pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric

Peter X. Gao
petergao@berkeley.edu

Akshay Narayan
akshay@berkeley.edu

Gautam Kumar
gautamk@berkeley.edu

Rachit Agarwal
ragarwal@berkeley.edu

Sylvia Ratnasamy
sylvia@berkeley.edu

Scott Shenker
shenker@berkeley.edu

University of California at Berkeley

ABSTRACT

The importance of minimizing flow completion times (FCT) in datacenters has led to a growing literature on new network transport designs. Of particular note is pFabric, a protocol that achieves near-optimal FCTs. However, pFabric's performance comes at the cost of generality, since pFabric requires specialized hardware that embeds a specific scheduling policy within the network fabric, making it hard to meet diverse policy goals. Aiming for generality, the recent Fastpass proposal returns to a design based on commodity network hardware and instead relies on a centralized scheduler. Fastpass achieves generality, but (as we show) loses many of pFabric's performance benefits.

We present `pHost`, a new transport design aimed at achieving both: the near-optimal performance of pFabric *and* the commodity network design of Fastpass. Similar to Fastpass, `pHost` keeps the network simple by decoupling the network fabric from scheduling decisions. However, `pHost` introduces a new distributed protocol that allows end-hosts to directly make scheduling decisions, thus avoiding the overheads of Fastpass's centralized scheduler architecture. We show that `pHost` achieves performance on par with pFabric (within 4% for typical conditions) and significantly outperforms Fastpass (by a factor of 3.8×) while relying only on commodity network hardware.

CCS Concepts

•**Networks** → **Transport protocols; Data center networks;**

CoNEXT '15 December 01–04, 2015, Heidelberg, Germany

DOI: 10.1145/2716281.2836086

Keywords

Datacenter network; Packet transport; Flow scheduling

1. INTRODUCTION

Users of Internet services are extremely sensitive to delays. Motivated by this, there has been a tremendous effort recently to optimize network performance in modern datacenters. Reflecting the needs of datacenter applications, these efforts typically focus on optimizing the more application-centric notion of flow completion time (FCT), using metrics such as a flow's *slowdown* which compares its FCT against the theoretical minimum (flow size in bytes divided by the access link bandwidth).

Recent research has produced a plethora of new datacenter transport designs [5,6,8,12–14,18]. The state-of-the-art is pFabric [6] that achieves close to theoretically minimal slowdown over a wide variety of workloads. However, to achieve this near-optimal performance, pFabric requires specialized network hardware that implements a specific packet scheduling and queue management algorithm[1]. There are two disadvantages to this: (i) pFabric cannot use commodity hardware, and (ii) pFabric's packet scheduling algorithm cannot be altered to achieve policy goals beyond minimizing slowdown — such goals may become relevant when the datacenter is shared by multiple users and/or multiple applications.

Countering this use of specialized hardware, the Fastpass proposal [14] uses commodity switches coupled with a flexible and fine-grained (close to per-packet) central scheduler. While this allows the network fabric to remain sim-

[1]Specifically, in pFabric, each packet carries the number of currently remaining (that is, un-ACKed) bytes in the packet's flow. A pFabric switch defines a flow's priority based on the packet from that flow with the smallest remaining number of bytes. The switch then schedules the oldest packet from the flow with the highest priority. Note that, within a flow, the oldest packet may be different from the packet with the fewest remaining bytes since packets transmitted later may record fewer remaining bytes.

ple and policy-agnostic, the resulting performance is significantly worse than that achieved by pFabric, especially for short flows (§4). In this paper, we ask: *Is it possible to achieve the near-ideal performance of pFabric using commodity switches?*

We answer this question in the affirmative with `pHost`, a new datacenter transport design that is simple and general — requiring no specialized network hardware, no per-flow state or complex rate calculations at switches, no centralized global scheduler and no explicit network feedback — yet achieves performance surprisingly close to pFabric.

In the next section, we provide the rationale for and overview of the `pHost` design. We provide the specifics of our design in §3. In §4 we evaluate `pHost`'s performance, comparing it against pFabric [6] and Fastpass [14]. We discuss related work in §5 and close the paper with a few concluding comments in §6.

2. PHOST OVERVIEW

We start with a brief review of modern datacenter networks (§2.1). We then describe the key aspects of `pHost`'s design (§2.2), and close the section with an intuitive description of why `pHost`'s approach works (§2.3).

2.1 Modern Datacenter Networks

Modern datacenter networks differ from traditional WAN networks in several respects.

- *Small RTTs*: The geographic extent of datacenter networks is quite limited, so the resulting speed-of-light latencies are small. In addition, switch forwarding latencies have dropped as cut-through switching has become common in commodity switches.
- *Full bisection bandwidth*: By using topologies such as Fat-Tree [3] or VL2 [11], datacenter networks now provide full bisection bandwidth [2, 16].
- *Simple switches*: Datacenter switches tend to be relatively simple (compared to high-end WAN routers). However, they do provide some basic features: a few priority levels (typically 8–10 [1, 5, 12]), ECMP and/or packet spraying (that is, randomized load balancing on a per-flow and/or per-packet basis [1, 10]), cut-through switching, and relatively small buffers.

`pHost` both assumes and exploits these characteristics of datacenter networks, as we elaborate on in §2.3.

2.2 Basic Transport Mechanism

We now provide a high-level overview of `pHost`'s transport mechanism. `pHost` is built around a host-based scheduling mechanism that involves requests-to-send (RTS), per-packet token assignment, and receiver-based selection of pending flows. These techniques have their roots in wireless protocols (e.g., 802.11) and capability-based DDoS mechanisms (e.g., SIFF [19]). Specifically:

- Each source end-host, upon a flow arrival, sends a request to send (RTS) packet to the destination of the flow. The RTS may contain information relevant for making scheduling decisions (such as flow's size, or which tenant the source belongs to, etc.).
- Once every packet transmission time, each destination end-host considers the set of pending RTSs (that is, RTSs for flows that still have bytes to send) and sends a "token" to one of the corresponding sources. The token allows the source to transmit one data packet from that flow, and may specify the priority level at which the packet is to be sent at. Thus, each destination host performs *per-packet* scheduling across the set of active flows *independent* of other destinations.
- Tokens expire if the source has not used the token within a short period after receiving the token (default being 1.5× MTU-sized packet transmission time). Each source may also be assigned a few "free tokens" for each flow, which need not be sent from the destination host.
- After each packet transmission, a source selects one of its unexpired tokens and sends the corresponding data packet. Thus, each source host also performs a selection across its set of active flows *independent* of our sources.
- Once the destination has received all data packets for a flow, it sends an ACK packet to the source.
- All control packets (RTS, tokens, ACKs) are sent at the highest priority.

