queue is not full, the packet is just added to the end of the queue and the metadata queue is updated. If the queue is full, we use a similar binary tree of comparators structure as in the dequeuing operation above, but this time to find the packet with the lowest priority. That packet is dropped from both the packet and metadata queues and the new packet is added to the end of the queue.

## 4.2 Rate Control Design

What about rate control? If the fabric schedules flows as discussed above, the need for rate control is minimal. In particular, we do not need rate control to prevent spurious packet drops due to bursts, as can occur for example in Incast [18] scenarios. Such events only impact the *lowest* priority packets at the time which can quickly be retransmitted without impacting performance (see §4.3). Further, we do not need to worry about keeping queue occupancies small to control queueing latency. Since packets are scheduled based on priority, even if large queues do form in the fabric, there would be no impact on the latency for high-priority traffic.

However, there is one corner case where a limited form of rate control is necessary. Specifically, whenever a packet traverses multiple hops only to be dropped at a downstream link some bandwidth is wasted on the upstream links that could have been used to transmit other packets. This is especially problematic when the load is high and multiple elephant flows collide at a downstream link. For example, if two elephant flows sending at line rate collide at a last-hop access link, half the bandwidth they consume on the upstream links is wasted. If such high loss rates persist, it would eventually lead to congestion collapse in the fabric. Note that packet drops at the ingress (the source NICs) are not an issue since they do not waste any bandwidth in the fabric.

We use the above insight to design an extremely simple rate control that we implement by taking an existing TCP implementation and throwing away several mechanisms from it. We describe the design by walking the reader through the lifetime of a flow:

- Flows start at line rate. Practically, this is accomplished by using an initial window size equal to the bandwidth-delay product (BDP) of the link (12 packets in our simulations).
- We use SACKs and for every packet acknowledgement we do additive increase as in standard TCP.
- There are no fast retransmits, dupACKs or any other such mechanisms. Packet drops are only detected by timeouts, whose value is fixed and small ($3\times$ the fabric RTT, which is around $45\mu$s in our simulations). Upon a timeout, the flow enters into slow start and `ssthresh` is set to half the window size before the timeout occurred.
- If a fixed threshold number of consecutive timeouts occur (5 in our current implementation), it indicates a chronic congestion collapse event. In this case, the flow enters into *probe mode* where it periodically retransmits minimum-sized packets with a one byte payload and re-enters slow-start once it receives an acknowledgement.

This is the entire rate control design. We do not use any sophisticated congestion signals (either implicit such as 3 dupACKs or explicit such as ECN, XCP etc), no complicated control laws (we use additive increase most of the time and just restart from a window of 1 if we see a timeout), nor do we use sophisticated pacing mechanisms at the end host. The only goal is to avoid excessive and persistent packet drops which this simple design accomplishes.

**Remark 2.** Our rate control design uses the minimal set of mechanisms that are actually needed for good performance. One could of course use existing TCP (with all its features) as well and only increase the initial window size and reduce the minimum retransmission timeout ($minRTO$).

## 4.3 Why this Works

Since pFabric dequeues packets according to priority, it achieves ideal flow scheduling as long as at each switch port and at any time one of the highest priority packets that needs to traverse the port is available to be scheduled. Maintaining this invariant is complicated by the fact that, sometimes, buffers overflow and packets must be dropped. However, when a packet is dropped in pFabric, by design, it has the lowest priority among all buffered packets. Hence, even if it were not dropped, its "turn" to be scheduled would not be until at least *all* the other buffered packets have left the switch. (the packet's turn may end up even further in the future if higher priority packets arrive while it is waiting in the queue.) Therefore, a packet can safely be dropped as long as the rate control is aggressive and ensures that it retransmits the packet (or sends a different packet from that flow) before all the existing packets depart the switch. This can easily be achieved if the buffer size is at least one bandwidth-delay product and hence takes more than a RTT to drain, providing the end-host enough time to detect and retransmit dropped packets. Our rate control design which keeps flows at line-rate most of the time is based on this intuition.

## 4.4 Implementation

A prototype implementation of pFabric including the hardware switch and the software end-host stack is beyond the scope of this paper and is part of our future work. Here, we briefly analyze the feasibility of its implementation.

### 4.4.1 Switch implementation

Priority scheduling and dropping are relatively simple to implement using well known and widely used hardware primitives because pFabric switches have very small buffers — typically about two BDPs worth of packets at each port which is less than ∼36KB for a 10Gbps 2-tier datacenter fabric. With a 36KB buffer, in the worst-case of minimum size 64B packets, we have 51.2ns to find the highest/lowest of at most ∼600 numbers, which translate to ∼40 clock cycles for today's switching ASICs. A straight-forward implementation of this using the binary comparator tree discussed in §4.1 requires just 10 ($\log_2(600)$) clock cycles, which still leaves 30 cycles for the flow-id compare operation. This can be done in parallel for all 600 packets, but it is preferable to do it sequentially on smaller blocks to reduce the required gates and power-draw. Assuming a 64 block compare that checks 64 flow-ids at a time (this is easy and commonly implemented in current switches), we require at most 10 clock cycles for all 600 packets. Hence we need a total of 20 clock cycles to figure out which packet to dequeue, which is well within the budget of 40 clock cycles. The analysis for the enqueuing is simpler since the only operation there is the operation performed by the binary tree of comparators when the queue is full. As discussed above, this is at most 10 clock cycles.

A number of optimizations can further simplify the pFabric switch implementation. For instance, we could use a hash of the 5-tuple as the flow-id (instead of the full 5-tuple) to reduce the width of the bit-wise flow-id comparators. A fairly short hash (e.g., 8–12 bits) should suffice since the total number of packets is small and occasional hash collisions only marginally impact the scheduling order. Moreover, if we restrict the priority assignments such that a flow's priority does not increase over time — for example by using absolute flow size as the priority instead of remaining flow size — we would not need the starvation prevention mechanism and could get rid of the flow-id matching logic completely. Our results indicate that using absolute flow size is almost as good as remaining flow size for realistic flow size distributions found in practice (§5.4.3).

Note that our switches do not keep any other state, nor are they expected to provide feedback, nor do they perform rate computations. Further, the significantly smaller buffering requirement lowers the overall switch design complexity and die area [4].

### 4.4.2 End-host implementation

pFabric's priority-based packet scheduling needs to extend all the way to the end-host to be fully effective. In fact, we think of the fabric as starting at the NIC (§3) and in our simulations we assume that the NIC queues also implement pFabric's priority scheduling/dropping mechanisms. An alternative design may push the contention to software queues by rate-limiting the traffic to the NIC (at line rate). Priority scheduling can then be implemented in software across active flows. This approach does not require NIC changes and also avoids dropping packets at the end-host but it requires more sophisticated software particularly at 10Gbps speeds.

The reader may also wonder about the feasibility of our rate control implementation. Specifically, our rate control frequently operates at line rate and uses a fixed retransmission timeout value typically set to 3×RTT which can be quite small (e.g., we use 45$\mu$s in our simulations). Such precise timers may be problematic to implement in current software. However our simulations show that the timeout can be set to larger values (e.g., 200–300$\mu$s for our simulated network) in practice without impacting performance (see §5.4.3 for details). Prior work has demonstrated the feasibility of such retransmission timers in software [18].

It is important to note that while our rate control design is based on TCP, we do not require that the rate control be done by the TCP stack in the kernel. In fact, we expect the near-ideal latency provided by pFabric to most benefit applications that are optimized to reduce the latency incurred at the end-host. Such applications (e.g., RAMCloud [15]) typically use techniques like kernel bypass to avoid the latency of going through the networking stack and implement some form of rate control in user-space. We believe our simple rate control is a nice fit in these scenarios.

Finally, another potential concern is burstiness of traffic and the impact of the very small buffers in pFabric switches. Specifically, mechanisms like Large Send Offload (LSO) can cause bursts of up to 64KB from a 10Gbps NIC [4]. Since our buffers are small, such bursting could lead to unacceptably high drop rates. However, such "batching" techniques cause latency spikes in software that would anyway be undesirable for latency-optimized software stacks. Since pFabric's rate control is very simple, we may not need such optimizations to reduce CPU overhead (especially if we bypass the kernel altogether and implement pFabric's rate control in user-space) and it may be possible to operate with smaller bursts on the order of a few KBs. Nevertheless, in §5.4.3, we show that moderately increasing the switch buffer size

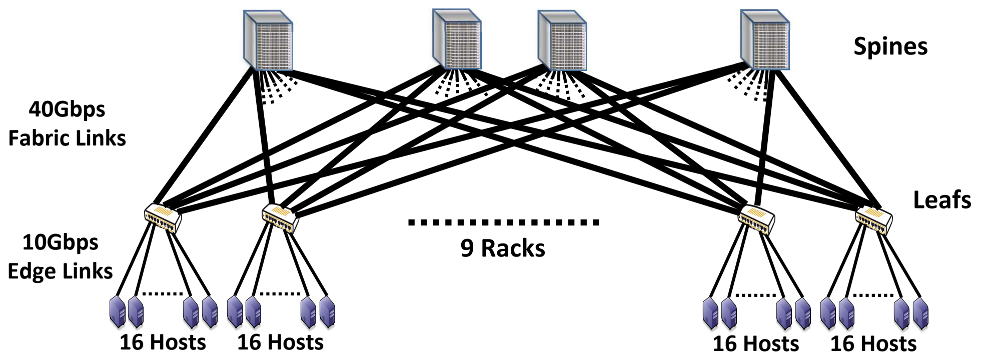


Figure 3: Baseline topology used in simulations.

has no negative impact on performance (which is expected since larger buffers do not hurt the technique in any way) and we could use this to increase tolerance to bursts. Further, the switch complexity is also not impacted too much. Following our switch implementation feasibility analysis in the previous section, with a buffer that is twice as large as before, the complexity of dequeueing a packet increases to 31 clock cycles (11 for finding the highest priority flow-id and 20 for doing the compare to find the earliest packet from that flow-id), which is well within the budget of 40 clock cycles that we have for dequeueing the packet.

