FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks

Abdul Kabbani
Google Inc.
Mountain View, CA, USA
akabbani@google.com

Balajee Vamanan
Purdue University
West Lafayette, IN, USA
bvamanan@ecn.purdue.edu

Jahangir Hasan
Google Inc.
Mountain View, CA, USA
jahangir@google.com

Fabien Duchene
Universite catholique de Louvain
Louvain-La-Neuve, Belgium
fabien.duchene@uclouvain.be

ABSTRACT

Datacenter networks provide high path diversity for traffic between machines. Load balancing traffic across these paths is important for both, latency- and throughput-sensitive applications. The standard load balancing techniques used today obliviously hash a flow to a random path. When long flows collide on the same path, this might lead to long lasting congestion while other paths could be underutilized, degrading performance of other flows as well. Recent proposals to address this shortcoming incur significant implementation complexity at the host that would actually slow down short flows (MPTCP), depend on relatively slow centralized controllers for rerouting large congesting flows (Hedera), or require custom switch hardware, hindering near-term deployment (DeTail).

We propose FlowBender, a novel technique that: (1) Load balances distributively at the granularity of flows instead of packets, avoiding excessive packet reordering. (2) Uses end-host-driven rehashing to trigger dynamic flow-to-path assignment. (3) Recovers from link failures within a Retransmit Timeout (RTO). (4) Amounts to less than 50 lines of critical kernel code and is readily deployable in commodity data centers today. (5) Is very robust and simple to tune. We evaluate FlowBender using both simulations and a real testbed implementation, and show that it improves average and tail latencies significantly compared to state of the art techniques without incurring the significant overhead and complexity of other load balancing schemes.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols

CoNEXT'14, December 2–5, 2014, Sydney, Australia.
ACM 978-1-4503-3279-8/14/12.
http://dx.doi.org/10.1145/2674005.2674985.

General Terms

Algorithms, Design, Performance

Keywords

Data Centers; ECMP; Load Balancing; TCP

1. INTRODUCTION

Datacenter networks are typically based on multistage fat-tree topologies which provide high bisection bandwidth via a large number of paths between any pair of hosts [5, 15]. Efficient utilization of these paths is critical to the performance of datacenter applications. If traffic distribution across paths is uneven, some paths may congest unnecessarily while others go underutilized, adversely affecting the throughput and tail latency of network flows. The impact of long tail latencies on datacenter applications is well understood and has been the subject of recent work [7, 21, 23]. In particular, large-scale online services such as Web search, retail, and advertising run under soft real-time requirements for improved user experience and revenue. Because a user-facing response is constructed by aggregating the results from thousands of servers, the tail latency of the individual flows directly affects response time and quality.

Equal Cost Multiple Path (ECMP) forwarding is the standard mechanism used today for spreading traffic across multiple paths in datacenter networks. ECMP randomly maps a given flow to one of the paths by hashing some fields in the packet headers. The fields for hashing are chosen such that all packets of a given flow follow the same path (i.e., flow to path mapping is static). ECMP works well in balancing the aggregate load across all paths when there are a large number of flows with sufficient entropy across the headers. However, in reality, datacenter traffic is often heavy tailed with a small number of long flows contributing a significant fraction of all traffic [8]. These few long flows may not have enough entropy to uniformly distribute the load. Therefore, a handful of long flows could collide on some path to create a long-lasting congestion while other paths go underutilized. Despite its shortcomings, ECMP has widespread adoption because it is supported by commodity off-the-shelf switches and works out of the box with standard unmodified TCP/IP stacks.

The solution space for load balancing traffic in datacenter networks can be classified along two independent dimensions: (1) whether the solution is static, or whether it dynamically reacts to current network conditions; and (2) the extent of out-of-order delivery of packets in a given flow. For example, ECMP is a static scheme with strict packet ordering. We argue that the most desirable design point in this space would be a solution that is dynamic and has very little to no re-ordering within a flow. Being dynamic makes the scheme robust by freeing it from assumptions about traffic properties (e.g., ECMP requires entropy), and a small degree of packet re-ordering keeps the end host implementation simple and close to stock TCP/IP stacks. Though a number of recent proposals have attempted to address the shortcomings of ECMP, none of them fall under the desirable design point we have described. Switch-driven schemes like *DeTail* [23] are dynamic but incur significant packet re-ordering and additionally require hardware changes to switch silicon which incur high cost and hinder incremental deployment. End-host-driven schemes like MPTCP [17] are dynamic but have significant end host complexity to manage their packet re-ordering. Random Packet Scatter (RPS) [13] is static and also incurs high degree of packet re-ordering.

We propose *FlowBender*, an end-host-driven load balancing scheme that is dynamic but incurs little to no packet re-ordering. FlowBender offers *both* simplicity and high-performance. Our central idea addresses the fundamental problem with ECMP: the flow-to-path assignment is *static*; there is no consideration for re-routing if it turns out that the current path is oversubscribed or even broken. Our solution is to transmit each flow as a single stream, through the same commodity ECMP-based network, until the flow is congested or disconnected, at which point the flow is *selectively re-routed*. We base FlowBender's design on the *key* observation that a handful of long flows account for a large fraction of network load (bytes) [8], and that addressing the load imbalance caused by ECMP's non-uniform hashing amounts *mainly* to re-balancing the long flows. Because long flows tend to span several Round-Trip Times (RTTs), the problem naturally lends itself to an end-to-end design, obviating the need to make hardware changes to switches. Also, because we change a flow's path (route) *selectively* and *only* in response to congestion, we drastically reduce out-of-order packet delivery, thereby nearly *eliminating* packet re-ordering effort.

FlowBender requires two components: (1) a mechanism to detect congestion or link failures at the host, and (2) a means for the end host to reroute a specific flow. For congestion and link failure detection, we improvise on Explicit Congestion Notification (ECN) which is commonly used in today's datacenters, in addition to standard TCP Timeouts. With ECN, senders monitor the fraction of marked packets to decide whether a flow needs to be re-routed. We chose end-to-end signals such as ECN and Timeouts as opposed to link-level, hop-by-hop signals such as Priority Flow Control (PFC) or queue lengths so that we can react to congestion and route failures at any hop within the route while preventing congestion spreading episodes such as Tree Saturation [11, 20, 12].

For re-routing a flow, we configure the hash function in the switches to compute its output based on an additional *flexible* field in the packet header such as Time-To-Live (TTL), in addition to the other fields the switch is already hashing upon. This flexible field would have its value updated independently for a flow *only* when re-routing is desired. The updated value would then be used for all the future packets in that flow (hence arriving in order) until the flow gets rerouted again. Therefore, the flexible field acts conceptually as a path ID for the flow, and changing it causes the flow to explore a new path through the network. Thus, while other competing schemes [23, 17, 13] might end up routing *each* packet differently, we choose to reroute a flow to a different path *selectively* and *only* during congestion or route failures, thereby avoiding the cost of excessive packet re-ordering.