`pHost`'s design has several degrees of freedom: the scheduling at the destination (which flows to send next token to), the scheduling at the source (which token to use after each packet transmission), the priority level at which each data packet is sent at, and the number of free tokens assigned to the sources. These can be configured to achieve different performance goals, without any modification in the network fabric. For instance, we demonstrate later that `pHost` is competitive with pFabric when the above degrees of freedom are configured to globally minimize slowdown, but can also be configured to optimize for performance metrics other than slowdown (*e.g.*, meeting flow deadlines, or achieving fairness across multiple tenants, etc.).

2.3 Why pHost works

Similar to prior proposals (e.g., pFabric [6]), we utilize the packet-spraying feature found in many commodity switches (in which packets are spread uniformly across the set of available routes) [1, 10]. Intuitively, using packet-spraying in a full-bisection-bandwidth network can eliminate almost all congestion in the core (§4), so we do not need sophisticated path-level scheduling (as in Fastpass) nor detailed packet scheduling in the core switches (as in pFabric). However, we do make use of the few levels of priority available in commodity switches to ensure that signaling packets suffer few drops.

While there is no congestion in the core, there can still be congestion at the destination host if multiple sources are sending flows to the destination at the same time. In pHost, this comes down to how the destination grants tokens in response to RTS requests from sources. Choosing flows that should be assigned tokens at any time is effectively a bipartite matching problem. A centralized scheduler could compute a match based on a global view (akin to how early routers managed their switch fabrics) but this would incur the complexity of a scalable centralized scheduler and the latency overhead of communication with that scheduler (§4). In pHost, we instead use a fully decentralized scheduler (once again, akin to router scheduler designs such as PIM [7] and iSlip). The resulting match may be imperfect, but we compensate for this in two ways. To avoid starvation at the source (if a destination does not respond with a token), we allow the source to launch multiple RTSs in parallel. Each source is also given a small budget of free tokens for each flow; this also allows sources to start sending without waiting for the RTT to hear from the destination. To avoid starvation at the destination (e.g., when a source does not utilize the token it was assigned), we use a back-off mechanism where (for a short time) a destination avoids sending tokens to a particular source if the source has not used the tokens it was recently assigned by the destination.

As we shall show, the combination of these techniques avoids starvation at the hosts, and allow pHost to achieve good network utilization despite a fully decentralized host-based scheduler.

3. DESIGN DETAILS

We now describe the details of pHost's design. At a high level, there are two main components to pHost's design: (i) the protocol that dictates *how* sources and destinations communicate by exchanging and using RTS, token and data packets, and (ii) the scheduling policy that dictates *which* sources and destinations communicate.

We start by describing the protocol that end-hosts implement (§3.1) and then elaborate on how this protocol ensures high network utilization (§3.2). We then describe how pHost supports flexible scheduling policies (§3.3). Finally we describe how pHost achieves reliable transmission in the face of packet drops (§3.4).

3.1 pHost Source and Destination

pHost end-hosts run simple algorithms for token assignment and utilization. The algorithm for the source is summarized in Algorithm 1. When a new flow arrives, the source immediately sends an RTS to the destination of the flow. The RTS may include information regarding the flow (flow size, deadline, which tenant the flow belongs to, etc) to be used by the destination in making scheduling decisions. The source maintains a per-flow list of "tokens" where each token represents the permission to send one packet to the flow's destination; we refer to this as the *ActiveTokens* list. The ActiveTokens list is initialized with a configurable number of "free tokens" (we elaborate on the role of free tokens in §3.2); all subsequent tokens can only be explicitly granted by the destination in response to an RTS.

Algorithm 1 pHost algorithm at Source.

```
if new flow arrives then
    Send RTS
    ActiveTokens ← FreeTokens          ▷ Add free tokens (§3.2)
else if new token T received then
    Set ExpiryTime(T)      ▷ Tokens expire in fixed time (§3.2)
    ActiveTokens ← T
else if idle then
    T = Pick(ActiveTokens)          ▷ pick unexpired token (§3.3)
    Send Packet corresponding to T
end if
```

When a source receives a token from the destination, it adds this token to its ActiveTokens list. Each token has an associated expiry time and the source is only allowed to send a packet if it holds an unexpired token for that packet (again, we elaborate on the role of token timeouts in §3.2). Whenever a source is idle, it selects a token from ActiveTokens based on the desired policy goals (§3.3) and sends out the packet for the corresponding token.

The high-level algorithm used at the destination is summarized in Algorithm 2. Each destination maintains the set of flows for which the destination received an RTS but has not yet received all the data packets; we refer to this as the *PendingRTS* list. When the destination receives a new RTS, it adds the RTS to PendingRTS immediately. Every (MTU-sized) packet transmission time, the destination selects an RTS from PendingRTS list based on the desired policy goals (§3.3) and sends out a token to the corresponding source. The token contains the flow ID, the packet ID and (optionally) a priority value to be used for the packet. As at the source, each token has an associated expiry time (we use a value of $1.5 \times$ MTU-sized packet transmission time). A token assigned by the destination is considered revoked if it is not utilized within its expiry time. This avoids congestion at the destination as sources cannot use tokens at arbitrary times. In addition, the destination maintains a count of the number of expired tokens for the flow (the difference between the number of tokens assigned and the number of packets received). If this count exceeds a threshold value, the flow is marked as "downgraded" which lowers the likelihood it will be granted tokens in the immediate future; we elaborate on the details and purpose of downgrading in §3.2. Finally, once the destination has received all the packets for a flow, it sends an ACK to the source and removes the RTS from PendingRTS list.

Algorithm 2 pHost algorithm at Destination.

```
if receive RTS R then
    PendingRTS ← R
else if idle then
    F = Pick(PendingRTS)              ▷ Pick an active flow (§3.3)
    Send Token T for flow F
    if F.#ExpiredTokens > Threshold then                ▷ (§3.2)
        Downgrade F for time t*                         ▷ (§3.2)
    end if
else if Received data for token T then
    Set Token T as responded
end if
```

Note that all control packets (RTS, token, ACK) in `pHost` are sent at the highest priority. Similar to TCP where each data packet generates an ACK packet, `pHost` uses one token per data packet. In addition, `pHost` uses just one RTS and one ACK packet per flow. All control packets in `pHost` are of 40 bytes; thus, the bandwidth and end-to-end latency overheads of control packets in `pHost` are minimal.

3.2 Maximizing Network Utilization

As discussed earlier, packet spraying in a full-bisection bandwidth network eliminates almost all congestion in the core. However, sources sending multiple RTSs in parallel (Algorithm 1) and destinations assigning one token per packet transmission time (Algorithm 2) may lead to network underutilization. For a centralized scheduler (as in Fastpass), avoiding this underutilization is easy since the scheduler has a global view of the network. To achieve high network utilization in a fully decentralized manner, however, `pHost` has to resolve two challenges. We discuss these challenges and how `pHost` resolves these challenges.