## 5 Evaluation

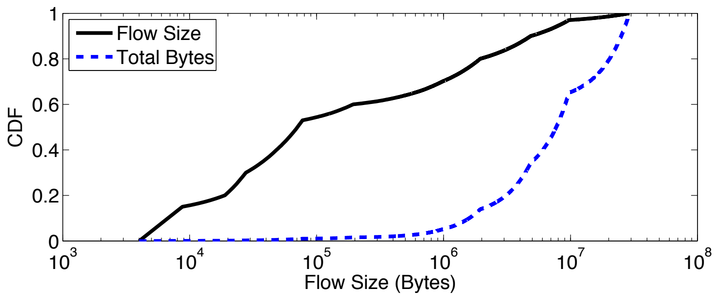
In this section we evaluate pFabric's performance using extensive packet-level simulations in the ns2 [14] simulator. Our evaluation consists of three parts. First, using carefully constructed micro-benchmarks, we evaluate pFabric's basic performance such as its loss characteristics, its ability to efficiently switch between flows that are scheduled one-by-one, and how it handles Incast [18] scenarios. Building on these, we show how pFabric achieves near-optimal end-to-end performance in realistic datacenter networks running workloads that have been observed in deployed datacenters [11, 3]. Finally, we deconstruct the overall results and demonstrate the factors that contribute to the performance.

### 5.1 Simulation Methodology

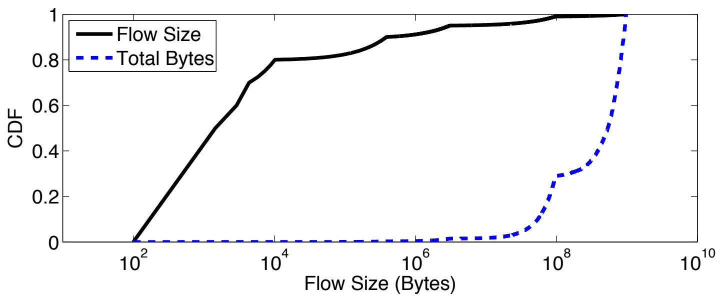
**Fabric Topology:** We use the leaf-spine topology shown in Figure 3. This is a commonly used datacenter topology [1, 11]. The fabric interconnects 144 hosts through 9 leaf (or top-of-rack) switches connected to 4 spine switches in a full mesh. Each leaf switch has 16 10Gbps downlinks (to the hosts) and 4 40Gbps uplinks (to the spine) resulting in a non-oversubscribed (full bisection bandwidth) fabric. The end-to-end round-trip latency across the spine (4 hops) is $\sim$14.6$\mu$s of which 10$\mu$s is spent in the hosts (the round-trip latency across 2 hops under a Leaf is $\sim$13.3$\mu$s).

**Fabric load-balancing:** We use *packet spraying* [10], where each switch sprays packets among all shortest-path next hops in round-robin fashion. We have also experimented with Equal Cost Multipathing (ECMP) which hashes entire flows to different paths to avoid packet reordering. Overall, we found that for all schemes, the best results are obtained with packet-spraying after fast retransmissions are disabled to cope with packet reordering (in fact, this is the reason we disabled 3 dupACKs in our rate control). Hence, we use packet spraying by default for all schemes.

**Benchmark workloads:** We simulate empirical workloads modeled after traffic patterns that have been observed in production datacenters. We consider two flow size distributions shown in Figure 4. The first distribution is from a datacenter supporting web search [3]. The second distribution is from a cluster running large data mining jobs [11]. Flows arrive according to a Poisson process and the source and destination for each flow is chosen uniformly at random. The flow arrival rate is varied to obtain a desired level of load in the fabric. Both workloads have a diverse mix of small and large flows with heavy-tailed characteristics. In the web search workload, over 95% of all bytes are from the 30% of the



(a) Web search workload



(b) Data mining workload

Figure 4: Empirical traffic distributions used for benchmarks. The distributions are based on measurements from real production datacenters [3, 11].

flows that are 1–20MB. The data mining workload is much more extremely skewed: more than 80% of the flows are less than 10KB and 95% of all bytes are in the ∼3.6% flows that are larger than 35MB. As we demonstrate in §5.4, this actually makes the data mining workload easier to handle because it is less likely that multiple large flows are concurrently active from/to one fabric port — reducing network contention. Hence, for most of our simulations, we focus on the more challenging web search workload.

**Performance metrics:** Similar to prior work [13, 20, 3] we consider two main performance metrics. For deadline-constrained traffic, we use the *application throughput* defined as the fraction of flows that meet their deadline. For traffic without deadlines, we use the flow completion time (FCT). We consider the average FCT across all flows, and separately for small and large flows. We also consider the 99th percentile flow completion time for the small flows. We normalize all flow completion times to the best possible completion time for that flow — the value achieved if that one flow is transmitted over the fabric at 10Gbps without any interference from competing traffic.

## 5.2 Schemes compared

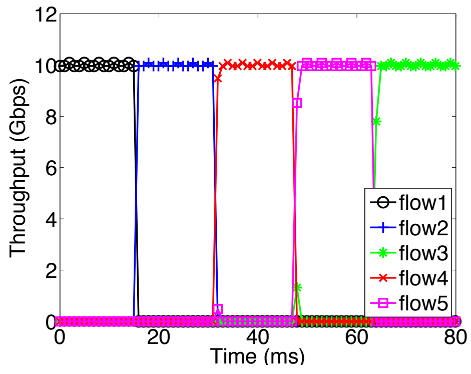
**TCP-DropTail:** A standard TCP-New Reno with Sack and DropTail queues.

**DCTCP:** The DCTCP [3] congestion control algorithm with ECN marking at the fabric queues.

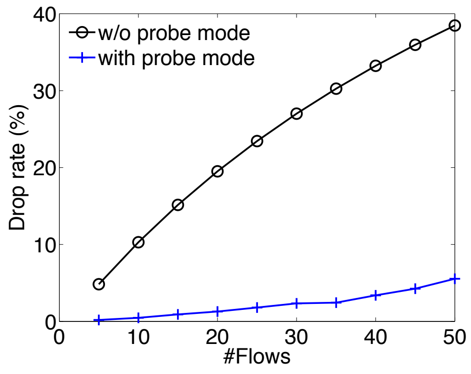
**pFabric:** The design described in this paper including both the switch and the minimal rate control. Unless otherwise specified, the *remaining flow size* is used as the priority for each packet.

**PDQ:** This is the best known prior approach for minimizing flow completion times or missed deadlines. Our implementation follows the design faithfully as described in [13] including the Early Start and Early Termination enhancements and is based on a copy of the source code we obtained from the authors of the PDQ paper.

**Ideal:** The *Ideal* scheduling algorithm described in §3. A central scheduler with a complete view of all flows preemptively schedules existing flows in nondecreasing order of size and in a maximal manner (see Algorithm 1). For this scheme, we conduct flow-level simulations (not packet-level) in Matlab according to the same exact sequence of flow arrivals used in our ns2 benchmarks.



(a) Flow switching



(b) Loss rate

Figure 5: (a) Per-flow throughput when 5 large flows are initiated simultaneously to one destination port. (b) Loss rate vs number of long-lived flows congesting a bottleneck link.

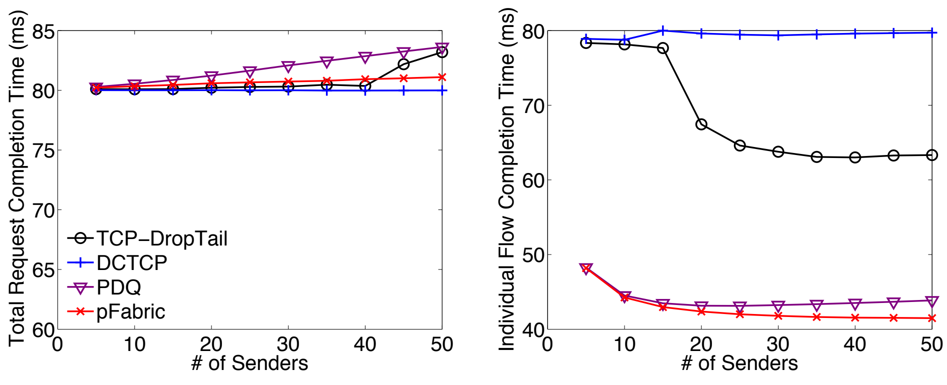
We experimentally determine the best settings for the relevant parameters for all schemes (summarized in Table 1). Note that the larger retransmission timeout for the other schemes compared to pFabric is because they have larger queues. In fact, the difference in retransmission timeout (200$\mu$s versus 45$\mu$s) is in proportion to the queue size difference (225KB − 36KB = 189KB ∼ 150$\mu$s at 10Gbps). Without this, spurious retransmissions would hurt the performance of these schemes. We evaluate the impact of pFabric's RTO in depth in §5.4.3.

| Scheme | Parameters |
|---|---|
| TCP-DropTail | $qSize = 225$KB<br>$initCwnd = 12$ pkts<br>$minRTO = 200\mu$s |
| DCTCP | $qSize = 225$KB<br>$markingThresh = 22.5$KB<br>$initCwnd = 12$ pkts<br>$minRTO = 200\mu$s |
| pFabric | $qSize = 36$KB<br>$initCwnd = 12$ pkts<br>$RTO = 45\mu$s |
| PDQ | $qSize = 225$KB<br>$RTO = 200\mu$s,<br>$K = 2$ (for Early Start)<br>$probingInterval = 15\mu$s |

Table 1: Default parameter settings in simulations.