To summarize, the key strengths of FlowBender's design are that it:

- Requires no changes to switch hardware (silicon).
- Amounts to *only* 50 lines of kernel code change.
- Requires simple re-configuration to ECMP hash functions (a handful of commands).
- Substantially outperforms ECMP and matches the performance of other more complex schemes.
- Incorporates robust end-to-end congestion notifications (ECN) and failure signals (Timeouts).
- Reroutes at the Round-trip Time (RTT) granularity and recovers from link failures essentially within an RTO.

We evaluate FlowBender using simulations and a real implementation. With our real implementation we show that FlowBender reduces the tail latency over ECMP by more than 40% on average for large flows. With our at-scale ns-3 [4] simulations with workloads representative of datacenter applications, we show that FlowBender reduces the mean and tail latency of all-to-all workloads by 73% and 93% respectively as compared to ECMP, while staying within 2% of other expensive schemes like DeTail and RPS. With storage-type workloads which generate incast, FlowBender still helps jobs complete in half to quarter the time relative to ECMP (i.e. in terms of the Avg. time for the last flow of an incast job to finish), while also closely matching the performance of other expensive schemes like DeTail and RPS (within 2% of their completion times on average and in terms of the tail). Thus, FlowBender makes a strong case for a non-intrusive, end-host- and flow-level load balancing using the classic end-to-end principle [19].

The remainder of the paper is organized as follows. In Section 2 we discuss related work on improving datacenter load balancing. Section 3 describes our FlowBender proposal in detail. We present our experimental methodology and results in Section 4 followed by several suggestions for further design and performance optimizations in Section 5, and we conclude in Section 6.

2. RELATED WORK

There is a plethora of work that focus on load balancing. While it is beyond the scope of this paper to cover all of them in detail, we present a brief summary of recent work in this area.

Equal Cost Multiple Path (ECMP) forwarding is the standard mechanism used in today's datacenters for load balancing traffic across multiple paths. The switch computes a hash function on the various fields of the packet header that identify a flow, and uses the resulting hash value to pick a port from the set of eligible ports to forward the packet along. Such static flow-to-path assignment can work well

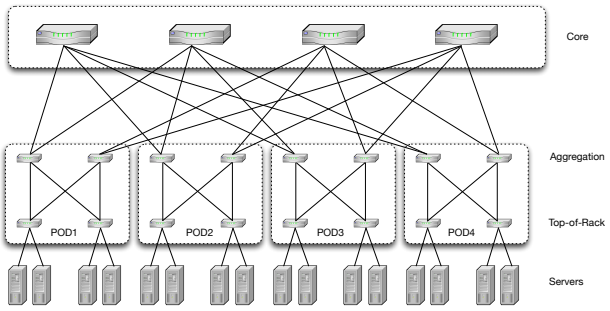

Figure 1: Fat-tree Datacenter Network Topology

when the flows are plentiful and short-lived. However, when there are relatively few long flows present and their hash values happen to collide, these flows suffer long-lasting congestion even though other eligible paths go under-utilized. The key problem with ECMP is that *the flow-to-path assignment is static; there is no opportunity for rerouting if it turns out that the current path is oversubscribed.*

MPTCP [17] addresses the problem by splitting a TCP flow into multiple subflows with different port numbers. The different port numbers, being part of the input to the ECMP hashing function, cause the subflows to hash to different paths through the network. The sender splits the original data stream across the subflows and monitors the RTT, throughput, and packet loss along the various subflows[22] to prioritize those with good performance. The receiver requires additional logic to stitch the arriving data back into the original data stream. While MPTCP shows improvements over ECMP for flows larger than 70KB, it incurs significant implementation complexity because of the more complex sender-side congestion control algorithm and the receiver-side data reassembly logic (more than 10,000 lines of kernel code), rendering its performance worse than ECMP's for smaller flows [18]. Moreover, MPTCP suffers from high CPU overhead, shooting up to around 40% on every client and 10% on every server [17], which is considered to be too costly for datacenters.

SPAIN [16], another host-based load balancing scheme, suggests pre-computing a set of paths spanning the entire network and mapping them to different VLANs. In order to achieve higher throughput and better fault-tolerance, a sender dynamically changes its VLAN if a route associated with the current VLAN fails or at constant intervals in order to shift a long-lived flow to another path to avoid load imbalance. In constrast, FlowBender incurs less overhead as it changes routes selectively and only if a flow is congested. Moreover, if a route has a lower capacity than another between two hosts, FlowBender has the natural tendency to send less traffic on the lower-capacity route (Section 4), whereas SPAIN obliviously uses both paths equally. Unlike SPAIN, FlowBender does not require any routing precomputations and its potential usage of VLANs is merely for adding hashing entropy. We elaborate on FlowBender's usage of VLANs in Section 3.

In contrast to MPTCP and SPAIN, Random Packet Spraying (RPS) [13] and DeTail [23] place the burden of load balancing the traffic upon the switches themselves at the packet level, requiring hardware changes at the switches. With RPS, a switch *randomly* selects an output port from a set of eligible outports *independently* for *every* packet. While RPS works well for symmetric topologies such as fat-trees (aside from the out-of-order packet delivery issues mentioned below), when asymmetries arise in a network due to incremental deployments or link failures, RPS causes severe throughput inefficiencies [23].

DeTail overcomes RPS's limitations by combining two distinct techniques that work synergistically to improve the tail latency and utilization of the network. First, DeTail employs Priority Flow Control (PFC) [1] which guarantees lossless packet-forwarding in the network, but is fraught with the risks of network deadlock and intra-priority head-of-line (HOL) blocking issues [11, 20, 12]. Second, DeTail load balances via a flavor of packet-level adaptive routing [14]. When a packet arrives at the switch, it chooses the less congested eligible port to send on. However, such adaptive routing design has a number of shortcomings. First, it requires custom switching silicon for monitoring queue length changes at line rates and for switching packets based on this information. Custom hardware incurs high cost and long turn-around time that hinders rapid deployment. Second, because DeTail performs load balancing on a per-packet granularity, it causes frequent out-of-order delivery of TCP segments. Accordingly, DeTail entirely disables fast retransmit in the TCP stack. However, a TCP stack with such modification will not operate well when the remote end of the connection is a legacy TCP stack (as could be the case with external or VM-based traffic) or when the intervening network lacks end-to-end PFC or has a large RTT. Furthermore, link-level schemes like DeTail cannot handle link failures that occur on the deterministic part of a path (i.e. when there is a single downlink towards the destination). In contrast, FlowBender, being end-to-end and host-based, can re-route around broken paths within an RTO (section 3).