Free tokens for new flow arrivals. Recall that upon a flow arrival, the source immediately sends an RTS to the corresponding destination. The first challenge is that the bandwidth at the source may be wasted until the token for the flow has been received (even if the destination is free, source cannot send the packets). This may have particularly adverse effect on short flow performance, since such a wait may be unnecessary. `pHost` avoids this overhead by assigning each source a few "free tokens" per flow (akin to TCP's initial congestion window) that can be used by the source without receiving any tokens from the corresponding destination.

Source downgrading and token expiry. To understand the second challenge, let us consider the case of Figure 1. When the source in the above example prefers the token for flow B in the second time unit, the bandwidth at the destination for flow A is wasted. Even worse, another source may have a flow C to send to the destination of flow A, but the destination continues sending tokens to the source of flow A, which in turn continues to prefer utilizing tokens for flow B. This may lead to long term wastage of bandwidth at both the destination of flow A and the source of flow C.

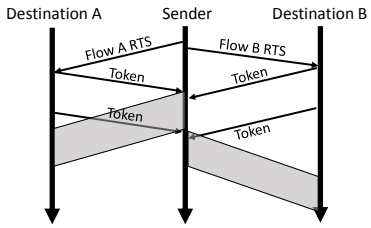


Figure 1: In this example, two flows A and B arrive at the source at roughly the same time, and two RTS are sent to the respective destinations. Suppose the source gets a token for flow A first. Since the source has only one token, it immediately consumes it by sending the corresponding data packet to destination A. Now suppose the source receives another token for flow A and a token for flow B while it is sending the data packet for flow A. The source now has two tokens, one for each flow. Suppose the source decides to utilize the token for flow B at this step.

`pHost` uses a source downgrading mechanism to prevent a destination from sending tokens to a source that does not respond with data packets. As mentioned earlier, `pHost` destinations maintain a count of the number of unexpired tokens for each source. If this count exceeds a threshold value in succession (default being a BDP worth of tokens), the destination *downgrades* the source and stops assigning tokens to that source. The source is left downgraded for a *timeout* period (default being 3× RTT). After the timeout period, the destination resends tokens to the source for the packets that were not received.

3.3 Local Scheduling Problem

Datacenter network operators today have to contend between a rich mix of tenants and applications sharing the network, each of which may have a different performance goal. For instance, the goal in some scenarios may be to optimize for tail latency (*e.g.*, web search and social networking) across all flows. In other scenarios (*e.g.*, multiple tenants), the goal may be to share network bandwidth fairly among the tenants. As `pHost` implements scheduling at end-hosts, it naturally provides algorithmic flexibility in optimizing for various performance metrics. In this subsection, we describe how this enables flexibility in terms of network resource allocation between users and applications, and to optimize the network for a wide variety of performance metrics.

Recall, from §3.1, that the `pHost` sources send a RTS to the destination expressing their intent of sending packets upon each flow arrival. The sources embed the information related to the flow (*e.g.*, flow size, deadlines, etc) within the RTS packet. The destinations then assign tokens to the flows, optionally specifying a priority level to be used for the packets in the flow. We describe how this design enables optimizing for three different performance objectives using end-host scheduling: (i) minimizing flow completion time [5, 6, 8, 13, 14]; (ii) deadline-constrained traffic [6, 12, 18]; and (iii) fairness across multiple tenants.

The optimal algorithm for minimizing flow completion time when scheduling over a single link is Shortest Remaining Processing Time (SRPT) scheduling, which prioritizes the flow with the fewest remaining bytes. Transport protocols [6, 14] that achieve near-ideal performance emulate this policy over a distributed network fabric by prioritizing flows that have least number of packets remaining to complete the flow. `pHost` can emulate SRPT over a distributed network fabric using the same scheduling policy — each destination prioritizes flows with least number of remaining packets while assigning tokens; the destination additionally allows the sources to send short flows with the second highest priority and long flows with the third highest priority (recall, control packets are sent with the highest priority). Note that this is significantly different from pFabric, which assigns packet priority to be the remaining flow size. Similarly, the sources prioritize flows with the fewest number of remaining packets while utilizing tokens; the sources also use any free tokens when idle. We show in §4 that using this simple scheduling policy at the end-hosts, `pHost` achieves performance close to that of state-of-the-art protocols when minimizing flow completion time.

Next, we discuss how `pHost` enables optimizing for deadline-constrained traffic [6, 12, 18]. The optimal algorithm for scheduling deadline-constrained flows over a single link is Earliest Deadline First (EDF), which prioritizes the flow with the earliest deadline. `pHost` can emulate EDF by having each source specify the flow deadline in its RTS. Each destination then prioritizes flows with earliest deadline (analogous to the SRPT policy) when assigning tokens; the sources, as earlier, prioritize flows with earliest deadline when utilizing tokens.

Indeed, pFabric [6] can emulate the above two policies despite embedding the scheduling policies within the network fabric. We now discuss a third policy which highlights the necessity of decoupling scheduling from the network fabric. Consider a multi-tenant datacenter network where tenant A is running a web search workload (most flows are short) while tenant B is running a MapReduce workload (most flows are long). pFabric will naturally prioritize tenant A's shorter flows over tenant B's longer flows, starving tenant B. `pHost`, on the other hand, can avoid this starvation using its end-host based scheduling. Specifically, the destinations now maintain a counter for the number of packets received so far from each tenant and in each unit time assign a token to a flow from the tenant with smaller count. While providing fairness across tenants, `pHost` can still allow achieving the tenant-specific performance goals for their respective flows (implementing scheduling policies for each tenant's flows).

3.4 Handling Packet drops

As we show in §4, the core idea of `pHost` end-hosts performing per-packet scheduling to minimize congestion at their respective access links leads to negligible number of packet drops in full-bisection bandwidth datacenter networks. For the unlikely scenario of some packets being dropped, per-packet token assignment in `pHost` lends itself to an extremely simple mechanism to handle packet drops. In particular, recall from §3.1, that each destination in `pHost` assigns a token *to a specific packet* identifying the packet ID along with the token. If the destination does not receive one of the packets until a token has been sent out for the last packet of the flow (or timeout), the destination simply reissues a token for the lost packet when the flow has the turn to receive a token. The source, upon receiving the token, retransmits the lost packet(s).

4. EVALUATION

In this section, we evaluate `pHost` over a wide range of datacenter network workloads and performance metrics, and compare its performance against pFabric and Fastpass.

4.1 Test Setup

Our overall test setup is identical to that used in pFabric [6][2]. We first elaborate on this setup — the network topology, workload and metrics; we then describe the protocols we evaluate and the default test configurations we use.