## 5.3 Basic Performance Measures

**Seamless switching between flows:** Can pFabric seamlessly switch between flows that need to be scheduled serially? To test this, we simultaneously generate 5 large transfers of size 20MB to a single destination host. Figure 5(a) shows the throughput achieved by each flow over time. We observe that the flows are indeed scheduled one-by-one and at each time, one flow grabs all the bottleneck's bandwidth (10Gbps). Note that this is the optimal flow scheduling in order to minimize the average flow completion time in this scenario. pFabric uses this scheduling even though the flows are all exactly the same size because the packet priorities are based on the *remaining* flow size. Hence, a flow that is initially lucky and gets more packets through gets the highest priority and dominates the other flows. The last of the



(a) Total Request (b) Individual Flows

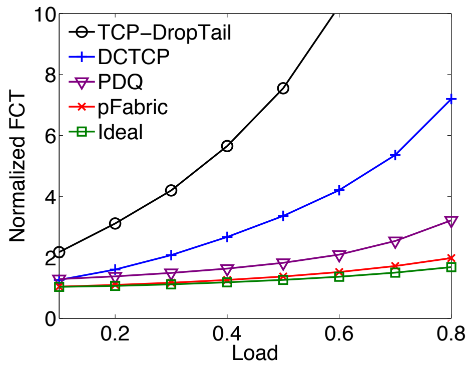
Figure 6: Total request and individual flow completion times in Incast scenario. Note that the range of the y-axis is different for the two plots.

5 flows completes after ∼80.15ms. This is only 150$\mu$s larger then the best possible completion time of 80ms for a 100MB transfer at 10Gbps. Hence, pFabric is able to seamlessly schedule one flow after the other with very little loss of throughput efficiency.

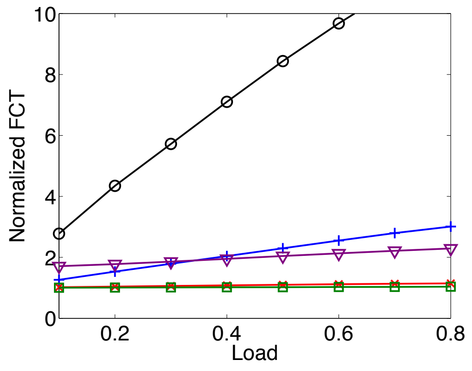
**Loss rate:** The previous simulation showed that pFabric can seamlessly switch between flows without loss of throughput, but what about loss rates? We repeat the previous simulation but stress the network by using up to 50 concurrent large flows to a single destination port, and measure the overall loss rate. We conduct the simulation both with and without pFabric's probe mode (discussed in §4.2). The results are shown in Figure 5(b). We observe that without probe mode, the loss rate rises sharply from ∼4.8% to ∼38.5% as the number of flows increases. This is because except for the high-priority flow, the packets of the other flows are all dropped at the bottleneck. Hence, each low-priority flow retransmits a full-sized (1500B) packet every $RTO = 45\mu$s which is eventually dropped. As expected, the probe mode significantly lowers the loss rate (to under 5.5% with 50 flows) since the low priority flows only periodically send a small probe packet (with a one byte payload) while waiting for the high priority flow to complete.

**Incast:** We now show pFabric's performance for Incast traffic patterns which occur in many large-scale web applications and storage systems and have been shown to result in throughput degradation for TCP [18, 3]. The incast pattern exhibits similar characteristics as the previous experiment where a large number of flows simultaneously transmit to a single destination. Similar to prior work [18], we create Incast by having a receiver node request a 100MB file that is striped across $N$ sender nodes. The senders respond with 100MB/$N$ of data simultaneously. The request completes when all the individual flows have finished. Once a request is complete, the client immediately initiates the next request. The simulation is run for 10,000 requests and we compute the average total request completion time and the average individual flow completion times.

The results for TCP-DropTail, DCTCP, PDQ and pFabric are shown in Figure 6. Note that all schemes use a small $minRTO$ which has been shown to greatly mitigate the Incast problem [18] (DCTCP additionally benefits from aggressive ECN marking [3]). Hence, considering the total request completion time, all schemes handle Incast fairly well. DCTCP does the best and achieves a near-ideal request complete time of 80ms across all number of senders. pFabric is almost as good achieving a total request completion time of 81.1ms at 50 senders. The small increase is due to the slight overhead of serially scheduling flows with pFabric. However, as expected, serial flow scheduling significantly improves the average *individual* flow completion times (Figure 6(b)) for pFabric compared to DCTCP and TCP-DropTail which are more fair across flows. PDQ also exhibits a similar behavior as pFabric since it aims to mimic the same kind of flow scheduling, however it has slightly higher overhead in flow switching and consequently shows slightly worse performance as the number of flows increases.



(a) Web search workload



(b) Data mining workload

Figure 7: Overall average normalized flow completion time for the two workloads at various loads.

## 5.4 Overall Performance

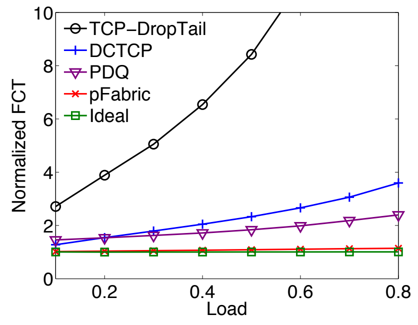
In this section we show pFabric's overall performance in large scale datacenter topologies with realistic workloads. We show that pFabric's ability to efficiently schedule flows in the order of their priorities (remaining flow size or deadline) enables it to achieve near-optimal performance for traffic scenarios with no deadlines as well as scenarios where there is a mix of deadline and no-deadline traffic. In the interest of space, after the overall performance results, we only show results for the deadline-unconstrained traffic for targeted experiments that highlight different aspects of pFabric's design and their impact on overall performance.

### 5.4.1 Deadline-unconstrained traffic

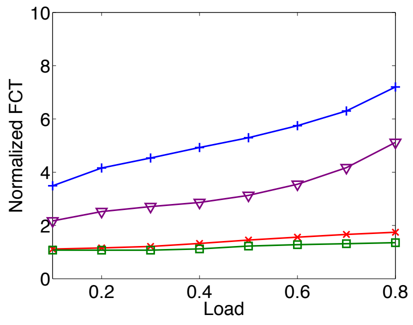
pFabric achieves near-optimal flow completion times for all flow sizes, loads and for both workloads in our simulations. Figure 7 shows the overall average flow completion times for the web search and data mining benchmarks as we vary the load from 10% to 80%. Recall that each flow's completion time is normalized to the best possible value that is achievable in an idle fabric for that flow. We observe that for both workloads the average FCT with pFabric is very close to that of the Ideal flow scheduling scheme and is significantly better than for the other schemes. pFabric's performance is within ∼0.7-17.8% of the Ideal scheme for the web search workload and within ∼1.7–10.6% for the data mining workload. Compared to PDQ, the average FCT with pFabric is ∼19-39% lower in the web search workload and ∼40-50% lower in the data mining workload. All schemes generally do better for the data mining workload, particularly at high load. This is because in the data mining workload, the largest ∼3.6% of flows contribute over 95% of all bytes (Figure 4(b)). These flows, though very large, arrive infrequently and thus it is rare that multiple of them are concurrently active at a particular fabric port and cause sustained congestion.

It is important to note that PDQ always requires one extra RTT of overhead for flow initiation (SYN/SYN-ACK exchange) before a flow can transmit. Because of this, PDQ's normalized FCT is at-least two for very small flows that can ideally complete in one RTT. For example, in the data mining workload where about 50% of all flows are one packet, it is not surprising that pFabric's average normalized FCT s 50% lower than PDQ.

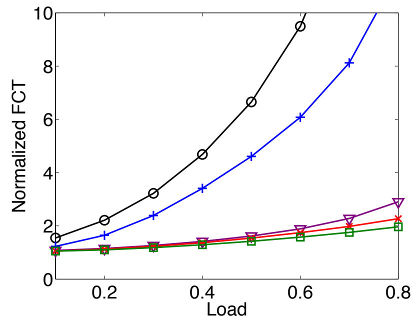
**FCT breakdown based on size:** We now breakdown the FCT stats across small (0, 100KB], medium (100KB, 10MB], and large (10MB, $\infty$) flows. The results are shown in Figures 8 and 9 for the two workloads. We plot the average (normalized) FCT in each bin and also the 99th percentile for the small flows (whose tail-latency is often critical in practice). The results show that for both workloads, pFabric achieves near-optimal average and 99th percentile FCT for the small flows: it is within ∼1.3–13.4% of the ideal average FCT and within ∼3.3–29% of the ideal 99th percentile FCT (depending on load). Compared to PDQ, the average FCT for the small flows with pFabric is ∼30-50% lower for the web search workload and ∼45-55% lower for the data mining workload with even larger improvements at the 99th percentile. Similarly, the average FCT for the medium flows with pFabric is very close to ideal for both workloads.



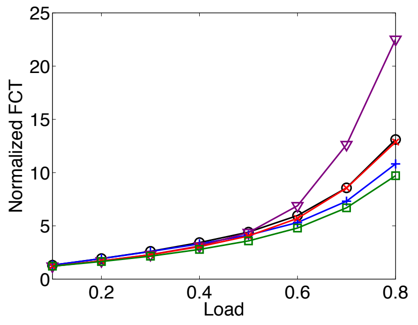
(a) (0, 100KB]: Avg



(b) (0, 100KB]: 99th prctile

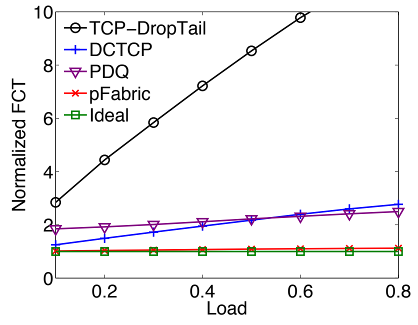


(c) (100KB, 10MB]: Avg

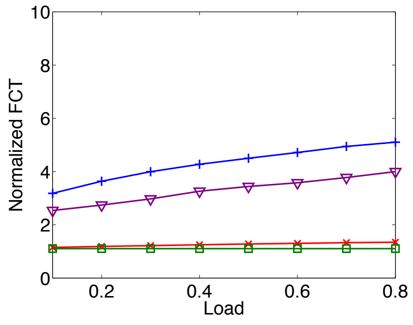


(d) (10MB, ∞): Avg

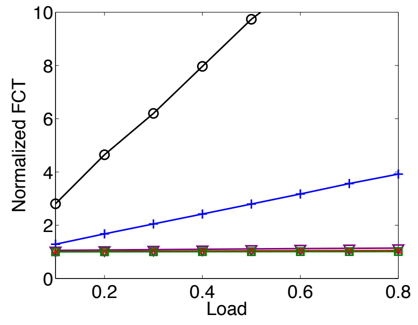
Figure 8: Web search workload: Normalized FCT statistics across different flow sizes. Note that TCP-DropTail does not appear in part (b) because its performance is outside the plotted range and the y-axis for part (d) has a different range than the other plots.