Centralized approaches [6, 9, 10] have also been proposed to address ECMP's shortcomings and are motivated by the observation that efficiently load balancing datacenter traffic flows entails load balancing the relatively fewer larger ones (given the heavy-tailed nature of datacenter flow size distributions [8]). While software-based switch controllers could be leveraged to globally allocate long flows to non-congested paths, they are not as prompt at handling sudden traffic changes compared to our RTT-based load balancing schemes as we discuss in the next section.

Next we describe FlowBender in details and highlight the major features that distinguish it from the schemes we have just summarized.

3. FLOWBENDER

We now explain the details of FlowBender, beginning with the insight that informs its design.

3.1 Flow versus Packet-level Load Balancing

FlowBender operates in between the extreme of statically sending a flow on one path (ECMP) and that of spreading its packets across multiple paths simultaneously (e.g. RPS, DeTail, and MPTCP). Efficient load balancing does not necessarily mean that we have to simultaneously spread a flow across several paths and then stitch it back together at the receiver, especially when there is no congestion to start with. Instead, we target the simpler mechanism of shifting (rerouting) the entire flow to a different path only once it

is congested or disconnected, thus avoiding sustained out of order packet delivery and any hardware updates to existing datacenter infrastructures.

3.2 Flow Control: Link versus Transport Layer

Link and transport-level congestion signals are important for realizing better rate control mechanisms, but they could be also leveraged for guiding our load balancing decisions as will be explained next. The question we are trying to answer now is: which congestion signals should we leverage?

Link-level notification mechanisms such as PFC have faster reaction times compared to RTT-based ones such as Explicit Congestion Notification (ECN), especially when the congestion point is close to the traffic source itself. When the congestion point is far away from the traffic source, however, such link-level schemes can notoriously result in congestion spreading trees [11, 20, 12]. Rather than worrying about this phenomenon and adding complexity into our switches for load balancing based on PFC signals, our argument is that something simple like ECN has been already demonstrated to be prompt enough in propagating congestion information back to the sources. Of course, relying on ECN end-to-end signals means that we are targeting those longer flows that take several roundtrips to finish, which happen to be those flows carrying most of the network traffic anyway [8]. Otherwise, in the hypothetical scenario where most of the traffic is generated by very small flows only, ECMP should perform quite well handling such traffic given the much higher hashing entropy involved.

Another important reason supporting our transport-layer choice is that, unlike link-level ones, algorithms such as TCP can quickly detect link failures end-to-end, which can influence FlowBender to promptly avoid broken paths as will be discussed in the next section.

With the clear motivation to pursue flow-level and RTT-based load balancing only once congestion occurs, we now proceed to discuss the details of FlowBender.

3.3 The Design of FlowBender

In designing FlowBender we resist any temptations to pursue complex optimization and aim for an ultra-simple approach that can work with today's commodity datacenter switches. We are strictly interested in an approach that avoids all of the following: added switch hardware complexity, sustained out of order packet delivery, high overhead due to complicated multi-pathing techniques, and very coarse load balancing timescales as is typically the case with centralized schemes. All of these issues can be avoided by an approach that *transmits a flow on a single path, through an ECMP-based network, and reroutes that individual flow only once it is congested*. The two questions that naturally follow this last sentence are (1) how is a flow perceived as congested, and (2) how is rerouting initiated in a simple ECMP-based network? We address these questions next.

3.3.1 Sensing Congestion

ECN is a standard congestion notification scheme supported in today's switches. Our FlowBender approach is not strictly tied to one way of triggering and handling this feedback vs another, but we chose to demonstrate it using state of the art DCTCP-style marking, where *a congested switch marks every packet exceeding a desired queue size threshold, and the TCP*[1] *sender keeps track of the fraction of ECN-marked ACKs every RTT*. If this fraction is larger than a certain threshold for any flow, this means that this flow is congested and should be rerouted as discussed next.

3.3.2 Rerouting Flows

The ECMP engines in switches are typically set to compute a hash function based on the packet header fields that uniquely identify a connection. However, the hash engines can be also normally configured to hash upon other fields in the packet as well (e.g. the VLAN ID field from the Ethernet header [2, 3]), and there are actually some already existing commercial platforms today that allow hashing based on programmable header offsets such as hashing based on the TTL value of a packet [2]. Per FlowBender, each TCP socket independently keeps track of the value V it should consistently insert into such a "flexible" hashing field (for every outgoing packet), and the socket updates V only once a flow is congested, thus effectively triggering sender-initiated rerouting then.

Note that from our FlowBender perspective, we are interested in "hacking" only one such flexible field that (i) does not require any pre-negotiations between the sender and the receiver for setting the value of (i.e. in contrast to MPTCP's port numbers) and (ii) does not cause packet delivery failures at the receiver or within the fabric if changed. An example of such a value is the TTL field (as long as its value V is large enough for a data center environment): whether it is set to say 90 or 163, a packet will have no issues getting routed and received properly but would probably traverse a different path in each case. The VLAN ID is another hashing input that is also easy to leverage as a flexible field, even when conventional VLANs are setup.[2]

To recap, each TCP socket sender (i.e. flow) independently keeps track of its value V and the per-RTT fraction of marked ACKs F. Once F exceeds a set threshold T, the current path is considered to be potentially congested and the packets of the corresponding flow are rerouted by changing the value V of the flexible field.

Before we move to discussing low-level details and optimizations, there is one more extremely important aspect of FlowBender that is yet to be emphasized: A packet could be at an advanced stage in its route where the only way for reaching the destination happens to be broken. In such a case, link-level schemes like DeTail or PacketScatter would be stuck unable to reroute around the broken path until the routing tables are updated (which is normally O(seconds) in a large-scale datacenter). In the case of our end-to-end FlowBender approach, however, rerouting can be also triggered once an RTO takes place, bringing the failure recovery time several orders of magnitude smaller! Of course, it is not guaranteed that the new value V would necessarily avoid broken routes from the first attempt. Actually the same could be said in the case of congestion-triggered rerouting, where the new route could be even more congested than the

[1]Datacenter operators have the option of running the transport layer they desire, and the vast majority of flows in a Datacenter are TCP flows anyway [7].

[2]Having a wide range for selecting the value V from usually means that there are more options for rerouting. We experimented with a wide range of values and have unsurprisingly found that even when we restricted each flow to 2 options only, FlowBender was extremely effective. We have empirically chosen a range of 8 options for V in our experiments.

previous one. The catch, however, is that as long as there is a statistical drift to avoid broken or congested routes, they will be avoided, even if this takes a couple of attempts before things are straightened out.