Network Topology. We use the same network topology as in pFabric [6]. The topology is a two-tier multi-rooted tree with 9 racks and 144 end-hosts. Each end-host has a 10Gbps access link and each core switch has nine 40Gbps links; each network link has a propagation delay of 200ns. The resultant network fabric provides a full bisection bandwidth of 144Gbps. Network switches implement cut-through routing and packet spraying (functions common in existing commodity switches [1,6,10]). pFabric assumes each switch port has a queue buffer of 36kB; we use this as our default configuration value but also evaluate the effect of per-port buffer sizes ranging from 6kB-72kB.

Workloads. We evaluate performance over three production traces whose flow size distributions are shown in Figure 2. All three traces are heavy-tailed, meaning that most of the flows are short but most of the bytes are in the long flows. The first two – "Web Search" [5] and "Data Mining" [11]) – are the traces used in pFabric's evaluation. The third "IMC10" trace uses the flow size distributions reported in a measurement study of production datacenters [9]. The IMC10 trace is similar to the Data Mining trace except in the tail: the largest flow in the IMC10 trace is 3MB compared to 1GB in the Data Mining trace. In addition to production traces, we also consider a synthetic "bimodal" workload that we use to highlight specific performance characteristics of the three protocols; we elaborate on this workload inline in §4.3. As in prior work [6, 8], we generate flows from these workloads using a Poisson arrival process for a specified target network load. We consider target network loads ranging from 0.5 – 0.8.

[2]We would like to thank the pFabric authors for sharing their simulator with us; our simulator (https://github.com/NetSys/simulator) builds upon theirs.

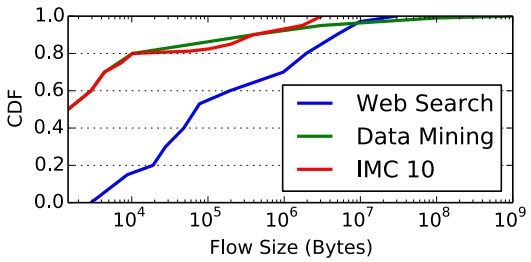


Figure 2: Distribution of flow sizes across workloads used in our evaluation. Note that short flows dominate all workloads; however, Data Mining and IMC10 workloads have significantly larger fraction of short flows when compared to the Web Search workload.

Traffic matrices. Our default traffic matrix is all-to-all, where each source host generates flows to each other host using above workloads. For completeness, in §4.3, we consider two additional traffic matrices. In permutation traffic matrix, each source sends flows to a single destination chosen uniformly at random without replacement. Finally, we also evaluate for an incast traffic matrix where each destination receives flows from a specified number of sources. We describe the setup for the latter two traffic matrices inline.

Performance metrics. The primary metric we focus on is that used in pFabric: mean *slowdown*, defined as follows: let `OPT(i)` be the flow completion time of flow `i` when it is the only flow in the network and let `FCT(i)` be the observed flow completion time when competing with other flows. Then, for flow `i`, the *slowdown* is defined as the ratio of `FCT(i)` and `OPT(i)`. Note that `FCT(i)` $\geq$ `OPT(i)`; thus, a smaller slowdown implies better performance. The mean and high percentile slowdowns are calculated accordingly across the set of flows in the workload. For completeness, in §4.3, we consider a range of additional metrics considered in previous studies, including "normalized" FCT, throughput, and the fraction of flows that meet their target deadlines; we define these additional metrics in §4.3.

Evaluated Protocols. We evaluate `pHost` against pFabric [6] and Fastpass [14]. For pFabric, we use the simulator provided by the authors of pFabric with their recommended configuration options: an initial congestion window of 12 packets, an RTO of 45μs, and the network topology described above. Unfortunately, a packet-level simulator is unavailable for Fastpass and hence we implemented Fastpass in our own simulator. Our implementation of Fastpass uses: (1) 40B control packets and an epoch size of 8 packets (Fastpass makes scheduling decisions every epoch with a recommended epoch interval of 8 MTU transmission times), (2) zero processing overhead at the centralized packet scheduler (i.e., we assume the scheduler solves the global scheduling problem infinitely fast); and (3) perfect time synchronization (so that all end-hosts are synchronized on epoch start and end times). Note that the latter two represent the *best-case* performance scenario for Fastpass.

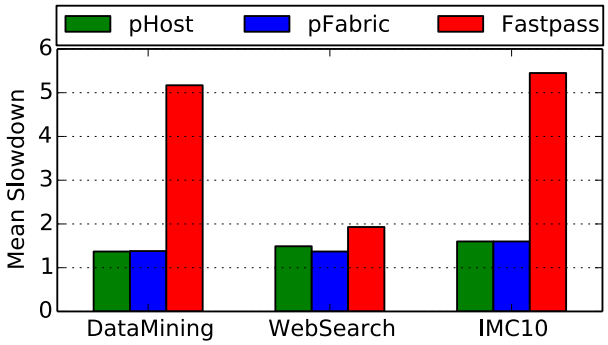


Figure 3: Mean slowdown of pFabric, `pHost`, and Fastpass across different workloads for our default configuration (0.6 load, per-port buffers of 36kB). `pHost` performs comparable to pFabric, and 1.3–4$\times$ better than Fastpass.

Default configuration. Unless stated otherwise, our evaluation use a default configuration that is based on an all-to-all traffic matrix with a network load of 0.6, a per-port buffer of 36kB at switches, and other settings as discussed above. For `pHost`, we set the token expiry time to be 1.5$\times$, source downgrade time to be 8$\times$ and timeout to be 24$\times$ MTU-sized packet transmission time (note that BDP for our topology is 8 packets). Moreover, we assign 8 free tokens to each flow. We evaluate the robustness of our results over a range of performance metrics, workloads, traffic matrices and parameter settings in §4.3.

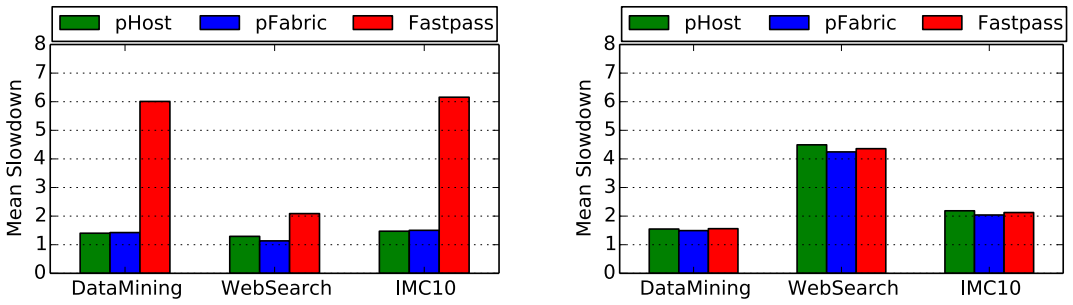
4.2 Performance Evaluation Overview

Figure 3 shows the mean slowdown achieved by each scheme for our three trace-based workloads. We see that the performance of `pHost` is comparable to that of pFabric. pFabric is known to achieve near-optimal slowdown [6]; hence these results show that `pHost`'s radically different design approach based on scheduling at the end-hosts is equally effective at optimizing slowdown.