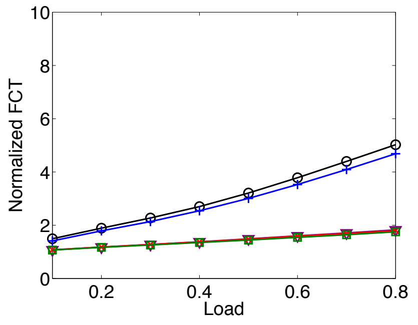
(a) (0, 100KB]: Avg



(b) (0, 100KB]: 99th prctile

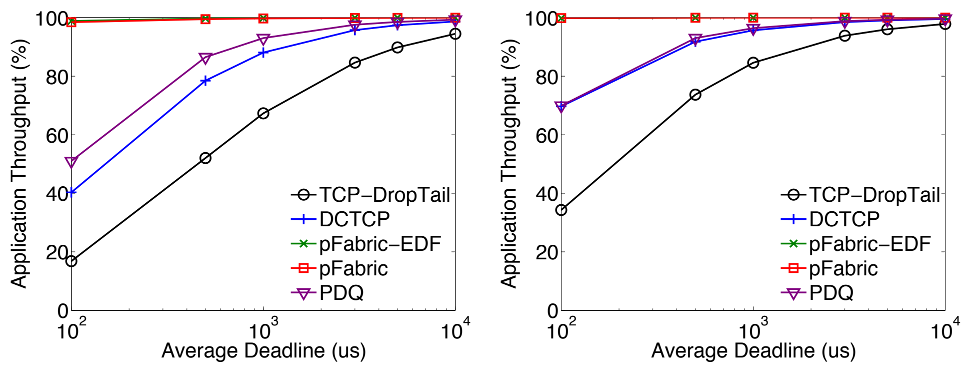


(c) (100KB, 10MB]: Avg



(d) (10MB, ∞): Avg

Figure 9: Data mining workload: Normalized FCT statistics across different flow sizes. Note that TCP-DropTail does not appear in part (b) because its performance is outside the plotted range.



(a) Web search workload (b) Data mining workload

Figure 10: Application Throughput for deadline traffic with various deadline settings at 60% load.

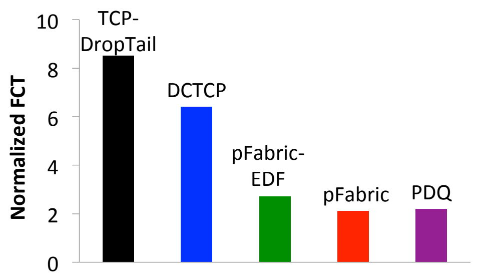
pFabric also achieves very good performance for the average FCT of the large flows, across all but the highest loads in the web search workload. pFabric is roughly the same as TCP and ∼30% worse than Ideal at 80% load for the large flows in the web search workload (for the data mining workload, it is within ∼3.3% of Ideal across all flows). This gap is mainly due to the relatively high loss rate at high load for this workload which wastes bandwidth on the upstream links (§4.2). Despite the rate control, at 80% load, the high initial flow rates and aggressive retransmissions cause a ∼4.3% packet drop rate in the fabric (excluding drops at the source NICs which do not waste bandwidth), almost all of which occur at the last hop (the destination's access link). However, at such high load, a small amount of wasted bandwidth can cause a disproportionate slowdown for the large flows [4]. Note that this performance loss occurs only in extreme conditions — with a challenging workload with lots of elephant flows and at very high load. As Figure 8(d) shows, under these conditions, PDQ's performance is more than 75% worse than pFabric.

### 5.4.2 Mix of deadline-constrained and deadline-unconstrained traffic

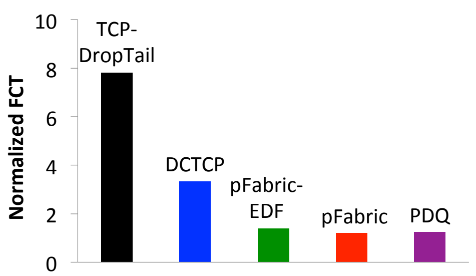
We now show that pFabric maximizes the number of flows that meet their deadlines while still minimizing the flow completion time for flows without deadlines. To perform this experiment, we assign deadlines for the flows that are smaller than 200KB in the web search and data mining workloads. The deadlines are assumed to be exponentially distributed similar to prior work [20, 13, 17]. We vary the mean of the exponential distribution (in different simulations) from 100$\mu$s to 100ms to explore the behavior under tight and loose deadlines and measure the Application Throughput (the fraction of flows that meet their deadline) and the average normalized FCT for the flows that do not have deadlines. We lower bound the deadlines to be at least 25% larger than the minimum FCT possible for each flow to avoid deadlines that are impossible to meet.

In addition to the schemes used for the baseline simulations with deadline-unconstrained traffic, we present the results for pFabric with *Earliest-Deadline-First (EDF)* scheduling. pFabric-EDF assigns the packet priorities for the deadline-constrained flows to be the flow's deadline quantized to microseconds; the packets of flows without deadlines are assigned priority based on remaining flow size. Separate queues are used at each fabric port for the deadline-constrained and deadline-unconstrained traffic with strict priority given to the deadline-constrained queue. Within each queue, the pFabric scheduling and dropping mechanisms determine which packets to schedule or drop. Each queue has 36KB of buffer.

Figure 10 shows the application throughout for the two workloads at 60% load. We picked this moderately high load to test pFabric's deadline performance under relatively stressful conditions. We find that for both workloads, both pFabric-EDF and pFabric achieve almost 100% application throughput even at the tightest deadlines and perform significantly better than the other schemes. For the web search workload, pFabric-EDF achieves an Application Throughput of 98.9% for average deadline of 100$\mu$s; pFabric (which is deadline-agnostic and just uses the remaining flow size as the priority) is only slightly worse at 98.4% (the numbers are even higher in the data mining workload). This is not surprising; since pFabric achieves a near-ideal FCT for the small flows, it can meet even the tightest deadlines for them.

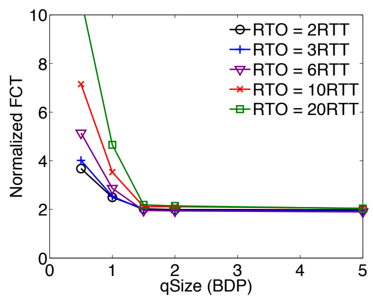


(a) Web search workload

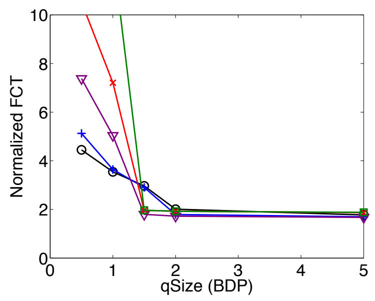


(b) Data mining workload

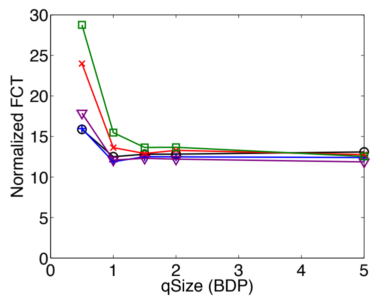
Figure 11: Average normalized FCT for non-deadline traffic at 60% load.



(a) Overall: Avg



(b) (0, 100KB]: 99th prctile



(c) (10MB, ∞): Avg

Figure 12: Web search workload at 80% load using a variety of queue size and retransmission timeout settings. qSize is normalized to BDP = 18KB and the RTO is normalized to RTT = 14.6$\mu$s.

As expected, PDQ achieves a higher application throughput than the other schemes. But it misses a lot more deadlines than pFabric, especially at the tightest settings. This is partly because of the one extra RTT of flow-initiation overhead that PDQ adds to every flow. Because of this, PDQ cannot meet some of the tighter deadlines for the small flows (that can ideally complete in 1 RTT). We verified that when the average deadline was 100$\mu$s, due to its fixed one RTT overhead, PDQ could not have met the deadline for 22.7% of the deadline-constrained flows (this number was 5.0% for the 500$\mu$s and 2.5% for the 1ms average deadline settings).

We also find that pFabric achieves the lowest average FCT for the flows without deadlines (Figure 11). pFabric-EDF is slightly worse as expected because it gives strict priority to the deadline-constrained traffic.

### 5.4.3 pFabric deep dive

In this section we dig deeper into pFabric's design in a series of targeted simulations. For brevity, the majority of simulations in this section use the web search workload since it is more challenging and allows for clearer contrasts. Also, we only show the results for the overall average FCT, the 99th percentile FCT for the small flows, and the average FCT for the large flows. These plots cover the range of behaviors we observe for different flows.