3.4 Recommended Parameter Settings and Simple Optimizations

FlowBender is very simple to tune. In its simplest form, the main parameter to configure is the T congestion threshold, which should be chosen to be small enough in order to alleviate congestion at its early onset. The smaller it is, the sooner we could be avoiding severe congestion episodes, but the more rerouting false alarms we might be getting. We have empirically varied T, the threshold for the percentage of marked ACKs, between 1% and 10% for our experiments and found FlowBender to be very effective across this range.[3] The reason FlowBender continued to be very effective at very low T thresholds is that it would not switch to a new route until at least one RTT has elapsed, thus causing packet reordering only among those packets transmitted around the rerouting instance (i.e. relatively few compared to the extent of reordering in the case of DeTail or RPS). In other words, false alarms, potentially due to bursty marking, would not be that expensive after all.

3.4.1 Even Less Reordering?

If an argument is made for a specific scenario or application where very little packet reordering instances could be tolerated, FlowBender could be readjusted to reroute only after a flow is consistently congested (T is exceeded) for N consecutive RTTs.[4] The pseudocode for the basic FlowBender algorithm together with this addition is shown below. We ran another instance of the experiments described in the next section with such an optimization, where N was set equal to 2, and saw very similar performance compared to the basic version described earlier.

```
for every RTT do
  F ← num_marked_pkts/total_pkts
  if F > T then
    num_congested_rtts + +
    if num_congested_rtts >= N then
      num_congested_rtts ← 0
      Change V
    end if
  else
    num_congested_rtts ← 0
  end if
end for
```

As is clear from its pseudo code and description, FlowBender's design is simple enough that its complete implementation requires only about 50 lines of kernel code on the hosts, and 5 lines of configuration code on the switches. Such simplicity is in stark contrast to the software complexity of schemes like MPTCP and the hardware complexity of DeTail.

[3]All of this while maintaining the default tcp_reordering threshold of 3 and confirming that the CPU overhead does not increase due to FlowBender.

[4]One might even want to exponentially average F across those N RTTs for a smoother reaction, but we don't think such optimizations are necessary at this stage.

3.4.2 Desynchronizing Flows

When a link is congested, it could be that two or more long flows would be triggered to reroute at the same time, and it's important to ensure that we are avoiding the situation where multiple simultaneous rerouting events would cascade into a rerouting wave in the fabric. Reacting at a very early congestion onset and having the RTT epochs for calculating F naturally jittered seems to be already quite helpful desynchronizing these events per the results discussed in the next section. Another design option for fortifying these jitters further is by avoiding that all flows reroute after exactly N consecutively congested RTTs, and randomly choosing the number to be something like N, $N+1$, or $N-1$ instead.

3.4.3 FlowBender beyond TCP

Even though our discussion in this paper has been focused on illustrating FlowBender's potential on top of TCP, the same load balancing principles could be easily applied to other transport protocols that sense congestion (whether CE-based, drop-based, RTT-based, etc). Moreover, for unreliable transport protocol flows such as UDP, one can argue that packet or burst-level "spraying," by changing the V value at the desired pace, would be extremely helpful for load-balancing the network traffic more effectively, especially given that applications that use UDP are usually robust or oblivious to packet reordering (of course, Weighted instead of Equal Cost Multi-pathing is required anyway to handle non-uniform link capacities and topological asymmetries).

3.4.4 Other Optimizations?

We clearly don't claim that FlowBender provides the absolutely best performance or would always lead to an optimal performance in theory. There are several optimizations that can be applied for improving FlowBender, as discussed in Section 5. However, at this stage, we are more interested in illustrating how this simplest version can still drastically improve datacenters performance!

4. EVALUATION

We evaluate FlowBender against existing schemes (i.e., ECMP, RPS, and DeTail) using both simulations and a real implementation. DeTail requires custom silicon modification so we rely on simulations to perform at-scale evaluations. We simulate production-like workloads in a typical datacenter-like topology. We also evaluate FlowBender on real testbed implementation as a proof-of-concept and to capture the effect of real system limitations and side-effects (e.g., Large Segment Offload (LSO)).

4.1 A Note on MPTCP

In our evaluations, we were not able to compare to MPTCP for the following reasons: (1) There is a lack of a reliable NS simulation package that we could leverage. (2) The latest publicly available MPTCP Linux release suffers from performance issues due to several implementation artifacts. These issues impact small flows drastically (similar to those reported in [13]), to the extent that the performance of MPTCP can be lower than a well-tuned ECMP network. Because the performance numbers capture the effect of implementation artifacts instead of the technical idea itself, such a comparison would be unfair to MPTCP and we opt

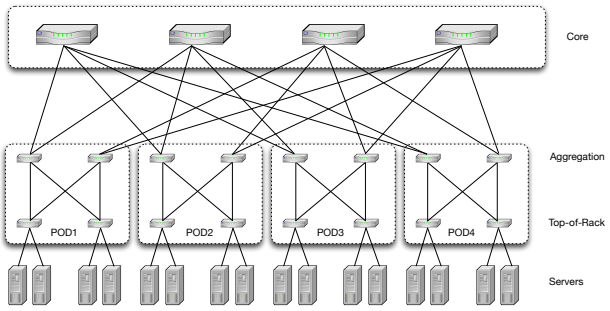

Figure 2: Fat-tree Datacenter Network Topology

to omit such a comparison. That said, even if those artifacts were to be adjusted appropriately, MPTCP's improvement relative to FlowBender's, if any, would be marginal because both DeTail and FlowBender closely match the performance of RPS (Section 4.2.3 and Section 4.2.5), which is optimal for the symmetric fat-tree datacenter topologies we evaluate.

In addition to quantitative performance, MPTCP possesses certain qualitatively undesirable design issues. For example, MPTCP has complex implementation, and because it pre-negotiates multiple routes for every connection it unnecessarily incurs higher overhead for very short flows.

4.2 Simulation

4.2.1 Methodology

Topology: We use ns-3 [4] simulations to evaluate FlowBender on production-like workloads. We model the network topology and the traffic after typical production deployments. We simulate a fat-tree network [15] analogous to that depicted in Figure 2. The network has 128 servers, organized into four pods, each having four Top of the Rack (ToR) and four aggregation switches, with eight core switches interconnecting the pods (overall oversubscription factor of four from servers to core switches). Because 10 Gbps ethernet is typical in today's datacenters, we use 10 Gbps point-to-point ethernet links across our entire network. All our switches use combined input-output queuing in order to better suit DeTail's requirements (the switch architecture does not impact ECMP, FlowBender or RPS). We configure the host delay to be 20 μs and the switch delay to be 1 μs, and we obtain a *baremetal RTT* of $2 \times 5 \times 1 + 4 \times 20 = 90\mu$s between two servers on different pods, which is realistic per today's datacenters RTTs.

ECMP: Because DCTCP provides faster congestion response and helps eliminate costly timeouts, we chose DCTCP as the base TCP stack upon which we build all the other schemes. We start with ns-3's TCP New Reno protocol, and implement DCTCP [7] on top of it, faithfully capturing all its optimizations. We set the parameters of DCTCP to match those in [7]: (1) g, the factor for exponential weighted averaging, is set to $\frac{1}{16}$; and (2) K, the buffer occupancy threshold for setting the CE-bit, is set to 90 KB (typical for 10 Gbps links). So, our base case (ECMP) is DCTCP running over a commodity datacenter network with ECMP-enabled switches.