Somewhat surprisingly, we see that slowdown with Fastpass is almost 4$\times$ higher than `pHost` and pFabric.[3] We can explain this performance difference by breaking down our results by flow size: Figure 4 shows the mean slowdown for short flows versus that for long flows. For long flows, all the three protocols have comparable performance; however, for short flows, both `pHost` and pFabric achieve significantly better performance than Fastpass. Since the three workloads contain approximately 82% short flows and 18% long flows, the performance advantage that pFabric and `pHost` enjoy for short flows dominates the overall mean slowdown.

That pFabric and `pHost` outperform Fastpass for short flows is (in retrospect) not surprising: Fastpass schedules flows in epochs of 8 packets, so a short flow must wait for at least an epoch ($\sim$ 10μs) before it gets scheduled. Further, the signaling overhead of control packets adds another round trip of delay before a short flow can send any packet. Nei-

[3]Note that the evaluation in [14] does not compare the performance of Fastpass to that of pFabric.



(a) Short flows: Mean Slowdown

(b) Long flows: Mean Slowdown

Figure 4: Breakdown of mean slowdown by flow size for pFabric, `pHost`, and Fastpass (all flows greater than 10MB (for Data Mining and Web Search workloads) and greater than 100KB (for IMC10 workload) are considered long flows). All the three schemes have similar performance for long flows; for short flows, however, `pHost` performs similar to pFabric and 1.3–4× better than Fastpass.

ther pFabric nor `pHost` incur this overhead on flow arrival. That all three protocols have comparable performance for long flows is also intuitive because, for a long flow, the initial waiting time in Fastpass (one epoch and one round trip time) is negligible compared to its total FCT.

The above results show that `pHost` can match the near-optimal performance of pFabric (without requiring specialized support from the network fabric) and significantly outperforms Fastpass (despite lacking the global view in scheduling packets that Fastpass enjoys). Next, we evaluate whether the above conclusions hold for a wider range of workloads, performance metrics and traffic matrices.

4.3 Varying Metrics and Scenarios

We now evaluate `pHost`– and how `pHost` compares to pFabric and Fastpass – over varying performance metrics, network load, traffic matrices, etc.

Varying Performance Metrics. Our evaluation so far has focused on mean slowdown as our performance metric. We now evaluate performance using five additional metrics introduced in prior work: (i) normalized flow completion time [8, 12–14], defined as ratio of the mean of `FCT(i)` and the mean of `OPT(i)`; (ii) network throughput, measured as the number of bytes delivered to receivers through the network over unit time normalized by the access link bandwidth; (iii) the 99 percentile in slowdown [6]; (iv) for deadline-constrained traffic, the fraction of flows that meet deadlines [6, 12, 18]; and (v) packet drop rates. Figure 5 shows our results using the above metrics.

NFCT. Figure 5(a) shows that all three protocols see similar performance as measured by NFCT; across all evaluated cases, the maximum difference in NFCT between any two protocols is 15%. This similarity is simply because the NFCT metric, as defined, is (unlike mean slowdown) dominated by the FCT of long flows. FCT for long flows is in turn dominated by the time to transmit the large number of bytes involved, which is largely unaffected by protocol differences and hence all three protocols have similar performance.

Throughput. The results for throughput (shown in Figure 5(b)) follow trend similar to NFCT results, again because overall throughput is dominated by the performance of long flows. Note that the network load (rate at which packets arrive at the sources) matches the throughput or network utilization (rate at which packets arrive at the destination) when the slowdown is 1. Since this is not the case, we expect the throughput to be less than the load; that is less than 6Gbps for 0.6 network load over a topology with 10Gbps access link bandwidth.

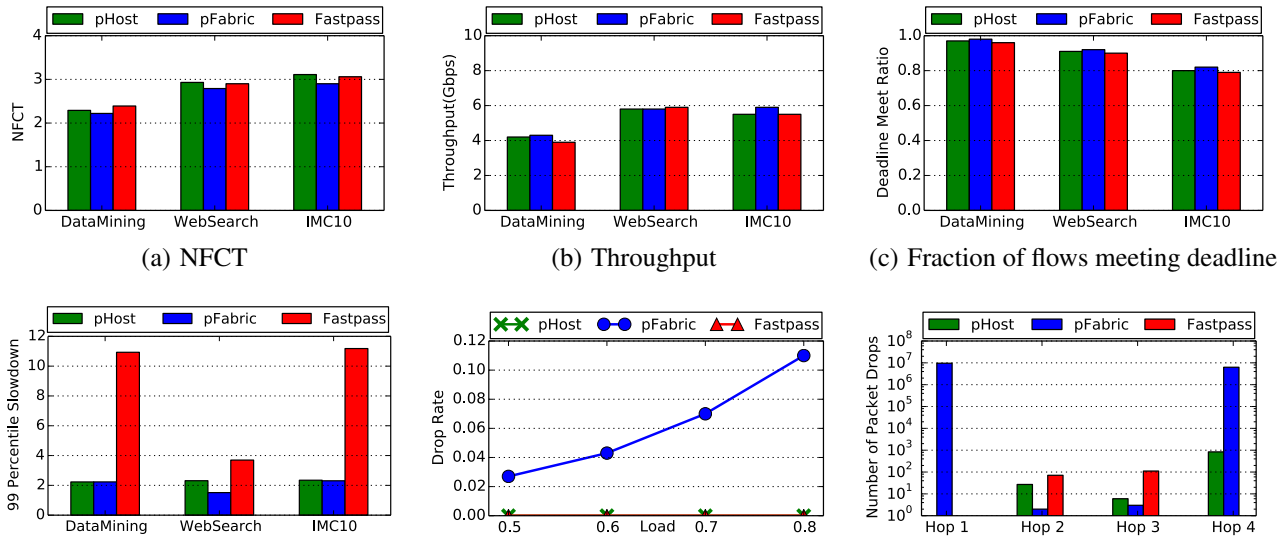
Deadlines. We now evaluate the performance of the three protocols over deadline-constrained traffic (Figure 5(c)). We assign a deadline to each flow using exponential distribution with mean 1000μs [6]; if the assigned deadline is less than 1.25× the optimal FCT to a flow, we set the deadline for that flow to be 1.25× its optimal FCT. We observe that all protocols achieve similar performance in terms of fraction of flows that meet their deadlines, with the maximum difference in performance between any two protocols being 2%.

We conclude that for applications that care about optimizing NFCT, throughput or deadline-constrained traffic, all three protocols offer comparable performance. The advantage of `pHost` for such applications lie in considerations other than performance: that `pHost` relies only on commodity network fabrics and that `pHost` avoids the engineering challenges associated with scaling a centralized controller.