**Impact of qSize and RTO:** We repeat the web search workload at 80% load for different pFabric switch buffer size ($qSize$) and retransmission timeout ($RTO$) settings. $qSize$ is varied between $0.5 \times BDP$ and $5 \times BDP$ (recall that the BDP is 18KB for our topology). $RTO$ is varied between $2 \times RTT$ and $20 \times RTT$ where $RTT$ is the baseline round-trip latency of the fabric (14.6$\mu$s). The results are shown in Figure 12. We observe a loss in performance for buffers smaller than one BDP. At the same time, increasing the buffer size beyond $2 \times BDP$ yields very little gains. This is intuitive since we need at least one BDP to allow enough time for retransmitting dropped packets without under-utilization (§4.3), but having just one BDP provides zero margin for error and requires perfect RTO estimation to avoid performance loss. As the plots show, making the buffer size slightly larger than one BDP gives more margin and allows the use of a simple, fixed RTO without performance loss. We recommend $qSize = 2 \times BDP$ and

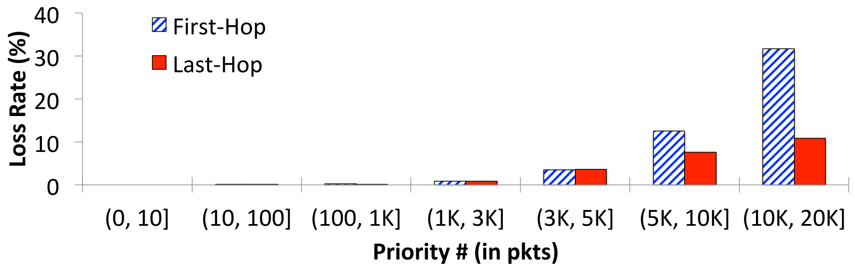


Figure 13: Packet loss rate at the first-hop (source NIC) and last-hop (destination access link) versus priority number for the web search workload at 80% load. The loss rate in the fabric's core is negligible.

$RTO = 3 \times RTT$ for pFabric based on these results. An RTO of $3 \times RTT$ is appropriate since with a buffer of $2 \times BDP$, the total round-trip delay when a packet is dropped (and the buffer is full) is $3 \times RTT$. The values we use in our simulations ($qSize = 36$KB, $RTO = 45\mu$s) follow this guideline.

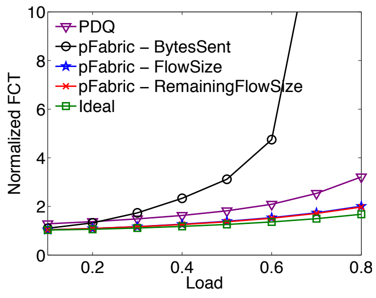
While the above guideline guarantees good performance in all cases, interestingly, Figure 12 suggests that for realistic workloads we can use a much larger RTO with almost no performance penalty. For instance, $RTO = 20 \times RTT$ ($\sim$290$\mu$s for our fabric) achieves nearly the same performance as $RTO = 3 \times RTT$ when $qSize = 2 \times BDP$. Relaxing the required retransmission timeout could be very useful in practice and simplify the pFabric host's implementation; as prior work has demonstrated [18], retransmission timers with a granularity of 200$\mu$s are easy to achieve in software.

The reason such large RTOs do not have significant impact (despite the small buffers) is that almost all packet drops in pFabric occur for the large flows which anyway have fairly high FCTs. To see this, we plot the packet loss rate versus the packet priority number for the baseline web search workload at 80% load in Figure 13. The plot shows that almost all losses are for flows larger than 3000 packets. But these flows are bandwidth-limited and necessarily take a long time to complete. For example, a 3000 packet flow (1500 bytes per packet) needs *at least* 3.6ms to complete at 10Gbps and thus is not severely impacted if the RTO is not very tight and adds $\sim$200$\mu$s of additional delay.

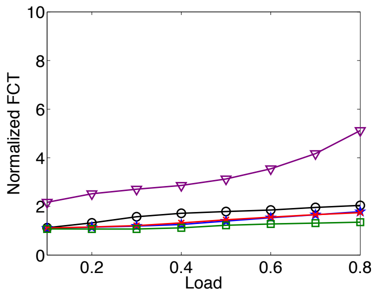
**Different priority assignment schemes:** Next, we compare three different schemes for assigning packet priorities with increasing degrees of complexity. For each packet transmitted, the priority field is set to be: (i) the number of bytes thus far sent from the flow; (ii) the flow size in bytes; or (iii) the remaining flow size in bytes (the default scheme in this paper). The first scheme is the simplest as it does not require knowledge of flow size. The second and third schemes both require flow size information, but the second is simpler since the priority number is decided once and remains constant for all the packets of a flow. As explained in §4.1, this scheme simplifies the pFabric switch implementation since we don't need the starvation prevention mechanism.

Figures 14 and 15 show a comparison of the three schemes and also PDQ. We find that using the flow size and remaining flow size as the packet priority achieve nearly indistinguishable overall average FCT for both workloads. This is not surprising; even though remaining flow size is conceptually closer to ideal (§3), for realistic workloads with a diverse range of flow sizes, most of the benefit is in scheduling the small flows before the large flows which both schemes achieve. The breakdowns for the small and large flows are also consistent with this finding. We do find that for the large flows ($>$ 10MB), the remaining flow size scheme achieves up to $\sim$15% lower average FCT than absolute flow size for the web search workload (Figure 14(c)).

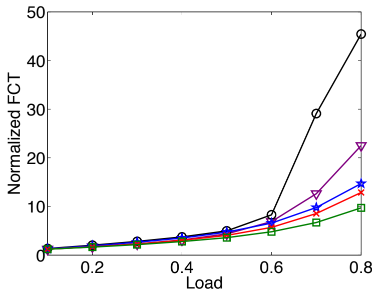
The performance of "BytesSent" is more varied. As expected, it is worse than the schemes with flow size knowledge. Yet, for the data mining workload, it still achieves a significantly lower overall average FCT than PDQ. In fact, we find that its average and tail FCT for the small flows ($<$ 100KB) is almost as good as default pFabric. However, its performance degrades for the larger flows. In particular, for the web search workload its performance completely breaks down at high load. This is because in this workload, especially at high load, it is common for multiple large flows to arrive and compete with an existing large flow during its lifetime. Each time this occurs, the BytesSent priority scheme essentially stops all existing flows (which have lower priority since they have sent more data) until the new flow "catches up". Hence, the large flows can take very long to complete. The takeaway is that the *BytesSent scheme should only be used in environments where a very small fraction of the flows are large and it is*



(a) Overall: Avg

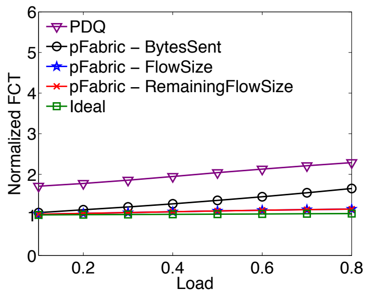


(b) (0, 100KB]: 99th prctile

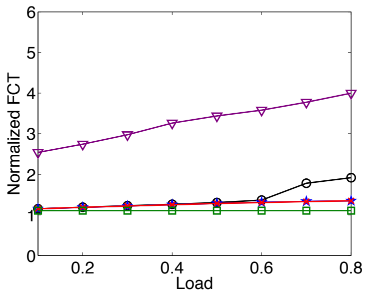


(c) (10MB, ∞): Avg

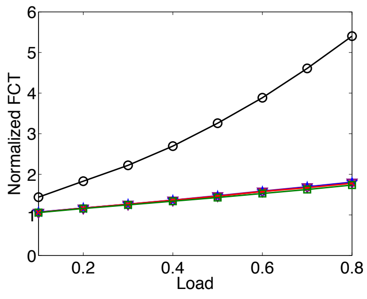
Figure 14: Web search workload using different priority assignment schemes.



(a) Overall: Avg

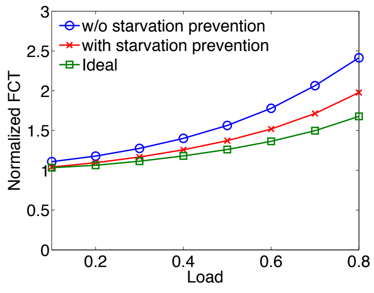


(b) (0, 100KB]: 99th prctile

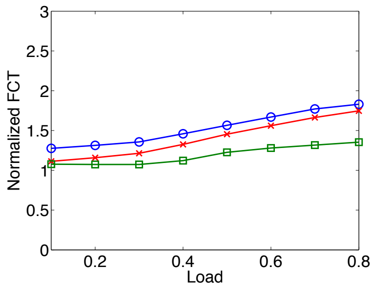


(c) (10MB, ∞): Avg

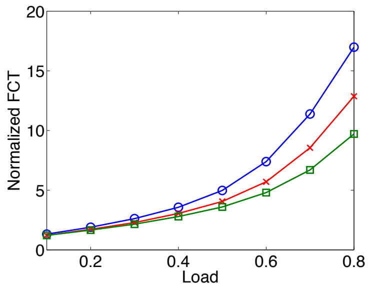
Figure 15: Data mining workload using different priority assignment schemes.



(a) Overall: Avg



(b) (0, 100KB]: 99th prctile



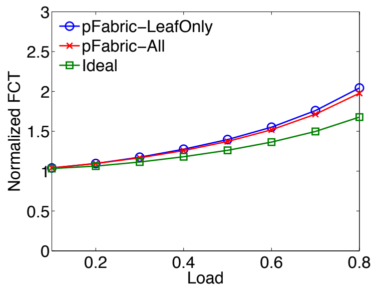
(c) (10MB, ∞): Avg

Figure 16: Impact of pFabric's starvation prevention (for the web search workload).

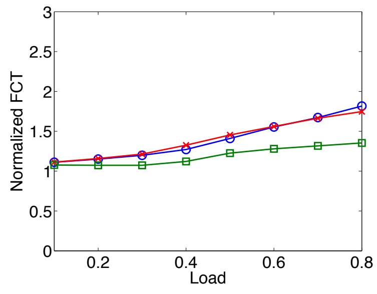
*rare that multiple such flows are concurrently active on a single path.*

**pFabric's starvation prevention mechanism:** Figure 16 compares the performance of pFabric with and without the starvation prevention mechanism discussed in §4.1. Recall that without starvation prevention, pFabric dequeues packets strictly according to their priority number, which may delay a packet from a flow for a long time if later packets of that flow have higher priority. The results show that starvation prevention improves the overall average FCT by ∼6-22% across different loads.