FlowBender: We implement FlowBender on top of DCTCP. We set T, the congestion threshold, to 5%, and N, the number of RTTs a sender must be congested before switching paths, to 1. Both DCTCP and FlowBender use an RTO_{min} of $10ms$.

DeTail: We use the same implementation from [23] in conjunction with DCTCP instead of the standard DropTail TCP. We set the pause and unpause thresholds to 20 KB and 10KB on the ingress queues respectively, per the guidelines in [1]. As suggested by the DeTail paper, we also disable TCP's fast retransmit to handle out-of-order packet delivery. The DeTail paper uses two thresholds of 16 KB and 64 KB to reduce the implementation complexity of determining the least congested output port at the switches, at the cost of a small performance loss (due to inaccuracy in the determination of the least congested output port). To abstract away the effect of this trade-off, we *generously* do a full comparison across all output port counters in DeTail to *always* find the minimally congested path without introducing *any* additional latency/throughput overhead due to this optimization. Thus, we compare FlowBender to the best-possible implementation of DeTail.

RPS: Our implementation of RPS consists of DCTCP running over switches that support RPS i.e., switches uniformly randomly choose a random output port from a set of eligible ports.

4.2.2 Functionality Verification

We start by evaluating FlowBender's efficiency in load balancing large flows to validate its functionality. In this experiment, we simultaneously initiate a small number of 250 MB flows from hosts on one ToR in a specific pod transmitting to hosts on another specific ToR in a different pod. We compare the average and worst completion times of the flows with FlowBender to that under ECMP. Because all flows are of equal size, as load balancing *improves*, both the mean and the maximum flow completion times improve. Furthermore, with better load balancing, we expect a *tighter* distribution of flow completion times i.e., the mean and the tail are close. Therefore, we can think of the ratio between the mean and the tail as a quantitative measure for quality of load balancing.

Table 1: FlowBender's flow completion times relative to ECMP's

Flows	ECMP (ms)		FlowBender (ms)	
	Mean	Max	Mean	Max
8	588	1950	294	367
16	1468	5220	580	740
24	2515	9238	897	1144

In this experiment, we vary the number of flows as 8, 16 and 24 flows, which translates to an average number of 1, 2 and 3 flows per route respectively. Consequently, we expect the best flow completion times to be roughly 200, 400, and 600 milliseconds respectively (modulo the round-trip time delay and assuming instantaneous rate convergence to the fair share with no slow-start delays). Table 1 shows the mean and maximum flow completion times. As expected, we see that FlowBender improves ECMP's mean by 2x and maximum flow completion times by 5-8x respectively. Also, the ratio of the maximum flow completion time to the mean

flow completion time is more than 3.3 with ECMP, while with FlowBender the ratio reduces to less than 1.3 implying a tighter latency distribution with lower variance.

4.2.3 All-to-All Workloads

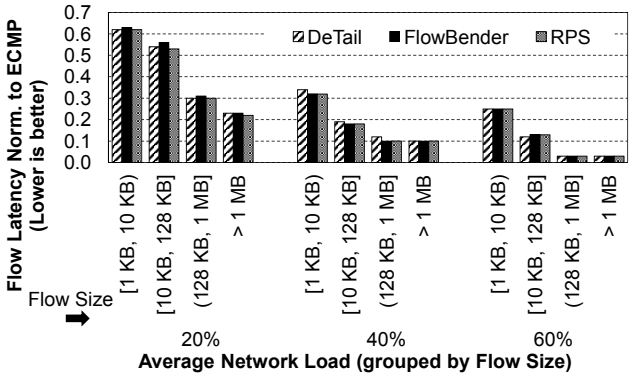

Figure 3: All-to-All workloads: Mean Latency Norm. to ECMP

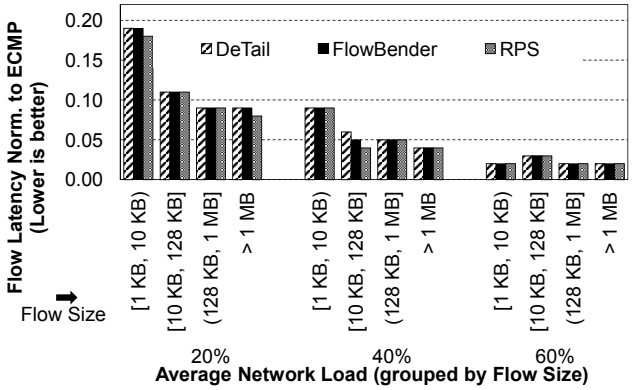

Figure 4: All-to-All workloads: Tail Latency Norm. to ECMP

We compare ECMP, RPS, DeTail, and FlowBender with traffic patterns typical of large-scale online services such as Web search. Our flow size distribution is heavy-tailed and is modeled based on the data from [8]. Every server randomly picks another server in the network to send data to, and flows arrive at the sender as a poisson process with the mean adjusted to produce the desired load. We vary the load (reported relative to the bisectional bandwidth), and compare the different schemes in terms of the mean and the $99^{th} percentile$ latency.

We show the means in Figure 3 and the 99^{th} percentile latencies in Figure 4, with each being binned across flow sizes: (a) less than or equal to 10KB, (b) greater than 10KB but less than or equal to 128KB, (c) greater than 128KB but less than or equal to 1MB, and (d) greater than 1MB. The load is plotted along the X-axis, and we plot the latency of RPS, DeTail, and FlowBender normalized to that of ECMP along the Y-axis. Across all of these loads, we see that all the three schemes i.e., DeTail, FlowBender and RPS *substantially* outperform ECMP. The high level takeaway from our experiments here is that though DeTail, RPS, and FlowBender achieve similar performance in general, FlowBender does not require *any* hardware change at the switches *and* is not strictly predesigned to operate in symmetric topologies, unlike DeTail and RPS.

4.2.4 Out-of-order Packet Delivery

FlowBender incurs *negligble* out-of-order packets relative to what DeTail and RPS incur. In fact, we monitored the number of out-of-order packets in all simulations and found that with FlowBender, the probability of a packet arriving out of order was about 0.006% higher than ECMP's. Compared to FlowBender's performance, DeTail had more than 97.9% the number of out-of-order packet delivery that RPS experienced.

Furthermore, DeTail suffers from the Tree Saturation problem due to its reliance on link-level PFC signals [11, 20, 12], and RPS cannot cope with asymmetric topologies and link failures [23].