Tail latency and drop rates. We now evaluate the three protocols for two additional performance metrics: tail latency for short flows and the packet drop rate.

99%ile Slowdown. Prior work has argued the importance of tail performance in datacenters and hence we also look at slowdown at the 99-percentile, shown in Figure 5(d). We see that for both pHost and pFabric, the 99%ile slowdown is around 2 (roughly 33% higher than the mean slowdown), while for Fastpass the slowdown increases to almost 2× the mean slowdown.

Drop rate. We now measure the drop rates for the three protocols. By design, one would expect to see very different behavior in terms of packet drops between pFabric and the



(a) NFCT (b) Throughput (c) Fraction of flows meeting deadline

(d) 99%-ile slowdown for short flows (e) Drop rate (f) Packet drops across hops

Figure 5: Performance of the three protocols across various performance metrics. See §4.3 for detailed discussion.

other two protocols — `pHost` and Fastpass. Indeed, pFabric is deliberately aggressive in sending packets, expecting the network to drop low priority packets in large numbers; in contrast, pHost and Fastpass explicitly schedule packets to avoid drops. Figure 5(e) shows the overall drop rate of each protocol under increasing network load for the Web Search workload. As expected, we see that pFabric has a high drop rate that increases with load while `pHost` and Fastpass see drop rates consistently close to zero even as load increases.

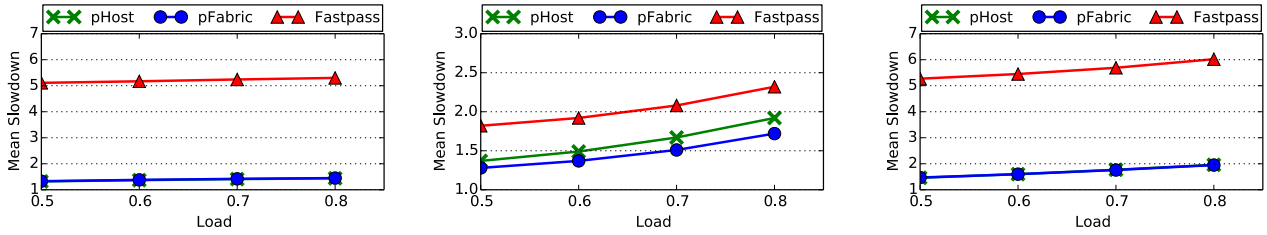
Figure 5(f) shows *where* drops occur in the network: we plot the absolute number of packet drops at each of the 4 hops in the network (end-host NIC queue, the aggregation switch upstream queue, the core switch queue, and the aggregation switch downstream queue); 511 million packets are injected into the network over the duration of the simulation (network load being 0.6). We see that for pFabric, the vast majority of packet drops occur in the first (61%) and last (39%) hop queue, with almost no drops in the two intermediate hops. In contrast, because pHost and Fastpass explicitly schedule packets, first and last hop drops are almost eliminated: both protocols experience zero drops at the first hop, the number of last hop drops for pHost and Fastpass are 836 and 0 packets respectively.

Finally, we note that the absolute number of drops within the network fabric is low for all three protocols: 33, 5 and 182 drops for pHost, pFabric, and Fastpass respectively, which represent less than 0.00004% of the total packets injected into the network. This confirms our intuition that full bisection bandwidth networks together with packet spraying avoids most congestion (and hence the need for careful scheduling) within the network fabric.

Varying network load. Our evaluation so far used traffic generated at 0.6 network load. We now evaluate protocol performance for network load varying from 0.5–0.9, as reported in prior work. Figure 9 presents our results across the three workloads and protocols.

We observe that the relative performance of the different protocols across different network loads remains consistent with our results from above. This is to be expected as the distribution of short versus long flows remains unchanged with varying load.

We also note that, in all cases, performance degrades as network load increases. Closer examination revealed that in fact the overall network becomes *unstable* at higher loads; that is, with the network operating in a regime where it can no longer keep up with the generated input load. In Figure 7, we aim to capture this effect. In particular, the x-axis plots the fraction of packets (out of the total number of packets over the simulation time) that have arrived at the source as the simulation progresses; the y-axis plots the fraction of packets (again, out of the total number of packets across the simulation time) that have arrived at the source but have not yet been injected into the network by the source ("pending" packets). In a stable network, the fraction of pending packets would remain roughly constant over the duration of the simulation showing that the sources inject packets into the network at approximately the same rate at which they arrive. We observe that, at 0.6 load, this number does in fact remain roughly constant over time. However, at higher load, this number increases as the simulation progresses, indicating that packets arrive faster than the rate at which sources can inject then into the network. Measuring slowdown when operating in this unstable regime is unwise since the measured value depends closely on the duration of the simulation (e.g., in our experiments at 0.8 load, we could obtain arbitrarily high slowdowns for pFabric by simply tuning the



(a) Data Mining (b) Web Search (c) IMC10

Figure 6: Performance of the three protocols across varying network loads.

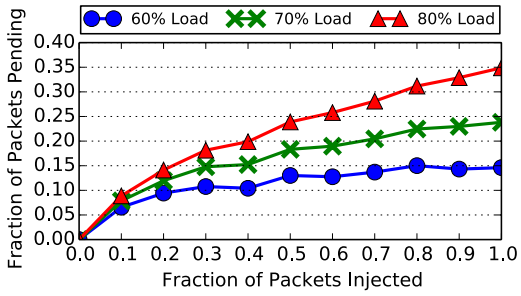


Figure 7: Stability analysis for pFabric. x-axis is the fraction of packets (out of total number of packets across the simulation) that have arrived at the source as the simulation progresses; y-axis is the fraction of packets (again, out of total number of packets across the simulation) that have yet not been injected into the network by the sources. pFabric is stable at 0.6 load, unstable beyond 0.7 load. We get similar results for `pHost` and Fastpass.

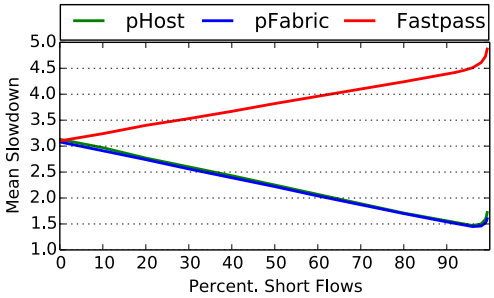


Figure 8: Mean slowdown of `pHost`, pFabric, and Fastpass in synthetic workload (with varying fraction of short flows). Both pFabric and `pHost` perform well when the trace is short flow dominated. Fastpass performs similar to `pHost` and pFabric when there are 90% long flows, but gets significantly worse as the fraction of short flows increases.