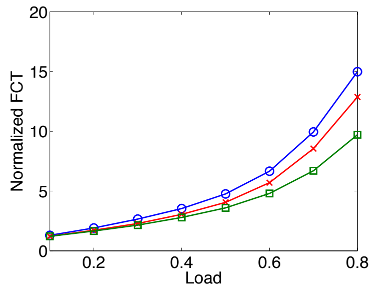
**pFabric only at the leaf tier:** We now consider the implications of deploying pFabric switches only at the leaf tier of the fabric. The spine switches employ standard drop-tail queues. We compare this configuration with having pFabric switches at both the leaf and spine tiers. The results are shown in Figure 17. We find that having pFabric only at the leaf tier is only marginally worse than having it everywhere. This is because most contention occurs at the leaf switches anyway since packet-spraying provides very good load-balancing (more on this below). This is a useful result which makes it easier to deploy pFabric in existing environments; it suggests that we can initially introduce pFabric for a subset of the switches and still get a significant benefit.



(a) Overall: Avg

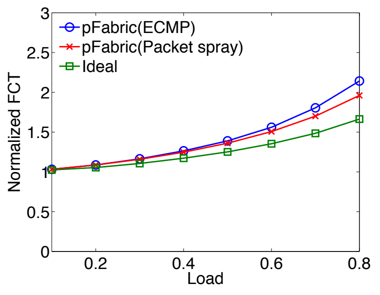


(b) (0, 100KB]: 99th prctile

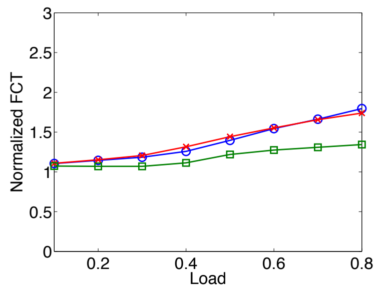


(c) (10MB, ∞): Avg

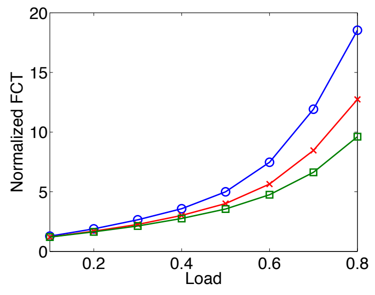
Figure 17: Impact of having pFabric switches only at the leaf tier (for the web search workload).



(a) Overall: Avg

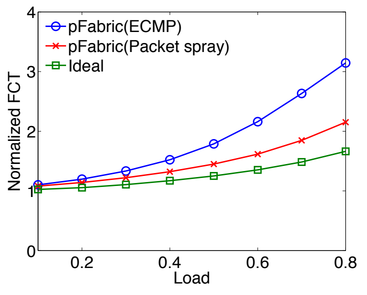


(b) (0, 100KB]: 99th prctile

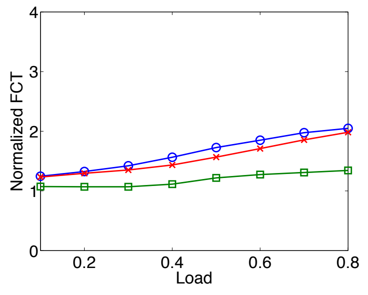


(c) (10MB, ∞): Avg

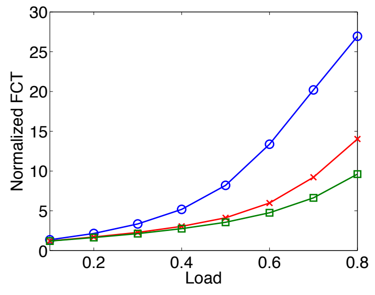
Figure 18: Web search workload using ECMP and packet-spraying load-balancing for baseline topology with 40Gbps leaf-spine links.



(a) Overall: Avg



(b) (0, 100KB]: 99th prctile



(c) (10MB, ∞): Avg

Figure 19: Web search workload using ECMP and packet-spraying load-balancing for “flat” topology with 10Gbps leaf-spine links.

**Impact of fabric load-balancing:** We now compare pFabric’s performance with different load-balancing mechanisms. We repeat the web search benchmark with ECMP and packet spraying load-balancing on two topologies: the baseline topology which uses 40Gbps leaf-spine links (Figure 3), and a “flat” topology which uses 10Gbps leaf-spine links. The latter topology employs 16 spine switches and has the same bisection bandwidth as the baseline topology. We report the results in Figures 18 and 19. We observe that ECMP and packet-spraying achieve similar overall performance for the baseline topology except for the average FCT of large flows at high load where ECMP is up to ∼44% worse. However, for the flat topology, we observe a fairly significant degradation even in the overall average FCT with ECMP. At 80% load, the overall average FCT with ECMP is ∼46% higher than with packet-spraying. Note that most of the performance degradation due to ECMP occurs for the large flows; pFabric achieves near-ideal performance for the small flows regardless of the load-balancing scheme. These results are consistent with prior findings (e.g., see [16]) and show that *efficient load-balancing techniques beyond ECMP (such as packet-spraying) improve the performance of large flows especially when there is no*

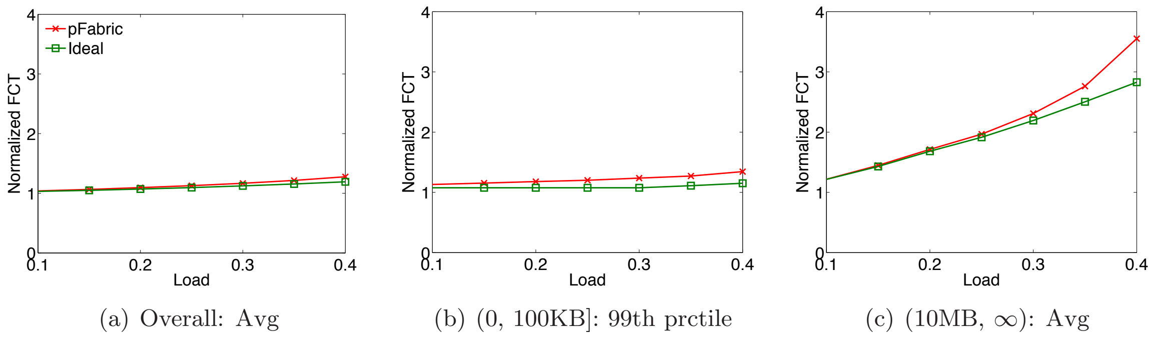


Figure 20: Web search workload on a 3:1 oversubscribed fabric topology. The load here is at the fabric's edge.

*speedup at the core links relative to the edge links.* Of course, good load-balancing is generally important for any datacenter transport solution and is complimentary to pFabric's mechanisms. We have used packet-spraying in this paper due to its simplicity and good performance, but other techniques [2, 16] can also be used with pFabric.

**Impact of oversubscription:** Finally we show that pFabric works well even for oversubscribed topologies. All prior results have been based on the full-bisection bandwidth fabric topology shown in Figure 3. We now consider the implications of having an oversubscription at the leaf switches. For this, we repeat the web search benchmark for a 3:1 oversubscribed fabric topology with 3 leaf switches, each with 48 10Gbps downlinks (to the hosts) and 4 40Gbps uplinks (to the spine). The results are shown in Figure 20. Note that the load is shown for the fabric's *edge*. Since about one-third of all traffic stays local to a leaf switch and two-thirds traverse the spine (recall that the source and destination for each flow is chosen at random), the load at the fabric's core is about $\frac{2}{3} \times 3 = 2$ times larger than at the edge. Hence, the 5–40% loads considered correspond to 10–80% load at the core. The results confirm that pFabric is very close to Ideal for the oversubscribed topology as well.

## 6 Incremental Deployment

Our goal in this paper has been to decouple flow scheduling and rate control and design the simplest possible mechanisms for both tasks. This results in very simple switch and rate control designs, but it does require some hardware changes. In this section we ask how far could we go with the same insight of decoupling flow scheduling from rate control using existing switches? Specifically, we consider using the available priority queues in today's switches and tackle the question of how end-hosts should set the priority field in the packet header to approximate SRPT-style flow scheduling in the fabric. Commodity switches typically support 4–8 class of service queues. Current practice is to use these to isolate entire traffic classes; for example, give higher priority to all traffic belonging to an important application (such as a realtime web application) over less important traffic (such as data backups). Clearly such crude mechanisms cannot minimize flow completion time and guarantee near-ideal latency for small delay-sensitive flows.

We consider a design that dynamically decides the priority based on flow sizes as we have done with pFabric. The basic idea is to set the highest priority for flows smaller than a particular size, the next highest priority for flows greater than the above size but less than a second threshold and so on, so we can approximate the same scheduling behavior as pFabric and thus minimize flow completion time. The are two key challenges with this approach: (i) *How many priority queues are necessary for good performance?* and (ii) *What flow size thresholds should be used for each priority queue?*

### 6.1 Assigning Flow Priorities

We now present a novel and principled approach to answering these questions. We use a simple queueing model to derive the optimal thresholds for minimizing the average FCT for a given flow size distribution (the flow size distribution is empirically measured and assumed to be known in advance). For simplicity,

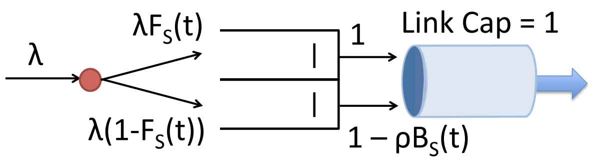


Figure 21: Queueing model for two-queue system. Flows arrive with rate $\lambda$ and have size $\mathbf{S} \sim \mathbf{F_S}(\cdot)$. $\rho$ is the total load and $\mathbf{B_S}(\cdot)$ is distribution of total bytes according to flow size. Flows smaller than threshold, $\mathbf{t}$, use the high-priority queue.