4.2.5 Partition Aggregate Workloads

In this experiment, we evaluate FlowBender's ability in reducing tail latency under synchronized Partition-Aggregate traffic patterns typical of datacenter applications like Web search. We use the same heavy-tailed flow size distribution here that is used in the previous all-to-all experiment. We initiate Partition-Aggregate jobs as a Poisson arrival process with mean such that the total utilization of the bisectional bandwidth is equal to 40%. Each Partition-Aggregate job corresponds to a 1MB transaction broken evenly across n workers that are spread randomly in the fabric with all workers responding simultaneously together with their data but essentially finishing at different times. In Figure 5, we vary n between 4 and 32 along the X-axis and plot the average job completion time normalized to ECMP's i.e., the average of flow completion times of flows that finish last in each job, along the Y-axis. As expected, FlowBender performs better with smaller fan-in degrees (larger flow sizes) than with larger fan-in degrees (smaller flow sizes). FlowBender's performance, like that of RPS and DeTail, suffers with a larger fan-in degree due to synchronized packet arrivals to the destination ToR with the receiver's last hop being *the* bottleneck i.e., multipathing does not help in principle. Overall, across different fan-in degrees, FlowBender achieves a reduction in the average job completion time by a factor of *four* in the best case (fan-in degree of 4), and by a factor of two in the worst case (fan-in degree of 32), and closely matches the performance of other expensive schemes such as DeTail and RPS.

4.2.6 Sensitivity Analysis

In this section, we perform sensitivity analysis of FlowBender, by varying two of its primary knobs: N, the number of RTTs a sender must be congested for before switching paths, and T, the congestion threshold.

In Figure 6, we vary N along the X-axis and show the mean latency across different flow sizes normalized to the default setting i.e., $N = 1$. As expected, as N increases, FlowBender's performance suffers as its response slows down with higher values of N. Nevertheless, the performance variation to N is *marginal*, and FlowBender exhibits robust performance across a broad range of values.

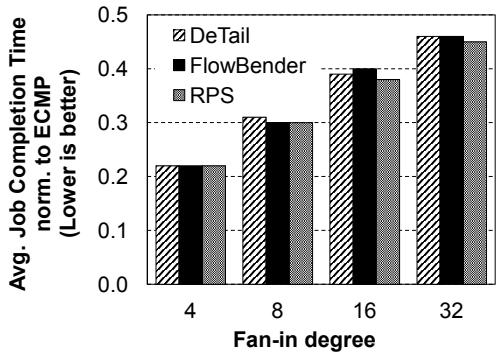

Figure 5: Partition Aggregate Workloads: Avg. Job Completion Time

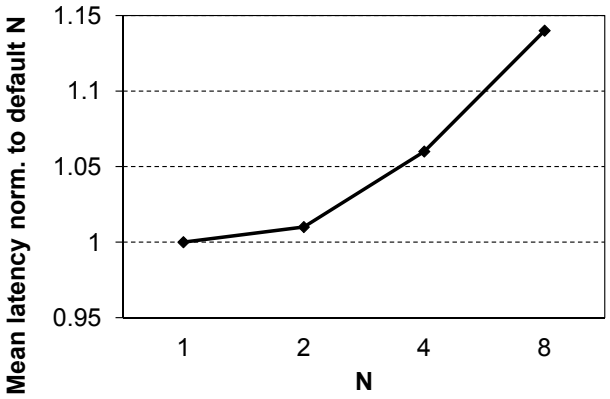

Figure 6: Sensitivity to N

In Figure 7, we vary T, the congestion threshold, along X-axis, and show the mean latency across different flow sizes normalized to the its default setting i.e., $T = 5\%$. For small values of T i.e., $T = 1\%$, FlowBender suffers marginal performance degradation as it responds to bursty spikes in the fraction of marked packets. We attain the best performance for $T = 5\%$. Increasing T beyond 5% slows down FlowBender's response, causing marginal performance degradation. Overall, similar to N, we see that FlowBender's performance is relatively robust across a wide range of T values.

4.3 Testbed Implementation

Our real implementation (testbed) has 15 ToR switches with 12 to 16 servers each. The servers are connected to the ToRs via $10Gbps$ links, and the ToRs are interconnected via 4 aggregation switches with one $10Gbps$ link to each of the 4 switches. In other words, each server has 4 distinct paths to reach any other server on the other ToR. Servers are running with Linux 3.0, including the aforementioned FlowBender changes (less than 50 lines of code added to the kernel) and the DCTCP implementation, and have their RTO set to $10msec$. We use standard ECMP-capable switches with a shared buffer space of $2MB$. The switches are configured with the CE marking threshold set to $90KB$.

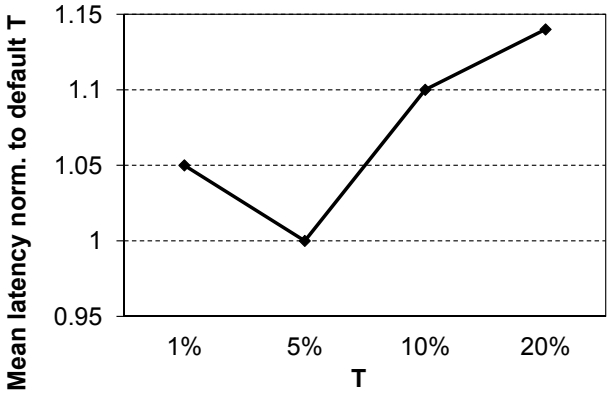

Figure 7: Sensitivity to T

4.3.1 *All-to-All Traffic*

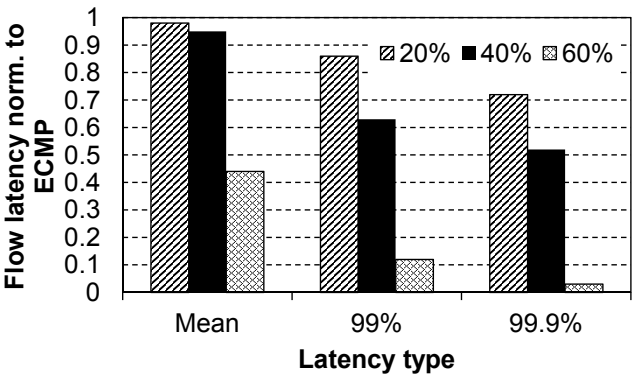

Figure 8: FlowBender's Latency Reduction at 20, 40, and 60% load (Bisectional)

In this experiment servers on one ToR initiate 1 MB flows randomly to any other server in the network with exponential inter-arrival times at a rate that cumulatively amounts to 20%, 40%, or 60% average utilization across the bisectional links. We initiate a total of 1.2 million flows, and wait for all flows to finish. We use the default TCP re-ordering threshold of 3, and monitor the out-of-order delivery numbers to ensure that FlowBender does not introduce *undesired CPU (processing) overhead*. To reconfirm that FlowBender does not lead to any abnormal packet re-ordering activity, we re-ran our experiments with a TCP re-ordering threshold of 30, and we didn't see any noticeable difference in performance.