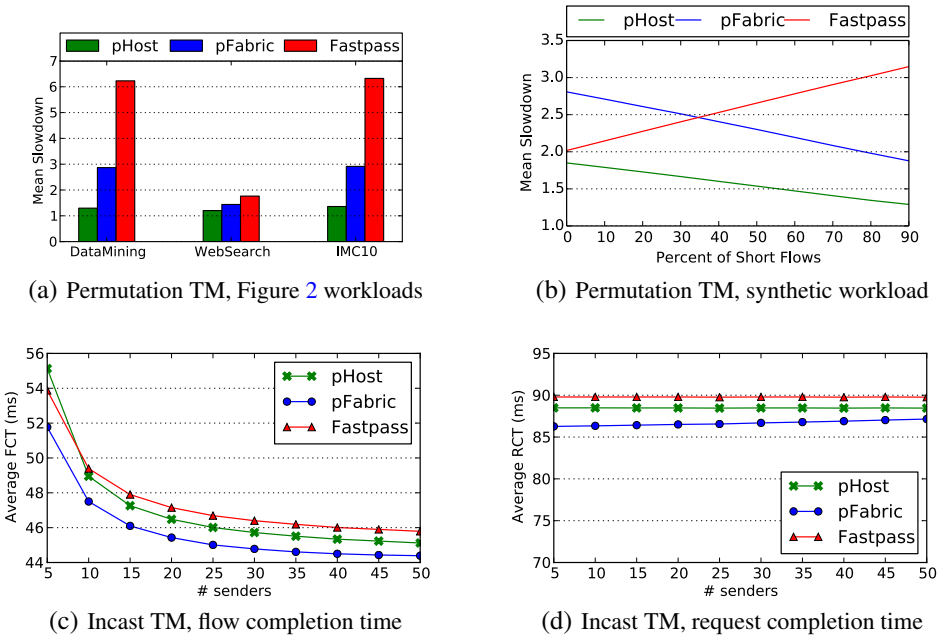
length of the simulation run). This is the reason we select a network load of 0.6 as our default configuration (compared to the 0.8 used in pFabric).

New Workloads. Our results so far were based on three traces used in prior work. While these reflect existing production systems, network workloads will change as applications evolve and hence we sought to understand how `pHost`'s performance will change for workloads in which the ratio of short *vs.* long is radically different from that observed today. We thus created a synthetic trace that uses a simple bimodal distribution with short (3 packet) flows and long (700 packet) flows and vary the fraction of short flows from 0% to 99.5%. We show the corresponding mean slowdown in Figure 8. We make two observations. The first is that `pHost` once again matches pFabric's performance over the spectrum of test cases. The second — and perhaps more interesting — observation is that the absolute value of slowdown (for all protocols) varies significantly as the distribution of short vs. long flows changes; for pFabric and pHost, the traces based on current workloads occupy the "sweet spot" in the trend. This shows that although pFabric and our own `pHost` achieve near-optimal performance (i.e., mean slowdown values close to 1.0) for existing traces, this is not the case for radically different workloads. Whether and how one might achieve better performance for such workloads remains an open question for future work.

Varying traffic matrices. Our evaluation so far has focused on all-to-all traffic matrix (TM). We now evaluate the performance of `pHost` for two other traffic matrices — permutation TM and incast TM.

In permutation TM, we match each source to a unique destination and all flows from the source are generated (using the workloads described earlier) only for that destination. Figure 9(a) and Figure 9(b) present results for the permutation TM for the three workloads from Figure 2 and for the synthetic workload (with varying fraction of short flows). We note that `pHost` significantly outperforms both pFabric and Fastpass for the permutation TM across all workloads. Note that for permutation TM, there is little or no contention across flows in the core. Thus, per-packet scheduling as in `pHost` and Fastpass performs significantly better than pFabric for long flows (extreme left point in Figure 9(b)). For short flows, Fastpass performance degrades due to centralized scheduler overheads; however, `pHost` avoids such overheads due to distributed scheduling and due to low contention across flows to the same destination. Overall, thus, `pHost` performs better than both pFabric and Fastpass across all workloads.

For incast TM, we use a setup similar to pFabric [6]. Specifically, for each incast "request", one destination receives a fixed amount of data from N sources chosen uniformly at random across the hosts in the network topology. We set the amount of data to be 100MB and vary N from 5–



(a) Permutation TM, Figure 2 workloads

(b) Permutation TM, synthetic workload

(c) Incast TM, flow completion time

(d) Incast TM, request completion time

Figure 9: Performance of the three protocols across various traffic matrices. `pHost` performs better than pFabric and Fastpass for Permutation TM, and within 5% of pFabric for incast TM.

50, with each source generating 100MB$/N$ amount of data in a single flow. We repeat this for 10000 requests. Figure 9(c) shows the average FCT for the incast TM. We note that each of `pHost`, pFabric and Fastpass perform within 7% of each other. Figure 9(d) shows the average request completion time (RCT) of each scheme. Similar to FCT, the performance difference between `pHost`, pFabric and Fastpass is less than 4%, and varying N (number of sources simultaneously sending to the receiver) has negligible impact on RCT. We also conducted experiments with varying amount of data (100–1000MB); while absolute numbers change, the trends remain the same. We conclude that each of these protocols have comparable performance for incast TM.

Varying switch parameters. We evaluate the impact of varying the per-port buffer size in switches. Figure 10 shows the mean slowdown with increasing switch buffer sizes for our Data Mining workload.[4] We see that none of the three schemes is sensitive to the sizing of switch buffers, varying less than 1% over the range of parameters evaluated even with tiny 6.2 KB buffers. We also evaluated variation of throughput with switch buffer sizes, for a workload with 100% long flows and 0.6 network load. The results for that evaluation were very similar with each protocol observing very little impact due to buffer sizes.

[4]For small buffer sizes ($<$ 36kB) pFabric's performance degrades if we use the default values for its parameters (initial congestion window and retransmission timeout). Hence, for each buffer size, we experimented with a range of different parameter settings for pFabric and select the setting that offers the best slowdown.

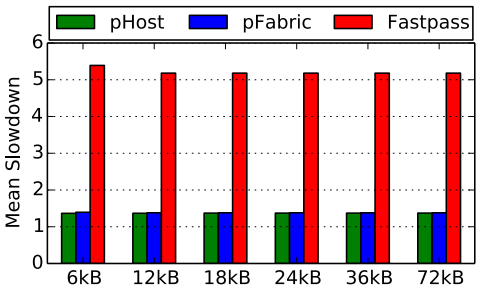


Figure 10: Both `pHost` and pFabric perform well even with tiny buffer sizes. Moreover, the performance of all the three protocols remains consistent across a wide range of switch buffer sizes.