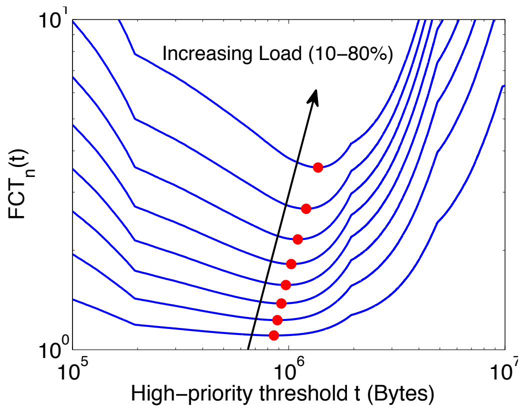


Figure 22: $\mathbf{FCT_n(t)}$ for the web search flow size distribution at loads 10–80%. The red circles show the optimal threshold at each load.

we present the derivation for the case of two priority queues, but it can be generalized to any number of priority queues.

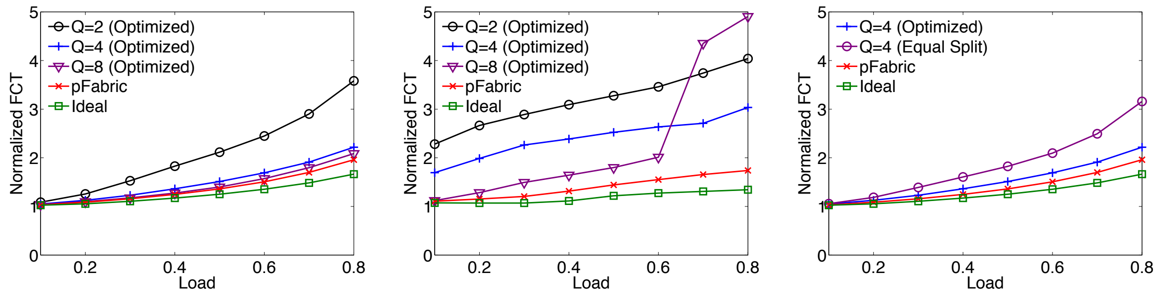
The queuing model is shown in Figure 21. Flows arrive to a link of capacity 1 according to a Poisson process of rate $\lambda$ and have size $S \sim F_S(\cdot)$ ($F_S$ is the CDF of $S$), imposing a total load of $\rho = \lambda E(S) < 1$. Flows smaller (larger) than threshold $t > 0$ are enqueued in the high-priority (low-priority) queue. Therefore, the arrival processes to the two queues are independent Poisson processes with rates $\lambda F_S(t)$ and $\lambda(1-F_S(t))$. The high-priority queue has strict priority and drains at rate 1. The low-priority queue uses the *remaining bandwidth* after servicing the high-priority traffic. Thus, its drain rate is $1 - \rho B_S(t)$ where $B_S(t) = \int_0^t x f_S(x)\,\mathrm{d}x / E(S)$ is the fraction of the overall bytes that belong to flows smaller than $t$. Note that in reality, the low-priority queue drains only when the high-priority queue is empty. However, this complicates the analysis since the two queues are dependent. By using the average drain rate of the low priority queue as its instantaneous drain rate, we greatly simplify the analysis.

The average normalized FCT (FCT divided by flow size) can be derived as a function of the threshold $t$ for this model:

$$\begin{aligned} FCT_n(t) &= F_S(t) + \frac{1}{1-\rho B_S(t)}(1-F_S(t)) \\ &+ \frac{\lambda}{2(1-\rho B_S(t))}\int_0^t x^2 f_S(x)\,\mathrm{d}x \int_0^t \frac{f_S(y)}{y}\,\mathrm{d}y \\ &+ \frac{\lambda}{2(1-\rho B_S(t))(1-\rho)}\int_t^\infty x^2 f_S(x)\,\mathrm{d}x \int_t^\infty \frac{f_S(y)}{y}\,\mathrm{d}y. \end{aligned} \quad (1)$$

The derivation is based on using the well-known Pollaczek-Khintchine formula [12] to compute the average waiting time for a flow in each priority queue (assuming $M/G/1$ queues) and can easily be generalized for any number of priority queues (see Appendix A for the proof in the general case). It is important to note that $FCT_n(\cdot)$ depends on the flow size distribution as well as the overall load $\rho$.

Figure 22 shows $FCT_n$ and the optimal threshold for the high-priority queue computed numerically for the web search flow size distribution (Figure 4(a)). The threshold varies between ∼880-1740KB as the load increases from 10% to 80%. Also, the figure suggests that the performance can be fairly



(a) Overall: Avg (b) (0, 100KB]: 99th prctile (c) (10MB, ∞): Avg

Figure 23: Web search benchmark with 2, 4, and 8 priority queues. Parts (a) and (b) show the average normalized FCT across all flows and the 99th percentile for the small flows. Part (c) compares the performance using the optimized thresholds with a heuristic which splits the flows equally in case of 4 queues.

sensitive to the chosen threshold, particularly at high load. We evaluate the sensitivity to the threshold using simulations in the next section.

**Remark 3.** The above derivation provides a principled way of determining thresholds for each priority, however it assumes that we know the exact flow size distribution in advance. Measuring the flow size distribution can be challenging in practice since it can change across both time and space. For instance, because of spatial variations (different flow size distributions at different switches), we may need to use different thresholds for the priority queues at each switch and further these thresholds may change over time.

## 6.2 Simulations

We now compare using a few priority queues in existing switches with pFabric. Our results confirm that while this mechanism provides good performance with a sufficient number of priority queues (around 8), it is still worse than pFabric and the performance is sensitive to the value of the thresholds used and also how the switch buffer is shared among the priority queues.

We simulate the web search workload (§5.1) for three scenarios with 2, 4, and 8 priority queues per fabric port. The queues at a port share a buffer pool of size 225KB (150 packets). We reserve 15KB (10 packets) of buffer per queue and the rest is shared dynamically on a first-come-first-serve basis. In each scenario, we use the optimal flow size thresholds for each priority queue as derived in §6.1. The results are shown in Figure 23. We observe that, as expected, the average overall FCT (part (a)) improves as we increase the number of priority queues and is close to pFabric's performance with 8 priority queues. We observed a similar trend in the average FCT across small, medium, and large flows (plots omitted). Figure 23(b) also shows that there is a significant increase in the 99th percentile FCT for the small flows at high loads in the 8-queue case. This is because with 8 queues, 80 out of the total 150 packets are reserved, leaving only 70 packets to be shared among the queues. Thus at high load, during some bursts, the high priority queue runs out of buffers and drops packets, increasing tail latency. This demonstrates the need for carefully tuning the buffer allocations for each priority queue for good performance.

**Sensitivity to thresholds:** Finally, we explore the sensitivity of the performance with a few priority queues to using the "right" thresholds for splitting traffic. Figure 23(c) shows a comparison of the 4-queue system with optimal thresholds with a reasonable heuristic that splits flows equally across the 4 queues: the smallest 25% of flows are assigned to the highest priority queue, second smallest 25% to the second highest priority, etc. The plot shows the average FCT across all flows. We find a fairly substantial improvement with the optimized thresholds. At 80% load, the average FCT is reduced by more than 30% with more substantial performance gaps for the tail latencies for short flows (we omit the figure for brevity). This confirms that the thresholds for splitting traffic across limited priority queues need to be chosen carefully. *By allowing an essentially unlimited number of priorities, pFabric does not require any tuning and is not sensitive to parameters such as thresholds (which may vary across time and space), minimum reserved buffer per priority queue, overall buffer size, etc.*

## 7 Discussion

pFabric, more generally, advocates a different design philosophy for datacenter networks. Our thought process is informed by the fact that the datacenter network is more an inter-connect for distributed computing workloads rather than a bit-pipe. Hence we believe that it is more important to orchestrate the network resource allocation to meet overall computing objectives, rather than traditional communication metrics such as throughput and fairness which TCP optimizes for. This leads us to a design ethos where flows (which are proxy for the datum needed to be exchanged in the compute tasks) become first-class citizens and the network fabric is designed to schedule them in a lightweight fashion to maximize application-layer objectives. pFabric is our first step in this direction. Below, we discuss some other common concerns that might come up with a design like pFabric.

**Starvation & Gaming:** A potential concern with strictly prioritizing small flows is that this may starve large flows. Further, a malicious user may game the system by splitting up her large flows to gain an advantage. Of course, these issues are not unique to pFabric; any system that implements SRPT-like scheduling has these concerns. That said, as prior work has also argued (see for example PDQ [13] and the references therein, particularly Bansal *et al.* [7]), under realistic heavy-tailed traffic distributions, SRPT actually improves the majority of flows (even the large flows) compared to TCP's fair sharing. This is consistent with our findings (e.g., Figures 8(d) and 9(d)). The intuition is that for heavy-tailed distributions, small flows contribute a small fraction of the overall traffic; hence prioritizing them has little impact on the large flows and in fact helps them because they complete quickly which reduces network contention. Nonetheless, if desired, an operator can put in explicit safeguards against starvation. For instance, she can place a cap the priority numbers so that beyond a certain size, all flows get the same base priority. Finally, our current design is targeted to private datacenters thus malicious behavior is out of scope. In public environments, further mechanisms may be needed to prevent abuse.

**Setting packet priorities:** In many datacenter applications flow sizes or deadlines are known at initiation time and can be conveyed to the network stack (e.g., through a socket api) to set priorities. In other cases, we expect that pFabric would achieve good performance even with imprecise but reasonable estimates of flow sizes. As shown in §6, with realistic distributions, most of the benefit can be achieved by classifying flows into a few (4-8) priority levels based on size. The intuition is that it suffices that at any instant at each switch the priority dequeueing order is maintained (which does not require that priorities be accurate, only that relative priorities across enqueued flows be largely correct).