In Figure 8, we show the *mean*, the 99^{th} percentile, and the 99.9^{th} percentile latencies along the X-axis, and FlowBender's completion time normalized to that of ECMP along the Y-axis. We use default parameter settings for FlowBender i.e., $N = 1$ and $T = 5\%$. FlowBender improves the 99^{th} and 99.9^{th} percentiles by $15 - 26\%$ and $34 - 45\%$ respectively, in comparison to ECMP. At 60% load, FlowBender's flows finish more than *twice* as fast as ECMP on average, and $87 - 96\%$ faster at the tail end.

A real implementation has a number of performance-affecting system-level details (e.g., application-level delays, kernel la-

tencies, offload inefficiencies, CPU power-saving modes, etc.) which are typically absent in simulations. Accordingly we see that the testbed results are not quantitatively equal to that of simulations. However the qualitative results are similar and support the claim that FlowBender offers drastic improvement over static schemes like ECMP.

4.3.2 Decongesting HotSpots

We now evaluate FlowBender's ability to decongest flows by re-routing them around hotspots. We simplify our traffic matrix in this case and initiate an all-to-all random shuffle of $1MB$ TCP flows from one ToR to another (all in the same direction). The aggregate TCP traffic generates $14Gbps$ from the sending ToR (arbitrarily spread across the 4 $10Gbps$ links). We also initiate 1 UDP flow between the same pair of ToRs, in the same direction as the TCP traffic, and rate limit it to $6Gbps$. The purpose of the UDP flow is to create a static (asymmetric) hot spot along one of the four paths given that this flow will not be re-routed or load balanced by FlowBender. We denote the path which this UDP flow hashes on by U.

Note that the aggregate TCP and UDP traffic on the four routes between the sending and the receiving ToRs amounts to 20Gbps. Hence, in an ideal setting, one would wish that the $14Gbps$ would have been equally split across the three paths other than U given that $14/3Gbps$ is still less than UDP's $6Gbps$ that was already routed on U. With ECMP, on one hand, U was unsurprisingly getting around quarter of the TCP traffic ($14/4 = 3.5Gbps$) obliviously mapped to it, thus ending up with around $9.5Gbps$ on average in total, driving that link practically unstable. FlowBender, on the other hand, succeeded in load balancing the traffic to a great extent with only around $1.5Gbps$ of the $14Gbps$ going on U. This experiment confirms FlowBender's ability to adaptively re-route around congestive hotspots in the network, and to respond to congestion created by non-TCP traffic.

The above experiment is also interesting from a Weighted Cost Multipathing (WCMP) perspective (as opposed to ECMP) in the context of asymmetric topologies, where in order to reach a certain destination group, the viable ports at a switch are configured with different forwarding weights so as not to prematurely oversubscribe those paths with lower capacities. One of the challenges with WCMP is to be able to reflect the different weights of the forwarding ports accurately, which is highly dependent on how many entries the forwarding table can accommodate as per current ECMP implementations (i.e. larger tables can represent the different weights with higher granularity). The significance of this experiment is in how even if the forwarding weights suffer some inaccuracy because of the forwarding table being constrained to have few entries only (as is the case with our testbed and most of the commodity switches), FlowBender is able to dynamically re-adjust the traffic on the different available paths such that those with lower capacities are not severely congested (i.e. more robustness to forwarding weight misconfigurations or chip limitations).

4.3.3 Topological Dependencies

Our simulation and testbed results above are based on a topology with 8 and 4 different paths between any pair of pods or ToRs respectively. A question that commonly arises here is: how helpful would FlowBender be when the path diversity between any pair of pods increases? In other words, what role does FlowBender play if the port density of each switch is, say, doubled, together with the number of servers per ToR, while keeping the over-subscriptions ratios the same (i.e. the path diversity quadruples)? The extent to which FlowBender helps clearly depends on the number of the available paths P, but it also depends on the number of those larger flows L that we're trying to spread out on those paths. More precisely, and particularly true in the limits, the performance improvement depends on the ratio $R = L/P$ of the two numbers as we show next, which is typically expected to remain constant given that as the bisectional capacity (i.e. P) is scaled up, the load, and hence, L would be also scaled up proportionally to maintain the same utilization.

Considering the micro benchmark basic validation results discussed earlier, where FlowBender is shown to do a good job in evenly spreading the flows across the different paths, FlowBender's performance improvement amounts to how bad ECMP's flow distribution performance was in the first place. Given the oblivious nature of ECMP, the distribution of the number of large flows per each route is, in steady state, a very straightforward binomial distribution with a mean R and a variance $R(1 - 1/P)$, which is therefore not that different for a reasonably large P. For example, varying P from 8 to 32 would increase the variance by less than 11% only and hence would have a negligible effect in practice. In fact, we reran our All-to-All experiments with a different fan-out degree, and the performance improvement due to FlowBender was almost the same.

5. FURTHER OPTIMIZATIONS AND FUTURE WORK

Our paper presents one of the most basic versions of FlowBender. As discussed earlier, we have intentionally resisted any temptations to optimize FlowBender in the interest of demonstrating how effective this simple idea is, and our evaluation results confirm our position for all the practical cases that we have discussed. Looking forward, however, FlowBender might be desired to operate in more challenging environments, potentially outside datacenters, and could benefit from some of the suggestions summarized below.

5.1 Stability

FlowBender does not monitor the load on all available paths before taking a rerouting decision. Instead, it randomly chooses a new path when it detects that the current path is congested. Therefore, the new path it reroutes to may be also congested. If that happens to be the case because say of an incast episode or because the network is highly congested in general, FlowBender will trigger yet another path change, and the rerouting process may repeat. In some pathological cases, such a design could continuously thrash from one path to the other if no further measures for preventing this are taken. Because every rerouting instance carries the potential of out-of-order delivery of packets to the receiver, such thrashing is undesirable.[5] Accordingly, FlowBender can be extended to limit the number of path

[5]Even though such artifacts are more likely to occur in an incast situation, the extent to which they occurred in our simulations was negligible as is evident from the improvement ratios discussed earlier. We do bring up the feature suggested herein, however, so as to add more confidence around FlowBender's ability to handle some of those quite

changes that could occur when the network is severely congested. More specifically, FlowBender can be constrained to switch paths for a maximum of S consecutive times before it goes into a *locked* state. In the locked state, it will pick one path out of the last S paths that had the lowest value of F, the fraction of ECN-marked ACKs, and will lock in to that path for the next U RTTs. At the end of the locked phase, FlowBender resumes business as usual, tracking F and switching paths if it exceeds T for N consecutive RTTs. Note that if we were to choose S and U as 5 and 10, respectively, with an N of 2 (which gives almost the same performance as the default $N = 1$ configuration), then we would be limiting the rerouting events to a maximum of 5 times in every $5 \times 2 + 10 = 20$ RTTs, thus significantly limiting the number of out-of-order packets that could be triggered by FlowBender.