4.4 Flexibility

Our results so far focused purely on performance goals. However, in addition to performance, datacenter operators must also satisfy policy goals – e.g., ensuring isolation or fairness between different tenants. Compared to pFabric, `pHost` offers greater flexibility in meeting such policy goals since `pHost` can implement arbitrary policies for how tokens are granted and consumed at end-hosts. To demonstrate `pHost`'s flexibility, we consider a multi-tenant scenario in which two tenants have different workload characteristics and the operator would like the two tenants to fairly share network bandwidth while allowing each tenant to optimize for slowdown within its share. To achieve this in `pHost`, we configure `pHost`'s token selection mechanism and packet priority assignment to enforce fairness between the two tenants. This is a minor change: we replace the SRPT priority function with one that does SRPT for flows within a tenant, but enforces that the tenant with fewer bytes scheduled so far

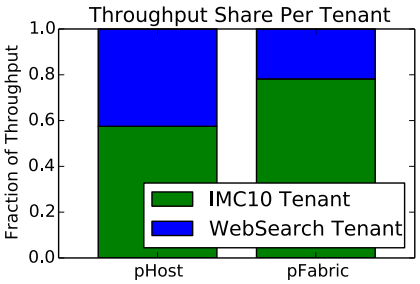


Figure 11: `pHost`, by decoupling flow scheduling from the network fabric, makes is easy to implement diverse policy goals (*e.g.*, fairness in a multi-tenant scenario). In this figure, one tenant gets greater throughput with pFabric (for reasons discussed in §4.4), while the throughput is more fairly allocated using `pHost`.

should be prioritized. Additionally we turn off data packet priorities (all packets go at the same priority) and remove "free token".

We evaluate a scenario in which one tenant's workload uses the IMC10 trace, while the other tenant's workload uses the Web Search trace. Both the tenants inject the flows in their trace at the beginning of the simulation and we measure the throughput each tenant achieves. Figure 11 plots how the overall throughput of the network is shared between the two tenants in pFabric *vs.* `pHost`.

We see that pFabric allows the IMC10 tenant to achieve significantly higher throughput than the Web Search tenant. This is expected because the IMC10 workload has shorter flows and a smaller mean flow size than the Web Search workload (see Figure 2) and hence pFabric implicitly gives the IMC10 tenant higher priority. In contrast, with pHost, the two tenants see similar throughput.

5. RELATED WORK

Our work is related to two key network transport designs proposed recently: pFabric [6] and Fastpass [14]. pFabric is a distributed transport mechanism that achieves near-optimal performance in terms of flow completion times; however, pFabric requires specialized hardward that embeds a specific scheduling policy within the network fabric. This approach not only has the disadvantage of requiring specialized network hardware, but also limits generality — the scheduling algorithm cannot be altered to achieve diverse policy goals. Fastpass aims at generality using commodity network fabric along with a centralized scheduler, but loses many of pFabric's performance benefits. `pHost` achieves the best of the two worlds: the near-optimal performance of pFabric, and the commodity network design of Fastpass.

We compare and contrast `pHost` design against other rate control and flow scheduling mechanisms below.

Rate Control in Datacenters. Several recent proposals in datacenter transport designs use rate control to achieve various performance goals, such as DCTCP [5], D^{2}TCP [17], D^3 [18], PDQ [12], PIAS [8], PASE [13]. Specifically, DCTCP uses rate control (via explicit network feedback) to minimize end-to-end latency for short flows. D^{2}TCP and D^3 use rate control to maximize the number of flows that can meet their respective deadlines. PDQ has goals similar to pFabric; while a radically different approach, PDQ has limitations similar to pFabric — it requires a complicated specialized network fabric that implements PDQ switches. While interesting, all the above designs lose the performance benefits of pFabric [6] either for short flows, or for long flows; moreover, many of these designs require specialized network hardware similar to pFabric. `pHost` requires no specialized hardware, no complex rate calculations at network switches, no centralized global scheduler and no explicit network feedback, and yet, performs surprisingly close to pFabric across a wide variety of workloads and traffic matrices.

Flow Scheduling in Datacenters. Hedera [4] performs flow scheduling at coarse granularities by assigning different paths to large flows to avoid collision. Hedera improves long flow performance, but ignores short flows that may require careful scheduling to meet performance goals when competing with long flows. Mordia [15] schedules at finer granularity ($\sim 100\mu$s), but may also suffer from performance issues for short flows. Indeed, 100μs corresponds to the time to transmit a $\sim$ 121KB flow in a datacenter with 10Gbps access link capacity. In the Data Mining trace, about 80% flows are smaller than that. Fastpass achieves superior performance by performing per-packet scheduling. However, the main disadvantage of Fastpass is the centralized scheduler that leads to performance degradation for short flows (as shown in §4). Specifically, Fastpass schedules an epoch of 8 packets($\sim 10\mu$s) in order to reduce the scheduling and signaling overhead. So no preemption can happen once the 8 packets are scheduled, which fundamentally limits the performance of flows that are smaller than 8 packets. pHost also performs per-packet scheduling but avoids the scalability and performance issues of Fastpass using a completely distributed scheduling at the end hosts.

6. CONCLUSION

There has been tremendous recent work on optimizing flow performance in datacenter networks. The state-of-the-art transport layer design is pFabric, that achieves near-optimal performance but requires specialized network hardware that embeds a specific scheduling policy within the network fabric. We presented `pHost`, a new datacenter transport design that decouples scheduling policies from the network fabric and performs distributed *per-packet* scheduling of flows using the *end-hosts*. `pHost` is simple — it requires no specialized network fabric, no complex computations in the network fabric, no centralized scheduler and no explicit network feedback — and yet, achieves performance surprisingly close to pFabric across all the evaluated workloads and network configurations.

Acknowledgements

We thank our shepherd, Costin Raiciu, and the anonymous reviewers for providing excellent feedback. This work is supported by NSF Grant 1117161, Grant 1343947, and Grant 1040838.

7. REFERENCES

[1] CISCO: Per packet load balancing. http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/pplb.html.

[2] A. Agache and C. Raiciu. GRIN: Utilizing the empty half of full bisection networks. In *Proc. of HotNets*, 2012.

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

[4] 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.

[5] 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.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proc. of SIGCOMM*, 2013.

[7] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS)*, 11(4):319–352, 1993.

[8] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *Proc. of NSDI*, 2015.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of IMC*, 2010.

[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 IEEE 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] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proc. of SIGCOMM*, 2012.

[13] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proc. of SIGCOMM*, 2014.

[14] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "Zero-Queue" datacenter network. In *Proc. of SIGCOMM*, 2014.

[15] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *Proc. of SIGCOMM*, 2013.

[16] C. Raiciu, M. Ionescu, and D. Niculescu. Opening up black box networks with CloudTalk. In *4th USENIX Conference on Hot Topics in Cloud Ccomputing*, 2012.

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

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

[19] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate ddos flooding attacks. In *IEEE Symposium on Security and Privacy*, 2004.