**Supporting multiple priority schemes:** In practice, datacenter fabrics are typically shared by a variety of applications with different requirements and a single priority scheme may not always be appropriate. This can easily be handled by operating the pFabric priority scheduling and dropping mechanisms within individual "higher-level" traffic classes in an hierarchical fashion. Traditional QoS mechanisms such as WRR are used to divide bandwidth between these high-level classes based on user-defined policy (e.g., a soft-real time application is given a higher weight than batch jobs), while pFabric provides near-optimal scheduling of individual flows in each class according to the class's priority scheme (remaining flow size, deadlines, etc).

**Other datacenter topologies:** We have focused on Fat-tree/Clos topologies in this paper as this is by far the most common topology in practice. However, since conceptually we think of the fabric as a giant switch with bottlenecks only at the ingress and egress ports (§3) we expect our results to carry through to any reasonable datacenter topology that provides uniform high throughput between ingress and egress ports.

**Stability:** Finally, the theoretical literature has demonstrated scenarios where size-based traffic prioritization may reduce the stability region of the network [19]. Here, stability is in the *stochastic* sense meaning that the network may be unable to keep up with flow arrivals even though the average load on each link is less than its capacity [9]. However, this problem is mostly for "linear" topologies with flows traversing different numbers of hops — intuitively it is due to the tradeoff between prioritizing small flows versus maximizing service parallelism on long routes. We have not seen this issue in our study and do not expect it to be a major concern in real datacenter environments because the number of hops is very uniform in datacenter fabrics, and the overall load contributed by the small (high-priority) flows is small for realistic traffic distributions.

## 8 Conclusion

This paper decouples the key aspects of datacenter packet transport — flow scheduling and rate control — and shows that by designing very simple mechanisms for these goals separately we can realize a minimalistic datacenter fabric design that achieves near-ideal performance. Further, it shows how surprisingly, large buffers or complex rate control are largely unnecessary in datacenters. The next step is to integrate a prototype implementation of pFabric with a latency-sensitive application to evaluate the impact on application layer performance. Further, our initial investigation suggests that further work on designing incrementally deployable solutions based on pFabric could be fruitful. Ultimately, we believe this can pave the path for widespread use of these ideas in practice.

**Acknowledgments:** We thank our shepherd, Jon Crowcroft, and the anonymous SIGCOMM reviewers for their valuable feedback. Mohammad Alizadeh thanks Tom Edsall for useful discussions regarding the practical aspects of this work.

## References

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

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proc. of NSDI*, 2010.

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

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of NSDI*, 2012.

[5] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. In *Proc. of HotNets*, 2012.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of SIGMETRICS*, 2012.

[7] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proc. of SIGMETRICS*, 2001.

[8] A. Bar-Noy, M. M. Halldórsson, G. Kortsarz, R. Salman, and H. Shachnai. Sum multicoloring of graphs. *J. Algorithms*, 2000.

[9] T. Bonald and L. Massoulié. Impact of fairness on Internet performance. In *Proc. of SIGMETRICS*, 2001.

[10] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *Proc. of INFOCOM*, 2013.

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

[12] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris. *Fundamentals of Queueing Theory.* Wiley-Interscience, New York, NY, USA, 4th edition, 2008.

[13] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of SIGCOMM*, 2012.

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

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

[16] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proc. of the SIGCOMM*, 2011.

[17] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.

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

[19] M. Verloop, S. Borst, and R. Núñez Queija. Stability of size-based scheduling disciplines in resource-sharing networks. *Perform. Eval.*, 62(1-4), 2005.

[20] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proc. of SIGCOMM*, 2011.

[21] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. of SIGCOMM*, 2012.

# Appendices

## A FCT Derivation for Queueing Model in Section 6.1

In this section we present the FCT calculation for the queueing model presented in §6.1. Specifically, we compute the average normalized FCT (FCT divided by flow size) for the general case with an arbitrary number of priority queues as a function of the flow size thresholds used for each queue. The result for two queues was given in Eq. (1).

Our approach is to derive the overall average normalized FCT by computing the average FCT for a flow in each priority queue. Since each queue is simply modeled as a $M/G/1$ queue, this can be done using the Pollaczek-Khintchine formula [12]. Recall that the Pollaczek-Khintchine formula provides the average waiting time for a $M/G/1$ queue:

$$\mathbb{E}(W) = \frac{\lambda \mathbb{E}(S^2)}{2(1-\rho)},$$

where $\lambda$ is the job arrival rate, $\mathbb{E}(S^2)$ is the second moment of the job service time distribution, and $\rho$ is the traffic load.

We begin with some definitions:

- $\boldsymbol{\lambda}$ : Overall flow arrival rate (flows arrive according to a Poisson process of rate $\lambda$).
- $\boldsymbol{F_S(\cdot)}$ : Flow size CDF. For simplicity, we assume $F_S(\cdot)$ is continuous and has a PDF denoted by $\boldsymbol{f_S(\cdot)}$.
- $\boldsymbol{\rho}$ : Overall traffic load, equal to $\lambda\mathbb{E}(S) < 1$ (the link capacity is 1).
- $\boldsymbol{k}$ : Number of priority queues ($k \geq 2$).
- $\underline{\boldsymbol{t}} = \boldsymbol{(t_0, t_1, ..., t_{k-1}, t_k)}$ : Vector of flow size thresholds corresponding to each priority queue. Flows with size $t_{i-1} < x \leq t_i$ are assigned to queue $i$ (for $1 \leq i \leq k$). Here $t_0 \triangleq 0$ and $t_k \triangleq \infty$ are defined for convenience.
- $\boldsymbol{\lambda_i}$ : Flow arrival rate to priority queue $i$. Since the $i^{th}$ queue receives the flows with size in $(t_{i-1}, t_i]$, it is evident that:

$$\lambda_i = \lambda(F_S(t_i) - F_S(t_{i-1})). \tag{2}$$

  Note that the arrival process to each queue is also Poisson.

- $\boldsymbol{\mu_i}$ : Service rate for priority queue $i$. This is set to the *remaining bandwidth* after queues $j < i$ (which have higher priority) have been served. Hence:

$$\mu_i = 1 - \rho \sum_{j=1}^{i-1} B_S^j, \tag{3}$$

  where

$$B_S^j = \frac{\int_{t_{j-1}}^{t_j} x f_S(x)\,\mathrm{d}x}{E(S)} \tag{4}$$

  is the *fraction of traffic* contained in flows with size in $(t_{j-1}, t_j]$.

- $\boldsymbol{\rho_i}$ : Traffic load for priority queue $i$, given by:

$$\rho_i = \frac{\rho B_S^i}{\mu_i}. \tag{5}$$

We can now use the Pollaczek-Khintchine formula to compute the average waiting time for a flow in priority queue $i$ (the time it takes for a new flow in queue $i$ to begin service):

$$\begin{aligned}\mathbb{E}(W_i) &= \frac{\lambda_i}{2(1-\rho_i)} \int_{t_{i-1}}^{t_i} \left(\frac{x}{\mu_i}\right)^2 \frac{f_S(x)}{F_S(t_i) - F_S(t_{i-1})} \, \mathrm{d}x \\ &= \frac{\lambda}{2(1-\rho_i)\mu_i^2} \int_{t_{i-1}}^{t_i} x^2 f_S(x) \, \mathrm{d}x \\ &= \frac{\lambda}{2(\mu_i - \rho B_S^i)\mu_i} \int_{t_{i-1}}^{t_i} x^2 f_S(x) \, \mathrm{d}x.\end{aligned}$$

Here the first step uses Eq. (2) and the second step follows from Eq. (5). Now, using Eq. (3), we have $\mu_i - \rho B_S^i = \mu_{i+1}$, and it follows that:

$$\mathbb{E}(W_i) = \frac{\lambda}{2\mu_{i+1}\mu_i} \int_{t_{i-1}}^{t_i} x^2 f_S(x) \, \mathrm{d}x. \tag{6}$$

Recall that we wish to compute the overall average normalized flow completion time (FCT divided by flow size). This is given by:

$$\mathbb{E}(FCT_n) = \sum_{i=1}^{k} \int_{t_{i-1}}^{t_i} \left(\frac{y}{\mu_i} + \mathbb{E}(W_i)\right) \frac{f_S(y)}{y} \, \mathrm{d}y, \tag{7}$$

where the $y/\mu_i$ term denotes the time it takes to complete a flow of size $y$ in queue $i$ after its service begins. Plugging in Eq. (6), we obtain:

$$\begin{aligned}\mathbb{E}(FCT_n) &= \sum_{i=1}^{k} \left( \int_{t_{i-1}}^{t_i} \frac{f_S(y)}{\mu_i} \, \mathrm{d}y + \frac{\lambda}{2\mu_{i+1}\mu_i} \int_{t_{i-1}}^{t_i} x^2 f_S(x) \, \mathrm{d}x \int_{t_{i-1}}^{t_i} \frac{f_S(y)}{y} \, \mathrm{d}y \right), \\ &= \sum_{i=1}^{k} \left( \frac{F_S(t_i) - F_S(t_{i-1})}{\mu_i} + \frac{\lambda}{2\mu_{i+1}\mu_i} \int_{t_{i-1}}^{t_i} x^2 f_S(x) \, \mathrm{d}x \int_{t_{i-1}}^{t_i} \frac{f_S(y)}{y} \, \mathrm{d}y \right). \end{aligned} \tag{8}$$

This completes the derivation. Note the for two priority queues ($k = 2$), we recover Eq. (1) by setting $t_0 = 0, t_1 = t, t_2 = \infty, \mu_1 = 1, \mu_2 = 1 - \rho B_S(t), \mu_3 = 1 - \rho$.

As demonstrated in §6.1, given the flow size distribution, $F_S(\cdot)$, and the desired number of priority queues, $k$, we can numerically compute Eq. (8) for different threshold values, $\underline{t}$, and traffic intensities, $\rho$, to find the optimal thresholds that minimize the overall average normalized FCT (see Figure 22).