5.2 Gradual Rerouting

Instead of shifting the traffic of a congested flow all at once from one route to another, the rerouting process can be attempted gradually. Of course, this is assuming the transport layer is adjusted to tolerate out-of-order packe delivery. For example, we can start by initially transmitting 10% of the traffic of a congested flow on a new path[6] while monitoring whether this has any effect on alleviating congestion. If congestion has been already mitigated by partially migrating to the new path, then this could be an indication that more traffic should be routed on the new path (e.g. go up to transmitting 20% of the traffic on the new path and so on). Otherwise, it could be that the desire to send on a different, uncongested path was not really met (potentially because of partially or fully overlapping with the existing path at one or more congested links), so the flow would attempt to send on a different new path instead (i.e. abandon the intermediate one).

The above discussion has focused on the case of load balancing across 2 subflows, which might be sufficient for most practical purposes, but the same approach can be extended to load balancing across more than two paths simultaneously.

5.3 Selective Rehashing

In our current implementation, we change the value of V for the flexible hashing input field obliviously once congestion occurs. Changing the value of V, however, does not always mean that the route will change as this really depends on the hashing function. That said, given sufficient knowledge about the hashing functions in the fabric, one can avoid this artifact by having each flow precompute a number of potential values for V that would result in hashing to different paths. The process for precomputing such values might be very challenging and time consuming to perform for general hashing functions, but if the network operators choose their hashing functions in a way such that flipping one of the hashing inputs bits would result in a different hashing output (e.g. XOR hash functions), then this process would become much easier to perform. Alternatively, flows can correlate the different RTT estimates or values for F corresponding to different V's in order to infer, with a high probability, how these V's map to different paths and avoid choosing a redundant value for rerouting.

5.4 Proactive Probing and Route Caching

We have demonstrated the reactive version of FlowBender, where a flow is rerouted only once congestion has been detected. Of course, one might argue that FlowBender is already quite prompt in rerouting when congestion arises, given its low congestion detection threshold T, but the load balancing performance could be still further improved by allowing a flow to proactively probe across the different paths. By probing we mean that a flow can periodically send a few of its packets with different V values, keeping track of their sequence numbers ranges, and check once they have been acked which of them have experienced congestion. One of those V's corresponding to probe packets which did not seem to be congested at all could be proactively selected as the basic V once F has exceeded a threshold T' smaller than T. Alternatively, a flow may attempt to keep track of some of those *better* V's to hash upon once congestion occurs, instead of choosing V obliviously, or might simply blacklist those highly congested V's and avoid revisiting them until sufficient time has elapsed.

5.5 Rerouting and Flow Control

Rerouting, in the load balancing sense, and flow control, in the congestion control sense, are two ways for handling congestion in multipath environments. When both mechanisms rely on the same congestion signal (e.g. ECN driving both, FlowBender and DCTCP), it becomes useful to decide on which of the two mechanisms should be triggered based on the nature of the congestion event. For example, if congestion is occuring at the receiving ToR (e.g. an incast event), then it is unlikely that rerouting the individual congested responses will help (though rerouting still did not have a noticeable negative impact on the overall performance per our simulation results). Alternatively, if a flow is mainly bottlenecked at a core switch while other core switches are lightly loaded, then slowing down the rate of this flow would be rather harmful when it could have been rerouted instead. In our evaluation for FlowBender, we did not attempt any optimizations along these lines given that DCTCP would not have kicked in aggressively yet once the very low threshold T would have been just exceeded. In the future, however, FlowBender and other congestion control and load balancing mechanisms could significantly benefit from more informative congestion control signals that can somehow distinguish whether congestion is occuring at the edge of the fabric and/or somewhere else.

6. CONCLUSION

In this paper, we proposed a new load balancing mechanism called FlowBender. The main motivation for introducing FlowBender is to overcome the limitations of oblivious hashing schemes such as ECMP, prominent in today's datacenters, without suffering from high packet re-ordering or requiring custom hardware changes and complicated host mechanisms that could offset any potential benefits. FlowBender is a host-based, congestion-driven scheme that distributively re-routes individual flows around congested hotspots only once congestion is experienced, end-to-end, or when link failures occur. FlowBender relies on ECN and ECMP

unlikely pathological scenarios that our evaluations did not cover.

[6]This can be done in a periodic manner to avoid CPU-intensive computations for generating random numbers (e.g. every 10th pkt is sent on a new path).

switch support, which is typical in today's datacenters, and implements the load-balancing logic at the end-hosts via a very straightforward kernel patch. In contrast to centralized flow scheduling schemes, FlowBender recovers from link failures after an RTO occurs and reroutes congested flows at the RTT granularity, several orders of magnitude faster than state of the art routing management schemes.

Our ns-3 [4] simulations, with workloads representative of datacenter applications, show that FlowBender *substantially* reduces the mean and tail latency compared to ECMP, while achieving performance very similar to that of other expensive or undesirable schemes like DeTail and RPS. In particular, our tail latencies are reduced by more than 5x to 20x relative to ECMP's, and our partition-aggregate jobs are about 2x to 4x faster on average. We have also evaluated FlowBender using a real implentation and found that it cuts the flow completion tail latencies by around 40% relative to ECMP's for large flows.

7. REFERENCES

[1] 802.1Qbb - priority-based flow control. `http://www.ieee802.org/1/pages/802.1bb.html`.

[2] Avoiding network polarization and increasing visibility in cloud networks using broadcom smart hash technology. `http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf`.

[3] Cisco cli command reference. `http://www.cisco.com/en/US/docs/wireless/asr_901/Command/Reference/Cmdref_asr901.html`.

[4] NS-3 network simulator. `http://www.nsnam.org/`.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[9] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM*, pages 1629–1637. IEEE.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[11] W. J. Dally. Virtual-channel flow control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, Mar. 1992.

[12] D. M. Dias and M. Kumar. Preventing congestion in multistage networks in the presence of hotspots. In *ICPP (1)'89*, pages 9–13, 1989.

[13] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2130–2138, 2013.

[14] J. Kim, W. J. Dally, and D. Abts. Adaptive routing in high-radix clos network. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[15] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, Oct. 1985.

[16] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010.

[17] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[18] C. Raiciu, C. Paasch, S. BarrÈ, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12), San Jose (CA)*, 2012.

[19] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, pages 277–288, 1984.

[20] J. R. Santos, Y. Turner, and G. (john Janakiraman. End-to-end congestion control for infiniband. In *In proceedings of Infocom03*, 2003.

[21] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[22] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.

[23] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 139–150, New York, NY, USA, 2012. ACM.